Contents

[ANGULAR 4](#_Toc80384924)

[OVERVIEW 4](#_Toc80384925)

[SETTING UP THE DEVELOPMENT ENVIRONMENT 4](#_Toc80384926)

[CREATING ANGULAR PROJECT 4](#_Toc80384927)

[FOLDER STRUCTURE 4](#_Toc80384928)

[APP MODULE 4](#_Toc80384929)

[ROOT COMPONENT IN DETAILS 5](#_Toc80384930)

[HOW ANGULAR APPLICATION IS BOOTSTRAPED 5](#_Toc80384931)

[CSS IN ANGULAR PROJECTS 6](#_Toc80384932)

[INSTALLING BOOTSTRAP CSS USING NPM 6](#_Toc80384933)

[ADDING IMAGE TO PROJECT 6](#_Toc80384934)

[CREATING THE FIRST CUSTOM COMPONENT 7](#_Toc80384935)

[COMPONENT SELECTORS 8](#_Toc80384936)

[DATA BINDING 8](#_Toc80384937)

[STRING INTERPOLATION - ONE WAY DATA BINDING 8](#_Toc80384938)

[PROPERTY BINDING - ONE WAY DATA BINDING 9](#_Toc80384939)

[EVENT HANDLING 9](#_Toc80384940)

[TWO WAY DATA BINDING [ngModel Directive] 10](#_Toc80384941)

[ANGULAR DIRECTIVES 10](#_Toc80384942)

[PRE-DEFINED DIRECTIVES 11](#_Toc80384943)

[ATTRIBUTE DIRECTIVES 12](#_Toc80384944)

[CUSTOM DIRECTIVES 13](#_Toc80384945)

[LOCAL REFEREENCE IN TEMPLATE 15](#_Toc80384946)

[COMMUNICATION BETWEEN COMPONENTS 16](#_Toc80384947)

[PARENT TO CHILD COMMUNICATION - PROPERTY BINDING AND @Input DECORATOR 16](#_Toc80384948)

[CHILD TO PARENT COMMUNICATION USING VIEWCHILD AND VIEWCHILDEN 19](#_Toc80384949)

[CHILD TO PARENT COMMUNICATION USING EVENTS - @Output 19](#_Toc80384950)

[@VIEWCHILD 21](#_Toc80384951)

[@VIEWCHILDREN 22](#_Toc80384952)

[VIEW ENCAPSULATION 24](#_Toc80384953)

[CONTENT PROJECTION 25](#_Toc80384954)

[CONTENT PROJECTION – ngContent 25](#_Toc80384955)

[MULTI SLOT CONTENT PROJECTION 25](#_Toc80384956)

[CONTENT PROJECTION - @ContentChild 27](#_Toc80384957)

[CONTENT PROJECTION - @ContentChildren 28](#_Toc80384958)

[ANGULAR LIFECYCLE HOOKS 28](#_Toc80384959)

[CREATING A SERVICE 29](#_Toc80384960)

[USING SERVICES 30](#_Toc80384961)

[HIERARCHICAL DEPENDENCY INJECTION 30](#_Toc80384962)

[INJECTING SERVICE INTO ANOTHER SERVICE 30](#_Toc80384963)

[CROSS COMPONENT COMMUNICATION USING SERVICES 31](#_Toc80384964)

[MAKING HTTP REQUEST 32](#_Toc80384965)

[BACKEND SET UP - SETTING UP FIREBASE 32](#_Toc80384966)

[POST REQUEST 34](#_Toc80384967)

[GET REQUEST 34](#_Toc80384968)

[FORMS 34](#_Toc80384969)

[TEMPLATE DRIVEN APPROACH 34](#_Toc80384970)

[ANGULAR MODULE 34](#_Toc80384971)

[ROUTING 36](#_Toc80384972)

[ROUTING TO MODULES 36](#_Toc80384973)

[ROUTING TO COMPONENTS 36](#_Toc80384974)

[SETTING UP ROUTE 37](#_Toc80384975)

[RELATIVE AND ABSOLUTE PATHS IN ROUTES 37](#_Toc80384976)

[NAVIGATING TO ROUTE PROGRAMATICALLY 37](#_Toc80384977)

[ROUTE PARAMETER PASSING IN ROUTE 38](#_Toc80384978)

[ROUTE OBSERVABLES 39](#_Toc80384979)

[QUERY STRING PARAMETERS AND FRAGMENTS - PASSING IN ROUTE 39](#_Toc80384980)

[CHILD ROUTES 40](#_Toc80384981)

[REDIRECTION AND WILD CHARACTERS IN ROUTE CONFIGURATION 41](#_Toc80384982)

[ROUTE GUARDS 42](#_Toc80384983)

[RXJS 42](#_Toc80384984)

[TERMINOLOGIES 42](#_Toc80384985)

[OBSERVABLES 42](#_Toc80384986)

[ANGULAR PIPES 45](#_Toc80384987)

[BUILT IN PIPES 46](#_Toc80384988)

[PARAMETERIZING AND CHAINING PIPES 46](#_Toc80384989)

[CREATING CUSTOM PIPES 47](#_Toc80384990)

[ANGULAR MODULES & OPTIMIZATION 48](#_Toc80384991)

[ANGULAR DEPLOYMENT 48](#_Toc80384992)

[ANGULAR UNIVERSAL 49](#_Toc80384993)

[WHAT IS ANGULAR UNIVERSAL? 49](#_Toc80384994)

[WHY ANGULAR UNIVERSAL? 49](#_Toc80384995)

[SETTING UP ANGULAR UNIVERSAL? 49](#_Toc80384996)

[SERVER AND CLIENT SIDE RENDERING AND RUNTIME INTERACTION 52](#_Toc80384997)

[ANGULAR UNIVERSAL – PRE- RENDERING 53](#_Toc80384998)

[APPLICATION SHELL 56](#_Toc80384999)

[STATE TRANSFER API 56](#_Toc80385000)

# ANGULAR

## OVERVIEW

|  |  |
| --- | --- |
|  | 1. Angular has component-based approach 2. Each component should be sufficient on its own 3. The components are registered using some tag/ selector |

## SETTING UP THE DEVELOPMENT ENVIRONMENT

To set up the development environment for Angular JS, we need to install

* **Node JS** – This provides the runtime environment for the angular application. It has bunch of utilities that help us to build and run the angular application
* Microsoft Visual Studio – Editor
* **Angular CLI** – The Angular CLI is a tool to initialize, develop, scaffold, and maintain Angular applications.

|  |  |
| --- | --- |
| **INSTALLING NODE**   1. node –v: This gives the version of the node installed 2. npm –v : This gives the version of the package manager installed. | **INSTALLING ANGULAR CLI**   1. To install the Angular CLI: **npm install -g @angular/cli** 2. To check completion and version: **ng –version** |

## CREATING ANGULAR PROJECT

|  |  |
| --- | --- |
| 1. To create a new Angular Project: **ng new <*project\_name*>.** 2. To run the Angular App : **ng serve** 3. Access the application from browser : <http://localhost:4200/>   **EXAMPLE**   1. ng new my-first-project 2. cd my-first-project 3. ng serve 4. ng serve / npm start will start a development server locally |  |

## FOLDER STRUCTURE

|  |  |
| --- | --- |
|  | 1. **src** : This folder will have our custom/source code will go. 2. **node\_module** : This will have all the dependencies installed /downloaded need to run the Angular App 3. **package.json** : This has the list of dependencies(similar to pom.xml) 4. **e2e** : Need for **end to end** testing of Angular 6 application. |

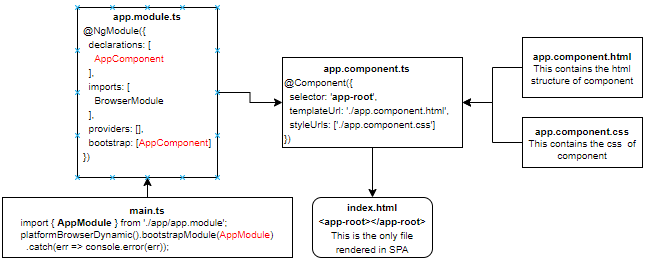
## APP MODULE

|  |  |
| --- | --- |
| **@NgModule** | * Angular modules are decorated with the @NgModule decorator. * This decorator is used to describe the module's own components, directives, pipes, services, etc. and make some of them public so that components external to this module can use them. |
| **declarations** | * Every component you build in the module must be registered here. If a component isn't placed here, Angular will not know how to load that component and will throw an exception. * Any custom pipes and directives you build must also be registered in the declarations array. * The components, directives, and pipes declared in declarations are private by default and thus, are only visible to the components in the module in which they are declared, this can be easily remedied by declaring them in the exports array. |
| **imports** | * The imports array is used to tell Angular the specific modules you want to make available within the current module. You must declare modules you want to make visible in the imports array; it is not enough to simply import a module using an import statement like this:   import {{ MyCustomModule }} from './custom/modules' |
| **exports** | The exports array is used for specifying specific components, directives, and pipes that should be made visible to other modules. Exported declarations are the module's public API. For example, if you have a module A and a module B and you want to make MyCustomComponent available to those modules, you would place the MyCustomComponent class in the exports array. |
| **providers** | The providers array is used to specify services and similar, injectable objects, that are available in the injector of this module. |
| **bootstrap** | The bootstrap array is where the components required to bootstrap the application are placed. Any components listed in the bootstrap array are also added to the entryComponents array. Typically, this only contains the root component (often defined in the root module, AppModule). |

## ROOT COMPONENT IN DETAILS

|  |  |  |
| --- | --- | --- |
|  | Any component in angular is consist of below file need for rendering:   1. Front End view(HTML markup) – This resides in the app.componet.html 2. The backend logic – This resides in app.component.ts file, which is a Typescript file/code. 3. The component can be used in an html file (index.html) using a tag. 4. Style : app.component.css (This css file is scoped to that component itself) 5. app.component.spec.ts : This file will all the test cases to unit test the component | |
| **TS FILE [app.component.ts]**  import { Component } from '@angular/core';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  title = 'first-project';  } | **USING THE COMPONENT (IN INDEX.HTML)**  <html lang="en">  <body>  <app-root></app-root>  </body>  </html> |

## HOW ANGULAR APPLICATION IS BOOTSTRAPED



The app module is the root module - which is bootstrapped by **main.ts** file

1. The app.module file has declaration of all the components which has to be known to angular before the app component is bootstrapped (bootstrap: [**AppComponent**])
2. The Module - bundles all the components.
   1. The components that must be bundled is declared in “declarations” array in @NgModule. This declaration happens automatically when we create a component using angular cli.
3. app component is the root component of complete project

## CSS IN ANGULAR PROJECTS

|  |  |
| --- | --- |
| * ***style.css*** = This css file is meant for global styling * For each component we create – there will be dedicated css also created for that component only * The CSS added in component specific css (e.g. app.component.css) is applied to that component only. |  |

### INSTALLING BOOTSTRAP CSS USING NPM

|  |  |
| --- | --- |
| INSTALL THE BOOTSTRAP CSS USING NPM: **npm install --save bootstrap@3**  **ADDING THE BOOTSTRAP MODULE TO PROJECT**   * GO TO angular.json file on the project * Add the bootstrap css path in the styles array |  |
| CSS can also be added using import statement in style.css file | EXAMPLE : To include bulma css library   1. npm install bulma 2. Go to style.css and write the import statement as : @import 'bulma/css/bulma.css'; |

### ADDING IMAGE TO PROJECT

|  |  |
| --- | --- |
| To add image to the project. The images can be kept in the asset folder of the project  REFERING IMAGES IN HTML : <img src="assets/tree.jpeg"> |  |

## CREATING THE FIRST CUSTOM COMPONENT

|  |  |
| --- | --- |
| 1. The custom component will be the child of root component. 2. The component can be used in the root component html (app.component.html) using its selector |  |
| **CLI Command t**o create a new component:   * ng generate component <component-name> * ng g c <component-name>   **CREATING A COMPONENT WITHOUT TEST FILE :** ng generate component <component-name> --spec false  **CREATING A COMPONENT IN A FOLDER :** ng generate component <folderpath>/<component-name>   * It will create 4 files for the new component * It has updated the app.module.ts * The new component has been created inside app folder. | |  |
| The module bundles all the components, so we need to declare all the components that has to be bundles into a module  So - Once the component is created – it will add the declaration in the root module   1. Import the component using import statement (Note – don’t had “.ts” extension) 2. Declare the component in the declaration section of Module 3. The complete object is then passed to NgModule decorator. | | import { **HelloComponent** } from './hello/hello.component';  @NgModule({  declarations: [ AppComponent,  HelloComponent  ],  imports: [ BrowserModule ],  providers: [],  bootstrap: [AppComponent]  }) |
| **HOW TO USE THE COMPONENT**   1. Get the selector name from hello-world.component.ts file 2. Go to root component html(app.component.html) 3. Use the selector using the selector in **app.component.html**   <app-hello-world></app-hello-world>   1. Everytime we use the tag – we are creating an instance of the corresponding TS clas | |  |

IMPORTANT NOTES: -

1. The component TS file is the entry point of any angular component. This bundles all the files of the component.
2. The TS file has been annotated with @Component to declare it as an Angular Component. Below are the details of @Component Annotation properties.

|  |  |
| --- | --- |
| **selector** | This is the name by which a component is registered with |
| **templateUrl** | This is the file where the component’s markup resides |
| **styleUrl** | It’s an array, where the CSS can be listed for a component |

### COMPONENT SELECTORS

#### ATTRIBUTE SELECTORS

|  |  |
| --- | --- |
| COMPONENT | HTML |
| @Component({  selector: '[app-server]',  templateUrl: './server.component.html',  styleUrls: ['./server.component.css']  })  export class ServerComponent implements OnInit { ..} | <div app-server>  </div> |

#### CLASS SELECTORS

|  |  |
| --- | --- |
| COMPONENT | HTML |
| @Component({  selector: '.app-server',  templateUrl: './server.component.html',  styleUrls: ['./server.component.css']  })  export class ServerComponent implements OnInit { } | <div class=”app-server”>  </div> |

## DATA BINDING

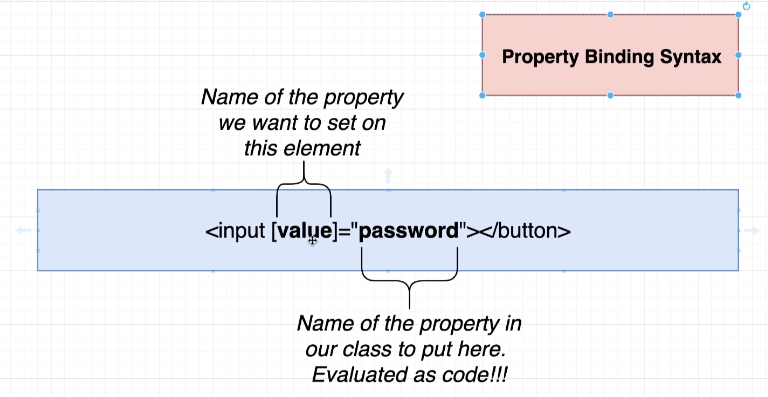
|  |  |
| --- | --- |
|  | The data binding can happen either of the ways  1. Typescript to HTML using String interpolation and Property binding  2. ***HTML to Typescript*** – Event Binding like click event  3. Two-way data binding can be achieved using ngModel. |

### STRING INTERPOLATION - ONE WAY DATA BINDING

Let’s create a new a component – “server” component to demonstrate data binding.

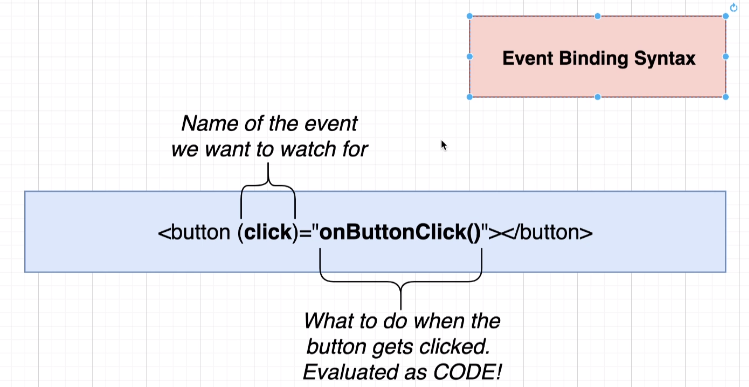
|  |  |
| --- | --- |
| COMPONENTS TS FILE – server.component.ts | COMPONENT HTML – server.component.html |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverId: number = 10;  serverStatus: string = "offline";  constructor() {}  getServerStatus() {  return this.serverStatus;  }  ngOnInit() {}  } | <p>Server {{ serverId }} is {{ getServerStatus() }}</p>.   1. The string interpolation operator can enclose any such variable which is a string or can be easily type converted to string like integer 2. It can also enclose methods |

### PROPERTY BINDING - ONE WAY DATA BINDING



|  |  |
| --- | --- |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  **allowedToAddSevers**: boolean = false;  constructor() {  setTimeout(() => {  this.allowedToAddSevers = true;  }, 2000);  }  ngOnInit() {}  } | <button class="btn btn-primary" [disabled]="!allowedToAddSevers">  Add Server  </button>  This one-way binding is with a HTML element property like **disabled** |

### EVENT HANDLING



|  |  |
| --- | --- |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverCreationStatus = "No Server Added !";  constructor() { }  serverAdded() {  this.serverCreationStatus = "Server Added";  } | ngOnInit() {}  } |
| <button class="btn btn-primary" **(click)="serverAdded()">**Add Server</button> |
| PASSING VALUE IN EVENT BINDING | |
| import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverName: string = "";  onCreateServer(event) {  this.serverName = event.target.value;  }  ngOnInit() {}  } | <input type="text" (input)="**onCreateServer($event)"** class="form-control" />   * Here we are binding input event * $event 🡪 is the reserved word |

### TWO WAY DATA BINDING [ngModel Directive]

1. ***To enable the ngModel directive. We need to add the FormsModule to the imports[] array in the AppModule.***
2. Add the import from @angular/forms in the app.module.ts file: **import { FormsModule } from '@angular/forms';**

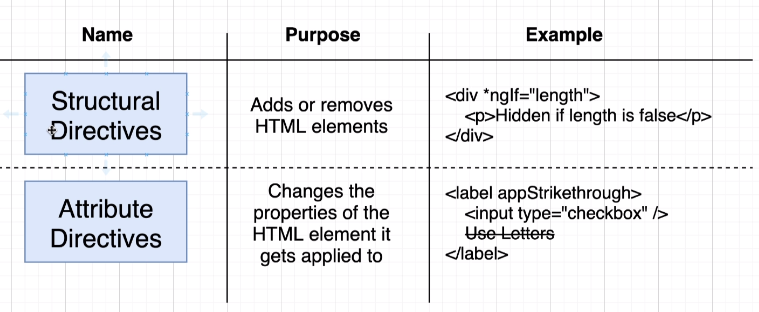
|  |  |  |
| --- | --- | --- |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  import { FormsModule } from "@angular/forms";  import { AppComponent } from "./app.component";  import { ServerComponent } from "./server/server.component";  import { ServersComponent } from "./servers/servers.component";  @NgModule({  declarations: [AppComponent, ServerComponent, ServersComponent],  imports: [BrowserModule, FormsModule],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverName: string = "";  constructor() {}  ngOnInit() {}  } | <input type="text" [(ngModel)]="serverName" /> |

## ANGULAR DIRECTIVES

**THE DIRECTIVES ARE USED TO MODIFY THE STRUCTURE OR PROPERTIES OF HTML ELEMENT**

* All structural directives start with \* e.g. \*ngIf,\*ngFor etc. otherwise called as attribute directive

|  |  |
| --- | --- |
| **ATTRIBUTE DIRECTIVES** | **STRUCTURAL DIRECTIVES** |
| It looks like normal HTML attribute | Looks like normal HTML attribute but has leading \* |
| Only affect /change the element they are added to like styling | Affect the whole area in the DOM(elements are added or removed) |



|  |  |
| --- | --- |
| Note – Two structural directives cannot be placed in the same element together | <div \*ngFor="let num in numbers" \*ngIf="num % 2 ==0">  </div |

### PRE-DEFINED DIRECTIVES

#### STURUCTURAL DIRECTIVES

|  |  |  |
| --- | --- | --- |
| **\*ngIf directive** | | |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  serverCreationStatus = "No Server Added !";  isServerAdded: boolean = false;  serverAdded() {  this.serverCreationStatus = "Server Added";  this.isServerAdded = true;  }  ngOnInit() {}  } | <button  class="btn btn-primary"  [disabled]="!allowedToAddSevers"  (click)="serverAdded()"  >  Add Server  </button>  <p \*ngIf="isServerAdded">{{ serverCreationStatus }}</p> | |
| If else directive : If we want to add else condition | <p \*ngIf="isServerAdded; **else noServer**">{{ serverCreationStatus }}</p>  <ng-template #noServer>  <p>No Server Added</p>  </ng-template> | |
| **\*ngFor** | | |
| @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"]  })  export class ServerComponent implements OnInit {  servers = [];  server = "";  constructor() {}  ngOnInit() {}  onCreateServer() {  console.log(this.server);  this.servers.push(this.server);  }  } | | <input type="text" class="form-control" [(ngModel)]="server" />  <button class="btn btn-primary" (click)="onCreateServer()">  Add servers  </button>  <ul>  <li \*ngFor="let server of servers; let serverNumber = index">  {{ serverNumber }} - {{ server }}  </li>  </ul> |

### ATTRIBUTE DIRECTIVES

#### ngClass and ngStyle

|  |  |
| --- | --- |
| APP.TS |  |
| export class AppComponent {  listOfNumbers: number[] = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]  }  APP.HTML  <div class="container">  <div class="row">  <div class="list-group">  <a href="#" class="list-group-item"  \*ngFor="let num of listOfNumbers"  [ngClass]="num%2==0?'active':''"  [ngStyle]="{color:num%2!=0 ?'red':''}">{{num}}</a>  </div>  </div>  </div> |
| * ngStyle expect a JavaScript object |

#### ng-container

#### NgSwitch

|  |  |
| --- | --- |
| TS FILE  import { Component, OnInit, Input } from "@angular/core";  @Component({  selector: "app-comment",  templateUrl: "./comment.component.html",  styleUrls: ["./comment.component.css"]  })  export class CommentComponent implements OnInit {  value = 5;  constructor() {}  ngOnInit() {}  } | HTML  <div [ngSwitch]="value">  <p \*ngSwitchCase="1">Value is 1</p>  <p \*ngSwitchCase="5">Value is 5</p>  <p \*ngSwitchDefault>default</p>  </div> |

### CUSTOM DIRECTIVES

#### CUSTOM ATTRIBUTE DIRECTIVES

|  |  |  |
| --- | --- | --- |
| CREATING A DIRECTIVE | | 1. **ng g d simpledirective** 2. ng generate directive simpledirective |
| APP MODULE | | GENERATED DIRECTIVE TS |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  import { FormsModule } from "@angular/forms";  import { AppComponent } from "./app.component";  import { SimpledirectiveDirective } from './simpledirective.directive';  @NgModule({  declarations: [AppComponent, SimpledirectiveDirective],  imports: [BrowserModule, FormsModule],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | | import { Directive } from "@angular/core";  @Directive({  selector: "[appSimpledirective]"  })  export class SimpledirectiveDirective {  constructor() {}  } |
| 1. The newly created directive must be declared at the App Module level |
| **CREATING AND USING AN ATTRIBUTE DIRECTIVE**   * In this example we will create a directive which will change the background color of the element on hovering it. * The directive always manipulates the HTML element, so to create a custom directive we need to get hold of the element itself to manipulate | | |
| TS FILE  import { Directive, ElementRef, OnInit, Renderer2 } from "@angular/core";  @Directive({  selector: "[appSimpledirective]"  })  export class SimpledirectiveDirective implements OnInit {  constructor(private elementRef: ElementRef, private renderer: Renderer2) {}  ngOnInit(): void {  this.renderer.setStyle( this.elementRef.nativeElement,"background-color", "red" );  }  } | HTML  <p **appSimpledirective**>This is from Directive</p> | |
| * ElementRef 🡪 This helps in getting hold of the element on which the directive is applied.   This can be also achieved using  this.elementRef.nativeElement.style.backgroundColor = "green";   * The renderer object have multiple methods for DOM manipulation (<https://angular.io/api/core/Renderer2>) | |

EXAMPLE 2

|  |  |
| --- | --- |
| DIRECTIVE (CUSTOM) | TEMPLATE CODE |
| import { Directive, HostListener, HostBinding } from '@angular/core';  @Directive({  selector: '[appDropdown]'  })  export class DropdownDirective {  constructor() { }  @HostBinding("class.open") isOpen: boolean = false;  @HostListener('click') toggleDropDown() {  this.isOpen = !this.isOpen;  }  } | <ul class="nav navbar-nav navbar-right">  <li class="dropdown" **appDropdown**>  <a class="dropdown-toggle" data-toggle="dropdown">Manage  <span class="caret"></span></a>  <ul class="dropdown-menu">  <li><a>Save Data</a></li>  <li><a>Fetch Data</a></li>  </ul>  </li>  </ul> |
| * In the above example – We are using a bootstrap drop down. To open the drop down we need to add “open” class in <li> * Step 1: We need to bind a variable with the class (open) – using Hostbinding * Step 2: Bind a function which will manipulate the variable on a binded event – using HostBinding | |

##### HOSTLISTENER AND HOSTBINDING DECORATOR – EXAMPLE 2

CUSTOM PROPERTY BINDING IN DIRECTIVES

|  |
| --- |
| <p appSimpledirective [mousehovercolor]="'yellow'" [mouseleavecolor]="'green'"> This is from Directive</p>  Here we are handing the color (unlike hardcoded in previous example) from the teamplate code itself. |
| import { Directive, ElementRef, OnInit, Renderer2, HostListener, Input } from "@angular/core";  @Directive({  selector: "[appSimpledirective]"  })  export class SimpledirectiveDirective implements OnInit {  constructor(private elementRef: ElementRef, private renderer: Renderer2) {}  @Input() mousehovercolor = "";  @Input() mouseleavecolor = "";  ngOnInit(): void {  this.renderer.setStyle( this.elementRef.nativeElement,"background-color","red" );  }  @HostListener("mouseover") mouseover() {  this.renderer.setStyle( this.elementRef.nativeElement,"background-color", **this.mousehovercolor** );  }  @HostListener("mouseleave") mouseleave() {  this.renderer.setStyle(this.elementRef.nativeElement,"background-color", **this.mouseleavecolor** );  }  } |

#### CUSTOM STRUCTURAL DIRECTIVES

The structural directive get transforms into ng-template as shown below

|  |
| --- |
| <div \*ngIf="condition"> This is ngIf</div> **🡪** <ng-template [ngIf]="condition"> This is ngIf</ng-template> |
| In general - Ng-template is basically display / hide the element based on the condition applied to it |

Let create a structural directive that behaves opposite of “ngIf”

|  |  |
| --- | --- |
| TS FILE(DIRECTIVE) | HTML |
| import { Directive, TemplateRef, ViewContainerRef, Input } from "@angular/core";  @Directive({  selector: "[appOppositeif]"  })  export class OppositeifDirective {  @Input() set appOppositeif(condition: boolean) {  if (!condition) this.vcRef.createEmbeddedView(this.tempRef);  else this.vcRef.clear();  }  constructor(  private tempRef: TemplateRef<any>,  private vcRef: ViewContainerRef  ) {}  } | <div **\*appOppositeif="false"**> This is appOppositeif </div> |
| 1. To manipulate the element, we mark setter of the structural directive 2. We inject TemplateRef because, ultimately all the structural directives are transformed into <ng-template> |

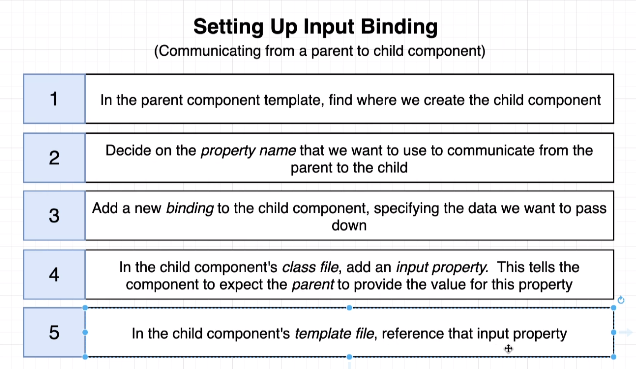
## LOCAL REFEREENCE IN TEMPLATE

* The local reference can be applied to any HTML element
* Once it is applied to the HTML element it will hold the reference o f that HTML element
* The scope of the local variable is limited to the template (HTML) where it applied. It can be passed to TS file from the template only .The TS file property cannot access to local reference unlike the two way binding.
* Once we get hold of the HTML element – so we can get or set the values of that the HTML element using local reference. It can be an alternative of two-way data binding – as shown below

|  |  |
| --- | --- |
| STUDENT REGISTRATION HTML | <div class="row">  <div class="form-group">  <label for="firstName">First Name</label>  <input type="text" class="form-control" id="firstName" #firstName>  </div>  <div class="form-group">  <label for="lastName">Last Name</label>  <input type="text" class="form-control" id="lastName" #lastName>  </div>  <button class="btn btn-primary" (click)="regStudent(firstName,lastName)">Register</button>  </div> |
| STUDENT REGISTRATION TS | import { Component, OnInit, EventEmitter, Output } from '@angular/core';  @Component({  selector: 'app-student-registration',  templateUrl: './student-registration.component.html',  styleUrls: ['./student-registration.component.css']  })  export class StudentRegistrationComponent implements OnInit {  @Output() registerStudent = new EventEmitter<{ firstName: string, lastName: string }>();  constructor() { }  ngOnInit() { }  regStudent(firstName: HTMLInputElement, lastName: HTMLInputElement) {  this.registerStudent.emit({  firstName: firstName.value,  lastName: lastName.value  });  firstName.value = "";  lastName.value = "";  } } |

## COMMUNICATION BETWEEN COMPONENTS

### PARENT TO CHILD COMMUNICATION - PROPERTY BINDING AND @Input DECORATOR



#### EXAMPLE 1

|  |  |
| --- | --- |
| **COMPONENT’S TS FILE** | **HTML FILE WHERE THE COMPONENT IS USED** |
| import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'app-date',  templateUrl: './date.component.html',  styleUrls: ['./date.component.css']  })  export class DateComponent implements OnInit {  user :any;  @Input("name") userName:string;  constructor() { }  ngOnInit() {  this.user={  'userName':this.userName,  'firstName':'John',  'lastName':'Doe',  'address' :'New Jersey',  'phones':[  '9650758731',  '9650758732'  ]  };  }  } | <app-date name="johnDoe"></app-date>   1. The data “name” has been passed from the view (html) to the component. 2. **@Input annotation** is used to pass the data and populate the member variable. 3. **Input** class must be imported from angular/core module. 4. The important point here is that we are communicating between components using property binding |

#### EXAMPLE 2

|  |  |  |  |
| --- | --- | --- | --- |
|  |  | | We will have 2 components   1. Comments Component(Parent)   This component will have a form to add comments   1. Comment Component (Child)   This will receive the value from the form and display the commets |
| COMMENTS COMPONENT | | | |
| COMMENTS TS  import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-comments",  templateUrl: "./comments.component.html",  styleUrls: ["./comments.component.css"]  })  export class CommentsComponent implements OnInit {  comments: string[] = [];  comment: string = "";  constructor() {}  ngOnInit() {}  addComment() {  this.comments.push(this.comment);  console.log(this.comments);  this.comment = "";  }  } | | COMMENTS HTML  <div class="container">  <div class="row">  <input type="text" class="form-element" [(ngModel)]="comment" />  <button class="btn btn-primary" (click)="addComment()">Add Comment</button>  </div>  <div class="row" \*ngFor="let comment of comments">  <app-comment [comment]="comment"></app-comment>  </div>  </div> | |
| COMMENT COMPONENT | | | |
| COMMENT TS  import { Component, OnInit, Input } from "@angular/core";  @Component({  selector: "app-comment",  templateUrl: "./comment.component.html",  styleUrls: ["./comment.component.css"]  })  export class CommentComponent implements OnInit {  @Input("comment") comment = "";  constructor() {}  ngOnInit() {}  } | | COMMENT HTML  <p>{{ comment }}</p>  Note :  Here the communication is happening from Parent to Child component. | |

#### EXAMPLE 3

|  |  |  |  |
| --- | --- | --- | --- |
| OUTPUT | FILE STRUCTURE | | COMPONENT STRUCTURE |
|  |  | |  |
| Blog.ts  export class Blog {  constructor(private blogTitle, private blogContent) { }  } | | | app.component.html |
| <div class="container">  <app-bloglist></app-bloglist></div> |
| Bloglist component | | | |
| Bloglist.component.ts | | Bloglist.component.html | |
| import { Component, OnInit } from '@angular/core';  import { Blog } from '../Blog';  @Component({  selector: 'app-bloglist',  templateUrl: './bloglist.component.html',  styleUrls: ['./bloglist.component.css']  })  export class BloglistComponent implements OnInit {  blogList: Blog[] = [];  constructor() { }  ngOnInit() {  this.blogList.push(new Blog("Blog Title 1", "Blog Content 1"));  this.blogList.push(new Blog("Blog Title 2", "Blog Content 2"));  this.blogList.push(new Blog("Blog Title 3", "Blog Content 3"));  this.blogList.push(new Blog("Blog Title 4", "Blog Content 4"));  this.blogList.push(new Blog("Blog Title 5", "Blog Content 5"));  this.blogList.push(new Blog("Blog Title 6", "Blog Content 6"));  }  } | | <app-blog [blogList]=blogList></app-blog>   1. We are passing an object ; List of blog objects | |
| Blog component | | | |
| Blog.component.ts | | Blog.component.html | |
| import { Component, OnInit, Input } from '@angular/core';  import { Blog } from '../Blog';  @Component({  selector: 'app-blog',  templateUrl: './blog.component.html',  styleUrls: ['./blog.component.css']  })  export class BlogComponent implements OnInit {  @Input() blogList: Blog[] = [];  constructor() { }  ngOnInit() {  }  } | | <div class="row" \*ngFor="let blog of blogList">  <div class="panel panel-default">  <div class="panel-body">  <div class="row">  {{blog.blogTitle}}  </div>  <div class="row">  {{blog.blogContent}}  </div>  </div>  </div> | |

## CHILD TO PARENT COMMUNICATION USING VIEWCHILD AND VIEWCHILDEN

* The @ViewChild and @ViewChildren decorator help in the communication between child component to parent component.
* This can used in conjunction with “local Reference” OR “Component Name” itself

### CHILD TO PARENT COMMUNICATION USING EVENTS - @Output

|  |  |
| --- | --- |
| * App component has 2 child components. * The student-registration component has a form to Add Student * Student-list component list all the added students * The app component has student array – where new student object is pushed when added and hence student-list component list all students |  |

HOW COMPONENTS COMMUNICATE?

1. The student-registration component send the newly added student by emitting a custom event (registerStudent)
2. The App component has captures the event and add it to student array
3. The student list component then shows all the added srudent using data binding
4. However – the communication among the components can also be manage by angular services as well.

|  |  |
| --- | --- |
| APP COMPONENT HTML | <div class="container"> <div class="row">  <app-student-registration (registerStudent)=addStudent($event)></app-student-registration>  </div>  <hr/>  <div class="row"><ul class="list-group">  <app-student-list \*ngFor="let student of students" [student]=student></app-student-list>  </ul> </div></div> |
| APP COMPONENT TS | export class AppComponent {  title = 'datacomm';  students = []; 🡸Maintains a student array shared by both registration and listing components  **addStudent**(studentData: { firstName: string, lastName: string }) {  this.students.push({  firstName: studentData.firstName,  lastName: studentData.lastName  });  }  } |
| STUDENT REGISTRATION HTML | <div class="row">  <div class="form-group">  <label for="firstName">First Name</label>  <input type="text" class="form-control" id="firstName" [(ngModel)]="firstName">  </div>  <div class="form-group">  <label for="lastName">Last Name</label>  <input type="text" class="form-control" id="lastName" [(ngModel)]="lastName">  </div>  <button class="btn btn-primary" **(click)="regStudent()">**Register</button>  </div> |
| STUDENT REGISTRATION TS | import { Component, OnInit, EventEmitter, Output } from '@angular/core';  @Component({  selector: 'app-student-registration',  templateUrl: './student-registration.component.html',  styleUrls: ['./student-registration.component.css']  })  export class StudentRegistrationComponent implements OnInit {  @Output() registerStudent = new EventEmitter<{ firstName: string, lastName: string }>();  firstName: string;  lastName: string;  constructor() { }  ngOnInit() { }  regStudent() {  this.registerStudent.emit({  firstName: this.firstName,  lastName: this.lastName  });  this.firstName = "";  this.lastName = "";  }  } |
| STUDENT LIST HTML | <li class="list-group-item">{{student.firstName}} , {{student.lastName}}</li> |
| STUDENT LIST TS | import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'app-student-list',  templateUrl: './student-list.component.html',  styleUrls: ['./student-list.component.css']  })  export class StudentListComponent implements OnInit {  constructor() { }  **@Input() student:** { firstName: string, lastName: string };  ngOnInit() { }  } |

### @VIEWCHILD

#### ACCESSING TEMPLATE HTML USING LOCAL REFERENCE

|  |  |  |
| --- | --- | --- |
|  | * @ViewChild decorator is helps in getting access to the component’s template HTML in the component’s TS File. * Lets say , we have a requirement to focus on the first name field on page load * The ViewChild decorator has an ability to get hold of component’s HTML * ViewChild uses “local reference” to get hold of the HTML element . * Note - The HTML reference will be ready and can be accessed in the ngAfterViewInit() life cycle hook | |
| STUDENT REGISTRATION COMPONENT TS | | STUDENT REGISTRATION COMPONENT HTML |
| import { Component, OnInit, EventEmitter, Output, ViewChild, ElementRef, AfterViewInit } from '@angular/core';  @Component({  selector: 'app-student-registration',  templateUrl: './student-registration.component.html',  styleUrls: ['./student-registration.component.css']  })  export class StudentRegistrationComponent implements OnInit, AfterViewInit {  @Output() registerStudent = new EventEmitter<{ firstName: string, lastName: string }>();  @ViewChild("firstName", { static: true }) firstName: ElementRef;  constructor() { }  ngOnInit() { }  ngAfterViewInit(): void {  this.firstName.nativeElement.focus(); **🡨 Focus on the field**  }  regStudent(firstName: HTMLInputElement, lastName: HTMLInputElement) {  this.registerStudent.emit({  firstName: firstName.value,  lastName: lastName.value  });  firstName.value = "";  lastName.value = "";  }  } | | <div class="row">  <div class="form-group">  <label for="firstName">First Name</label>  <input type="text" class="form-control" id="firstName" #firstName>  </div>  <div class="form-group">  <label for="lastName">Last Name</label>  <input type="text" class="form-control" id="lastName" #lastName>  </div>  <button class="btn btn-primary" (click)="regStudent(firstName,lastName)">Register</button>  </div> |

In the above example we are using local reference to access the HTML element of component’s template using @ViewChild decorator. This decorator can also be used to access the child element in the template as well

#### ACCESSING CHILD ELEMENT USING VIEW CHILD

|  |  |  |
| --- | --- | --- |
| CHILD TS | PARENT HTML | |
| export class DateComponent implements OnInit {  constructor() { }  today: Date = new Date();  ngOnInit() {  }  } | <app-date></app-date>  <p>User Name : {{uname}}</p>   * <app-date> is of type DateComponent which can be accessed using @ViewChild decorator. * This cannot be accesses before **ngAfterViewInit()** life cycle hook of the component | |
| PARENT TS | | * If we have more than on <app-date> child component – The @ViewChild can able to access only the first element. * To sccess all the <app-date> child element we need to use another decorator **@ViewChildren** |
| export class ChildcompComponent implements OnInit, **AfterViewInit** {  @Input() uname: string = "";  @ViewChild(DateComponent, { static: true }) dateComponent: DateComponent;  ngOnInit() { }  ngAfterViewInit(): void {  setInterval(() => {  this.dateComponent.today = new Date();  }, 1000);  }} | |

### @VIEWCHILDREN

|  |  |  |
| --- | --- | --- |
| @ViewChildren | This decorator will allow the parent to select the all instances of the child component | |
|  | In this application     1. App component has two child component student-registration and student-listing 2. The app component has 2 buttons to select and deselect the students in the list 3. The background (color changes to green on click of “select All” and “Deselect all ” will remove the background color. 4. @ViewChildren decorator helps in manipulating the child component (student-listing) from App Component. 5. App component is using local reference to access the child element | |
| APP HTML | | |
| <div class="container">  <div class="row">  <app-student-registration (registerStudent)=addStudent($event)></app-student-registration>  </div>  <hr/>  <div class="row">  <div class="col-md-1">  <button class="btn btn-success btn-xs" \*ngIf="students.length >0" (click)="selectAllStudent()">Select All</button>  </div>  <div class="col-md-1">  <button class="btn btn-info btn-xs" \*ngIf="students.length >0" (click)="deSelectAllStudent()">DeSelect All</button>  </div>  </div>  <hr/>  <div class="row">  <ul class="list-group">  <app-student-list \*ngFor="let student of students" [student]=student #studentList></app-student-list> 🡨 **This local reference will be used in @ViewChildren() decorator**  </ul>  </div>  </div> | | |
| APP TS  import { Component, ViewChildren, QueryList } from '@angular/core';  import { StudentListComponent } from './student-list/student-list.component';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent {  @ViewChildren('studentList') studentList: QueryList<StudentListComponent> 🡨 This will give the list of all StudentListComponent(Child Component)  title = 'datacomm';  students = [];  addStudent(studentData: { firstName: string, lastName: string }) {  this.students.push({  firstName: studentData.firstName,  lastName: studentData.lastName  });  }  selectAllStudent() {  this.studentList.forEach(element => { 🡨 **Iterating the list**  element.selectStudent(); 🡨 Calling the function in child component(StudentListComponent)  });  }  deSelectAllStudent() {  this.studentList.forEach(element => { 🡨 **Iterating the list**  element.deSelectStudent(); 🡨 Calling the function in child component(StudentListComponent)  });  }  } | | |
| STUDENT LIST TS | | STUDENT LIST HTML |
| import { Component, OnInit, Input } from '@angular/core';  @Component({  selector: 'app-student-list',  templateUrl: './student-list.component.html',  styleUrls: ['./student-list.component.css']  })  export class StudentListComponent implements OnInit {  selectionClass: boolean = false;  constructor() { }  @Input() student: { firstName: string, lastName: string };  ngOnInit() {  }  selectStudent() {  this.selectionClass = true;  }  deSelectStudent() {  this.selectionClass = false;  }  } | | <li class="list-group-item"  [ngClass]="selectionClass ? 'list-group-item-success':''">  {{student.firstName}} , {{student.lastName}} </li> |

#### ACCESSING ALL CHILD ELEMENT USING @VIEWCHILD

|  |  |  |
| --- | --- | --- |
| CHILD TS | PARENT HTML | 1. @ViewChildren decorators has a list of all child instance embedded into the HTMl template 2. As its QueryList – so we need to loop though the elements to access the child element. |
| export class DateComponent implements OnInit {  constructor() { }  today: Date = new Date();  ngOnInit() {  } } | <app-date></app-date>  <app-date></app-date>  <p>User Name : {{uname}}</p> |
| PARENT TS  export class ChildcompComponent implements OnInit, AfterViewInit {  @Input() uname: string = "";  @ViewChildren(DateComponent) dateComponents: QueryList<DateComponent>;  ngOnInit() {  }  ngAfterViewInit(): void {  this.dateComponents.forEach(element => {  setInterval(() => {  element.today = new Date();  }, 1000);  });  }  } | |

## VIEW ENCAPSULATION

|  |  |
| --- | --- |
| COMPONENT | * As a defult behavior the style in the CSS of a component is applied only that component only. * But if we want to override theis default behavior and apply the css globally. |
| import { Component, OnInit, ViewEncapsulation } from "@angular/core";  @Component({  selector: "app-blogs",  templateUrl: "./blogs.component.html",  styleUrls: ["./blogs.component.css"],  encapsulation: ViewEncapsulation.None  })  export class BlogsComponent implements OnInit {  constructor() {}  ngOnInit() {}  }  CSS  p { color: red; } |

|  |  |
| --- | --- |
| ViewEncapsulation.None | The CSS applied to the component will be applied globally |
| ViewEncapsulation.Emulated | Default Behavior |
| ViewEncapsulation.Native | Emulated behavior will be applied in those browsers which supports it otherwise applied globally |
| ViewEncapsulation.ShadowDom |  |

## CONTENT PROJECTION

### CONTENT PROJECTION – ngContent

|  |  |  |
| --- | --- | --- |
| Headline Component | When we write an HTML between the component’s selector. Angualar ignores it and pull the HTML in its own template (HTML)  When we want angular the HTML between the component’s selector we can use **<ng-content>** | |
| <app-headline>  <h1>Student Registration</h1>  </app-headline> |
| HEADLINE COMPONENT HTML | APP COMPONENT HTML | This kind of HTML can be used to create container components. For example, headline component can have any HTML element into it. |
| <div class="row">  <ng-content></ng-content>  </div> | <app-headline>  <h1> Student Registration</h1>  </app-headline> |

|  |  |
| --- | --- |
|  | * The content projection can be leveraged to create templates * For example, lets say we need to create a template which has fixed header and footer but different body content * We can use <ng-content> to create such templates. It acts a placeholder for content |

### MULTI SLOT CONTENT PROJECTION

1. The content can be selected too to be projected in a specifc location. The selection can be done ***class , attribute selector or id selector***

#### CONTENT PROJECTION – CLASS SELECTOR

|  |  |
| --- | --- |
| APP HTML | CARD COMPONENT HTML |
| <div class="container">  <app-template1>  <div class="row">  <h1>Page Heading</h1>  <hr/>  </div>  <h2 class="articleHeading">Article heading</h2>  <p class="articleContent">Article Content</p>  <div class="row">  <hr/>  <p>Page footer  </p>  </div>  </app-template1>  </div> | <div class="panel panel-default">  <div class="panel-heading">  <ng-content select=".articleHeading"></ng-content>  </div>  <div class="panel-body">  <ng-content select=".articleContent"></ng-content>  </div>  </div> |

#### CONTENT PROJECTION – ATTRIBUTE SELECTOR

|  |  |
| --- | --- |
| TEMPLATE HTML | The ng content can also select a specific HTML content based on the selector |
| <div class="container">  <div class="row">  <ng-content select="[header]"> </ng-content>  </div>  <div class="row">  <div class="col-sm-3">  <ng-content select="[leftNav]"> </ng-content>  </div>  <div class="col-sm-1"> </div>  <div class="col-sm-8">  <ng-content select="[body]"> </ng-content>  </div>  </div>  <div class="row">  <hr/>  <ng-content select="[footer]"> </ng-content>  </div>  </div> |
| APP HTML | |
| <app-template>  <app-headline header>  <h1>Student Registration</h1>  <hr>  </app-headline>  <div class="row" leftNav>  <app-student-registration (registerStudent)=addStudent($event)></app-student-registration>  </div>  <div class="row" body>  <div class="row">  <div class="col-sm-1">  <button class="btn btn-success btn-xs" \*ngIf="students.length >0" (click)="selectAllStudent()">Select All</button>  </div>  <div class="col-sm-1">  <button class="btn btn-info btn-xs" \*ngIf="students.length >0" (click)="deSelectAllStudent()">DeSelect All</button>  </div>  </div>  <hr/>  <div class="row">  <ul class="list-group">  <app-student-list \*ngFor="let student of students" [student]=student #studentList></app-student-list>  </ul>  </div>  </div>  <div class="row" footer> Footer </div>  </app-template> | |

### CONTENT PROJECTION - @ContentChild

* The **@ContentChild** decorator can get hold of child of the projected content.

|  |  |  |
| --- | --- | --- |
| APP COMPONENT | CARD COMPONENT | |
| <div class="container">  <app-card>  <div class="row">  <h1>Page Heading</h1>  <hr/>  </div>  <h2 class="articleHeading">Article heading</h2>  <p class="articleContent">  <app-date></app-date>  <span>Article Content</span>  <div class="row">  <hr/>  <p>Page footer  </p>  </div>  </app- card >  </div> | <div class="panel panel-default">  <div class="panel-heading">  <ng-content select=".articleHeading"></ng-content>  </div>  <div class="panel-body">  <ng-content select=".articleContent"></ng-content>  </div>  </div> | |
|  | * <app-date> component is the child component in the projected component |
| CARD COMPONENT TS | |
| export class Template1Component implements OnInit, AfterContentInit {  @ContentChild(DateComponent, { static: true }) dateComponent: DateComponent;  constructor() { }  ngOnInit() { }  ngAfterContentInit(): void {  setInterval(() => {  this.dateComponent.today = new Date();  }, 1000)  } } | |

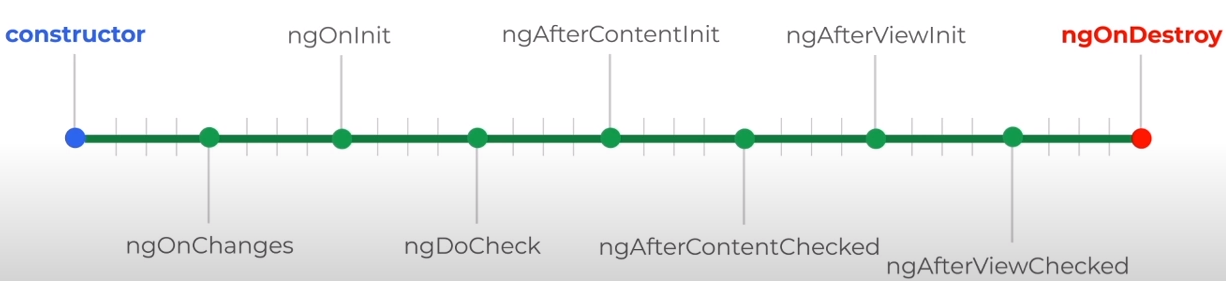
### CONTENT PROJECTION - @ContentChildren

* If we have multiple child component of projected content we can hold of them using @ContentChildren decorator

|  |  |
| --- | --- |
| APP COMPONENT | CARD COMPONENT |
| <div class="container">  <app-card>  <div class="row">  <h1>Page Heading</h1>  <hr/>  </div>  <h2 class="articleHeading">Article heading</h2>  <p class="articleContent">  <app-date></app-date>  <app-date></app-date>  <span>Article Content</span>  <div class="row">  <hr/>  <p>Page footer  </p>  </div>  </app- card >  </div> | <div class="panel panel-default">  <div class="panel-heading">  <ng-content select=".articleHeading"></ng-content>  </div>  <div class="panel-body">  <ng-content select=".articleContent"></ng-content>  </div>  </div> |
| export class Template1Component implements OnInit, AfterContentInit {  @ContentChildren(DateComponent) dateComponents: QueryList<DateComponent>;  constructor() { }  ngOnInit() { }  ngAfterContentInit(): void {  this.dateComponents.forEach(element => {  setInterval(() => {  element.today = new Date();  }, 1000)  });  }} |
|  | |

## ANGULAR LIFECYCLE HOOKS

When angular encounters a selector – it instatitate the component. In this creation process it go through various phases. Angular gives and opportunity to hook methods in these phases.



|  |  |
| --- | --- |
| ngOnChange() | This is the first life cycle hook. This gets called multiple times as well when ever any property bounded by @Input is changed |
| ngOnInit() | Called when the component is intitailzed. In this hook the properties are ready to be accessible. |
| ngDoCheck() | Called multiple times whenever it detects a change in the component’s template. e. g property value.This is called when a event is occurred , like click event |
| ngAfterContentInit() | Called when the projected content (ng-cotent) is initialized |
| ngAfterContentChecked() | Called when a change is detected in the projected content (ng-content) |
| ngAfterViewInit() | Called when component is fully intialialied and view is rendered |
| ngAfterViewChecked() | Called when component is fully intialialied and all its childs view is initialized ` |
| ngOnDestroy() | When we move from one page to another(Route change). The component of the previous pages is destroyed. This life cycle hook is called right before the component is destroyed. It can be used for some clean up activity |

## CREATING A SERVICE

|  |  |  |
| --- | --- | --- |
| CREATING A SERVICE(CLI COMMAND)  [SEVICES CREATED IN APP MODULE] | ng generate service <service\_name>  E.g. 🡪 ng generate service test OR ng g s <***service\_name***> | |
| * This will generate only 2 files because service are something, which contains the backend logic and has no view – so no corresponding **html or module** files are generated of it * The Services are annotated with **@Injectable** * Since the services contains the business logic which will be used other components in the module. For that we need to declare the service in component’s provider’s section * If the service is a global service – it can be added as providers app module. * The Service is a centralized location – where our business logic will reside – and can be shared among multiple components. | | |
| GENERATED FILES    SERVICE  import { Injectable } from '@angular/core';  @Injectable({  providedIn: 'root'  })  export class LoggingService {  constructor() { }  logToConsole(message:string){  console.log(message);  }  } | | AT COMPONENT LEVEL  import { Component, OnInit } from "@angular/core";  import { LoggingService } from "../logging.service";  @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"],  providers: [LoggingService]  })  export class ServerComponent implements OnInit {  constructor(private loggingService: LoggingService) {}  } |
| SERVICE INJECTED IN APP COMPONENT IN APP.MODULE.TS  @NgModule({  declarations: [AppComponent, HelloWorldComponent],  imports: [BrowserModule, ViewModule],  providers: [TestService],  bootstrap: [AppComponent]  }) |
| **Note – Here the “**LoggingService**”** has been declared at component level and “TestService” is declared at @NgModule level . Refer *HIERARCHICAL DEPENDENCY INJECTION* for more details on it. | | |

### USING SERVICES

* The services can be used in the component using **DEPENDECY INJECTION**

|  |  |
| --- | --- |
| SERVICE  @Injectable({  providedIn: "root"  })  export class TestService {  constructor() {}  printToConsole(args: string) {  return args;  }  } | Let’s say we have a method in the service “printToConsole” , which we want to use /call in one of the component.  Angular will inject the service instance into the constructor argument as shown below |
| export class AppComponent {  title = "training";  constructor(private svc: TestService) {  svc.printToConsole("Hello !!");  }  } |
| Note – This is the new syntax for Angular 6+ version. This is same as declaring at Globally at @NgModule level . The "new syntax" does offer one advantage though: Services can be loaded lazily by Angular (behind the scenes). This can lead to a better performance and loading speed. | |

### HIERARCHICAL DEPENDENCY INJECTION

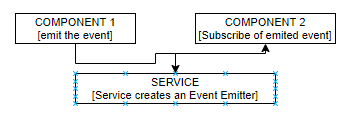
|  |  |
| --- | --- |
|  | 1. The Service can be declared as providers at each component level or at the topmost level NgModule level 2. When the service is declared at component level then component and its child component gets – the same instance of service 3. When the Service is declared as NgModule(topmost level)- Same instance of service is being used by the complete application. 4. In the diagram- App module and all its child components gets the an instance of “GlobalService” service. whereas ParentComponent and ChildComponent gets the same instance of **ParentService** service. |

### INJECTING SERVICE INTO ANOTHER SERVICE

* LoggingService has been injected in ServerService – which is being used by app server component
* We can inject service-1 to a service-2 only if service-2 is marked as Injectable

|  |  |  |
| --- | --- | --- |
| SERVICE 1 [LOGGING SERVICE] | SERVICE 2 [SERVER SERVICE] | USING SERVICE |
| import { Injectable } from "@angular/core";  @Injectable({  providedIn: "root"  })  export class LoggingService {  constructor() {}  logToConsole(message: string) {  console.log(message);  }  } | import { Injectable } from "@angular/core";  import { LoggingService } from "./logging.service";  @Injectable({  providedIn: "root"  })  export class ServerService {  constructor(private loggingService: LoggingService) {}  serverLogging(message: string) {  this.loggingService.logToConsole(message);  }  } | import { Component, OnInit } from "@angular/core";  import { LoggingService } from "../logging.service";  import { ServerService } from "../server.service";  @Component({  selector: "app-server",  templateUrl: "./server.component.html",  styleUrls: ["./server.component.css"],  providers: [ServerService]  })  export class ServerComponent implements OnInit {  constructor(private serverService: ServerService) {}  ngOnInit() {}  onCreateServer() {  this.serverService.serverLogging("Server Added=" + this.server);  }  } |

### CROSS COMPONENT COMMUNICATION USING SERVICES



* Service just creates the instance of event emitter
* This instance is levarged to emit and subscribe for the event

|  |  |  |
| --- | --- | --- |
| SERVICE | COMPONENT 1 (Emits the data) | |
| import { Injectable, EventEmitter } from '@angular/core';  import { Recipe } from './recipe.model';  @Injectable({  providedIn: 'root'  })  export class RecipeService {  private recipes: Recipe[] = [  new Recipe('Test Recipe', 'Test Description', '),  new Recipe('Test Recipe1', 'Test Description2',’’), ];  selectedRecipe = new EventEmitter<Recipe>();  constructor() { }  getRecepies() {  return this.recipes.slice();  }  } | import { Component, OnInit, Output, EventEmitter } from '@angular/core';  import { Recipe } from '../recipe.model';  import { RecipeService } from '../recipe.service';  @Component({  selector: 'app-recipe-list',  templateUrl: './recipe-list.component.html',  styleUrls: ['./recipe-list.component.css']  })  export class RecipeListComponent implements OnInit {  recipes: Recipe[] = [];  constructor(private recipeService: RecipeService) { }    showRecipeDetails(recipe: Recipe) {  this.recipeService.selectedRecipe.emit(recipe);  }  } | |
| COMPONENT 2 (Subscribe for the data) | | MODEL |
| import { Component, OnInit, Input } from '@angular/core';  import { Recipe } from '../recipe.model';  import { RecipeService } from '../recipe.service';  @Component({  selector: 'app-recipe-detail',  templateUrl: './recipe-detail.component.html',  styleUrls: ['./recipe-detail.component.css']  })  export class RecipeDetailComponent implements OnInit {  recipe: Recipe;  constructor(private recipeService: RecipeService) { }  ngOnInit(): void {  this.recipeService.selectedRecipe.subscribe((recipe: Recipe) => {  this.recipe = recipe;  });  }  } | | export class Recipe {  constructor(public name: string, public description: string, public imagePath: string) {  }  } |

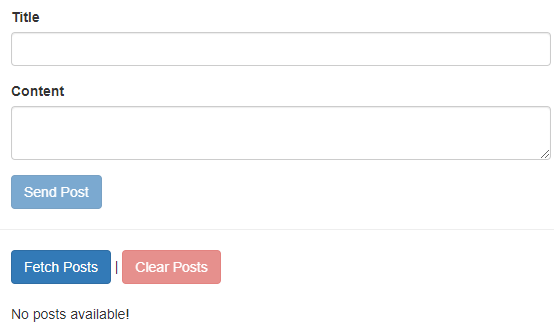
## MAKING HTTP REQUEST

### BACKEND SET UP - SETTING UP FIREBASE

|  |  |  |
| --- | --- | --- |
|  | To set up Db in firebase   1. Access the firebase url with a valid google account - <https://console.firebase.google.com/> 2. Create a firebase project using “Add Project”.Give a name to the project . 3. To Set up database in the firebase project . Click on the project name 🡪 Database 🡪 Realtime Database🡪Create Database | |
| CREATING REALTIME DB   1. Create database 🡪Start test mode 🡪Enable |  | |
|  | | 1. The base path of the REST API will be   [**https://angular-udemy-6a41c.firebaseio.com/**](https://angular-udemy-6a41c.firebaseio.com/)   1. The data will be stored and fetch in JSON format |

To understand the HTTP requests in Angular we will follow a common HTML template and perform different HTTP operation on the same.

SAMPLE FORM:



* This form can able to CREATE, READ , DELETE post stored in the Firebase backend DB

|  |
| --- |
| SAMPLE HTML  <div class="container">  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3">  <form #postForm="ngForm" (ngSubmit)="onCreatePost(postForm.value)">  <div class="form-group"> <label for="title">Title</label> <input type="text" class="form-control" id="title" required ngModel name="title" /> </div>  <div class="form-group"> <label for="content">Content</label> <textarea class="form-control" id="content" required ngModel name="content" ></textarea> </div>  <button class="btn btn-primary" type="submit" [disabled]="!postForm.valid" > Send Post </button> </form>  </div>  </div>  <hr />  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3"> <button class="btn btn-primary" (click)="onFetchPosts()"> Fetch Posts </button> | <button class="btn btn-danger" [disabled]="loadedPosts.length < 1" (click)="onClearPosts()" > Clear Posts </button> </div>  </div>  <div class="row">  <div class="col-xs-12 col-md-6 col-md-offset-3">  <p>No posts available!</p>  </div>  </div>  </div> |
| SAMPLE TS FILE  import { Component, OnInit } from '@angular/core';  import { HttpClient } from '@angular/common/http';  @Component({  selector: 'app-root',  templateUrl: './app.component.html',  styleUrls: ['./app.component.css']  })  export class AppComponent implements OnInit {  loadedPosts = [];  constructor(private http: HttpClient) { }  ngOnInit() { }  onCreatePost(postData: { title: string; content: string }) {  // Send Http request  }  onFetchPosts() {  // Send Http request  }  onClearPosts() {  // Send Http request  }  } |

### POST REQUEST

|  |  |
| --- | --- |
| onCreatePost(postData: { title: string; content: string }) {  this.http  .post(  'https://angular-udemy-6a41c.firebaseio.com/posts.json',  postData  ) .subscribe(responseData => {  console.log(responseData);  });  } | * post() returns an observable for which we need to subscribe * We no subsbcription is done – the request will not be sent. |

### GET REQUEST

* For any HTTP Operation we need to import HttpClientModule on the App Module level.

|  |  |
| --- | --- |
| STEP 1 : IMPORT THE HTTP MODULE IN APP MODULE | STEP 2: INJECT THE SERVICES in COMPONENT |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  import { HttpClientModule } from "@angular/common/http";  import { AppComponent } from "./app.component";  @NgModule({  declarations: [AppComponent],  imports: [BrowserModule, **HttpClientModule**],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | INJECT THE SERVICES OF HttpClientModule IN COMPONENT CONSTRUCTOR  import { Component } from "@angular/core";  import { HttpClient } from "@angular/common/http";  @Component({  selector: "app-root",  templateUrl: "./app.component.html",  styleUrls: ["./app.component.css"]  })  export class AppComponent {  title = "training";  constructor(private http: HttpClient) { }  ngOnInit() {  this.http  .get("https://api.github.com/users/avishekhsinhaRepo")  .subscribe(response => console.log(response));  }  }  Note 🡪subscribe() is like a “then()” in promise |

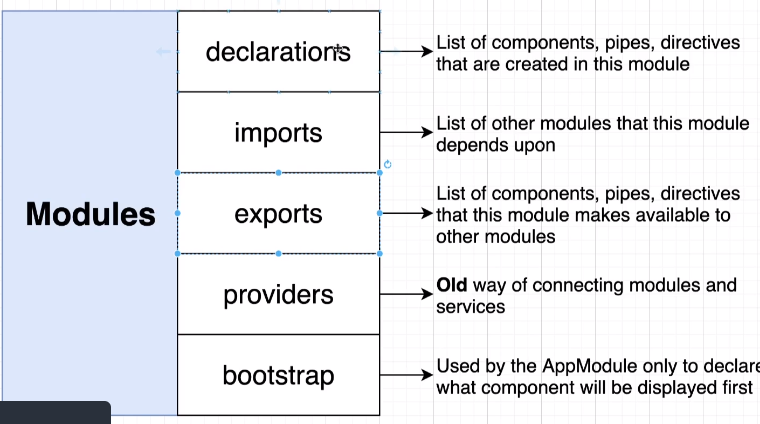
## FORMS

Angular offer2 different approaches for handling forms

1. Template Driven approach
2. Reactive approach

### TEMPLATE DRIVEN APPROACH

## ANGULAR MODULE



|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| 1. Module can be considered as a namespace or container (like java package) 2. @NgModule 🡪 Annotation is used to define a module 3. Each Angular module has module.ts file 4. The module.ts file has a list of    1. The components the module has (declarations)    2. The Angular Services it has (providers)    3. The list of other module which it is importing (imports), we do so if we have dependency on other module. So importing module gives a way use the classes of other module | | | | @NgModule({  declarations: [  AppComponent,  HelloWorldComponent  ],  imports: [  BrowserModule  ],  providers: [],  bootstrap: [AppComponent]  }) | |
|  | | | | CREATING A NEW MODULE (CLI COMMAND)   * ng generate module <module\_name>   Ex.   * ng generate module view   This command will create a blank module “**view**” | |
|  | |  | CREATING THE COMPONENT IN THE NEWLY CREATED MODULE   * ng generate component <module\_name>/<component\_name> * Ex -ng generate component view/address   This CLI command will create an “address” component in the **view** module and add the entry of the “address” component in module.ts file of “view” module | | |
|  | | | Below is the hierarchy we have created so far, so if App Module want to use AddressComponent, it has to import the “view” module i.e.   1. Adding a imports entry in “**app.module.ts**” of view module 2. View Module has to expose the Address component using export | | |
| app.module.ts (import the Module)  @NgModule({  declarations: [  AppComponent,  HelloWorldComponent  ],  imports: [  BrowserModule,  ViewModule  ],  providers: [],  bootstrap: [AppComponent]  }) | view.module.ts (expose the component using export)  @NgModule({  declarations: [AddressComponent],  imports: [CommonModule],  exports: [AddressComponent]  }) | | | | app.component.html  <app-hello-world></app-hello-world>  <app-address></app-address> |
| Note   1. We import the module if we want to use the component of other module 2. The other module must export their components | | | | |

## ROUTING

|  |  |
| --- | --- |
| CREATING ANGULAR APP WITH ROUTING ENABLED | ng new comps –routing |

|  |  |
| --- | --- |
|  |  |
| **MAPPING OF URLS IN ROUTING CAN BE DONE**   * + Urls mapped to angular modules. Each mapped url will routed to a angular module   + Urls mapped to angular component. Each mapped url will routed to a angular component | |

### ROUTING TO MODULES

|  |  |
| --- | --- |
| CREATING A ROUTING MODULE | ng g m <***module\_name***> --routing |

### ROUTING TO COMPONENTS

|  |  |
| --- | --- |
| Application to be developed using Route is Student Management System It will have 4 screens   1. Home Screen [StudentsComponent] 2. Add student Screen[AddStudentComponent] - User can add student name 3. List Student Screen[ListStudentComponent] – List all the added Student 4. Edit Student Screen[EditStudentComponent] – Edit a selected student |  |

### SETTING UP ROUTE

CONFIGURING ROUTE IN APP MODULE

|  |  |
| --- | --- |
| import { Routes, RouterModule } from "@angular/router";  const appRoutes: Routes = [  { path: "", component: StudentsComponent },  { path: "add", component: AddStudentComponent },  { path: "list", component: ListStudentComponent },  { path: "edit", component: EditStudentComponent }  ];  @NgModule({  declarations: [  AppComponent, StudentsComponent,  ListStudentComponent, EditStudentComponent,  AddStudentComponent  ],  imports: [BrowserModule, AppRoutingModule, RouterModule.forRoot(appRoutes)],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {}  HTML [APP ROOT HTML]  <div class="container">  <h2>Student Management</h2>  <ul class="nav nav-tabs">  <li routerLinkActive="active" [routerLinkActiveOptions]="{ exact: true }">  <a routerLink="/">Home</a>  </li>  <li routerLinkActive="active">  <a routerLink="add">Add Student</a>  </li>  <li routerLinkActive="active"><a routerLink="list">List Student</a></li>  <li routerLinkActive="active"><a [routerLink]="['edit'] >Edit Student</a></li>  </ul>  <div class="row">  <router-outlet></router-outlet>  </div>  </div> | 1. The **appRoutes** is an array of objects which has the configuration of Route url versus the corresponding component that will show up on this route url 2. Then finally the **appRoutes** array is added to the import section of the @NgModule of App root component   RouterModule.forRoot(appRoutes)]   1. The route link are further configure into HTML using **routerLink** directive 2. routerLink can also be passed as an array of string   [routerLink]="['edit']  Note –   1. When the routerLink is a relative path – it always with respect to current path – so it appends the url to the current path 2. When the routerLink is a absolute path – it always with respect to root path [e.g. - [http://localhost:4200](http://localhost:4200/)] – so it appends the url to the root path.   ADVANTAGE OF PASSING ROUTE LINK IN AN ARRAY   * [routerLink]="['edit'] 🡪rootPath/edit * routerLink]="['edit',’subLink’] 🡪rootPath/edit/subLink * Example:   <a class="list-group-item" \*ngFor="let user of users;let id = index" [routerLink]="['/users',id,user.name]">  URL GENERATED: /users/2/Chris   1. All the component will appear on the page dynamically on the page , when the route url is changed. ***router-outlet directive becomes the place holder for the component***     <router-outlet></router-outlet>   1. routerLinkActive 🡪 This takes the value of the class name which get add to the element when the give route us active. 2. routerLinkActiveOptions 🡪 |

### RELATIVE AND ABSOLUTE PATHS IN ROUTES

### NAVIGATING TO ROUTE PROGRAMATICALLY

|  |  |
| --- | --- |
| import { Router } from "@angular/router";  @Component({  selector: "app-list-student",  templateUrl: "./list-student.component.html",  styleUrls: ["./list-student.component.css"]  })  export class ListStudentComponent implements OnInit {  constructor(private studentService: StudentService, private router: Router) {}  navigateToEdit() {  // some complex logic – like DB fetch or update  this.router.navigate(["edit"]);  }  } | 1. We use navigate() function to programmatically navigate to different route 2. We follow this approach when we want to do some complex logic – before navigating. 3. The navigate() always loads the route from the root path- irrespective of – whether the path is relative or absolute |

* Since by default the navigate method always load the route with respect to root , but if we want to load the route with respect to a given route we have to pass “relativeTo” parameter to the “navigate()” method

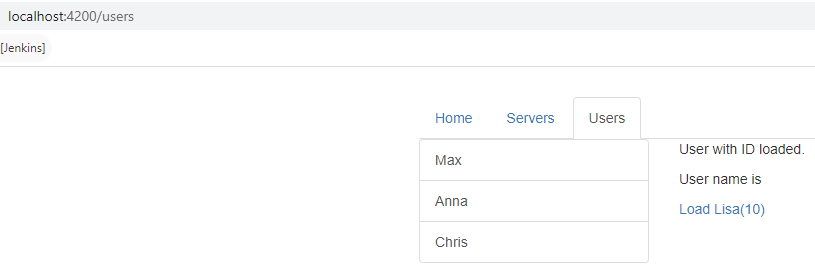
|  |  |
| --- | --- |
| **NAVIGATE METHOD - DEFAULT** | **NAVIGATE METHOD – LOADING ROUTE RELATIVILY** |
| navigateToEdit() {  this.router.navigate(["edit"]);  }  This will navigate to “edit” route with respect to root i.e  <http://localhost:4200/edit> | export class ServersComponent implements OnInit {  constructor(private serversService: ServersService, private router: Router, private route: ActivatedRoute) { }  onReloadServer() {  this.router.navigate(['servers'], { relativeTo: this.route });  }  }  This will navigate to “servers” route with respect to current route. If we are on edit route. Navigate method will take us to  <http://localhost:4200/edit>/**servers** |

### ROUTE PARAMETER PASSING IN ROUTE

|  |  |
| --- | --- |
| CONFIGURING ROUTE in APP MODULE | CONSUMING PARAMETERS IN COMPONENT |
| import { BrowserModule } from "@angular/platform-browser";  import { NgModule } from "@angular/core";  import { AppRoutingModule } from "./app-routing.module";  import { StudentDetailsComponent } from "./student-details/student-details.component";  const appRoutes: Routes = [  { path: "details/**:name**", component: StudentDetailsComponent }  ];  @NgModule({  declarations: [  AppComponent,  StudentDetailsComponent  ],  imports: [  BrowserModule,  AppRoutingModule,  RouterModule.forRoot(appRoutes)  ],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule {} | import { Component, OnInit } from "@angular/core";  import { StudentService } from "../student.service";  import { ActivatedRoute } from "@angular/router";  @Component({  selector: "app-student-details",  templateUrl: "./student-details.component.html",  styleUrls: ["./student-details.component.css"]  })  export class StudentDetailsComponent implements OnInit {  constructor(  private studentService: StudentService,  private route: ActivatedRoute  ) {}  student = "";  studentNameParam = "";  ngOnInit() {  this.studentNameParam = this.route.snapshot.params["name"];  this.student = this.studentService.getStudent(this.studentNameParam);  }  } |
|  | |

### ROUTE OBSERVABLES

* The above technique to retrieve the route parameters can be when we are loading the components with the route change
* This will break when we have route parameter changing asynchronously – like loading the same route with different route parameter.



* In the above example to “Load Lisa” link will load the same route (users) with different route parameters. So it’s a kind of async operations which might happen sometime in future.
* For this kind of operation – we need a observable to handle it.

|  |  |
| --- | --- |
| import { Component, OnInit } from '@angular/core';  import { ActivatedRoute, Params } from '@angular/router';  @Component({  selector: 'app-user',  templateUrl: './user.component.html',  styleUrls: ['./user.component.css']  })  export class UserComponent implements OnInit {  user: { id: number, name: string };  constructor(private activateRoute: ActivatedRoute) { }  ngOnInit() {  this.user = {  id: this.activateRoute.snapshot.params['id'],  name: this.activateRoute.snapshot.params['name']  }  this.activateRoute.params.subscribe((user: Params) => {  this.user = {  id: user.id,  name: user.name  }  })  }  }  HTML  <a [routerLink]="['/users','10','Lisa']">Load Lisa(10)</a> | * this.activateRoute.params is a observable for which we need to subscribe to handle async change of route * Usually we need to unsubscribe from the observable in the onDestroy() life cycle hook. Since its an Angular managed observable – we don’t have to do it explicitly. Angular will do it for us. |

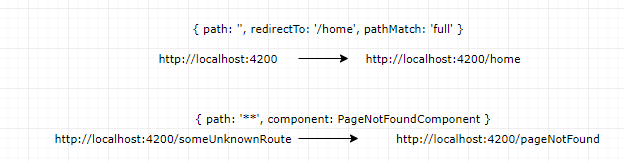
### QUERY STRING PARAMETERS AND FRAGMENTS - PASSING IN ROUTE

|  |  |
| --- | --- |
| FROM TEMPLATE | OUTPUT |
| <a class="list-group-item"  \*ngFor="let user of users;let id = index" [routerLink]="['/users',id,user.name]" **[queryParams]={admin:true,hasAccess:true}**  fragment="loading" >  {{ user.name }}  </a> | * The query parameters are passed as an object |
| PROGRAMATICALLY- SETTING THE QUERY PARAMETERS | export class HomeComponent implements OnInit {  constructor(private router: Router, private route: ActivatedRoute) { }  onLoadServer() {  this.router.navigate(['/servers'],  { queryParams: { allowEdit: 1 },  fragment: 'loading' });  }  } |
| RETRIVING THE QUERY STRING PARAMETERS | |
| * The queryParams (this.route.queryParams) and this.route.fragment) are observables also – so they too can be subscribed to . | let allowEdit = this.route.snapshot.queryParams['allowEdit'];  console.log(allowEdit);  console.log(this.route.snapshot.fragment); |

### CHILD ROUTES

|  |  |
| --- | --- |
| **CHILD ROUTES** | **SERVERS COMPONENT HTML** |
| const appRoutes: Routes = [  { path: '', component: HomeComponent },  { path: 'users', component: UsersComponent },  {  path: 'servers', component: ServersComponent, children: [  { path: ':id/edit', component: EditServerComponent },  { path: ':id', component: ServerComponent },  ]  },  { path: 'users/:id/:name', component: UserComponent },  ];  @NgModule({  declarations: [  AppComponent,  HomeComponent,  UsersComponent,  ServersComponent,  UserComponent,  EditServerComponent,  ServerComponent  ],  imports: [  BrowserModule,  FormsModule,  RouterModule.forRoot(appRoutes)  ],  providers: [ServersService],  bootstrap: [AppComponent]  })  export class AppModule { } | <router-outlet></router-outlet>   1. EditServerComponet and ServerComponent is getting treated as child route of Servers Component 2. So to load the child Route – We need to add the <router-outlet> in the Parent Component Template(ServersComponent) |

### REDIRECTION AND WILD CHARACTERS IN ROUTE CONFIGURATION



|  |  |
| --- | --- |
| const appRoutes: Routes = [  { path: '', redirectTo: '/home', pathMatch: 'full' },  { path: 'home', component: HomeComponent },  {  path: 'users', component: UsersComponent, children: [  { path: ':id/:name', component: UserComponent }  ]  },  {  path: 'servers', component: ServersComponent, children: [  { path: ':id/edit', component: EditServerComponent },  { path: ':id', component: ServerComponent },  ]  },  { path: '\*\*', component: PageNotFoundComponent },  ];  @NgModule({  declarations: [  AppComponent,  HomeComponent,  UsersComponent,  ServersComponent,  UserComponent,  EditServerComponent,  ServerComponent,  PageNotFoundComponent  ],  imports: [  BrowserModule,  FormsModule,  RouterModule.forRoot(appRoutes)  ],  providers: [ServersService],  bootstrap: [AppComponent]  })  export class AppModule { } | * The “\*\*” route must be the last route in the route configuration list * By default, Angular matches paths by prefix. That means, that the following route will match both /recipes  and just /   { path: '', redirectTo: '/somewhere-else' }  Angular will give an error here, because that's a common gotcha: This route will now ALWAYS redirect you! Why?  Since the default matching strategy is "prefix" , Angular checks if the path you entered in the URL does start with the path specified in the route. Of course every path starts with ''  (Important: That's no whitespace, it's simply "nothing").  To fix this behavior, you need to change the matching strategy to "full" :  { path: '', redirectTo: '/somewhere-else', pathMatch: 'full' }  Now, you only get redirected, if the full path is ''  (so only if you got NO other content in your path in this example). |

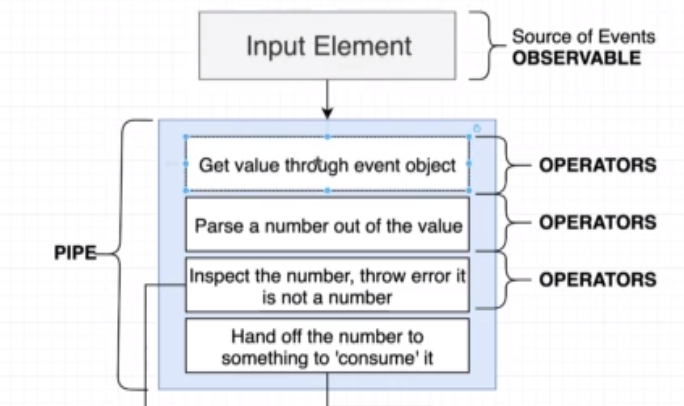
### ROUTE GUARDS

## RXJS

* It’s a separate library from Angular
* We use this instead of promise or async/await for handling async stuff.

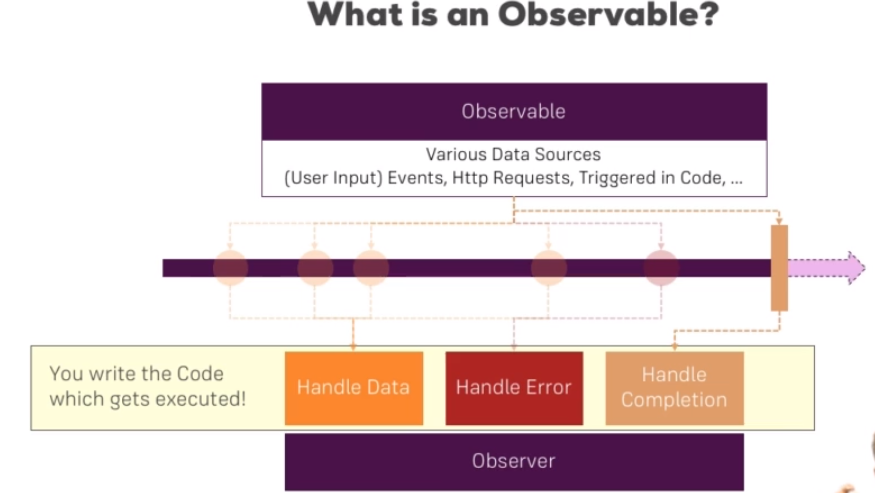
### TERMINOLOGIES

1. **OBSERVABLE**: The entity which emit the event/data
2. **OPERATOR**: The invidualal operation which act on emited data for processing
3. **PIPE**: The series of preprocessting activity is called pipes



1. OBSERVER:
   1. The data / event handler which finally gets the data after pre-processing.
   2. The pipe terminates in case of error and further handled by Observer as a error handler.

### OBSERVABLES



* Angular uses Observables from a third-party package RxJS
* Obserables can be consided as a data source like user input (which triggers an event), HTTP requests etc..
* Observers are the piece of logic which listens to the observables.(code in subscribe function)
* Observables are basically used to handle asynchronous tasks like user event or response from an API . Earlier we use Promises to do so.

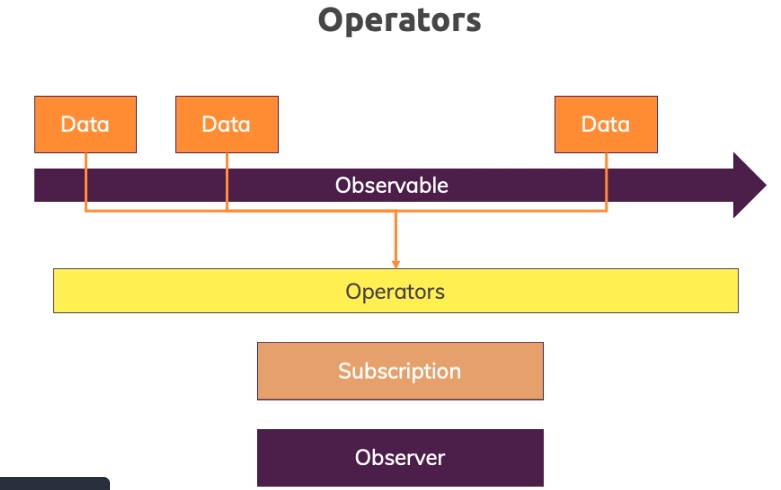
#### OBSERVABLES – EXAMPLE –(INTERVAL)

|  |  |
| --- | --- |
| import { Component, OnInit, OnDestroy } from '@angular/core';  import { interval, Subscription } from 'rxjs';  @Component({  selector: 'app-home',  templateUrl: './home.component.html',  styleUrls: ['./home.component.css']  })  export class HomeComponent implements OnInit, OnDestroy {  private intervalSubscriber: Subscription;  constructor() { }  ngOnInit() {  this.intervalSubscriber = interval(2000).subscribe((data) => {  console.log(data);  });  }  ngOnDestroy(): void {  this.intervalSubscriber.unsubscribe();  }  } |  |

#### BUILDING CUSTOM OBSERVABLES

|  |  |
| --- | --- |
| import { Component, OnDestroy, OnInit } from '@angular/core';  import { Subscription, Observable } from 'rxjs';  @Component({  selector: 'app-home',  templateUrl: './home.component.html',  styleUrls: ['./home.component.css']  })  export class HomeComponent implements OnInit, OnDestroy {  private firstObsSubscription: Subscription;  constructor() { }  ngOnInit() {  const customIntervalObservable = Observable.create(observer => {  let count = 0;  setInterval(() => {  observer.next(count);  if (count === 5) {  observer.complete();  }  if (count > 3) {  observer.error(new Error('Count is greater 3!'));  }  count++;  }, 1000);  }); | this.firstObsSubscription = customIntervalObservable.subscribe(data => {  console.log(data);  }, error => {  console.log(error);  alert(error.message);  }, () => {  console.log('Completed!');  });  }  ngOnDestroy(): void {  this.firstObsSubscription.unsubscribe();  }  } |
| Observer.next(data) 🡪 The data in the next method is emitted to success callback of the observer  observer.error(new Error(message)) 🡪This will terminate the observer with an error and call the error callback of the observer.  observer.complete() 🡪 This will complete the observable and calles the completion callback of the observer |
| OBSERVABLE AND OBSERVER MAPPING | |
|  | |

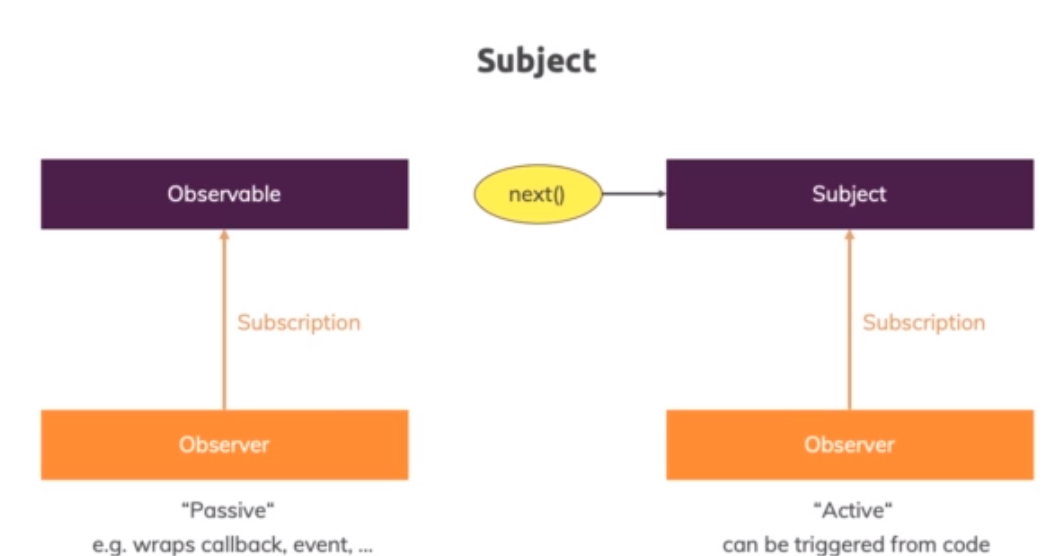
#### OPERATORS – OBSERVABLES

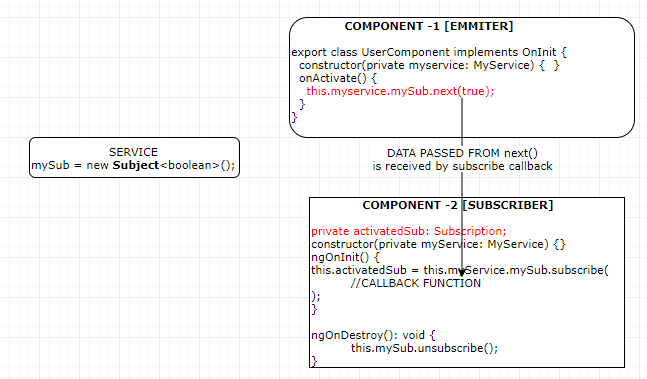


* When observable emits data. This might be possible that the subscriber needs the pre-processed data .
* The pre-processing of emited data before it is sent to obserser is done by Operators.
* List of operators : <https://www.learnrxjs.io/learn-rxjs/operators/complete>

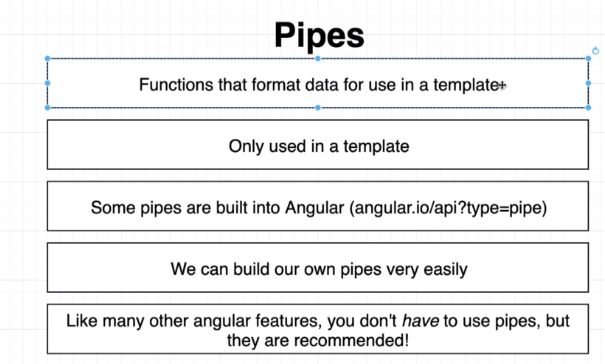
#### IMPORTANT AND COMMONLY USED OPERATORS

#### SUBJECTS - OBSERVABLES





## ANGULAR PIPES



* Angular pipes are used to transform the output in the template(HTML)
* Angular has build in pipes (as in below eample)- <https://angular.io/api?type=pipe>
* We can create custom pipes

### BUILT IN PIPES

|  |  |
| --- | --- |
| The value of username becomes an input for the “uppercase” pipe |  |
| TS FILE  import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-simplepipe",  templateUrl: "./simplepipe.component.html",  styleUrls: ["./simplepipe.component.css"]  })  export class SimplepipeComponent implements OnInit {  constructor() {}  todaysDate = new Date();  ngOnInit() {}  } | HTML  <div class="container">  <p>{{ todaysDate | date }}</p>  <label for="userName">User Name</label>  <input type="text" [(ngModel)]="userName" name="userName" />  <p>Filtered Name: {{ userName | uppercase }}</p>  </div> |

### PARAMETERIZING AND CHAINING PIPES

* The pipes can be parameterized too using “**colon**” sign . Let’s consider an example where we want to filter the output of the Date
* todayDate = new Date() 🡪 Sun May 17 2020 12:47:43 GMT+0530 (India Standard Time)

|  |  |
| --- | --- |
| PARAMETERING PIPES | |
| PIPE | OUTPUT |
| {{ todayDate | date}} | May 17, 2020 |
| {{ todayDate | **date:’fullDate’**}} | Sunday, May 17, 2020 |
| PIPE CHAINING  In pipe chaining output of one pipe becomes input for another chained pipe. Its gets executed from left to right | |
| {{ todayDate | date:’fullDate’|uppercase}} | SUNDAY, MAY 17, 2020 |

### CREATING CUSTOM PIPES

* We can create custom pipes to for some custom functionality. Let’s say we want to create a pipe which will add “…” if the number of character is more than 10 character.

|  |  |  |
| --- | --- | --- |
| CREATING PIPES | * ng g p shortenpipe * ng generate pipe shortenpipe | |
| DECLARE THE PIPE IN APP MODULE | shortenpipe.pipe.ts | |
| import { BrowserModule } from '@angular/platform-browser';  import { NgModule } from '@angular/core';  import {FormsModule} from '@angular/forms'  import { AppComponent } from './app.component';  import { ShortenpipePipe } from './shortenpipe.pipe';  @NgModule({  declarations: [  AppComponent,  ShortenpipePipe  ],  imports: [  BrowserModule,  FormsModule  ],  providers: [],  bootstrap: [AppComponent]  })  export class AppModule { } | import { Pipe, PipeTransform } from "@angular/core";  @Pipe({  name: "shortenpipe"  })  export class ShortenpipePipe implements PipeTransform {  transform(**value**: any, ...args: any[]): any {  if (value != undefined && value.length > 10) {  return value.substring(0, 10).concat("...");  }  return value;  }  } | |
| HTML  <div class="container">  <p>{{ todaysDate | date: "fullDate" | uppercase }}</p>  <label for="userName">User Name</label>  <input type="text" [(ngModel)]="userName" name="userName" />  <p>  <strong>  Name: {{ userName | uppercase | shortenpipe }}  </strong>  </p>  </div> | |
| 1. Pipe can be declared using @Pipe decorator, which has a name property to assign a name to the property. 2. The output of userName | uppercase becomes input for ”shortenpipe” pipe | | |
| PARAMETERIZING CUSTOM PIPE | | |
| TS FILE  import { Pipe, PipeTransform } from "@angular/core";  @Pipe({  name: "shortenpipe"  })  export class ShortenpipePipe implements PipeTransform {  transform(value: any, limit): any {  if (value != undefined && value.length > limit) {  return value.substring(0, 10).concat("...");  }  return value;  }  } | | HTML  <div class="container">  <p>{{ todaysDate | date: "fullDate" | uppercase }}</p>  <label for="userName">User Name</label>  <input type="text" [(ngModel)]="userName" name="userName" />  <p> <strong>  Name: {{ userName | uppercase | shortenpipe: 10 }}  </strong> </p></div> |

EXAMPLE 2: FILTERING OUTPUT: Filtering the output (names ) based on filter entered

|  |  |  |
| --- | --- | --- |
| TS FILE (PIPE) | COMPONENT HTML | |
| import { Pipe, PipeTransform } from "@angular/core";  @Pipe({  name: "filternames"  })  export class FilternamesPipe implements PipeTransform {  transform(value: any, filterString: string): any {  if (filterString == undefined) {  return value;  }  const filteredNames = [];  value.forEach(element => {  if (element.indexOf(filterString) > -1) {  filteredNames.push(element);  }  });  return filteredNames;  }  } | <div class="container">  <label for="userName">Filter Name</label>  <input type="text" [(ngModel)]="**filterString**" name="userName" />  <p \*ngFor="let uname of userNames | filternames: filterString">  {{ uname }}  </p>  </div> | |
| COMPOENNT TS  import { Component, OnInit } from "@angular/core";  @Component({  selector: "app-simplepipe",  templateUrl: "./simplepipe.component.html",  styleUrls: ["./simplepipe.component.css"]  })  export class SimplepipeComponent implements OnInit {  constructor() {}  userNames = [];  ngOnInit() {  this.userNames.push("John");  this.userNames.push("Mike");  this.userNames.push("Baker");  this.userNames.push("Stephen");  this.userNames.push("Rob");  this.userNames.push("Tina");  }  } | INITIAL VIEW | FILTERED VIEW |

## ANGULAR MODULES & OPTIMIZATION

## ANGULAR DEPLOYMENT

## ANGULAR UNIVERSAL

### WHAT IS ANGULAR UNIVERSAL?

* Angular universal is node-based rendering engine for Angular which is used for Server side rendering or build time pre-rendering of an Angular Application.
* A normal Angular application executes in the browser, rendering pages in the DOM in response to user actions. Angular Universal executes on the server, generating static application pages that later get bootstrapped on the client. This means that the application generally renders more quickly, giving users a chance to view the application layout before it becomes fully interactive.

### WHY ANGULAR UNIVERSAL?

#### PERFORMANCE BENEFITS

Server rendering generates the full HTML for a page on the server in response to navigation. This avoids additional round-trips for data fetching and templating on the client since it’s handled before the browser gets a response.



This will quickly load the initial page, which will give the user a chance to quickly view the initial layout until a complete angular application becomes fully interactive.

#### SOCIAL MEDIA CRAWLERS

#### SEO (SEARCH ENGINE OPTIMIZATION)

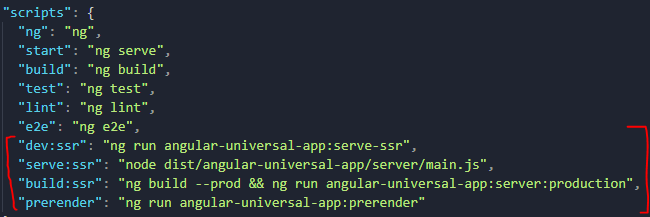
### SETTING UP ANGULAR UNIVERSAL?

|  |  |
| --- | --- |
| CLI COMMAND TO SET UP ANGULAR UNIVERSAL | ng add @nguniversal/express-engine |



#### IMPACT ON APP AFTER ANGULAR UNIVERSAL SET UP

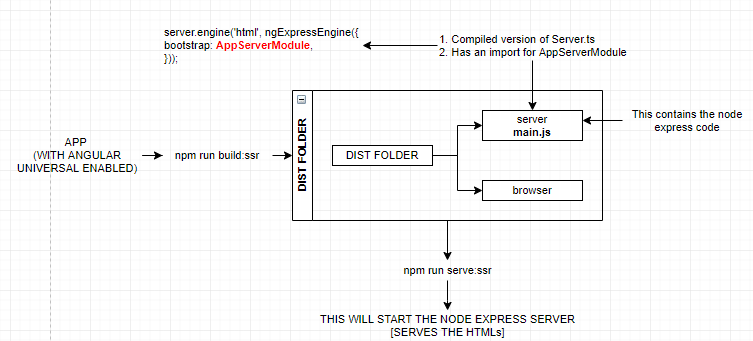
##### CHANGES IN PACKAGE.JSON



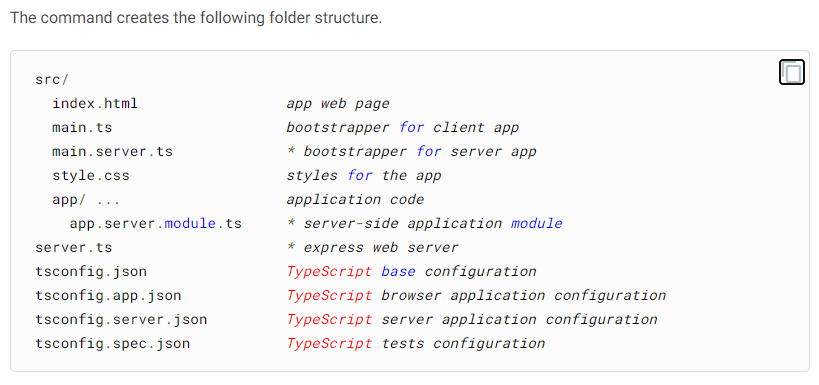
* After enabling angular universal – it adds some angular universal npm script in package.json.
* ssr: Stand for Server Side Rendering

|  |  |
| --- | --- |
| dev:ssr | * It runs the application in hot reload mode. * Used basically during development as it tracks the changes in files. * This runs the server side rendered version of the application. |
| serve:ssr | * This script start the express server – which will serve the HTML on demand |
| build:ssr | * This npm script builds the PROD version of server-side rendering on the Application. It first builds the browser version which then helps in creating the server version of the application * The output of the commands is – It creates 2 folders in the dist directory i.e “server” and “browser”. * The “browser” is client-side rendering version of the application and “server” contains the server side rendering version of the app. * The client version in-turn helps in generating the server-side version of the app. * The main.js in in the server folder will start the Node Express server – which will be responsible for serving the HTMLs |
| prerender |  |

##### NPM SCRIPT CHAIN – PROD BUILD



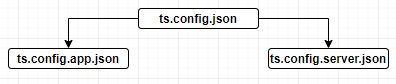
#### CHANGES IN FOLDER STRUCTURE



###### APP.SERVER.MODULE.TS

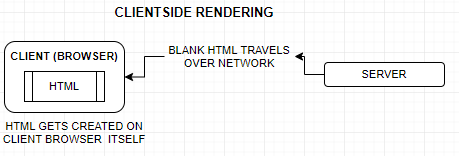
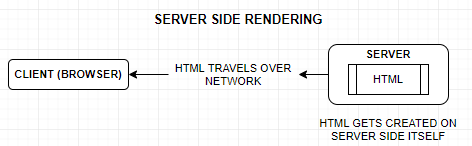
|  |  |
| --- | --- |
| app.server.module.ts  import { NgModule } from '@angular/core';  import { ServerModule } from '@angular/platform-server';  import { AppModule } from './app.module';  import { AppComponent } from './app.component';  @NgModule({  imports: [  AppModule,  ServerModule,  ],  bootstrap: [AppComponent],  })  export class **AppServerModule** {} | * In a general angular app . it has just one App module(app.module.ts). Enabling angular universal generates another module (app.server.module.ts) * The **AppServerModule** module imports the AppModule (this imports the entire app module component as well) and ServerModule * This actually creates two application in a angular app. A server side app and client side app. |

###### TS.CONFIG



* It creates 2 typescript config files – both inherits a base typescript config file “ts.config.json”
  + ts.config.app.json – For client-side rendering application
  + ts.config.server.json – For server-side rendering application

### SERVER AND CLIENT SIDE RENDERING AND RUNTIME INTERACTION

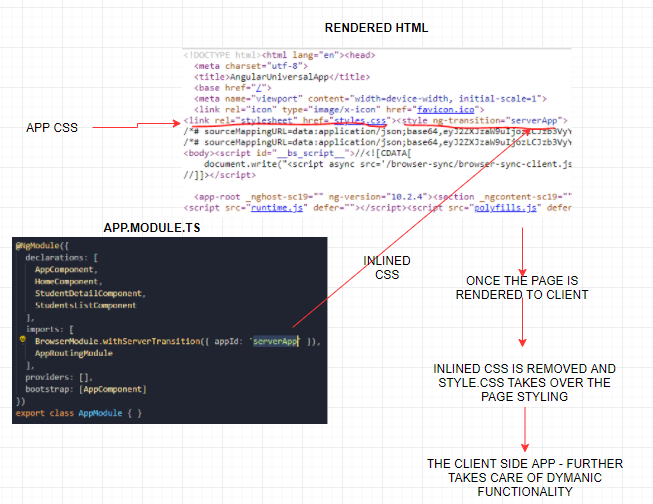


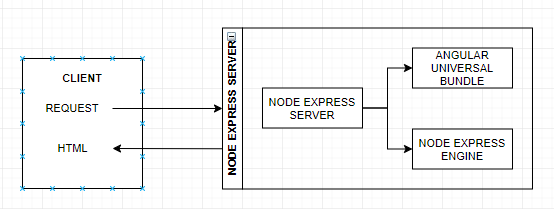
|  |  |
| --- | --- |
| HTML RESPONSE (CLIENT-SIDE RENDERING) | HTML RESPONSE (SERVER-SIDE RENDERING) |
| * In the client side rendering the HTML passed from the server is an empty HTML with <app-root></app-root> and Java script * The JS futher generates the HTML dynamically on client browser. | * In the server side rendering the entire HTML passed from the server to the client browser. It happens in the similar ways as some server side scripting language works(JSP) * No further HTML generation is needed on the client side. |

#### RUNTIME INTERACTION BETWEEN CLIENT AND SERVER-SIDE APP

1. The server-side module imports all the App module components – so the server rendering engine knows what HTML it must generates.
2. Due to the inlined css the end user never feels that the interface of app is broken.
3. The “appId” identifier helps angular to remove the inlined style from finally rendered page.

BrowserModule.withServerTransition({ **appId**: **'serverApp'** })





* Whenever a request goes from a browser – it goes to the node express server
* The node express server then make us of Angular Universal Bundle and Node Express engines to generate the HTML , which finally rendered back to the client.

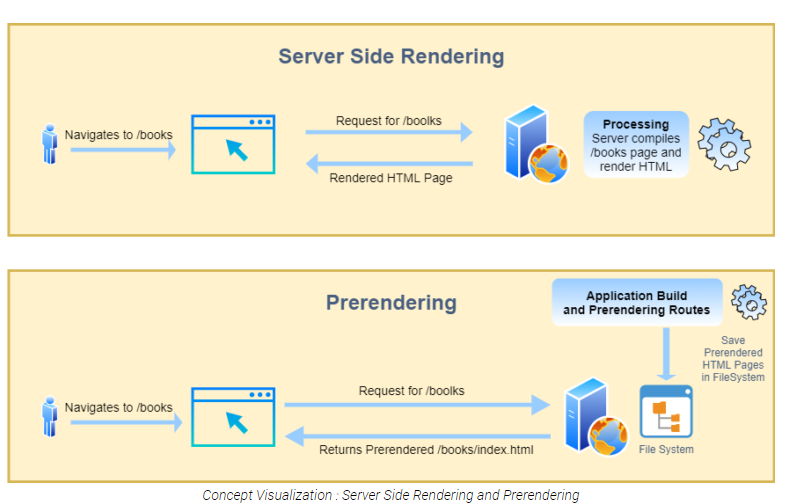
### ANGULAR UNIVERSAL – PRE- RENDERING

* Apart from getting the HTML from the server on demand. Angular universal can also be used to pre-render the HTML as well.
* **Server-Side Rendering (SSR):**  rendering a client-side or universal app to HTML on the server.
* **Prerendering**: running a client-side application at build time to capture its initial state as static HTML.

In both rendering techniques, we generate static HTML pages which can be easily crawled by crawlers.

Just the major difference is the location where HTML pages are generated. In Server-Side Rendering Technique when the user navigates to the URL => for example - /books route, the server compiles the application(render on a server) and sends the generated HTML page back to the client.

While In Prerendering Technique, All routes of the application are compiled at build time and saved as static HTML pages on file-system. These HTML pages can be served to the client using CDN. So whenever the user navigates to any route, he/she will get prerendered HTML pages.

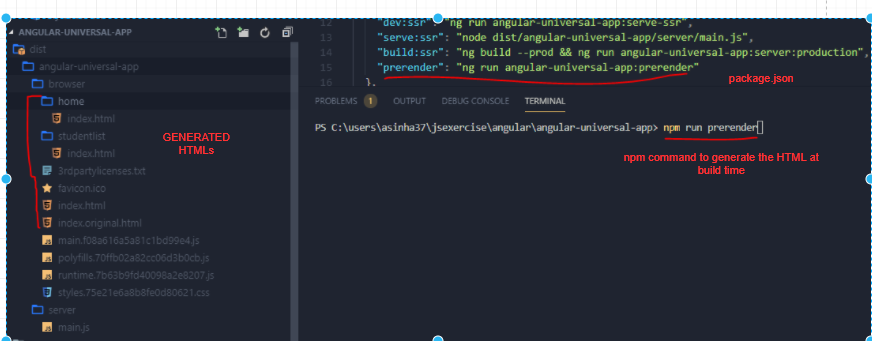


#### RUNNING PRE-RENDERING – BUILD TOOL

NPM SCRIPT : npm run prerender

#### CONCEPT OF PRE-RENDERING

* Usually the server-side render HTML are sent to the client on demand by the express server during the runtime.
* Pre-rendering help us to generates the HTML during the build time itself (in the dist folder). So, when we are prerendering the HTML we truly don’t need the express server. A simple HTTP server is enough to render the HTMLs
* To prerender the HTML – we need to know the all the entry points upfront of the application



* The Prerender build tool, generates the HTML along with the inline CSS as well.
* It generated the HTMLs (in the dist folder ) of static entry points. The dynamic entry points are handles differently.

#### SERVING THE PRE-RENDERED HTML TO CLIENT

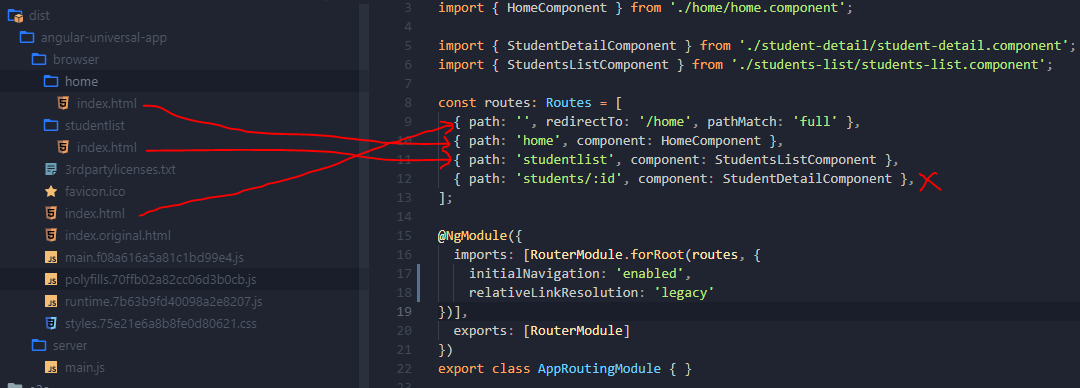
* The prerendered HTMLs can be servered by static http servers as well , rather than using a express server
* To run the HTTP server
  + Install the http-server dependency : npm i http-server
  + To run the application
    - Add the script in package.json :

serve:prerender": "http-server -c-1 dist/angular-universal-course/browser"

* Running the script: npm run serve:prerender

#### PRERENDERING ROUTES

* For routes the prerender build script generates the HTML of static routes.
* The prerender build script first scans the App routing module and first generates the HTMLs of static route
* The dynamic route prerendering is handled diffrently



#### PRERENDERING DYNAMIC ROUTES

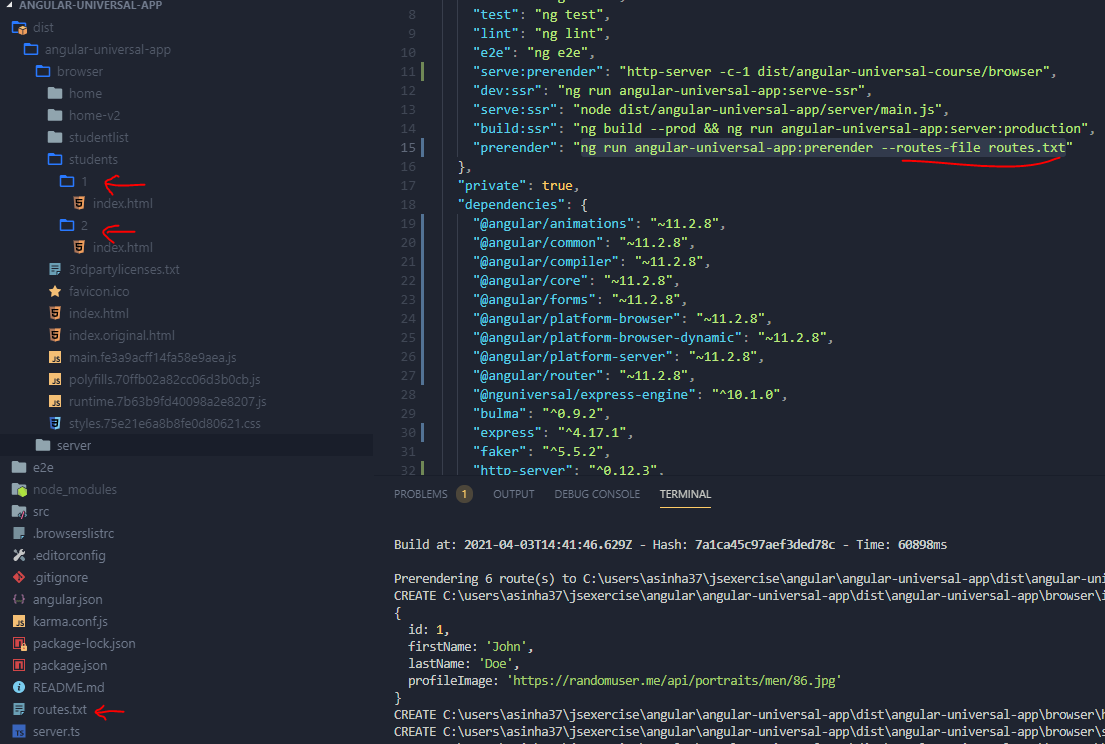
* Its easy for angular to create the prerendered HTMLs for static route. But has no way to know the HTML of the dynamic routes.
* Angular has a different way to handle dynamic routes. We must tell angular possible dynamic route upfront. The different ways to declare the dynamic routes are
  + The dynamic route can eb declared in package.json file
  + Or, dynamic routes can be configured in a text file.

##### PACKAGE.JSON CONFIGURATION

The routes can be added in pre-render npm script in package.json

##### TEXT FILE CONFIGURATION CONFIGURATION

|  |  |
| --- | --- |
| We need to create a .txt file e.g. **route.txt** in the application root directory and declare all the possible dynamic routes. |  |
| Then to generate the pre-rendered HTML run the following npm script | ng run angular-universal-app:prerender --**routes-file routes.txt** |



### APPLICATION SHELL

### STATE TRANSFER API