Table of Contents

[Keyboard Shortcuts 1](#_Toc172105807)

[Typescript 1](#_Toc172105808)

[Typescript Setup 1](#_Toc172105809)

[Variable Declaration 1](#_Toc172105810)

[Function Declaration 2](#_Toc172105811)

[Named Functions: 2](#_Toc172105812)

[Arrow Functions: 2](#_Toc172105813)

[Named Function Expressions: 2](#_Toc172105814)

[Function Parameters 2](#_Toc172105815)

[Optional parameters: 2](#_Toc172105816)

[Required Parameters: 3](#_Toc172105817)

[Rest parameters: 3](#_Toc172105818)

[Generic Functions 3](#_Toc172105819)

[Classes 3](#_Toc172105820)

[Creating an instance of the class (an object). 4](#_Toc172105821)

[Constructors 4](#_Toc172105822)

[Access Modifiers and Inheritance 5](#_Toc172105823)

[Getters and Setters 6](#_Toc172105824)

[Interfaces 7](#_Toc172105825)

[Interface Implementation 8](#_Toc172105826)

[Angular 10](#_Toc172105827)

[Angular Setup And Installation 10](#_Toc172105828)

[Adding Styling Frameworks to your project 11](#_Toc172105829)

[Component Syntax 11](#_Toc172105830)

[component attributes 11](#_Toc172105831)

[Binding Syntax 12](#_Toc172105832)

[i) Interpolation 12](#_Toc172105833)

[ii) Property binding 12](#_Toc172105834)

[iii) Event binding 12](#_Toc172105835)

[Directives 12](#_Toc172105836)

[Structural Directives 12](#_Toc172105837)

[Attribute Directives 13](#_Toc172105838)

[Pipes 13](#_Toc172105839)

[Component Communication 14](#_Toc172105840)

[i) Using @Input and @Output 14](#_Toc172105841)

[ii) Using @ViewChild, @ViewChildren and @ContentChild 15](#_Toc172105842)

[@ViewChild 15](#_Toc172105843)

[ng-template 15](#_Toc172105844)

[@ViewChildren 16](#_Toc172105845)

[@ContentChild 17](#_Toc172105846)

[iii) Using Services 17](#_Toc172105847)

[Lifecycle Hooks 18](#_Toc172105848)

[i) ngOnInit 18](#_Toc172105849)

[ii) ngOnDestroy 18](#_Toc172105850)

[iii) ngOnChanges 19](#_Toc172105851)

[iv) ngDoCheck 20](#_Toc172105852)

[v) ngAfterViewInit 20](#_Toc172105853)

[vi) ngAfterViewChecked 20](#_Toc172105854)

[vii) ngAfterContentInit 20](#_Toc172105855)

[viii) ngAfterContentChecked 20](#_Toc172105856)

[Dependency Injection [Providers] 21](#_Toc172105857)

[i) Class-Based Providers [Services] 21](#_Toc172105858)

[Creating a Service 21](#_Toc172105859)

[Injecting the Service to a Component 22](#_Toc172105860)

[Resolution Modifiers 23](#_Toc172105861)

[i) @Self() 23](#_Toc172105862)

[ii) @SkipSelf() 23](#_Toc172105863)

[iii) @Optional() 23](#_Toc172105864)

[iv) @Host() 24](#_Toc172105865)

[ii) Value-Based Providers 25](#_Toc172105866)

[Example of a custom api 25](#_Toc172105867)

[Example of a browser api 26](#_Toc172105868)

[Angular Http and Observables 27](#_Toc172105869)

[Http 27](#_Toc172105870)

[Procedure for Setting up Http in your project 27](#_Toc172105871)

[Observables 29](#_Toc172105872)

[“Observer” (Stream Tracker) methods 29](#_Toc172105873)

[Performing CRUD operations with Endpoints 30](#_Toc172105874)

[GET 32](#_Toc172105875)

[POST 32](#_Toc172105876)

[PUT 33](#_Toc172105877)

[DELETE 35](#_Toc172105878)

[REQUEST 36](#_Toc172105879)

[Unsubscribing from Rxjs streams 37](#_Toc172105880)

[Using “Subscription” Property 37](#_Toc172105881)

[Async Pipe 37](#_Toc172105882)

[Rxjs Operators 38](#_Toc172105883)

[shareReplay 38](#_Toc172105884)

[catchError 39](#_Toc172105885)

[Map 39](#_Toc172105886)

[mergeMap 41](#_Toc172105887)

[switchMap 41](#_Toc172105888)

[exhaustMap 41](#_Toc172105889)

[Modifying HttpRequests 41](#_Toc172105890)

[HttpHeaders (Adding Headers to HttpRequests) 41](#_Toc172105891)

[HttpInterceptors 42](#_Toc172105892)

[App Initialiser 43](#_Toc172105893)

[Angular Router 45](#_Toc172105894)

[Configuring Custom Static Routes 45](#_Toc172105895)

[Dynamic Routes 46](#_Toc172105896)

[Nested Routes 48](#_Toc172105897)

[“Router” Service (Triggering Navigation from one component to another after some function) 49](#_Toc172105898)

[Route Events 49](#_Toc172105899)

[- NavigationStart 50](#_Toc172105900)

[- NavigationEnd 50](#_Toc172105901)

[Route Guards 51](#_Toc172105902)

[canActivate() 51](#_Toc172105903)

[canActivateChild() 53](#_Toc172105904)

[canLoad() 54](#_Toc172105905)

[canDeactivate() 54](#_Toc172105906)

[Resolve (Data Prefetching) 56](#_Toc172105907)

[Angular Material 58](#_Toc172105908)

[Adding Angular Material 58](#_Toc172105909)

[Adding Angular Material Schematics (Navigation Component) 58](#_Toc172105910)

[Angular Forms 59](#_Toc172105911)

[Template-Driven Forms 59](#_Toc172105912)

[Form Attributes 59](#_Toc172105913)

[Form Initialisation 61](#_Toc172105914)

[Two-way Data Binding 62](#_Toc172105915)

[Reseting Forms 62](#_Toc172105916)

[Reactive Forms 63](#_Toc172105917)

[Configuring and initializing the form in the .ts file 63](#_Toc172105918)

[Rendering the form on the Template 64](#_Toc172105919)

[Setting up Angular Material for use in the form 68](#_Toc172105920)

[Adding and Removing Controls Dynamically To Your Reactive Form 68](#_Toc172105921)

[Using FormBuilder’s “.array” 68](#_Toc172105922)

[Using AddControl Method 70](#_Toc172105923)

[Using RemoveControl Method 71](#_Toc172105924)

[Validators in Reactive Forms 72](#_Toc172105925)

[Resetting Reactive Forms 73](#_Toc172105926)

[PatchValue and SetValue 73](#_Toc172105927)

[Adding Mat-Error at the form control level 74](#_Toc172105928)

[Listening To Changes In the Form Controls’ Values 75](#_Toc172105929)

[Custom Directives With Forms 76](#_Toc172105930)

[Creating a Custom Directive 76](#_Toc172105931)

[Linking Directives with Components 78](#_Toc172105932)

[Custom Validators 78](#_Toc172105933)

[For Template-Driven Forms 78](#_Toc172105934)

[For Reactive Forms 80](#_Toc172105935)

[Feature Modules 82](#_Toc172105936)

[Setting up feature modules 82](#_Toc172105937)

[Lazy Loading 84](#_Toc172105938)

[Manual Configuration 84](#_Toc172105939)

[Lazy Loading Using Angular CLI 85](#_Toc172105940)

[Custom Pipe (Adding a Filter to Data Using a Form Control) 87](#_Toc172105941)

[Global Error Handling 88](#_Toc172105942)

[Testing 90](#_Toc172105943)

**Angular**

## Keyboard Shortcuts

Ctrl + P (On VsCode and Chrome) – to open the search file panel (within the “sources” tab of the inspector in chrome.)

Ctrl + Space – To add an import

Ctrl + Shift + [ - To fold a block of code

Ctrl + Shift + ] - To unfold a block of code

# Typescript

## Typescript Setup

npm init – Creates a package.json file. This keeps track of all the packages being used in your project.

npm install –g typescript - Installs typescript on your computer and adds it to your system environment variables.

tsc –init – Creates a tsconfig file in your project

tsc – runs the compiler and converts the .ts file to a .js file.

Node some\_file – Runs the converted .js file.

## Variable Declaration

let name: string;

let age: number;

let isValid: boolean; // Has a default value of “undefined ”

let employeeList: string[]; // equivalent to “let employeeList: Array<string>;” - known as generics

let dob: string = “25”;

let result = parseInt(dob);

const enum Color {

Red,

Green,

Blue

}

let c: Color = Color:Blue;

let swapNums: [firstNum: number, secondNum: number]; // This are called tuples

function swapNumbers(num1: number, num2: number): [number,number] {

return [num2,num1];

}

let someValue: any; // Accepts values of any type. Also, any variable that is not explicitly given a type when being declared is automatically given a type of “any”

## Function Declaration

### Named Functions:

*function add(num1: number,num2: number): number {*

*return num1 + num2;*

*}*

### Arrow Functions:

*let difference = (num1: number,num2: number): number => num1 - num2;*

### Named Function Expressions:

*let product = function (num1: number,num2: number): number {*

*return num1 \* num2;*

*}*

## Function Parameters

### Optional parameters:

*function add(num1: number,num2: number, num3?: number): number {*

*return num3 ? num1 + num2 + num3 : num1 + num2;*

*}*

### Required Parameters:

*function add(num1: number,num2: number, num3 = 10): number {*

*return num1 + num2 +num3;*

*} // If no value is passed during the function call, num3 is assigned a value of 10*

### Rest parameters:

*function add(num1: number, num2: number, …num3: number[]): number {*

*return num1 + num2 + num3.reduce((a, b) => a + b, 0);*

*}*

Let numbers = [1, 2, 3, 4, 5];

add(1, 2, …numbers); // Add 1, 2 and the rest of the numbers.

add(1,2,3,4,5,6) // Also adds 1, 2, and the rest of the numbers

## Generic Functions

These are used when you do not want to decide the type of parameter and return values before the function is actually used.

*function getItems<Type>(items : Type[]): Type[] {*

*return new Array<Type>().concat(items);*

*}*

For such a function, these will work:

let concatenatedNumbers = getItems<number>([1,2,3,4,5]);

let concatenatedString = getItems<string>([“1”, “2”, “3”, “4”, “5”])

## Classes

class Employee {

id!: number; // The “!” is to indicate that the property with not be assigned a default value immediately. Without it and without a default value, an error occurs.

name!: string;

address!: string;

}

### Creating an instance of the class (an object).

let john = new Employee();

john.id = 1;

john.name = ‘John’;

john.address = ‘Highway 71’;

console.log(john); // Returns *“Employee {id: 1, name: ’John’, address: ‘Highway 71’}”*

### Constructors

These are responsible for creating an instance of a class (object);

There are two types:

A default constructor – With no parameters

A parameterized constructor – With parameters

In typescript, you cannot have both.

class Employee {

id: number; // Notice the missing “!”, because below, in the constructor, we have defined some default values.

name: string;

address: string;

constructor(id: number, name: string, address: string) {

this.id = id;

this.name = name;

this.address = address;

} // This requires that whenever creating a new class instance, the parameters MUST be passed at the same time. Therefore:

}

let john = new Employee(1, “John”, “Highway 71”);

### Access Modifiers and Inheritance

class Employee {

#id: number; // Equivalent to ‘private id” - To be accessed from only within the class

protected name: string; // To be accessed from within the class as well as its children.

address: string;

constructor(id: number, name: string, address: string) {

this.#id = id;

this.name = name;

this.address = address;

} // This requires that whenever creating a new class instance, the parameters MUST be passed at the same time. Therefore:

getNameAndAddress() : string { // The keyword “function” is not necessary

return `${this.name} stays at ${this.address}`;

}

Static getEmployeeCount() : number{ // “static” means that the function can be

Return 50; accessed by invoking the class name (without having to create a new class instance).

}

}

let john = new Employee(1, “John”, “Highway 71”);

john. // Only provides *“john.address”* and *“john.getNameAndAddress”* but not *“john.id”* nor *“john.name”.*

*Employee.getEmployeeCount() // Returns 50.*

class Manager extends Employee{

constructor(id: number, name: string, address: string) {

super(number, name, address); // Required if extending another class.

}

}

let mike = new Manager(2,”Mike”,”Amboseli court”);

mike.getNameAndAddress() returns “Mike stays at Amboseli court”.

### Getters and Setters

class Employee {

#id: number;

constructor(id: number, name: string, address: string) {

this.#id = id;

this.name = name;

this.address = address;

}

Get empId() : number {

return this.#id; // Recall that “id” is a private property. This allows access to the same from outside the class.

}

Set empId(id: number) {

this.#id = id; // This allows the private “id” property to be modified from outside the class.

}

}

let john = new Employee(1,”John”,”Thika”); // The class instance is created via the constructor.

console.log(john.empId()); // Returns “1” even though outside the class.

John.empId(10); // sets the #id of john to 10.

## Interfaces

Interface User {

name: string;

age?: number; // Optional parameter

id: number;

email: string;

}

let user: User = {name: ”John”, id: 1, email: “test@gmail.com”}

interface Employees extends User { // Interfaces and classes cannot have the same name.

salary: string;

}

let employee = Employees = {name: ”John”, id: 1, email: [test@gmail.com](mailto:test@gmail.com), salary:40000}

### Interface Implementation

You can define a function inside an interface and implement/describe it within a class. You will, however, have to export the interface (anything that to be used outside of the file from which it is defined has to be exported) and import the same from within the class file. Also, interfaces are defined outside of classes.

export Interface User {

name: string;

age?: number; // Optional parameter

id: number;

email: string;

}

export Interface Login {

Login(): User; // Function called “Login” that returns a value of type User”

}

From within the class file:

import { User, Login } from ‘./interface.ts’ // Or whatever the name of the file

import \* as UserLogin from ‘./interface.ts’ // Similar to the above but discouraged.

class Employee {

constructor() {

.

.

}

Login(): User { // If the second import statement is used, then this “Login” function implementation should be declared as “Login(): UserLogin.Login {}”

return {name: ”John”, age: 20, id:1, email: “test2@gmail.com”}

}

}

# Angular

## Angular Setup And Installation

*npm I @angular/cli –g* – Install angular globally.

*npm install -g @angular/cli@14.2.8* - To install a specific version of angular (14 in this case) and globally

*ng version* – to check for the version of angular and node installed.

*node -v* to check for the version of node installed

*ng new workspaceName --minimal* – To create an empty workspace (no projects/applications)

*ng new workspaceName –prefix=somePrefix* – To create an non-empty workspace and change the “prefix” value in the “angular.json” file to “somePrefix” so that selectors can be as “some-refix-root” rather than the default “app-root”, for example.

*npm i* – To install the packages into the current workspace specified in the package.json file.

*npm audit fix* --force – To fix the vulnerabilities after npm install.

*ng generate/g app projectName* – To create the angular application/project. (Use when the workspace is empty.)

*npx browserslist* – Returns all the browsers your project will be supported on.

*ng serve -o* – Compiles the project and renders it on watch mode, i.e, listens for any changes in the source code and recompiles after the same.

*ng generate/g component/c someComponent* – Generates a component within the “app” folder if not inside some other component, otherwise, it creates it within the component in focus within the terminal (angular CLI).

*ng add @ng-bootstrap/ng-bootstrap* – To install bootstrap styling framework to your project.

*ng add @angular/material* – To install angular material to your project

*nvm –v* – To check of “node version manager” installed.

*nvm list* – To check the versions available on your machine

*nvm use 18.17.1* – To change the version of node being used in the project to a specific version.

### Adding Styling Frameworks to your project

1. Import the file in the ”styles.scss” file. E.g.

*@import '../node\_modules/bootstrap/dist/css/bootstrap.min.css';*

1. Add the path of the styling file (as above) to the angular.json file within the “projects.architect.build.styles” array as:

*"styles": [*

*"src/styles.scss",*

*"node\_modules/bootstrap/dist/css/bootstrap.min.css"*

*]*

Changes made outside the src folder are not watched. You will have to re-serve before the changes are reflected.

Any components, directives and pipes need to be registered in the declarations array of the AppModule class/ root @NgModule.

## Component Syntax

### component attributes

1. Selector – custom html tag to render a component within another.
2. templateUrl – html file for the component
3. template: - inline html markup within the .ts file
4. stylesUrls: css, scss, sass styling files for the component.
5. styles: inline styling for the component

### Binding Syntax

There are three ways to display the data from the .ts file to the html file of the component:

#### Interpolation

Use {{variableName}} to bind the variable to the html tag. E.g *“<h1>Welcome to {{hotelName}}</h1>*”

#### Property binding

Html tags have properties. Eg. Divs have “innerText”. You can bind the ts variables to this tag properties. E.g if numberOfRooms = 10;

*“<div [innerText]=”numberOfRooms” ></div>*”

#### Event binding

Use the banana syntax to assign event listeners to html elements. Eg:

“*<button (click)="toggle()">Toggle</button>*”

### Directives

There two kinds:

1. Structural directives – have an asterisk (\*) e.g. \*ngIf, \*ngFor, \*ngSwitch. These can modify the DOM (add or remove elements from the DOM)
2. Attribute directives – Do not have an asterisk. E.g ngClass, ngStyle. These cannot modify the DOM.

#### Structural Directives

Preceded by an asterisk \*.

1. \*ngIf – Adds elements to the DOM when some condition evaluates to true. This is different from the “hidden” attribute because that may hide elements, but these hidden elements are still in the DOM, being less secure and being more computationally intensive. E.g.

“*<div \*ngIf=”rooms.availableRooms > 0”> Rooms List</div>*”

1. \*ngFor – Loops through the elements in an array, rendering each one. It has some other attributes: even, odd, index. E.g

*<tr \*ngFor=”let room of roomsList; let e = even; let o = odd; let I = index” ></tr>*

1. \*ngSwitch – Similar to the switch statement in regular programming. E.g:

*<div [ngSwitch]=”role”>*

*<div \*ngSwitchCase=”’User’”>Welcome User</div>*

*<div \*ngSwitchCase=”Admin”>*

*<hinv-rooms></hinv-rooms >*

*</div>*

*<div \*ngSwitchDefault>You are not authorized to view this page</div>*

*</div>*

#### Attribute Directives

Surrounded by square brackets [].

1. ngClass – Used to add a class name to a html element when a certain condition is met. E.g.

*<tr [ngClass]=”e: “even”: “odd””>*

*</tr>*

1. ngStyle - Used to add a css/scss style to a html element when a certain condition is met. E.g.

*<tr [ngStyle]=”e ? “even”: “odd””>*

*</tr>*

### Pipes

Used for data transformation. E.g. converting dates from one format to another.

Built in Pipes:

1. DatePipe - E.g. <td>{{room.checkintime | date}}</td>
2. CurrencyPipe: E.g. <th>{{ room.price | currency : 'JPY'}}</th>
3. UpperCasePipe: E.g. <th>{{ room.ammenities | uppercase}}</th>
4. LowerCasePipe: E.g. <th>{{ room.ammenities | lowercase}}</th>
5. TitleCasePipe: E.g. <th>{{ room.ammenities | titlecase}}</th>
6. DecimalPipe: <td>{{ room.rating | number : '2.1-2'}}</td>
7. PercentagePipe E.g. <td>{{ i+1 | percent}}</td>         1 is 100%
8. JsonPipe: E.g. {{roomList | json}}
9. SlicePipe: <tr \*ngFor="let room of roomsList | slice : 0:2; let e = even; let o = odd; let i = index" [ngClass]=" e ? 'even' : 'odd' " [ngStyle]="{'padding' : e ? '100px' : '200px'}">

To Be Discussed:

1. AsyncPipe
2. Custom Pipe

### Component Communication

Ways of Component Communication

#### Using @Input and @Output

Parent Component

*<div \*ngIf="rooms?.availableRooms && (rooms.availableRooms ?? 0) > 0"> <!--Nullish coalescing and optional pairing-->*

*<hinv-rooms-list [roomsList]="roomsList" (selectedRoom)="selectRoom($event)"></hinv-rooms-list>*

*</div>*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*selectedRoom!: roomList;*

*selectRoom(room: roomList) {*

*this.selectedRoom = room;*

*}*

Child Component

*<tr \*ngFor="let room of roomsList let e = even; let o = odd; let i = index" [ngClass]=" e ? 'even' : 'odd' " [ngStyle]="{'padding' : e ? '100px' : '200px'}">*

*<td><button class="btn btn-primary" (click)="selectRoom(room)">Select</button></td>*

*</tr>*

*\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_*

*@Input() roomsList?: roomList[];*

*@Output() selectedRoom = new EventEmitter<roomList>();*

*selectRoom(room: roomList) {*

*this.selectedRoom.emit(room);*

*}*

#### Using @ViewChild, @ViewChildren and @ContentChild

##### @ViewChild

Takes the first instance of the child and renders it on the parent.

@ViewChild(HeaderComponent { static : false }) header!: HeaderComponent;   // Placed in the parent’s .ts file.  "HeaderComponent" is the class name of the child component

//{ static: false} is the default - means that the child component is initialised during the ngInit of the parent

<hinv-header></hinv-header>     <!-- Placed in the parents HTML file -->

<hinv-header></hinv-header>     <!—This second one does not render -->

##### ng-template

This is a tag that will not render anything on its own. It is a placeholder. It is used to dynamically load other components at runtime using @ViewChild, i.e. rather than using the child’s selector tag, such “hinv-header”, and consequently rendering the component by default, this rendering is delayed until explicitly specified in the parent’s ts file.

<div #nametemplate> <!-- #nametemplate is a template reference using which the html tag can be accessed from the .ts file -->

</div>

 <div>

    <!-- #user is also template reference using which the html tag can be accessed from the .ts file -->

    <ng-template #user>

    </ng-template>

 </div>

@ViewChild('user', { read: ViewContainerRef }) vcr! : ViewContainerRef;

@ViewChild('nametemplate', {static: true}) elementName!: ElementRef;

  ngOnInit(): void {

    this.elementName.nativeElement.innerHTML = 'HELLO';

  }

  ngAfterViewInit(): void {       // Static is false by default hence the "ngAfterViewInit"

    const componentRef = this.vcr.createComponent(RoomsComponent);

    componentRef.instance.numberOfRooms = 1000;

  }

##### @ViewChildren

Used when you want to do different things with the different instances of the child class.

@ViewChildren(RoomsListComponent) roomslistComponent!: QueryList<RoomsListComponent>

  // For ViewChildren, the static property is always "false" and cannot be changed.

<hinv-rooms-list></hinv-rooms-list>

<hinv-rooms-list></hinv-rooms-list>

<hinv-rooms-list></hinv-rooms-list>

<!--Renders all three -->

##### @ContentChild

Used when wanting to render actual content but whose structure is defined, like a template. E.g in the “FAQs” section of a website where the template can be the same, with only the questions and answers changing.

* Create a component that will be a container
* Within this container’s .ts file, specify the components it is going to take within itself.

@Component({

  selector: 'hinv-container',

  templateUrl: './container.component.html',

  styleUrls: ['./container.component.scss'],

  // providers: [RoomsServicesService]

})

export class ContainerComponent implements OnInit, AfterContentInit {

  @ContentChild(EmployeeComponent) employee!: EmployeeComponent;

  @ContentChild(RoomsComponent) room!: RoomsComponent;

* Within the container’s .html file, specify the order in which the sub-components will be ordered.

<ng-content select="hinv-employee"></ng-content>

<ng-content></ng-content>

<ng-content select="hinv-rooms"></ng-content>

<!--This order of tags is what will be followed when rendering in spite of the order of tags when this container is being called from within another component-->

* Within another component, say app, pass the container to the template.

 <hinv-container>

    <hinv-rooms></hinv-rooms>

    <hinv-employee></hinv-employee>

    <h1>This is ng-content</h1>     <!--This doesn't work with other tags-->

 </hinv-container>

#### Using Services

Change Detection\*

### Lifecycle Hooks

These are methods called, for every component, as a component goes through the various stages of being loaded on the user interface.

#### ngOnInit

Called once, after the component and its input properties have been initialized

This is called after the component is initialized (by the constructor – which should only be modified to inject some services/ should not contain any other code otherwise)

ngOnInit(): void {

    this.roomsList = [{

      roomId: 1,

      roomType: "Deluxe Room",

      ammenities: "Kitchen, Washing machine, TV",

      image: "",

      price: 2000,

      checkInTime: new Date('28-Feb-2024'),

      checkOutTime: new Date('5-Mar-2024'),

      rating: 12.345,

    }];

    this.title = 'Rooms List';

    this.numberOfRooms = this.roomsList.length;

  }

#### ngOnDestroy

This is called whenever the specific component is removed from the DOM. E.g when using \*ngIf and the condition evaluates to false.

It is used when doing memory management

<div \*ngIf="hideRooms"> <!-- To test the ngOnDestroy of <hinv-rooms-list> -->

    <hinv-rooms-list [roomsList]="roomsList" [title]="title" (selectedRoom)="selectRoom($event)"></hinv-rooms-list>

</div>

// This is implemented on hinv-rooms-list

  ngOnDestroy(): void {

    console.log('The component is being destroyed');

  }

#### ngOnChanges

Called whenever an input property of the component changes.

Receives a SimpleChanges object that contains the previous and current values of the changed properties.

Only works with classes that implement @Input (children classes). E.g.

@Component({

  selector: 'hinv-rooms-list',

  templateUrl: './rooms-list.component.html',

  styleUrls: ['./rooms-list.component.scss'],

  changeDetection: ChangeDetectionStrategy.OnPush

})

export class RoomsListComponent implements OnInit, OnChanges {

  @Input() roomsList?: roomList[];

  @Input() title!: string;

  @Output() selectedRoom = new EventEmitter<roomList>();

  constructor() { }

  ngOnChanges(changes: SimpleChanges): void {

    if(changes['roomsList']['firstChange'] != true) {

      this.title = 'Updated ' + this.title;

    }

  }

#### ngDoCheck

ngDoCheck is called during every change detection cycle, which can occur frequently in Angular applications. This is triggered whenever any event happens in the application. Not just when the view’s “@Input”s are being checked for changes.

export class RoomsComponent implements OnInit, DoCheck {

ngDoCheck(): void {

    console.log("Some event has been triggered")

  }

#### ngAfterViewInit

Called once, after the component's view and its children's views have been initialized.

This is used with the @ViewChild component communication method such that it is called, by the parent/calling component, when the child component has been fully initialized.

@ViewChild(HeaderComponent, { static: false}) header!: HeaderComponent;   // Placed in the parents .ts file.  "HeaderComponent" is the class name of the child component

  //{ static: false} is the default - means that the child component is initialised during the ngInit of the parent

  ngAfterViewInit(): void {

    console.log(this.header);

  }

#### ngAfterViewChecked

This hook is called after the view and its children have been checked for changes. It is invoked every time change detection runs after the initial check.

#### ngAfterContentInit

Works together with the @ContentChild decorator.

#### ngAfterContentChecked

## Dependency Injection [Providers]

### Class-Based Providers [Services]

A service is a reusable class within which all the business logic, e.g., getting data from and api or posting the same, related to some particular class(es) is placed. This logic can then be accessed from within the (various) component(s) using dependency injection.

#### Creating a Service

To create a service:

* Navigate to the path where you want to create it from the terminal.
* Run “ng generate service <serviceName>” or “ng g s serviceName>”

import { Injectable } from '@angular/core';

import { roomList } from '../rooms';

@Injectable({

providedIn: 'root'

// Takes care of:

  // - Implicitly Adding this "RoomsServicesService" to the "Providers" section of the app.module.ts file.

// - Create a single instance of the class even if the service is injected into multiple components.

// providedIn: 'any'

  // - Also creates a single instance of the class even if the service is injected into multiple components and modules. However, if the modules are lazily loaded, it creates a separate instance for the same.

// - Also overwrites any injection token that is initialized in app.module for each module that specifies a different value for the injection token. Otherwise, if provided in = "root", the value of the injection token does not change.

})

export class RoomsServicesService {

  constructor() { }

  roomsList: roomList[] = [{

    roomId: 1,

    roomType: "Deluxe Room",

    ammenities: "Kitchen, Washing machine, TV",

    image: "",

    price: 2000,

    checkInTime: new Date('28-Feb-2024'),

    checkOutTime: new Date('5-Mar-2024'),

    rating: 12.345,

  }

  ];

  getRooms() {

    return this.roomsList;

  }

}

#### Injecting the Service to a Component

@Component({

  selector: 'hinv-rooms',

  templateUrl: './rooms.component.html',

  styleUrls: ['./rooms.component.scss'],

  providers: [RoomsServicesService]     // Creates a separate instance of the service for this component, separate from the globally available instance.

})

export class RoomsComponent implements OnInit, AfterViewInit, AfterViewChecked {

// The private limits the injected service to the .ts file. Public would make the same accessible even on the template (make it possible to manipulate using interpolation binding syntax) which is not a good practice.

constructor(private roomsService: RoomsServicesService) { }

  ngOnInit(): void {

    this.roomsList = this.roomsService.getRooms();

  }

#### Resolution Modifiers

##### @Self()

This one tells the component injecting a service to only look for the same from within the level of that component, not to look for the same from the parent, upto the root components.

For this reason, the component needs to import its own instance of the service class for the same to be accessible/ usable. Otherwise a “No provider for RoomsServicesService found in NodeInjector” error is retuned

@Component({

  selector: 'hinv-employee',

  templateUrl: './employee.component.html',

  styleUrls: ['./employee.component.scss'],

  providers: [ RoomsServicesService]

})

export class EmployeeComponent implements OnInit {

  employeeName: string = 'John';

  constructor(@Self() private roomsService: RoomsServicesService) { }

##### @SkipSelf()

This modifier instructs angular to skip the component within which this modifier is used when looking for the service in the dependency resolution tree.

##### @Optional()

This one is used when the service being injected is not provided in the root component.

import { Injectable } from '@angular/core';

@Injectable() // Missing { providedIn: root }

export class LoggerService {

  constructor() { }

  log(msg: string) {

    console.log(msg);

  }

}

export class HeaderComponent implements OnInit {

  title: string = '';

  constructor(@Optional() private loggerService: LoggerService) { }

  ngOnInit(): void {

    this.loggerService?.log("Header and Logger Linked");  // The log will not work because the service is not provided anywhere

  }

}

##### @Host()

This is used with container components such that all the children components rendered within this container do not go beyond this container when looking for their injected services within the dependency resolution tree.

This requires that the container define the “provider” in its @component decorator, and pass the service within the same.

@Component({

  selector: 'hinv-container',

  templateUrl: './container.component.html',

  styleUrls: ['./container.component.scss'],

  providers: [RoomsServicesService]

})

export class ContainerComponent implements OnInit, AfterContentInit {

  @ContentChild(EmployeeComponent) employee!: EmployeeComponent;

  @ContentChild(RoomsComponent) room!: RoomsComponent;

  constructor(@Host() private roomsService: RoomsServicesService) { }

<ng-content select="hinv-employee"></ng-content>

<ng-content></ng-content>

<ng-content select="hinv-rooms"></ng-content>

@Component({

  selector: 'hinv-employee',

  templateUrl: './employee.component.html',

  styleUrls: ['./employee.component.scss'],

})

export class EmployeeComponent implements OnInit {

  employeeName: string = 'John';

  constructor(private roomsService: RoomsServicesService) { }

  ngOnInit(): void {

  }

}

### Value-Based Providers

These are used to provide some constants that will be used throughout the project. E.g. as providers of api endpoints that can then be injected into other class-based providers (services).

#### Example of a custom api

* Create an object to hold the API, and declare it as a constant

export const environment = {

  production: false,

  apiEndPoint: 'http://localhost:8080/api/v1'

};

* Create a file to store the injection tokens.

import { InjectionToken } from "@angular/core";

import { AppConfig } from "./appconfig.interface";

import { environment } from "src/environments/environment";

export const APP\_SERVICE\_CONFIG = new InjectionToken<AppConfig>('app.config'); // 'app.config' is an arbitrary name.

export const APP\_CONFIG: AppConfig = {

    apiEndpoint: environment.apiEndPoint

}

export interface AppConfig {

    apiEndpoint: string;

}

* Add the injection tokens to ‘app.module.ts’ under the ‘providers’ section.

providers: [

    {

      provide: APP\_SERVICE\_CONFIG,

      useValue: APP\_CONFIG

    }

  ],

* Import the token from within the class (component/service) you want to use it.

export class RoomsServicesService {

  constructor(@Inject(APP\_SERVICE\_CONFIG) private config: AppConfig) {

    console.log("Rooms service is initialised");

    // console.log(environment.apiEndPoint);

    console.log(this.config.apiEndpoint)

  }

#### Example of a browser api

export const localStorageToken = new InjectionToken<any>('local storage', {

    providedIn: 'root',

    factory() {

        return localStorage;

    },

});

constructor(@Inject(localStorageToken) private localStorage: any) {

  }

  ngOnInit(): void {

    // this.elementName.nativeElement.innerHTML = 'HELLO';

    console.log(this.localStorage);

    this.localStorage.setItem('name', 'hilton hotel');

    setTimeout(() => {

      // Remove the token from localStorage after the specified time

      this.localStorage.removeItem('name');

  }, 5000);

  }

## Angular Http and Observables

### Http

*Monitored in the browser under the “network” tab of the inspector*

#### Procedure for Setting up Http in your project

* Set up the HttpClientModule in app.module.ts under “imports”

import { HttpClientModule } from '@angular/common/http'

imports: [

    BrowserModule,

    AppRoutingModule,

    FormsModule,

    HttpClientModule

  ],

* Set up a proxy

Create a file called “proxy.conf.json” under the “src” directory and paste:

{

    "/api": {

        "target": "http://localhost:3000",

        "secure": false

    }

}

Within the ”angular.json” file, under “serve”, paste "proxyConfig": "src/proxy.conf.json". E.g:

"serve": {

          "builder": "@angular-devkit/build-angular:dev-server",

          "configurations": {

            "production": {

              "browserTarget": "hotelInventoryApp:build:production"

            },

            "development": {

              "browserTarget": "hotelInventoryApp:build:development",

              "proxyConfig": "src/proxy.conf.json"

            }

          },

          "defaultConfiguration": "development"

        }

* Import HttpClient and add it to your constructor for the service where you want to make the API calls.

import { HttpClient } from '@angular/common/http';

@Injectable({

  providedIn: 'root'

})

export class RoomsServicesService {

  constructor(private httpClient: HttpClient) {

  }

* Call the endpoint and transform the data into the desired format from within your service(s)

  getRooms() {

    return this.http.get<roomList[]>('/api/rooms');

  }

* From within your component(s), get the data from the service method responsible and subscribe to the rxjs stream of data.

  ngOnInit(): void {

    this.roomsService.getRooms().subscribe((rooms) => {

      this.roomsList = rooms;

    });

  }

### Observables

#### “Observer” (Stream Tracker) methods

Create a new stream as an object of type “Observable”. This stream provides a tracker called an “observer” that tracks the stream of data.

This observer can call three methods:

* next() – Called when new data is added is added to the stream of data.
* complete() – called when the stream of data is finished.
* error() – called where there is an error.

import { Observable } from 'rxjs';

export class RoomsComponent implements OnInit, AfterViewInit, AfterViewChecked {

stream = new Observable((observer) => {

    observer.next('user1');

    observer.next('user2');

    observer.next('user3');

    observer.next('user4');

    observer.complete();

    observer.error('error');

});

Once these rxjs observables are subscribed to, they return three kinds of responses:

* next
* complete
* error

ngOnInit(): void {

this.stream.subscribe({

      next: (value) => console.log(value),

      complete: () => console.log('complete'),

      error: (err) => console.log(err)

    });

}

### Performing CRUD operations with Endpoints

Consider the following endpoints:

import { Controller, Get, Post, Body, Patch, Param, Delete, Put } from '@nestjs/common';

import { RoomsService } from './rooms.service';

import { CreateRoomDto } from './dto/create-room.dto';

import { UpdateRoomDto } from './dto/update-room.dto';

@Controller('api/Rooms')

export class RoomsController {

  constructor(private readonly roomsService: RoomsService) {}

  @Post()

  create(@Body() createRoomDto: CreateRoomDto) {

    return this.roomsService.create(createRoomDto);

  }

  @Get()

  findAll() {

    return this.roomsService.findAll();

  }

  @Get(':id')

  findOne(@Param('id') id: string) {

    return this.roomsService.findOne(id);

}

}

With the following implementations:

import { Injectable } from '@nestjs/common';

import { CreateRoomDto } from './dto/create-room.dto';

import { UpdateRoomDto } from './dto/update-room.dto';

import { Room } from './entities/room.entity';

import { v4 } from 'uuid';

@Injectable()

export class RoomsService {

  rooms: Room[] = [

    {

      roomNumber: '1',

      roomType: 'Deluxe Room',

      amenities: 'Air Conditioner, Free Wi-Fi, TV, Bathroom, Kitchen',

      price: 500,

      photos:

        'https://images.unsplash.com/photo-1518791841217-8f162f1e1131?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=800&q=60',

      checkinTime: new Date('11-Nov-2021'),

      checkoutTime: new Date('12-Nov-2021'),

      rating: 4.5,

    },

    {

      roomNumber: '2',

      roomType: 'Deluxe Room',

      amenities: 'Air Conditioner, Free Wi-Fi, TV, Bathroom, Kitchen',

      price: 1000,

      photos:

        'https://images.unsplash.com/photo-1518791841217-8f162f1e1131?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=800&q=60',

      checkinTime: new Date('11-Nov-2021'),

      checkoutTime: new Date('12-Nov-2021'),

      rating: 3.45654,

    },

    {

      roomNumber: '3',

      roomType: 'Private Suite',

      amenities: 'Air Conditioner, Free Wi-Fi, TV, Bathroom, Kitchen',

      price: 15000,

      photos:

        'https://images.unsplash.com/photo-1518791841217-8f162f1e1131?ixlib=rb-1.2.1&ixid=eyJhcHBfaWQiOjEyMDd9&auto=format&fit=crop&w=800&q=60',

      checkinTime: new Date('11-Nov-2021'),

      checkoutTime: new Date('12-Nov-2021'),

      rating: 2.6,

    },

  ];

  create(createRoomDto: CreateRoomDto) {

    console.log(createRoomDto);

    createRoomDto.roomNumber = v4();

    console.log(createRoomDto);

    this.rooms = [...this.rooms, createRoomDto];

    console.log(this.rooms);

    return this.rooms;

  }

  findAll() {

    return this.rooms;

  }

  findOne(id: string) {

    return this.rooms.find((room) => room.roomNumber === id);

  }

}

#### GET

Within the calling service

getRooms() {

    return this.http.get<roomList[]>('/api/rooms');

  }

Within the component where it’s used:

ngOnInit(): void {

    this.roomsService.getRooms().subscribe((rooms) => {

      this.roomsList = rooms;

    });

#### POST

Within the calling service:

addRoom(room: object) {

    return this.http.post<roomList[]>('/api/rooms', room)

  }

Within the component where it’s used:

addRoom() {

    const room = {

      // roomId: '1',

      roomType: "Deluxe Room",

      ammenities: "Kitchen, Washing machine, TV",

      image: "",

      price: 2000,

      checkInTime: new Date('28-Feb-2024'),

      checkOutTime: new Date('5-Mar-2024'),

      rating: 12.345,

    }

    // this.roomsList.push(room);     - This violates the "immutability" property in state meangement.

    // this.roomsList = [...this.roomsList, room];     // This creates a new object compared to the one above that modifies the existing one.

    this.roomsService.addRoom(room).subscribe((rooms) => {

      this.roomsList = rooms;

      this.numberOfRooms = this.roomsList.length;

    });

  }

#### PUT

Within the calling service:

editRoom(room: roomList) {

    return this.http.put<roomList[]>(`/api/rooms/${room.roomId}`, room);

  }

Within the component where it’s used:

HTML

<tr \*ngFor="let room of roomsList; let e = even; let o = odd; let i = index" [ngClass]=" e ? 'even' : 'odd' " [ngStyle]="{'padding' : e ? '100px' : '200px'}">

   <td><button class="btn btn-primary" (click)="editRoom(room)">Edit</button></td>

</tr>

TS

editRoom(room: roomList) {

    const edittedRoom = {

      roomId: room.roomId,

      roomType: "Master Suite",

      ammenities: "Jacuzzi, Kitchen, Washing machine, TV",

      image: "",

      price: 15000,

      checkInTime: new Date('28-Feb-2024'),

      checkOutTime: new Date('5-Mar-2024'),

      rating: 50,

    }

    // this.roomsList.push(room);     - This violates the "immutability" property in state meangement.

    // this.roomsList = [...this.roomsList, room];     // This creates a new object compared to the one above that modifies the existing one.

    this.roomsService.editRoom(edittedRoom).subscribe({

      next: (rooms) => this.roomsList = rooms

    });

  }

Within the backend’s implementation:

Controller

@Put(':id')

  update(@Param('id') id: string, @Body() updateRoomDto: UpdateRoomDto) {

    return this.roomsService.update(id, updateRoomDto);

  }

Service

update(id: string, updateRoomDto: UpdateRoomDto) {

    let indexToUpdate = this.rooms.findIndex((room) => { return room.roomId === updateRoomDto.roomId });

    console.log(indexToUpdate);

    this.rooms[indexToUpdate].roomType = updateRoomDto.roomType;

    this.rooms[indexToUpdate].amenities = updateRoomDto.amenities;

    this.rooms[indexToUpdate].price = updateRoomDto.price;

    this.rooms[indexToUpdate].photos = updateRoomDto.photos;

    this.rooms[indexToUpdate].checkinTime = updateRoomDto.checkinTime;

    this.rooms[indexToUpdate].checkoutTime = updateRoomDto.checkoutTime;

    this.rooms[indexToUpdate].rating = updateRoomDto.rating;

    return this.rooms;

  }

#### DELETE

Within the calling service:

 deleteRoom(id: string) {

    return this.http.delete<roomList[]>(`/api/rooms/${id}`);

  }

Within the component where it’s used:

HTML

<tr \*ngFor="let room of roomsList; let e = even; let o = odd; let i = index" [ngClass]=" e ? 'even' : 'odd' " [ngStyle]="{'padding' : e ? '100px' : '200px'}">

   <td><button class="btn btn-primary" (click)="deleteRoom(room)">Delete</button></td>

</tr>

TS

deleteRoom(room: roomList) {

    this.roomsService.deleteRoom(room.roomId ?? "").subscribe((rooms) => this.roomsList = rooms);

}

Within the backend’s implementation:

Controller

@Delete(':id')

  remove(@Param('id') id: string) {

    return this.roomsService.remove(id);

  }

Service

remove(id: string) {

    // return this.rooms.filter((room) => room.roomId !== id);

    let indexToDelete = this.rooms.findIndex((room) => { return room.roomId === id });

    this.rooms.splice(indexToDelete,1);

    return this.rooms;

}

#### REQUEST

This one provides a lot more information about the http requests and responses. Mainly it returns some information at various stages of the Http communications.

Within the calling service:

getPhotos() {

    const request = new HttpRequest('GET', `https://jsonplaceholder.typicode.com/photos`, {

      // reportError: true,

      reportProgress: true

    });

    return this.http.request(request);

  }

Within the component:

ngOnInit(): void {

    this.roomsService.getPhotos().subscribe((event) => {

      switch(event.type) {

        case HttpEventType.Sent: {      // The connection to the endpoint is being made

          console.log('Request has been made');

          break;

        }

        case HttpEventType.ResponseHeader: {      // The http request header has been made and sent

          console.log('Request successfully made');

          break;

        }

        case HttpEventType.DownloadProgress: {      // Data is being received from the endpoint

          this.totalBytesLoaded += event.loaded;

          console.log(this.totalBytesLoaded/(1024\*1024) + ' megabytes loaded');

          break;

        }

        case HttpEventType.Response: {      // The data transfer is finished and a response header has been returned.

          console.log(event.body);

          break;

        }

      }

    });

}

### Unsubscribing from Rxjs streams

#### Using “Subscription” Property

Subscribing to data streams is very computationally expensive and therefore unsubscribing from these streams after a component is destroyed is imperative

“subscription” property

subscription!: Subscription;

ngOnInit(): void {

    // this.roomsService.getRooms().subscribe((rooms) => {

    //   this.roomsList = rooms;

    // });

    this.subscription = this.roomsService.getRooms$.subscribe((rooms) => {

      this.roomsList = rooms;

    });

}

ngOnDestroy(): void {

    if(this.subscription) {

      this.subscription.unsubscribe();

    }

  }

#### Async Pipe

It is a good practice to use this “async” pipe to render data rather than subscribing to the same, with the potential of forgetting to unsubscribe. Async takes care of the unsubscription on its own.

Assign the http response to an observable.

// The "$" is used to denote that the variable holds an observable

  rooms$ = this.roomsService.getRooms$;

Use the “async” pipe to convert the observable into data

<!-- The use of "as" allows for this piping to happen once. Otherwise it would also have to happen at [roomsList]="rooms$ | async"-->

<div \*ngIf="rooms$ | async as rooms">

    <hinv-rooms-list [roomsList]="rooms" [title]="title" (selectedRoom)="selectRoom($event)"></hinv-rooms-list>

</div>

Prepare the child component to accept the data or a null if unavailable.

@Input() roomsList?: roomList[] | null;

### Rxjs Operators

There are many.

#### shareReplay

This operator is used to cache the response from the endpoint being called so that, if there are multiple request being made to the same API and yet the data there is not changing, data from this cache is what is used and the call is therefore on made once.

// Once you subscribe to a stream of data you cannot make any modifications. That is why we use the pipe operator.

  getRooms$ = this.http.get<roomList[]>('/api/rooms').pipe(

    shareReplay(1)

  );

ngOnInit(): void {

    // this.roomsService.getRooms().subscribe((rooms) => {

    //   this.roomsList = rooms;

    // });

    this.roomsService.getRooms$.subscribe((rooms) => {

      this.roomsList = rooms;

    });

}

#### catchError

error$ = new Subject<string>();

getError$ = this.error$.asObservable();

getRooms$ = this.http.get<roomList[]>('/api/room').pipe(

    shareReplay(1),

    catchError((err) => {

    // console.log(err);

    this.error$.next(err.message);    // If "next()" is called from within the component, ChangeDetection is triggered once again so avoid using it within components.

    return of([]);

  })

);

Get the error from within the component.

getError$ = this.roomsService.getError$;

For development purposes, display the error in the template:

<h1>{{ getError$ | async}}</h1>

#### Map

roomsCount$ = this.roomsService.getRooms$.pipe(

    map((rooms) => rooms.length)

  );

<h1>

    <!-- {{ getError$ | async}} -->

    Number of rooms: {{ roomsCount$ | async }}

</h1>

**Using Rxjs Map Operators**

Consider the posting service:

import { HttpClient } from '@angular/common/http';

import { Injectable } from '@angular/core';

@Injectable({

  providedIn: 'root'

})

export class BookingService {

  constructor(private http: HttpClient) { }

  bookRoom(booking : any) {

    return this.http.post('https://jsonplaceholder.typicode.com/posts', booking)

  }

}

And the component implementation:

ngOnInit(): void {

    // console.log("Booking initiated");

    this.bookingForm = this.fb.group({

      // roomId: new FormControl({value: "2", disabled: true}),

      roomId: [{ value: "2", disabled: true }],

      // bookingId: new FormControl(''), Same thing as below

      bookingId: [''],

      guestEmail: ['', [Validators.email, Validators.required]],

      .

.

.

tnc: new FormControl(false, {validators: [Validators.requiredTrue]})

    }

    this.bookingForm.valueChanges.pipe(

      // mergeMap((data) => {

// switchMap((data) => {

exhaustMap((data) => {

        return this.roomBookingService.bookRoom(data);

      })).subscribe((data) => {

        console.log(data)

      });

  }

##### mergeMap

This operator is used to merge the emissions of multiple observables into a single observable stream. It is often used when dealing with scenarios where multiple asynchronous operations need to be executed concurrently, and the order of their completion doesn't matter. It subscribes to all inner observables and emits their values in the order they are emitted.

##### switchMap

This operator is used to cancel the previous inner observable subscription and switch to a new one whenever a new value is emitted by the source observable. It is useful when you want to ensure that only the latest inner observable is being processed, particularly in scenarios such as typeaheads or search functionalities, where you may want to discard previous pending requests when a new one is made.

##### exhaustMap

This operator is used to ignore new source values while the current inner observable is still processing. It is often used in scenarios where you want to ensure that certain actions are not triggered multiple times concurrently, for example, preventing multiple HTTP requests from being made simultaneously until the previous one completes.

### Modifying HttpRequests

#### HttpHeaders (Adding Headers to HttpRequests)

Create a “HttpHeaders” object

headers = new HttpHeaders({'token': '786876'}).append('token2','ghgjhgkjhk').append('token3','ghgjhgkjhk');

Pass this object to the API call.

getRooms$ = this.http.get<roomList[]>('/api/rooms', {

    headers: this.headers,

)

return this.http.post<roomList[]>('/api/rooms', room, {

      headers: this.headers,

})

#### HttpInterceptors

These lie between the client and server and can be used to modify the http request and/or response in a similar manner as above but even more, e.g., when wanting to append the timestamp/token etc. to every request.

* Create the HttpInterceptors, (within the “app” folder):

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app>* ***ng g interceptor request***

* Add the interceptor to App.module.ts, under “providers”:

providers: [

    {

      provide: APP\_SERVICE\_CONFIG,

      useValue: APP\_CONFIG

    },

    {

      provide: HTTP\_INTERCEPTORS,

      useClass: RequestInterceptor,

      multi: true

    }

],

* Clone the request (because you cannot modify the original one), and modify it as you want.

import { Injectable } from '@angular/core';

import {

  HttpRequest,

  HttpHandler,

  HttpEvent,

  HttpInterceptor,

  HttpHeaders

} from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class RequestInterceptor implements HttpInterceptor {

  constructor() { }

  intercept(request: HttpRequest<unknown>, next: HttpHandler): Observable<HttpEvent<unknown>> {

    if (request.method === 'POST') {

      const newRequest = request.clone(

        {

          headers: new HttpHeaders({ 'token': '786876' }).append('token2', 'ghgjhgkjhk').append('token3', 'ghgjhgkjhk') // This header will be appended to every httprequest of type "POST"

        }

      );

      return next.handle(newRequest);

    } else {

      return next.handle(request);

    }

  }

}

### App Initialiser

This is used when you want to load some data, say from local storage or an API, before the app is initialized (before the first page is initialized).

* Create a service, say “init”

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app>* ***ng generate service init***

* Within the service, make a http call

import { HttpClient } from '@angular/common/http';

import { Injectable } from '@angular/core';

import { tap } from 'rxjs';

@Injectable({

  providedIn: 'root'

})

export class InitService {

  config: any;    // "any" does not need the '!:' to allow for a null value initialisation

  constructor(private http: HttpClient) { }

  init() {

    return this.http.get('./assets/config.json').pipe(tap((config) => this.config = config)); // Create the actually file and populate it with some data

  }

}

* Within “app.module.ts”, create a function (called a factory function) that takes the newly-created service class as a parameter, and returns the function created within the class. This should be done outside the “AppModule” class within “app.module.ts”

import { InitService } from './init.service';

function initFactory(initService: InitService) {

  return () => initService.init();

}

* Register the factory under “provide” within the app.module.ts’s “@NgModule” decorator.

{

      provide: APP\_INITIALIZER,

      useFactory: initFactory,

      deps: [InitService],

      multi: true

    }

* To test that the initialization was successful, in our example, for example,
  + check the network tab and see that the file being gotten was indeed gotten successfully
  + Inject the service into app.component and “console.log” the property being assigned the file gotten.

export class AppComponent implements AfterViewInit {

  constructor(

    private initService: InitService

  ) {

    console.log(initService.config);

  }

## Angular Router

### Configuring Custom Static Routes

If when creating the project you choose “Yes” to “Use routing”, then routing is configured for you in app.module.ts.

To set up custom routes,

* Go to “app.routing.model.ts” and add the same into the “Routes” array as:

const routes: Routes = [

  { path: 'rooms', component: RoomsComponent },

  { path: 'employees', component: EmployeeComponent },

{ path: 'input', component: InputCheckComponent },

{ path: '', redirectTo: '/rooms', pathMatch: 'full' }     // The default route

{ path: '\*\*', component: NotFoundComponent }     // Page 404 Not Found

];

* Go to app.component.ts and add a “<router-outlet>” tag as:

<router-outlet></router-outlet>

* Test the paths by typing them manually in the URL as:

*“http://localhost:4200/rooms”*

* Add anchor tags to the templates with “routerLink” attributes specifying the routes they will target

 <a [routerLink]="'rooms'">Rooms</a>

 <a [routerLink]="'employees'">Employees</a>

 <a [routerLink]="'input'">Inputs</a>

 <router-outlet></router-outlet>

*N/B: Switching Views Using RouterLinks Results in the component, being hidden, being destroyed*

### Dynamic Routes

This are routes that are configured from the components themselves rather than from “app-routing.module.ts”. E.g. To redirect to some page after selecting some item from a list of items, say, in an e commerce website.

Configure “app-routing.module.ts” to handle links that accept parameters:

*N/B: The parameterized route must be above the wildcard route*

{ path: 'rooms', component: RoomsComponent },

{ path: 'rooms/:id', component: RoomBookingComponent },     // For Dynamic Routes

{ path: '\*\*', component: NotFoundComponent },     // Page 404 Not Found

Configure the component(s) to target the dynamic route, passing the necessary argument(s) to the same.

<tr \*ngFor="let room of roomsList; let e = even; let o = odd; let i = index" [ngClass]=" e ? 'even' : 'odd' " [ngStyle]="{'padding' : e ? '100px' : '200px'}">

   <!-- <td><button class="btn btn-primary" (click)="selectRoom(room)">Select</button></td> -->

   <td><button class="btn btn-primary" [routerLink]="['/rooms', room.roomId]">Book</button></td>

        </tr>

Get the parameters from within the targeted view

// id: Number = 0;

  // id$: Observable<Number> = this.router.params.pipe(map((param) => param['id']));

  id$ = this.router.paramMap.pipe(map((param) => param.get('id')));     // "paramMap" allows you to get multiple parameters from the routerLink and handles the situation when the field passed does not exist in the routerLink

  constructor(private router: ActivatedRoute) { }

// ActivatedRoute is a service that comes in-built within angular

  ngOnInit(): void {

    // this.router.params.subscribe((param) => {

    //   this.id = param['id']);

    // });    - "Caution: Avoid Subscribing"

    // this.id = this.router.snapshot.params['id'];     "Caution: Snapshot value are not changed so using may result in issues if the view is not changed"

  }

For the template, render. E.g:

{{id$ | async}}

### Nested Routes

These are routes within other routes such that the inner routes are children routes.

To configure nested routes:

* Add the “children” parameter to the parent route

const routes: Routes = [

  { path: 'rooms',

    component: RoomsComponent,

    children: [

      { path: 'add-room', component: AddRoomsComponent }, // Place this route before the dynamic route otherwise that dynamic route will always be called.

      { path: ':id', component: RoomBookingComponent }

    ],     // For Dynamic Routes

  }

];

* Add the “<router-link></router-link>” tag to the template of the parent view so that the child view can be rendered next to/below/above the parent view, depending on the placement of the tag.

<div \*ngIf="rooms$ | async as rooms"> <!-- To test the ngOnDestroy of <hinv-rooms-list> -->

    <hinv-rooms-list [roomsList]="rooms" [title]="title" (selectedRoom)="selectRoom($event)"></hinv-rooms-list>

</div>

<router-outlet></router-outlet>

### “Router” Service (Triggering Navigation from one component to another after some function)

Sometimes you may want to navigate to a specific route after carrying out some function from within a component’s ts file:

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

import { Router } from '@angular/router';

@Component({

  selector: 'hinv-login',

  templateUrl: './login.component.html',

  styleUrls: ['./login.component.scss']

})

export class LoginComponent implements OnInit {

  constructor(

    private route: Router,

  ) { }

  ngOnInit(): void {

  }

  login() {

    if(this.loginService.login(this.email, this.password)) {

      // this.route.navigate(['rooms', 'add-room']);

      // this.route.navigateByUrl('/rooms/add-room');    // Same thing as above

      this.route.navigateByUrl('/rooms');    // Same thing as above

    }

  }

}

### Route Events

These are like the lifecycle hooks of routes, called during various stages of navigation.

They include:

* GuardsCheckEnd
* GuardsCheckStart
* NavigationCancel

#### NavigationStart

* NavigationError
* NavigationSkipped

#### NavigationEnd

* ResolveEnd
* ResolveStart
* RoutesRecognized

// This is done inside app.component.ts

constructor(

    @Inject(localStorageToken) private localStorage: any,

    private initService: InitService,

    private router: Router

  ) {

    console.log(initService.config);

  }

  ngOnInit(): void {

    this.router.events.subscribe((event) => {

      console.log(event);

    });

this.router.events.pipe(

      filter((event): event is NavigationStart => event instanceof NavigationStart)

    ).subscribe((event) => {

      console.log('Navigation Started');

    });

    this.router.events.pipe(

      filter((event): event is NavigationEnd => event instanceof NavigationEnd)

    ).subscribe((event) => {

      console.log('Navigation Completed');

    });

  }

### Route Guards

These, as the name suggests, are used to protect certain routes so that they are only usable if certain conditions are met.

To create a guard, run the command:

***ng generate guard <guardName>***

And then select the method(s) you want from the list that appears using space “ ” and enter.

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app\guards>* ***ng generate guard login***

*? Which interfaces would you like to implement? CanActivate, CanLoad*

*CREATE src/app/guards/login.guard.spec.ts (336 bytes)*

*CREATE src/app/guards/login.guard.ts (652 bytes)*

#### canActivate()

This is used to enable the routes configuredso that a user can navigate to them.

To implement the same, say “to activate all the routes after logging in”:

* Generate the guard and choose “canActivate” from the list of methods provided.
* Go to your …routing.ts file and add “canActivate[<guardName>]” to the routes you want to guard.

const routes: Routes = [

  { path: 'login', component: LoginComponent },

  { path: 'employees', component: EmployeeComponent, canActivate: [LoginGuard]},

  { path: 'input', component: InputCheckComponent, canActivate: [LoginGuard]},

  { path: 'search', component: SearchComponent, canActivate: [LoginGuard]},

  { path: 'rooms', loadChildren: () => import('./rooms/rooms.module').then((m) => m.RoomsModule), canActivate: [LoginGuard]},

  { path: '', redirectTo: '/login', pathMatch: 'full' },

  { path: 'booking', loadChildren: () => import('./booking/booking.module').then(m => m.BookingModule), canActivate: [LoginGuard]},     // The default route

  { path: '\*\*', component: NotFoundComponent },     // Page 404 Not Found

];

* Create a service for the login component and within it configure a Boolean flag that is set to true if the login criteria is met. E.g:

export class LoginService {

  isLoggedIn: boolean = false;

  constructor(

    private route: Router

  ) { }

  login(email: string, password: string): boolean {

    if(email === 'admin@gmail.com' && password === 'admin') {

      // this.route.navigate(['rooms', 'add-room']);

      // this.route.navigateByUrl('/rooms/add-room');    // Same thing as above

      this.isLoggedIn = true;

    }

    return this.isLoggedIn;

  }

}

* Call the service from the component and specify the navigation intended in the event the login criteria is met.

login() {

    if(this.loginService.login(this.email, this.password)) {

      // this.route.navigate(['rooms', 'add-room']);

      // this.route.navigateByUrl('/rooms/add-room');    // Same thing as above

      this.route.navigateByUrl('/rooms');    // Same thing as above

    }

  }

* Create a constructor and import the login service created from within the guard and configure the “canActivate” method to return true, based on the service’s method return, or to return a router navigation path in the event the service-method evaluates to false.

export class LoginGuard implements CanActivate, CanLoad {

  constructor(

    private loginService: LoginService,

    private route: Router

  ){

  }

  canActivate(

    route: ActivatedRouteSnapshot,

    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

// This isLoggedIn is reset to false every time the page is reloaded and so the page will be redirected to the login route every time this happens.

    // Also typing the url manully in the url bar will also redirect the page to the login route

    return this.loginService.isLoggedIn ? true : this.route.navigateByUrl('/login');

  }

}

#### canActivateChild()

This is similar to canActivate but used for children routes, i.e. routes configured within the “children” array of some parent route:

const routes: Routes = [

  // { path: 'rooms',

  { path: '',

    component: RoomsComponent,

    canActivateChild: [RoomsGuard],

    children: [

      { path: 'add-room', component: AddRoomsComponent }, // Place this route before the dynamic route otherwise that dynamic route will always be called.

      { path: ':id', component: RoomBookingComponent }

    ],     // For Dynamic Routes

  }

];

constructor(

    private loginService: LoginService

  ) {

  }

  canActivateChild(

    childRoute: ActivatedRouteSnapshot,

    state: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

    return this.loginService.isAdmin;

  }

#### canLoad()

This guard is used with lazily-loaded routes and modules such that a route is downloaded only when the guard evaluates to true.

canLoad(

    route: Route,

    segments: UrlSegment[]): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

    return this.loginService.isLoggedIn;

  }

const routes: Routes = [

  { path: 'rooms', loadChildren: () => import('./rooms/rooms.module').then((m) => m.RoomsModule), canLoad: [LoginGuard], canActivate: [LoginGuard] },

];

#### canDeactivate()

This guard is used to toggle when a component can be navigated away from based on some condition. For example, you cannot navigate away from a form if there are unsaved changes

import { Injectable } from '@angular/core';

import { ActivatedRouteSnapshot, CanDeactivate, RouterStateSnapshot, UrlTree } from '@angular/router';

import { Observable } from 'rxjs';

import { BookingComponent } from '../booking.component';

import { MatSnackBar } from '@angular/material/snack-bar';

@Injectable({

  providedIn: 'root'

})

// Target the component you want to control under "CanDeactivate<....>"" and "component: ...",

export class BookingGuard implements CanDeactivate<BookingComponent> {

  constructor(private \_snackBar: MatSnackBar) { }

  canDeactivate(

    component: BookingComponent,

    currentRoute: ActivatedRouteSnapshot,

    currentState: RouterStateSnapshot,

    nextState?: RouterStateSnapshot): Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree {

    // Check that the form is untouched and return true when navigation away is allowed

    if (component.bookingForm.pristine) {

      return true;

    } else {

      // open a snack bar to notify the user that there are unsaved changes

      this.\_snackBar.open('YOU HAVE UNSAVED CHANGES', 'DISCARD');

      return false;

    }

  }

}

Configure the route to the component with the guard:

const routes: Routes = [{ path: '', component: BookingComponent, canDeactivate: [BookingGuard] }];

Add “MatSnackBarModule” to app.module.ts

import { MatSnackBarModule } from '@angular/material/snack-bar';

imports: [

    .

.

.

    MatSnackBarModule

  ],

#### Resolve (Data Prefetching)

Sometimes you may want to delay the loading of a page until the data for the same is loaded. If the data is too huge, this may, otherwise, result in the view loading as a blank page that you may want to avoid. To delay this loading, you may use a resolve guard on the route that leads to the view/component.

To use this guard:

* Create a service to get the data from the API:

getComments() {

    return this.http.get<Comments[]>('https://jsonplaceholder.typicode.com/comments');

  }

* Create a guard, inject the service into the guard, implement the Resolve interface that returns the data gotten from the API, in its correct format:

import { Injectable } from '@angular/core';

import { ActivatedRouteSnapshot, CanActivate, Resolve, RouterStateSnapshot, UrlTree } from '@angular/router';

import { Observable } from 'rxjs';

import { Comments } from '../comments.interface';

import { CommentsService } from '../comments.service';

@Injectable({

  providedIn: 'root'

})

export class CommentsGuard implements Resolve<Comments[]> {

  constructor(private commentsService: CommentsService) {

  }

  resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Comments[] | Observable<Comments[]> | Promise<Comments[]> {

    return this.commentsService.getComments();

  }

}

* Within the component you intend to use it, import the ActivatedRoute module and pluck the data from the data property of the same.

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

import { CommentsService } from './comments.service';

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

import { pluck } from 'rxjs';

import { Comments } from './comments.interface';

@Component({

  selector: 'hinv-comments',

  templateUrl: './comments.component.html',

  styleUrls: ['./comments.component.scss']

})

export class CommentsComponent implements OnInit {

  constructor(

    private commentsService: CommentsService,

    private activatedRoute: ActivatedRoute

  ) {

  }

  // Comments$ = this.commentsService.getComments();

  // Comment$ = this.activatedRoute.data.pipe(pluck('comments'));

  Comments : Comments[] = [];

  ngOnInit(): void {

    this.activatedRoute.data.subscribe((data) => {

      this.Comments = data['comments'];

    });

  }

}

* Render the data in the template:

<table class="table">

    <thead>

        <tr>

            <th>postId</th>

            <th>it</th>

            <th>name</th>

            <th>email</th>

        </tr>

    </thead>

    <tbody>

        <!-- <tr \*ngFor="let comment of Comment$ | async"> -->

        <tr \*ngFor="let comment of Comments">

            <td>{{comment.postId}}</td>

            <td>{{comment.it}}</td>

            <td>{{comment.name}}</td>

            <td>{{comment.email}}</td>

        </tr>

    </tbody>

</table>

## Angular Material

### Adding Angular Material

ng add @angular/material

ng generate @angular/material:navigation <component-name> - To add navigation

### Adding Angular Material Schematics (Navigation Component)

ng generate @angular/material:navigation <component-name>

* Place the newly-created component’s selector in app.component.html. Say

<hinv-app-nav></hinv-app-nav>

* Replace the href links in the nav-component’s html with routerLinks and sepecify the paths as:

    <mat-nav-list>

      <a mat-list-item routerLink="rooms">Rooms</a>

      <a mat-list-item routerLink="employees">Employees</a>

      <a mat-list-item routerLink="input">Inputs</a>

    </mat-nav-list>

* Place the router-outlet tag in the section specified within the nav component:

<mat-sidenav-content>

    <mat-toolbar color="primary">

      <button

        type="button"

        aria-label="Toggle sidenav"

        mat-icon-button

        (click)="drawer.toggle()"

        \*ngIf="isHandset$ | async">

        <mat-icon aria-label="Side nav toggle icon">menu</mat-icon>

      </button>

      <span>hotelInventoryApp</span>

    </mat-toolbar>

    <!-- Add Content Here -->

    <router-outlet></router-outlet>

  </mat-sidenav-content>

## Angular Forms

In angular there are two types of forms:

* Template-Driven Forms
* Reactive forms

They both do the same thing and choosing between them is based on personal preference.

### Template-Driven Forms

These heavily depend on HTML for their functionality. They use “ngModel” for two-way data binding and “ngSubmit” for form submission:

#### Form Attributes

<!--Form and Form Controls' States-->

{{ roomForm.pristine | json}}   <!--The form is untouched. Only true when the form is created. Once any data is entered it turns to false and cannot go back until the view is refreshed-->

{{ roomForm.dirty | json}}      <!--The form is touched-->

{{ roomForm.valid | json}}      <!--All the form controls' validation requiremets are met-->

{{ roomForm.invalid | json}}    <!--Some of the form controls' validation requiremets are not met-->

{{ roomForm.value | json}}      <!--The collection of inputs entered into the form-->

{{ roomAmenities.valid | json}}

<form #roomForm="ngForm" (ngSubmit)="addRoom()">

    <div \*ngIf="success" class="alert" [ngClass]=" success ? 'alert-success' : 'alert-danger' ">

        {{successMessage}}

    </div>

    <div class="form-group">

        <input required type="text" class="form-control" [(ngModel)]="room.roomType" name="roomType" placeholder="Room Type">

    </div>

    <div class="form-group">

<!-- To handle validation  -->

        <input #roomAmenities="ngModel"

            required minlength="5" maxlength="50" pattern="^Hel\*"

            type="text" class="form-control" [(ngModel)]="room.ammenities" name="amenities"

            placeholder="Amenities"

" placeholder="Amenities">

<div class="alert alert-danger" \*ngIf="roomAmenities.errors">

            {{ roomAmenities.errors | json}}

        </div>

    </div>

    <div class="form-group">

        <input required type="text" class="form-control" [(ngModel)]="room.image" name="image" placeholder="Photo Url">

    </div>

    <div class="form-group">

        <input required type="number" class="form-control" [(ngModel)]="room.price" name="price" placeholder="Price per Night">

    </div>

    <div class="form-group">

        <input required type="datetime" class="form-control" [(ngModel)]="room.checkInTime" name="checkInTime"

            placeholder="Check In Time">

    </div>

    <div class="form-group">

        <input required type="datetime" class="form-control" [(ngModel)]="room.checkOutTime" name="checkOutTime"

            placeholder="Check Out Time">

    </div>

    <div class="form-group">

        <input required type="number" class="form-control" [(ngModel)]="room.rating" name="rating" placeholder="Rating">

    </div>

    <div class="form-group">

        <button [disabled]="roomForm.invalid" class="btn btn-primary">Add Room</button>

    </div>

</form>

#### Form Initialisation

To initialize the form properties:

room: roomList = {

    roomType : '',

    ammenities : '',

    image : '',

    price : 0,

    checkInTime : new Date(),

    checkOutTime : new Date(),

    rating : 0,

};

success!: boolean;

successMessage!: string;

Some build-in validators include:

* required
* minlength
* maxlength
* min
* max
* email
* pattern – For regular expressions

#### Two-way Data Binding

To access the controls use the “#” notation and initialize the same as “ngModel”. E.g.:

    <div class="form-group">

<!-- To handle validation  -->

        <input #roomAmenities="ngModel"

            required minlength="5" maxlength="50" pattern="^Hel\*"

            type="text" class="form-control" [(ngModel)]="room.ammenities" name="amenities"" placeholder="Amenities">

<div class="alert alert-danger" \*ngIf="roomAmenities.errors">

            {{ roomAmenities.errors | json}}

        </div>

    </div>

#### Reseting Forms

To reset the form (changing its “pristine” state to true):

<form #roomForm="ngForm" (ngSubmit)="addRoom(roomForm)">

</form>

addRoom(roomForm: NgForm) {

    this.roomsService.addRoom(this.room).subscribe(

      (data) => {

        this.success = true

        this.successMessage = "Room Added Successfully";

        console.log(data);

        // roomForm.reset();     // To reset it with no default values

        roomForm.reset({        // To reset it with default values

          roomType : '',

          ammenities : '',

          image : '',

          price : 0,

          checkInTime : new Date(),

          checkOutTime : new Date(),

          rating : 0,

        });

      },

      (err) => {

        this.success = false

        this.successMessage = 'Room failed to add'

      }

    );;

  }

### Reactive Forms

#### Configuring and initializing the form in the .ts file

Reactive forms are forms created using Typescript rather than HTML.

* Import “ReactiveFormsModule” into the module you intend to use them

@NgModule({

  declarations: [

    BookingComponent

  ],

  imports: [

    CommonModule,

    BookingRoutingModule,

    ReactiveFormsModule

  ]

})

export class BookingModule { }

* Declare the form and set it to type “FormGroup”, importing the class in the process.
* Inject the “FormBuilder” service that will be used to create the form, grouping the formControls.
* Initialise the object of type “FormGroup” by invoking the formBuilders “group” method and pass the form controls you want to render, along with their validators

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

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

@Component({

  selector: 'hinv-booking',

  templateUrl: './booking.component.html',

  styleUrls: ['./booking.component.scss']

})

export class BookingComponent implements OnInit {

  bookingForm!: FormGroup;

  constructor(private fb: FormBuilder) { }

  ngOnInit(): void {

    console.log("Booking initiated");

    this.bookingForm = this.fb.group({

      // roomId: new FormControl({value: "2", disabled: true}),

      roomId: [{value: "2", disabled: true}],

      // bookingId: new FormControl(''), Same thing as below

      bookingId: [''],

      roomId: new FormControl(''),

      guestEmail: new FormControl(''),

      checkinDate: new FormControl(''),

      checkoutDate: new FormControl(''),

      bookingStatus: new FormControl(''),

      bookingAmount: new FormControl(''),

      bookingDate: new FormControl(''),

      mobileNumber: new FormControl(''),

      guestName: new FormControl(''),

      guestAddress: new FormControl(''),

      guestCity: new FormControl(''),

      guestState: new FormControl(''),

      guestCountry: new FormControl(''),

      guestZipCode: new FormControl(''),

      guestCount: new FormControl(''),

    });

  }

}

#### Rendering the form on the Template

* Now that the form is already-created, render it on the HTML of the component. You will have to use the “[formGroup]” directive to tell the HTML use only render the already-created from, rather than create a new one:

<form [formGroup]="bookingForm" (ngSubmit)="bookRoom()">

    <!--Import MatFormFieldModule and MatInputModule from within the module.ts file-->

    <mat-form-field appearance="fill">

        <mat-label>Room Id</mat-label>

        <input

            type="number"

            matInput

            formControlName="roomId"

            placeholder="Room ID">

    </mat-form-field>

    <mat-form-field appearance="fill">

        <mat-label>Email</mat-label>

        <input

            type="email"

            matInput

            formControlName="guestEmail"

            placeholder="Enter your Email">

    </mat-form-field>

    <mat-form-field appearance="fill">

        <mat-label>Full Name</mat-label>

        <input

            type="text"

            matInput

            formControlName="guestName"

            placeholder="Enter your Full Name">

    </mat-form-field>

    <mat-form-field appearance="fill">

        <mat-label>Mobile Number</mat-label>

        <input

            type="text"

            matInput

            formControlName="mobileNumber"

            placeholder="Enter your Mobile Number">

    </mat-form-field>

    <mat-form-field appearance="fill">

        <mat-label>Booking Amount</mat-label>

        <input

            type="number"

            matInput

            formControlName="bookingAmount"

            placeholder="Enter the Booking Amount">

    </mat-form-field>

    <!--Import MatDatepickerModule, MatNativeDateModule from within the module.ts file-->

    <mat-form-field>

        <mat-label>Booking Date</mat-label>

        <input matInput [matDatepicker]="bookingDatePicker" formControlName="bookingDate">

        <mat-datepicker-toggle matIconSuffix [for]="bookingDatePicker"></mat-datepicker-toggle>

        <mat-datepicker #bookingDatePicker></mat-datepicker>

    </mat-form-field>

    <mat-form-field>

        <mat-label>CheckIn Date</mat-label>

        <input matInput [matDatepicker]="checkinDatePicker" formControlName="checkinDate">

        <mat-datepicker-toggle matIconSuffix [for]="checkinDatePicker"></mat-datepicker-toggle>

        <mat-datepicker #checkinDatePicker></mat-datepicker>

    </mat-form-field>

    <mat-form-field>

        <mat-label>Check Out Date</mat-label>

        <input matInput [matDatepicker]="checkoutDatePicker" formControlName="checkoutDate">

        <mat-datepicker-toggle matIconSuffix [for]="checkoutDatePicker"></mat-datepicker-toggle>

        <mat-datepicker #checkoutDatePicker></mat-datepicker>

    </mat-form-field>

    <!--A for can have only one "[FormGroup]". Therefore all the nested forms within it should be identified using "formGroupName"-->

    <div formGroupName="address">

        <mat-accordion class="example-headers-align" multi>

            <mat-expansion-panel>

                <mat-expansion-panel-header>

                  <mat-panel-title>

                    Address

                  </mat-panel-title>

                </mat-expansion-panel-header>

                <mat-form-field appearance="fill">

                    <mat-label>Address Line 1</mat-label>

                    <input

                        type="text"

                        matInput

                        formControlName="addressLine1"

                        placeholder="Enter your Primary Address">

                </mat-form-field>

                <mat-form-field appearance="fill">

                    <mat-label>Address Line 2</mat-label>

                    <input

                        type="text"

                        matInput

                        formControlName="addressLine2"

                        placeholder="Enter your Secondary Address">

                </mat-form-field>

                <mat-form-field appearance="fill">

                    <mat-label>City</mat-label>

                    <input

                        type="text"

                        matInput

                        formControlName="city"

                        placeholder="City">

                </mat-form-field>

                <mat-form-field appearance="fill">

                    <mat-label>State</mat-label>

                    <input

                        type="text"

                        matInput

                        formControlName="state"

                        placeholder="State">

                </mat-form-field>

                <mat-form-field appearance="fill">

                    <mat-label>Country</mat-label>

                    <input

                        type="text"

                        matInput

                        formControlName="country"

                        placeholder="Country">

                </mat-form-field>

                <mat-form-field appearance="fill">

                    <mat-label>Zip Code</mat-label>

                    <input

                        type="text"

                        matInput

                        formControlName="zipCode"

                        placeholder="Zip Code">

                </mat-form-field>

            </mat-expansion-panel>

        </mat-accordion>

    </div>

    <!--Import MatButtonModule from within the module.ts file-->

    <div>

        <button mat-raised-button color="primary">Book Room</button>

    </div>

</form>

#### Setting up Angular Material for use in the form

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

import { CommonModule } from '@angular/common';

import { BookingRoutingModule } from './booking-routing.module';

import { BookingComponent } from './booking.component';

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

import { MatInputModule } from '@angular/material/input';

import { MatFormFieldModule } from '@angular/material/form-field';

import { MatDatepickerModule } from '@angular/material/datepicker';

import { MatNativeDateModule } from '@angular/material/core';

import { MatButtonModule } from '@angular/material/button';

@NgModule({

  declarations: [

    BookingComponent

  ],

  imports: [

    CommonModule,

    BookingRoutingModule,

    ReactiveFormsModule,

    MatFormFieldModule, // There are the imports necessary to render the angular material form fields

    MatInputModule,

    MatDatepickerModule,

    MatNativeDateModule,

    MatButtonModule

  ]

})

export class BookingModule { }

#### Adding and Removing Controls Dynamically To Your Reactive Form

##### Using FormBuilder’s “.array”

* Within the larger form group, create a form control but initialize it as a form array that is in turn initialized with another form group having some arbitrary form controls.

ngOnInit(): void {

    console.log("Booking initiated");

    this.bookingForm = this.fb.group({

      checkinDate: [''],

      checkoutDate: [''],

      guests: this.fb.array([this.fb.group({

        guestName: [''],

        age: ['']

      })])

    });

  }

* Create a button in the template that clicked adds and a new form group of the innermost type.

<div>

   <button mat-raised-button color="primary" type="button" (click)="addGuest()">Add Guest</button>

</div>

* Implement the form group addition functionality such that the button pushes a new form group into the form array/ form control of the parent form group. You will, however, need to use a getter so as to get the form array since it is defined privately within the parent form group.

get guests() {

  return this.bookingForm.get('guests') as FormArray;

}

addGuest(){

  this.guests.push(this.fb.group({

  guestName: [''],

    age: ['']

  }));

}

* Render the form array in the template with the following structure:
  + A div for the form array/ form control of the parent form group identified using “formArrayName”
  + A div to handle the looping of the form array
  + An inner nested form group to handle the controls defined, identified using “formGroupName” and initialized as the index of the loop
  + Form Controls of the inner form group

<div formArrayName="guests">

        <mat-accordion class="example-headers-align" multi>

            <mat-expansion-panel>

                <mat-expansion-panel-header>

                  <mat-panel-title>

                    Guests

                  </mat-panel-title>

                </mat-expansion-panel-header>

                <div \*ngFor="let guest of guests.controls; let i=index">

                    <div [formGroupName]="i">

                        <mat-form-field appearance="fill">

                            <mat-label>Guest Name</mat-label>

                            <input

                                type="text"

                                matInput

                                formControlName="guestName"

                                placeholder="Enter the full name of the guest">

                        </mat-form-field>

                        <mat-form-field appearance="fill">

                            <mat-label>Age</mat-label>

                            <input

                                type="number"

                                matInput

                                formControlName="age"

                                placeholder="Enter the age of the guest">

                        </mat-form-field>

                        <div>

                            <button mat-raised-button color="primary" type="button" (click)="removeGuest(i)">Remove Guest</button>

                        </div>

                    </div>

                </div>

            </mat-expansion-panel>

        </mat-accordion>

    </div>

##### Using AddControl Method

* Create a button that adds a single form control to the form group

<div>

   <button mat-raised-button color="primary" type="button" (click)="addPassport()">Add Passport</button>

</div>

addPassport() {

 this.bookingForm.addControl('passport', new FormControl(''));

   console.log(this.bookingForm.getRawValue());

}

* Conditionally render the form control using \*ngIf and using the “get” method to check whether the field exists

<mat-form-field appearance="fill" \*ngIf="bookingForm.get('passport')">

        <mat-label>Passport Number</mat-label>

        <input

            type="number"

            matInput

            formControlName="passport"

            placeholder="Enter your Passport Number">

    </mat-form-field>

##### Using RemoveControl Method

* Add a button that removes the form desired form control

<div>

        <button mat-raised-button color="primary" type="button" (click)="deletePassport()">Delete Passport</button>

    </div>

deletePassport() {

    this.bookingForm.removeControl('passport');

    console.log(this.bookingForm.getRawValue());

  }

* If you are removing a control within a form Array, use the “removeAt” Method

<div \*ngFor="let guest of guests.controls; let i=index">

  <div [formGroupName]="i">

<div>

         <button mat-raised-button color="primary" type="button" (click)="removeGuest(i)">Remove Guest</button>

      </div>

</div>

</div>

removeGuest(i: number) {

  this.guests.removeAt(i);

}

#### Validators in Reactive Forms

Use the “Validators” module, rather than the template-driven “Validator”.

Have a lot of the same validators as template-driven forms.

* required
* requiredTrue
* minlength
* maxlength
* min
* max
* email
* pattern – For regular expressions

To add the Validators

* Import “Validators” from “@angular/forms”.
* Pass the validated in the form controls initialization

import { FormArray, FormBuilder, FormControl, FormGroup, Validators } from '@angular/forms';

ngOnInit(): void {

    // console.log("Booking initiated");

    this.bookingForm = this.fb.group({

      guestEmail: ['', [Validators.email, Validators.required]],

      guestName: ['', [Validators.minLength(5), Validators.required]],

      tnc: new FormControl(false, {validators: [Validators.requiredTrue]})

    });

  }

#### Resetting Reactive Forms

bookRoom() {

    this.bookingForm.reset({

      roomId: '',

      guestName: '',

      address: {

        addressLine1: '',

        addressLine2: '',

        zipCode: '',

      },

      guests: [],

      tnc: false

    })

  }

#### PatchValue and SetValue

These are used to initialize the form controls with data, say from and API.

The difference is that setValue is very strict – requires all fields to be initialized, whereas patchValue is more flexible, allowing for some controls to be omitted.

  ngOnInit(): void {

    // console.log("Booking initiated");

    this.bookingForm = this.fb.group({

      // roomId: new FormControl({value: "2", disabled: true}),

      roomId: [{ value: "2", disabled: true }],

      // bookingId: new FormControl(''), Same thing as below

      bookingId: [''],

.

.

.

// Call the getBookingData method after initialising the form

    this.getBookingData();

  }

  getBookingData(): void {

    this.bookingForm.patchValue({

      guestEmail: 'test@gmail.com',

      checkinDate: new Date('20-May-2024'),

      checkoutDate: new Date('21-May-2024'),

    });

  }

### Adding Mat-Error at the form control level

<mat-form-field appearance="fill">

   <mat-label>Email</mat-label>

     <input

      type="email"

      matInput

    formControlName="guestEmail"

      placeholder="Enter your Email">

<!-- Target a base level control within the parent form group-->

<mat-error \*ngIf="bookingForm.get('guestEmail')?.hasError('email')">The email provided is not valid</mat-error>

</mat-form-field>

<div formGroupName="address">

  <mat-accordion class="example-headers-align" multi>

      <mat-form-field appearance="fill">

        <mat-label>Address Line 1</mat-label>

            <input

          type="text"

               matInput

               formControlName="addressLine1"

               placeholder="Enter your Primary Address">

<!--Targetting a control within a nested form-->

            <mat-error \*ngIf="bookingForm.get('address.addressLine1')?.hasError('required')">Address Line 1 is Required</mat-error>

      </mat-form-field>

</mat-expansion-panel>

</mat-accordion>

</div>

<div formArrayName="guests">

   <mat-accordion class="example-headers-align" multi>

      <mat-expansion-panel>

         <mat-expansion-panel-header>

            <mat-panel-title>

              Guests

            </mat-panel-title>

         </mat-expansion-panel-header>

      <div \*ngFor="let guest of guests.controls; let i=index">

      <div [formGroupName]="i">

        <mat-form-field appearance="fill">

          <mat-label>Guest Name</mat-label>

              <input

                  type="text"

                  matInput

                  formControlName="guestName"

                  placeholder="Enter the full name of the guest">

<!-- Target a control within a form array-->

<mat-error \*ngIf="bookingForm.get(['guests', i, 'guestName'])?.hasError('required')">Address Line 1 is Required</mat-error>

        </mat-form-field>

</mat-expansion-panel>

   </mat-accordion>

</div>

### Listening To Changes In the Form Controls’ Values

ngOnInit(): void {

   // console.log("Booking initiated");

   this.bookingForm = this.fb.group({

     guestEmail: ['', [Validators.email, Validators.required]],

     checkinDate: [''],

      .

.

     tnc: new FormControl(false, {validators: [Validators.requiredTrue]})

   },{

     updateOn: 'blur'

// Can also be 'submit' or 'Change' (default e.g. for ebery key stroke).

     // Can also be applied on individual controls

// Used, for example, to delay the showing off error messages until perhaps the form is submitted

   });

   this.getBookingData();

    // "FormModule" provides a property "valueChanges" that is a stream of changes on the input fields of a form. By default, This is done for every keystroke

   this.bookingForm.valueChanges.subscribe((data) => {

     console.log(data);

    });

  }

The same can be done with template-driven forms using “ngModelOptions”:

<div class="form-group">

        <input #roomAmenities="ngModel"

            required minlength="5" maxlength="50" pattern="^Hel.\*"

            type="text" class="form-control" [(ngModel)]="room.ammenities" name="amenities"

            placeholder="Amenities"

<!-- To Toggle the value change listening -->

            [ngModelOptions]="{ updateOn: 'blur' }">

        <div class="alert alert-danger" \*ngIf="roomAmenities.dirty && roomAmenities.errors">

            {{ roomAmenities.errors | json}}

        </div>

### Custom Directives With Forms

Directives are almost the same thing as components except they do not have templates.

#### Creating a Custom Directive

To create a custom directive, run *“ng generate directive <directive name>”.*

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp>* ***ng generate directive hover***

*CREATE src/app/hover.directive.spec.ts (220 bytes)*

*CREATE src/app/hover.directive.ts (140 bytes)*

*UPDATE src/app/app.module.ts (2871 bytes)*

import { Directive, ElementRef, HostListener, Input, OnInit, Renderer2 } from '@angular/core';

@Directive({

  selector: '[hinvHover]'

})

export class HoverDirective implements OnInit{

  @Input() color: string = 'red';

  // Get the element being passed to this directive using "ElementRef"

  // You can modify the element's styles and/or properties using another api called "renderer2".

  constructor(private element: ElementRef, private renderer: Renderer2) {

    console.log(this.element.nativeElement);

    console.log(this.renderer);

  }

  ngOnInit(): void {

    // Two ways to modify the elements being passed.

    this.element.nativeElement.style.backgroundColor = this.color;

    this.renderer.setStyle(

      this.element.nativeElement,

      'font-size',

      '20px');

    this.renderer.setProperty(

      this.element.nativeElement,

      'disabled',

      true);

  }

  // Hostlistener is used to listen to events that happen on the parent component.

  @HostListener('mouseenter') onMouseEnter() {

    this.renderer.setStyle(

      this.element.nativeElement,

      'backgroundColor',

      'blue');

  }

  @HostListener('mouseleave') onMouseLeave() {

    this.renderer.setStyle(

      this.element.nativeElement,

      'backgroundColor',

      'white');

  }

}

#### Linking Directives with Components

To link this directive to a component element, add the directive’s selector to the html component:

<form>

    <div class="form-group">

        <!-- Add the directives's selector "hinvHover" to pass this element to the directive -->

        <input hinvHover color="yellow" type="email" class="form-control" [(ngModel)]="email" name="email" placeholder="Please provide your email">

    </div>

    <div class="form-group">

        <!-- Add the directives's selector "hinvHover" to pass this element to the directive -->

        <input hinvHover color="yellow" type="password" class="form-control" [(ngModel)]="password" name="password" placeholder="Enter your password">

    </div>

    <div class="form-group">

        <button class="btn btn-primary" (click)="login()">Login</button>

    </div>

</form>

### Custom Validators

#### For Template-Driven Forms

To create a custom validator,

* create a custom directive

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app\emailValidator>* ***ng generate directive emailValidator***

*CREATE src/app/emailValidator/email-validator.directive.spec.ts (257 bytes)*

*CREATE src/app/emailValidator/email-validator.directive.ts (158 bytes)*

*UPDATE src/app/app.module.ts (2986 bytes)*

* Implement “Validator” (for template-driven forms) on the class and implement the “validate” control.

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

import { AbstractControl, NG\_VALIDATORS, ValidationErrors, Validator } from '@angular/forms';

@Directive({

  selector: '[hinvEmailValidator]',

  providers: [{

    provide: NG\_VALIDATORS,

    useExisting: EmailValidatorDirective,

    multi: true,

  }]

})

export class EmailValidatorDirective implements Validator{

  constructor() { }

  // This takes the controls being passed as a parameter

  // And Returns the error, if some error condition is met, else it returns null if there is no error.

  validate(control: AbstractControl<any, any>): ValidationErrors | null {

    const value = control.value as string;    // To cast the input from the input control to a string.

    if(value.includes('test')) {

      return {invalid: true}    // The error should be an object with a key and value.

    } else {

      return null;

    }

  }

  // registerOnValidatorChange?(fn: () => void): void {

  //   throw new Error('Method not implemented.');

  // }

}

* Add the directive selector to the template and get the errors if any:

<input #emailControl="ngModel" hinvEmailValidator hinvHover color="yellow" type="email" class="form-control" [(ngModel)]="email" name="email" placeholder="Please provide your email">

  <div>

      {{ emailControl.errors | json}}

   </div>

#### For Reactive Forms

**Adding Custom Validation**

* Create a class that will handle the validators
* Create static methods within the class (methods that belong to the class rather than the instances/objects created from it. This way the methods can be called by directly invoking the class name)

import { AbstractControl } from "@angular/forms";

export class CustomValidator {

    static ValidateName(control: AbstractControl) {

        const value = control.value as string;

        if(value.includes('test')) {

            return {

                invalidName: true

            }

        } else {

            return null;

        }

    }

    // The difference is that you pass the character and return a function

    static ValidateSpecialChar(char: string ) {

        return ((control: AbstractControl) => {

            const value = control.value as string;

            if(value.includes(char)) {

                return {

                    invalidSpecialChar: true

                }

            } else {

                return null;

            }

        })

    }

    static ValidateDate(control: FormGroup) {

        // Get the dates from the controls directly

        const checkinDate: any = new Date(control.get('checkinDate')?.value);

        const checkoutDate: any = new Date(control.get('checkoutDate')?.value);

        const difference = Math.ceil(checkoutDate - checkinDate);

        console.log(difference)

        if(difference < 0) {

            // Add an error to the "checkoutDateControl"

            control.get('checkoutDate')?.setErrors({

                invalidDate: true

            })

            // Add an error to the Form as well

            return {

                invalidDate: true

            }

        } else {

            return null;

        }

    }

}

* Invoke the validator:

**At Control Level**

guestName: ['', [Validators.minLength(5), Validators.required, CustomValidator.ValidateName]],

**At Form Level**

, {

      updateOn: 'blur',

      validators: [CustomValidator.ValidateDate]

    }

## Feature Modules

### Setting up feature modules

Registering everything inside “app.module.ts” may result in problems debugging because of the size of the file. Also, for the sake of lazy loading, it is advisable to separate features into different modules.

To create a module:

**Ng generate module <moduleName> --routing –flat=true**

The “--flat” is used to generate the “module.ts” files in the same directory without creating a folder for the same.

The “---routing” is to create a “.routing.ts” file for the module so that you can register the routes related to the feature in the same file, separate from “app.routing.ts”

Once created, move all the components that are related to the now-created module and remove their declarations from “app.module.ts”.

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

import { CommonModule } from '@angular/common';

import { RoomsRoutingModule } from './rooms-routing.module';

import { RoomsListComponent } from './rooms-list/rooms-list.component';

import { RoomBookingComponent } from './room-booking/room-booking.component';

import { RoomsComponent } from './rooms.component';

import { AddRoomsComponent } from './add-rooms/add-rooms.component';

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

import { HeaderModule } from '../header/header.module';

@NgModule({

  declarations: [

    RoomsComponent,

    RoomsListComponent,

    RoomBookingComponent,

    AddRoomsComponent,

  ],

  imports: [

    CommonModule,

    RoomsRoutingModule,

    FormsModule,

    HeaderModule

  ]

})

export class RoomsModule { }

Also, remove all the routes specific to the module from app-routing.module.ts and move them to the module’s “-routing.module.ts” file.

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

import { RouterModule, Routes } from '@angular/router';

import { RoomsComponent } from './rooms.component';

import { AddRoomsComponent } from './add-rooms/add-rooms.component';

import { RoomBookingComponent } from './room-booking/room-booking.component';

const routes: Routes = [

  { path: 'rooms', component: RoomsComponent },

  { path: 'rooms/add-room', component: AddRoomsComponent },

  { path: 'rooms/:id', component: RoomBookingComponent },     // For Dynamic Routes

];

@NgModule({

  imports: [RouterModule.forChild(routes)],

  exports: [RouterModule]

})

export class RoomsRoutingModule { }

If there are any components outside the module that are used by any of the components within, create another module for the component in question.

Note, You will have to export the component being used outside and remove the same from “app.module.ts”.

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

import { CommonModule } from '@angular/common';

import { HeaderComponent } from './header.component';

@NgModule({

  declarations: [

    HeaderComponent

  ],

  imports: [

    CommonModule

  ],

  exports: [

    HeaderComponent

  ]

})

export class HeaderModule { }

### Lazy Loading

#### Manual Configuration

It is advisable to not only split the entire project into separate feature modules , but also to configure the same so that they may be downloaded separately and when needed.

For instance, if there is a module that maintains the user profile this module can be configured to be downloaded only when the user clicks on the router link that leads to that view/component.

Lazy loading also reduces the main bundle size so that users using the deployed product and having slow 3G networks can do so efficiently.

To configure lazy loading for a specific module:

* Remove the same from the “imports” section in “app.module.ts”
* Call the module as part of a route in “app.routing.ts” or other file using the “loadChildren” property.

const routes: Routes = [

  { path: 'login', component: LoginComponent },

  { path: 'employees', component: EmployeeComponent },

  { path: 'input', component: InputCheckComponent },

  { path: 'rooms', loadChildren: () => import('./rooms/rooms.module').then((m) => m.RoomsModule) },

  { path: '', redirectTo: '/login', pathMatch: 'full' },     // The default route

  { path: '\*\*', component: NotFoundComponent },     // Page 404 Not Found

];

* If the lazily-loaded module has a “.routing.module.ts” file of its own, remove the content in the parent route, to an empty string:

// { path: 'rooms',

  { path: '',

    component: RoomsComponent,

    children: [

      { path: 'add-room', component: AddRoomsComponent }, // Place this route before the dynamic route otherwise that dynamic route will always be called.

      { path: ':id', component: RoomBookingComponent }

    ],     // For Dynamic Routes

  }

];

#### Lazy Loading Using Angular CLI

If you are starting the project from scratch and know which modules you will lazily load, you can create the same already configured using angular CLI.

To create a component and its modules and lazily load the same use the command:

**ng generate module <moduleName> --route=”<routeName>” –routing –module=<parentModule>**

E.g.

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app>* ***ng generate module booking --route="booking" --routing --module="app"***

*CREATE src/app/booking/booking-routing.module.ts (347 bytes)*

*CREATE src/app/booking/booking.module.ts (364 bytes)*

*CREATE src/app/booking/booking.component.html (22 bytes)*

*CREATE src/app/booking/booking.component.spec.ts (606 bytes)*

*CREATE src/app/booking/booking.component.ts (281 bytes)*

*CREATE src/app/booking/booking.component.scss (0 bytes)*

*UPDATE src/app/app-routing.module.ts (1039 bytes)*

To create a module and then a component within that module using angular CLI:

* Create the module first – The missing “—flat” means a folder will be created for the module

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app>* ***ng generate module search***

*CREATE src/app/search/search.module.ts (192 bytes)*

* Create the component. This will update the created “.module.ts” file do add the component in its “declarations”:

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app>* ***ng generate component search***

*CREATE src/app/search/search.component.html (21 bytes)*

*CREATE src/app/search/search.component.spec.ts (599 bytes)*

*CREATE src/app/search/search.component.ts (277 bytes)*

*CREATE src/app/search/search.component.scss (0 bytes)*

*UPDATE src/app/search/search.module.ts (269 bytes)*

## Custom Pipe (Adding a Filter to Data Using a Form Control)

* Create a pipe that will handle the filtering logic using “ng generate pipe <pipeName>”.

*PS D:\Projects\Angular\Introduction To Angular\hotelInventoryApp\src\app\rooms\filters****> ng generate pipe filter***

*CREATE src/app/rooms/filters/filter.pipe.spec.ts (187 bytes)*

*CREATE src/app/rooms/filters/filter.pipe.ts (217 bytes)*

*UPDATE src/app/rooms/rooms.module.ts (1063 bytes)*

* To the filter method, pass the data and filter parameter, and return the data in the format/type intended.

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

import { roomList } from '../rooms';

@Pipe({

  name: 'filter'

})

export class FilterPipe implements PipeTransform {

  transform(rooms: roomList[], price: number): roomList[] {

    return rooms.filter((room) => {

      return room.price > price

    });

  }

}

* Apply the filter on the template

<tbody>

   <tr \*ngFor="let room of roomsList | filter : price; let e = even; let o = odd; let i = index" [ngClass]=" e ? 'even' : 'odd' " [ngStyle]="{'padding' : e ? '100px' : '200px'}">

.

.

.

   </tr>

</tbody>

@Input() price: number = 0;

<div>

    <!-- Use [formControl] because there is no FormGroup Associated with the control -->

    <input type="number" class="form-control" [formControl]="priceFilter" placeholder="Price Of Room">

</div>

<div \*ngIf="rooms$ | async as rooms"> <!-- To test the ngOnDestroy of <hinv-rooms-list> -->

    <hinv-rooms-list [roomsList]="rooms" [price]="priceFilter.value ? priceFilter.value : 0" [title]="title" (selectedRoom)="selectRoom($event)"></hinv-rooms-list>

</div>

## Global Error Handling

Angular comes with a Error Handling service whose error handling method interface which you can implement.

* Create a file within the “src/app” directory, and name it, say “errorhandler.service.ts” and within it implement the method:

import { ErrorHandler } from "@angular/core";

export class GlobalErrorHandler implements ErrorHandler {

    handleError(error: any): void {

        console.log(error);

        // This is just an example. You can do more complex things with the error such save them to a database that can then be used during system improvement after User Acceptance Testing.

    }

}

* Provide the class to “app.module.ts”:

providers: [

    {

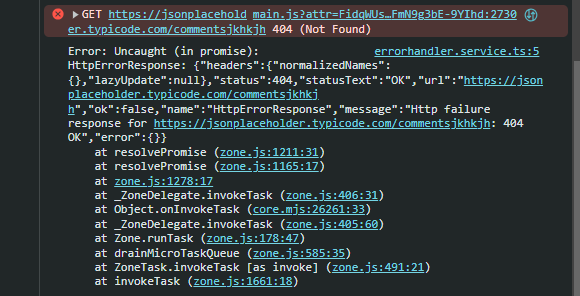
      provide: ErrorHandler,

      useClass: GlobalErrorHandler

    }

  ],

* For testing purposes, introduce an error, say misspell an API endpoint and watch how the method works:



## Testing

Angular provides karma, which is a testing framework that you can use to test your application.

To run the tests run, ***“ng test”*** in the terminal and follow the link provided to render the karma-jasmine report on the browser.

There is a section “Spec List”, that show the list of all the “.spec.ts” files used for the testing, as well as each file’s success status.

There is another section “Failures” that shows details of all the test that have failed.

To write a test case.

* Consider the following class:

export class Calculator {

    add(num1: number, num2: number) : number {

        return num1 + num2;

    }

    subtract(num1: number, num2: number) : number {

        return num1 - num2;

    }

    multiply(num1: number, num2: number) : number {

        return num1 \* num2;

    }

}

* A test can be written as:

import { Calculator } from "./testservices";

// A test case should describe a scenario.

describe('CalculatorTest', () => {

    // The test should be written like a story, with each test case being given an expected value

    it('should add two numbers', ()=>{

        const service = new Calculator();

        expect(service.add(2,2)).toBe(4);

    });

    it('should subtract two numbers', ()=>{

        const service = new Calculator();

        expect(service.subtract(2,2)).toBe(0);

    });

    it('should multiply two numbers', ()=>{

        const service = new Calculator();

        expect(service.multiply(2,2)).toBe(4);

    });

});