**Introduction to Services**

* Used for not repeating the code
* A service in Angular is a class that contains some functionality that can be reused across the application
* Services
  + Used to share the code across components of an application
  + Used to make HTTP requests
  + No separate decorator to indicate that a class is a service class

app.component.ts

title="Angular"

app.component.html

<h1>Welcome to {{title}}</h1>

<employee-list></employee-list>

<employee-detail></employee-detail>

employee-list.component.ts

template:

<h2>Employee List</h2>

<ul \*ngFor="let emp of employees">

<li>{{emp.name}}</li>

</ul>

export class…{

public employees=[

{"id":1,"name":"John","age":25},

{"id":2,"name":"Jack","age":35},

{"id":3,"name":"Steve","age":32},

{"id":4,"name":"Elena","age":27}

];

* In employee list component, we are displaying only the names of the employees.
* Suppose if we need to display all the details of the same employees in a separate view, let's say in a separate component, then again we need to copy all employee data to that component model as well.

employee-detail.component.ts

template:

<h2>Employee List</h2>

<ul \*ngFor="let emp of employees">

<li>{{emp.id}}{{emp.name}}{{emp.age}}</li>

</ul>

export class…{

public employees=[

{"id":1,"name":"John","age":25},

{"id":2,"name":"Jack","age":35},

{"id":3,"name":"Steve","age":32},

{"id":4,"name":"Elena","age":27}

];

* Solution works, but not efficient.
  + Violating the below principles
    - DRY(Do not Repeat Yourself)
    - Single Responsibility Principle
      * Component class should have only one responsibility -> to control view's logic, but here it is responsible for creating the data i.e., employees array.
* **Service**
  + A class with a specific purpose
  + Need of service
    - To share data
      * Responsible for providing data to components
    - Implement application logic
      * E.g. Calculating age from dob, this logic we can write and reuse.
    - External interaction
      * To connect with databases
  + Naming convention - .service.ts

* Dependency Injection
  + Mechanism where the required resources will be injected into the code automatically.
  + It can add the functionality of components at run-time.
  + It is a design pattern in which a class receives its dependencies from external sources rather than creating themselves.
  + Allows developers to reuse the code across application.
* 3 steps
  + Creation of a service class (Employee service class)
  + Register the service with Angular's built-in Injector.
  + Declare the service as a dependency in class which needs it. (employee-list and employee-detail)
  + **Creating a service**
    - **ng g s employee**

employee-service.ts

import {Injectable}…

@Injectable()

export class EmployeeService{

constructor(){}

getEmployees(){

return[

{"id":1,"name":"John","age":25},

{"id":2,"name":"Jack","age":35},

{"id":3,"name":"Steve","age":32},

{"id":4,"name":"Elena","age":27}];

1. **Registering with Injector**
   * **Register in App Module**

app.module.ts

@NgModule({

--

providers:[EmployeeService];

---

1. **Declaring as dependency in EmpList and EmpDetail**

employee-list.component.ts

export class ---{

public employees=[];

constructor(private empSer:EmployeeService){}

ngOnInit(){

this.employees=empSer.getEmployees;

}

}

employee-detail-component.ts

export class ---{

public employees=[];

constructor(private empSer:EmployeeService){}

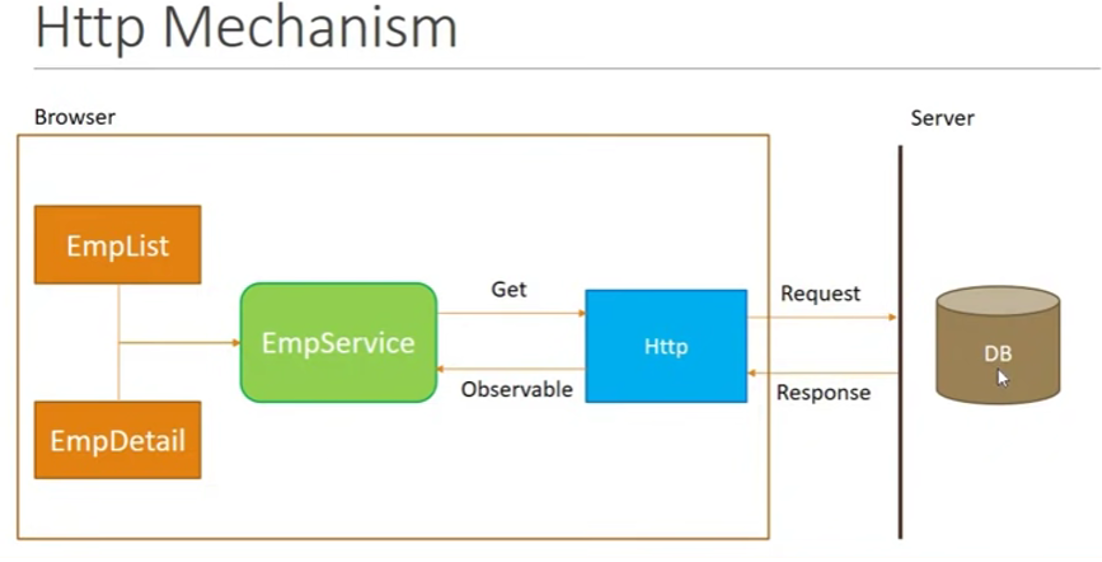
ngOnInit(){

this.employees=empSer.getEmployees;

}

}

**Server Communication using HttpClient**



* Instead of writing data directly in the service class, we need to get the data from a web server. (In our example, instead of real web server, we will use an external json file stored in assets folder, assuming the folder exists in a webserver)
* Steps:
  + Add HttpClientModule in the imports property of app.module.ts
  + Http Get request from service (EmployeeService)
  + Receive the observable (response from server-collection-asynchronous data) and cast into an array.(employees array)
  + Subscribe to the observable from the components that need the service.(EmployeeList & EmployeeDetails)
  + Assign the array (employees array) to a local variable.

app.module.ts

---

imports:[

BrowserModule,

HttpClientModule

],

---

* Inside src->right click assets folder,
  + create new folder - Data
  + Right click Data folder
    - Create New file - employees.json

employees.json

[

{"id":1,"name":"John","age":25},

{"id":2,"name":"Jack","age":35},

{"id":3,"name":"Steve","age":32},

{"id":4,"name":"Elena","age":27}

];

* Right click app folder -> create a new file -> employee.ts
  + Inside this, we can write a class or interface to cast the Observable object into our own class or interface type.

Employee.ts

export class Employee{

id:number=0;

name:string=" ";

age:number=0;

}

employee.service.ts

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

export class ---{

private empUrl:string="./assets/data/employees.json";

constructor(private http:HttpClient){}

getEmployees():Observable<Employee[]>{

return this.http.get<Employee[]>(this.empUrl);

}

* getEmployees() method returns Employee data as a Observable.
* Using get() method of HttpClient class to the json file to fetch the data. HttpClient receives the JSON response as of Employee[] object.
* The http.get<Employee[]>() method will automatically convert the received JSON data from the file to an array of Employee type.

employee-list.component.ts

export class ---{

employees!:Employee[];

constructor(private empSer:EmployeeService)

{}

getList(){

this.empSer.getEmployees().subscribe(

employees=>this.employees=employees)

ngOnInit():void{

this.getList();

}

}

* Invokes the service class method getEmployees() which makes an http call to employees.json file and the response is returned.
* Similar code in employee-details.component.ts

**Introduction to Template Driven Forms**

* Used to create small to medium sized forms.
* Easy to use.
* Minimal component code and more html code.
* Created by writing templates and binding directives and behavior to the templates.
* The template-driven approach makes use of built-in directives to build forms such as,
  + ngModel
  + ngModelGroup
  + ngForm
  + ngSubmit

Syntax:

<form #formRef = "ngForm" (ngSubmit) = "onSubmit()">

<input class="form-control" name="controlname" [(ngModel)]="classVariable" #variable ="ngModel"/>

</form>

* #formRef="ngForm" is a template variable that has an instance of ngForm or Angular from.

* (ngSubmit) = "onSubmit()" : On submitting the form, onSubmit() method gets invoked.

* [(ngModel)] = "classVariable": Binds the data entered in the input field of the form to the class variable specified in the corresponding .ts file

* #variable = "ngModel": Another template variable which is bound to an instance of ngModel. This helps to track the state of the input field by checking if the user touched the control, if the value changed, or if the value became invalid.

* We should make our module dependent on FormsModule.

* ng generate component Register

* Register.component.html

<div class="container">

    <form #registerForm="ngForm" (ngSubmit)="apply()">

        <div class="form-group">

            <label for="uname">Username</label>

            <input type="text" class="form-control" id="uname" required

            pattern="[a-zA-Z][a-zA-Z ]+"

            [(ngModel)]="userName" name="userName" #uname="ngModel"/>

            <div [hidden]="uname.valid||uname.pristine"

            class="alert alert-danger">

            <div [hidden]="!uname.hasError('required')">Name is required</div>

            <div [hidden]="!uname.hasError('pattern')">Only alphabetsallowed</div>

            </div>

        </div>

        <div class="form-group">

            <label for="pwd">Password</label>

            <input type="password" class="form-control" id="pwd" required

            [(ngModel)]="Password" name="Password" #pwd="ngModel"/>

            <div [hidden]="pwd.valid||pwd.pristine"

            class="alert alert-danger">

            Password is required

            </div>

            </div>

            <div class="form-group">

                <button class="btn btn-primary" type="submit"

                [disabled]="!registerForm.valid"> Register

                </button>

            </div>

    </form>

    <div \*ngIf="errorMessage!=null">

        <span class="text-danger">{{errorMessage}}</span>

    </div>

    <div \*ngIf="successMessage!=null">

        <span class="text-success">{{successMessage}}</span>

    </div>

</div>

* Register.component.ts

export class RegisterComponent implements OnInit {

  userName:string="";

  Password:string="";

  errorMessage:any="";

  successMessage:any="";

  apply(){

    this.errorMessage=null;

    this.successMessage=null;

    if(this.userName=='admin' && this.Password=='admin123'){

      this.successMessage="User Logged in successfully";

    }

    else{

      this.errorMessage="User not allowed";

    }

    console.log(this.userName)

    console.log(this.Password)

  }

  constructor() { }

  ngOnInit(): void {

  }

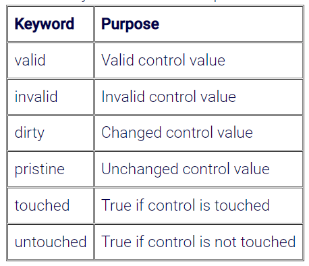
}

**Angular Validations**

* required
* minlength
* maxlength
* pattern
* email

 Angular creates a FormControl for each and every field, which has ngModel directive field applied.

* The FormControl exposes the state of form elements like,



* 2 ways to get reference to the FormControl
  + Use template variable sampleForm
    - sampleForm.controls.fullname.valid
  + Create a new local variable
    - #fname="ngModel"