Contents

Service

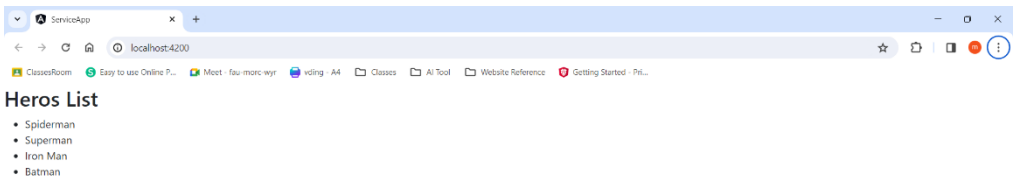
- Lets create a component “HeroNamesComponent”.
- In this component we will be showing the list of hero names.
    - **hero-name.component.ts**

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

@Component({
  selector: 'app-hero-names',
  templateUrl: './hero-names.component.html',
  styleUrls: ['./hero-names.component.css']
})
export class HeroNamesComponent {
  heroDetails = [
    {name: 'Spiderman', cinematicWorld: 'MCU'},
    {name: 'Superman', cinematicWorld: 'DC'},
    {name: 'Iron Man', cinematicWorld: 'MCU'},
    {name: 'Batman', cinematicWorld: 'DC'},
  ]
}
```

    - **hero-name.component.html**

```html
<h2>Heros List</h2>
<ul>
    <li *ngFor="let hero of heroDetails">{{hero.name}}</li>
</ul>
```



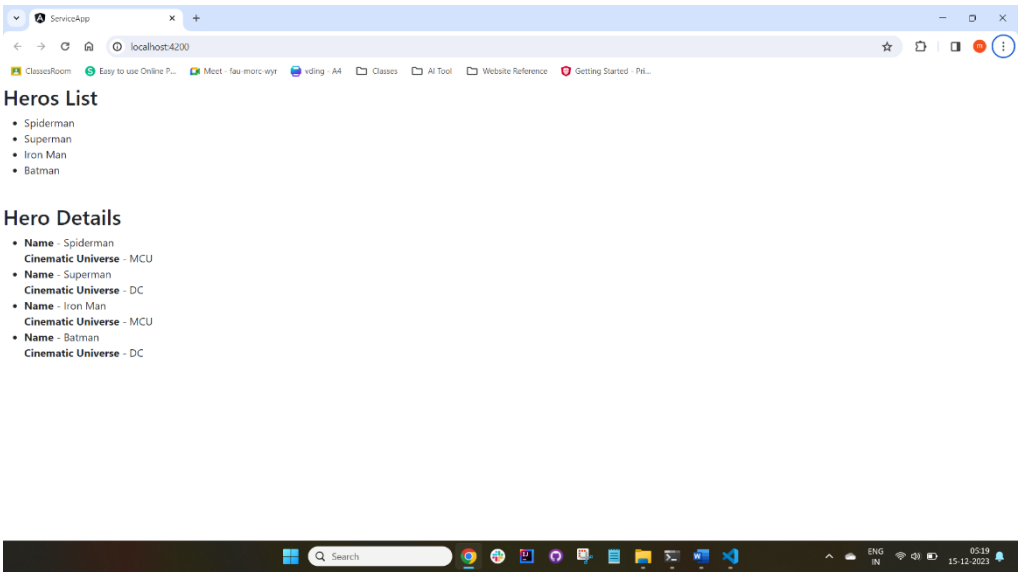
- Now lets create another components “HeroDetailsComponent”
- In this component we will be showing the details of the heros.
    - hero-details.component.ts

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

@Component({
  selector: 'app-hero-details',
  templateUrl: './hero-details.component.html',
  styleUrls: ['./hero-details.component.css']
})
export class HeroDetailsComponent {
  heroDetails = [
    {name: 'Spiderman', cinematicWorld: 'MCU'},
    {name: 'Superman', cinematicWorld: 'DC'},
    {name: 'Iron Man', cinematicWorld: 'MCU'},
    {name: 'Batman', cinematicWorld: 'DC'},
  ]
}
```

    - hero-details.component.html

```html
<h2>Hero Details</h2>
<ul>
    <li *ngFor="let hero of heroDetails">
        <b>Name</b> - {{hero.name}} <br>
        <b>Cinematic Universe</b> - {{hero.cinematicWorld}}
    </li>
</ul>
```



- Now if we see the above 2 code there is an issue that is code reusability.
- If we see in the both the component.ts file there is a common array “heroDetails”.
- Hence we are violating the following principles.

Principles

- Do Not Repeat Yourself (DRY).
- Single responsibility Principle.

To overcome the above principles the better solution is using **“Services”.**

Services

- A class with a specific purpose.
- Why?
  - Share Data.
  - Implement application logic.
  - External Interaction (connection with dB etc..).
- Naming Conventions - .service.ts

Dependency Injection (DI)

- Code Without DI
  - Now lets create 3 classes Engine, Tire, Car
  - Lets assume that a car is build with the help of tire and engine.

```
class Engine{
  constructor(){}
}

class Tires{
  constructor(){}
}

class Car{
  engine: any;
  tires: any;
  constructor(){
    this.engine = new Engine();
    this.tires = new Tires();
  }
}
```

  - Now lets assume Engine evolves and constructor can now accept a new parameter may be like engine type petrol or diesel.

```
class Engine{
  constructor(engineType:any){}
}

class Tires{
  constructor(){}
}
```

```
class Car{
  engine: any;
  tires: any;
  constructor(){
    this.engine = new Engine();
    this.tires = new Tires();
  }
}
```

  - Since we are changing our Engine our car is broken.
  - So to repair our car we need to send a parameter to the engine constructor.
  - Similarly If tire car accepts a new parameter the car needs to be fixed again.

- So the drawback in our code are
  - **Not flexible**
    - If new dependences come or change the car class need to be modified or changes.
  - **Not suitable for testing**
    - How many new cars you create we will be getting of same engine and same set of tire.
    - So we can't able to create a car with different engine or tire or with old, new engine or with old, new tire combination.

- DI as a design pattern
  - DI is coding pattern in which a class received its dependencies from external source rather than creating them itself.
  - Without DI

```
class Car{
  engine: any;
  tires: any;
  constructor(){
    this.engine = new Engine();
    this.tires = new Tires();
  }
}
```

  With DI

```
class Car{
  engine: any;
  tires: any;
  constructor(engine: any,tire: any){
    this.engine = engine;
    this.tires = tire;
  }
}
```

  - If we compare both the code without and with DI we can observe that
    - In without DI car class is creating the dependences.
    - In DI its just consumes the dependences instead of creating. The consumed dependencies are been created externally.
  - Hence by consuming the dependencies both the drawbacks are been resolved.
  - So to create a Car the modified code with DI will be like below.
  - Type 1 – No parameters required for both engine and tire.

```
export class CarComponent {
  engine = new Engine();
  tire = new Tires();
  car = new Car(this.engine, this.tire);
}

class Engine{
  constructor(){}
}

class Tires{
  constructor(){}
}

class Car{
  engine: any;
  tires: any;
  constructor(engine: any,tire: any){
```

```
    this.engine = engine;
    this.tires = tire;
  }
}
```

- Type 2 – Engine evolves and require a parameter engine type.

```
export class CarComponent {
  engine = new Engine("petrol");
  tire = new Tires();
  car = new Car(this.engine, this.tire);
}

class Engine{
  constructor(engineType:any){}
}

class Tires{
  constructor(){}
}

class Car{
  engine: any;
  tires: any;
  constructor(engine: any,tire: any){
    this.engine = engine;
    this.tires = tire;
  }
}
```

- Type 3 – Tire also gets evolves and require tire type parameter.

```
export class CarComponent {
  engine = new Engine("petrol");
  tire = new Tires("flat");
  car = new Car(this.engine, this.tire);
}

class Engine{
  constructor(engineType:any){}
}

class Tires{
  constructor(tireType:any){}
}

class Car{
  engine: any;
  tires: any;
```

```
  constructor(engine: any,tire: any){
    this.engine = engine;
    this.tires = tire;
  }
}
```

- So now we can now even create 1 or more different type of cars.

```
engine1 = new Engine("petrol");
tire1 = new Tires("flat");
car1 = new Car(this.engine1, this.tire1);

engine2 = new Engine("diesel");
tire2 = new Tires("rugged");
car2 = new Car(this.engine2, this.tire2);

car3 = new Car(this.engine1, this.tire2);
car4 = new Car(this.engine2, this.tire1);
```

- Now even dependencies increases there will be no problem just we need to consume in car class and build.

```
engine = new Engine("petrol");
tire = new Tires("flat");
depA = new dependency();
depB = new dependency();
depC = new dependency();
car = new Car(this.engine, this.tire, this.depA, this.depB,
this.depC);
```

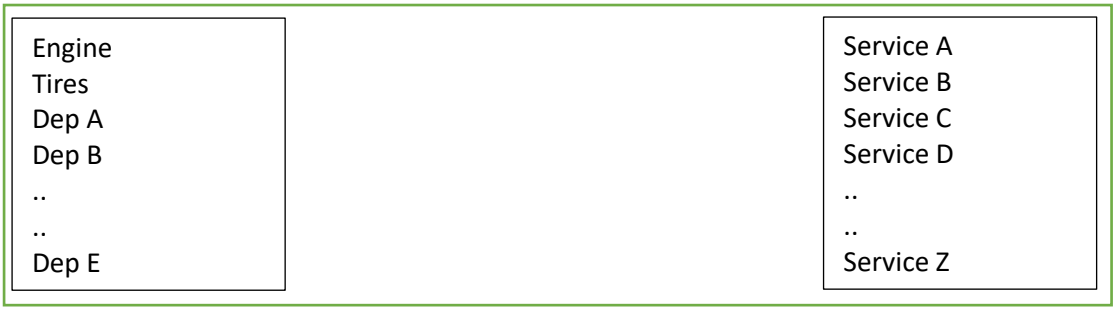
- Even a dependency will be having parameter of another dependency i.e dependency will be dependent on another dependency.

```
engine = new Engine("petrol");
tire = new Tires("flat");
depA = new dependency();
depB = new dependency();
depC = new dependency();
depD = new dependency(depC);
car = new Car(this.engine, this.tire, this.depA, this.depB,
this.depC, this.depD);
```

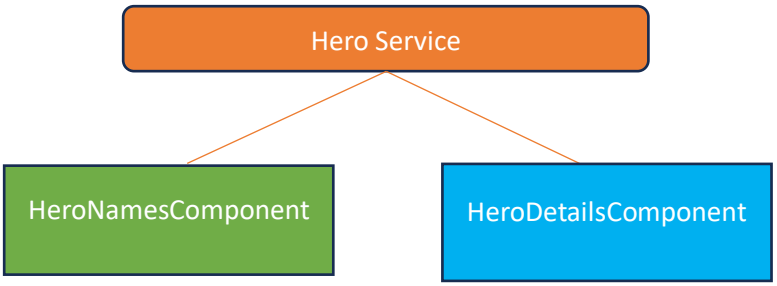
- So due to this will be somewhat difficult to developer to create individual dependencies or dependencies depend on another dependencies.
- This is where angular DI frameworks helps.

- DI as a framework

Injector



  - Steps to apply Injector in our hero code.
    - Define the HeroService class
    - Register with Injector
    - Declare as dependency in HeroNamesComponent and HeroDetailsComponent



Using Services

1. ### Creating HeroService

   - Command to create service
     ng g s <serviceName>
   - After creating the hero service it will be look like below

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

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  constructor() { }
}
```

- Now we will be creating a method getHeros() which return the array of heros.

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

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  constructor() { }

  getHeros(){
    return[
      {name: 'Spiderman', cinematicWorld: 'MCU'},
      {name: 'Superman', cinematicWorld: 'DC'},
      {name: 'Iron Man', cinematicWorld: 'MCU'},
      {name: 'Batman', cinematicWorld: 'DC'},
    ]
  }
}
```

2. ### Register HeroService

   - Now we need to register our service other wise our service will be considered as a class as other classes.
   - To register our service we need to register in provider array in app.module.ts because app.module is parent for all the components in our application so services registered here can be applicable for all components without any restrictions.

3. ### Mention dependency for HeroService in the component

   - As we discussed earlier the dependency is specified in the constructor.

```
heroDetails:any = [];
```

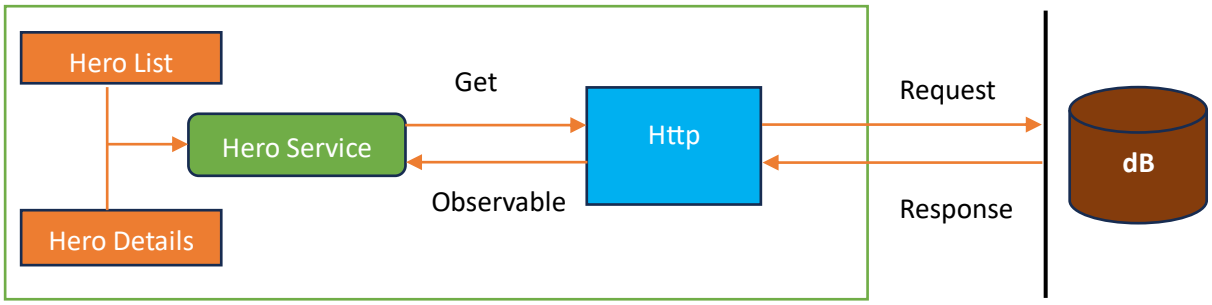
```
constructor(
  private _heroService:HeroService
){}
ngOnInit(): void {
  this.heroDetails = this._heroService.getHeros();
}
```

Injectable decorator

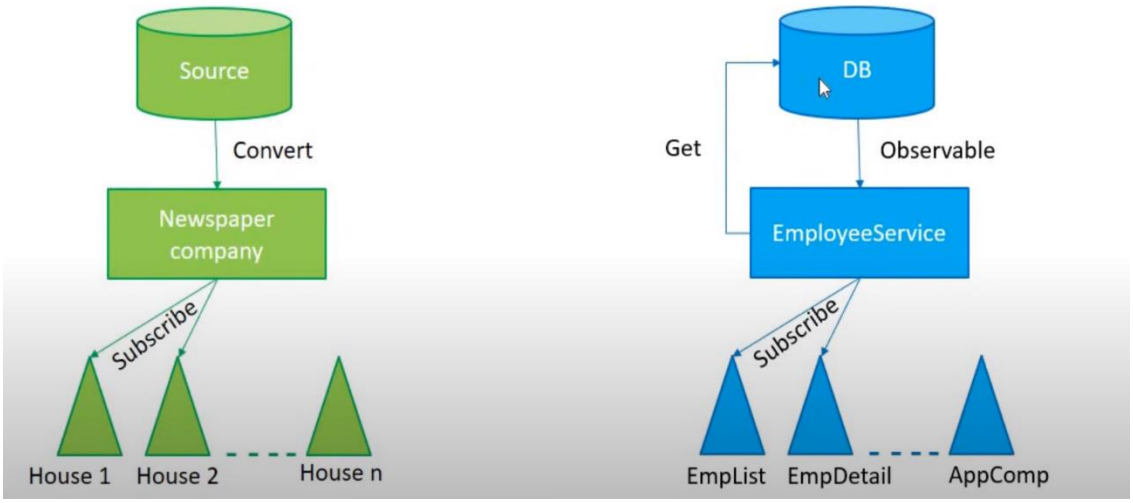
- Injectable decorator tells angular that this service might itself has injected dependencies.
- So, if you ever want to inject a service into another service injectable decorator is must.
- If we observe a component class does not have injectable decorator but still make use of another service well this is because of component decorator **@Component.**
- That lets angular know I might have dependencies and might make use of the dependency injection system.
- Since service do not have component decorator, we will be using **Injectable decorator**.

HTTP and Observables

- Right now, we are hard coding the array in our service.
- But in real world experience we need to fetch the data from web server (dB).
- So now we will replace the hard coded data with HTTP call to the server.
- HTTP Mechanism



- Observables



  - A sequence of items that arrives asynchronously over time
  - Http call – single item
  - Single item – Http response

- Steps to apply http mechanism
  - HTTP, Observales and RXJS
  1. HTTP Get request from service.
  2. Receive the observable and cast it into a variable.
  3. Subscribe to the observable from component
  4. Assign the variable in step 2 to a local variable in a component subscribed in step 3.
  - RXJS
    - Reactive Extension for JavaScript
    - External Library to work with Observables

- Fetch Data using HTTP

1. HTTP get Request from hero.service.ts

    - Need to import **HttpClientModule** in **import array** in **app.module.ts**.

```
import { HttpClientModule } from '@angular/common/http'
imports: [
    HttpClientModule
],
```

    - This module provides a modern, RxJS-based API for making HTTP requests and handling responses.
    - To use http in our service we need to declare it as a dependency in the constructor of our service (**hero.service.ts**).

```
import { HttpClient } from '@angular/common/http';
constructor(
    private http: HttpClient
) { }
```

    - Now we are ready to fetch data using HTTP so in the get function (**getHeros(){}**) lets make the request.

```
getHeros(){
    return this.http.get();
}
```

    - Basically, get request takes in a URL as its argument. Since, we are not using actual web server so we will create a file contains hero data and assume that file as URL.
    - Create a folder of name JSON which contains a .json(**heroDetails.json**) file in assests folder.

```
[
    {"name": "Spiderman", "cinematicWorld": "MCU"},
    {"name": "Superman", "cinematicWorld": "DC"},
    {"name": "Iron Man", "cinematicWorld": "MCU"},
    {"name": "Batman", "cinematicWorld": "DC"}
]
```

- Now in my service (**hero.service.ts**) created a new property _url which contain the path of above json file and that property is given to the get request.

```
export class HeroService {

  private _url:string = "/assets/json/heroDetails.json"

  constructor(
    private http: HttpClient
  ) { }

  getHeros(){
    return this.http.get(this._url);
  }
}
```

- Later, when you have a actual working webserver replace this url with that.

2. **Cast the observable which we receive as response to an hero array**
   - Present if we see our get method (**getHeros(){}**) returns of type observable.
   - But, for our request we but cast this observable to array format of heros.
   - So, first let's create a hero interface (Interface defines the structure of an object. It specifies the properties that the object must have, along with their types).
   - Create a folder called **Models** and create a file **hero.model.ts**

```
export interface HeroModel{
    name?:string,
    cinematicWorld?: string
}
```

   - Now we have a hero type that observable can cast into.
   - Now I will add type to get request in my serve (**hero.service.ts**) that is array of type HeroModel and import HeroModel

```
getHeros(){
    return this.http.get<HeroModel[]>(this._url);
  }
```

   - Now **getHeros()** will return an observable of type hero array.
   - Make sure to import observable from rxjs as well

```
getHeros(): Observable<HeroModel[]>{
    return this.http.get<HeroModel[]>(this._url);
}
```

3. Subscribe Observable in our components
   hero-name.component.ts

```
this._heroService.getHeros().subscribe(
      response => { this.heroDetails = response }
);
```

- HTTP Error Handling
  - To handle exceptions on an observable we make use of catchError & throwError operator.

```
import { HttpClient, HttpErrorResponse } from
'@angular/common/http';
import { Injectable } from '@angular/core';
import { Observable, catchError, throwError } from 'rxjs';
import { HeroModel } from 'src/app/Models/hero.modal';

@Injectable({
  providedIn: 'root'
})
export class HeroService {

  private _url: string = "/assets/json/heroDetails.json";

  constructor(private http: HttpClient) { }

  getHeros(): Observable<HeroModel[]> {
    return this.http.get<HeroModel[]>(this._url)
      .pipe(
        catchError(this.errorHandler)
      );
  }

  errorHandler(error: HttpErrorResponse): Observable<never> {
    // Using throwError with an instance of Error
    return throwError(new Error(error.message || "Server
Error"));
  }
}
```

  - Now add error in the component file

    hero-name.component.ts

```
this._heroService.getHeros().subscribe(
      response => { this.heroDetails = response },
      error => { this.errorMessage = error}
);
```