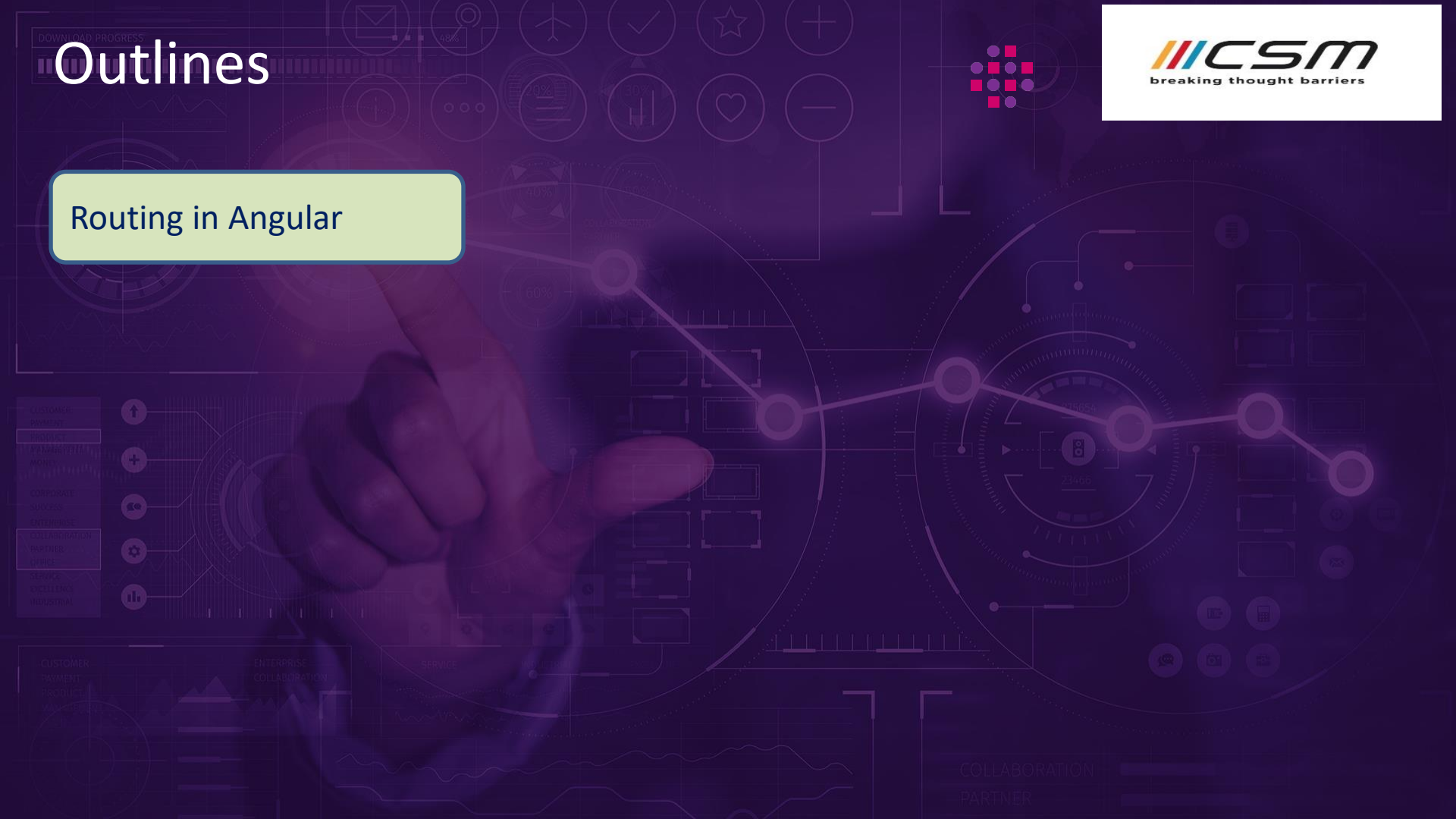
CSM
breaking thought barriers

Angular Training

Session -15

Outlines

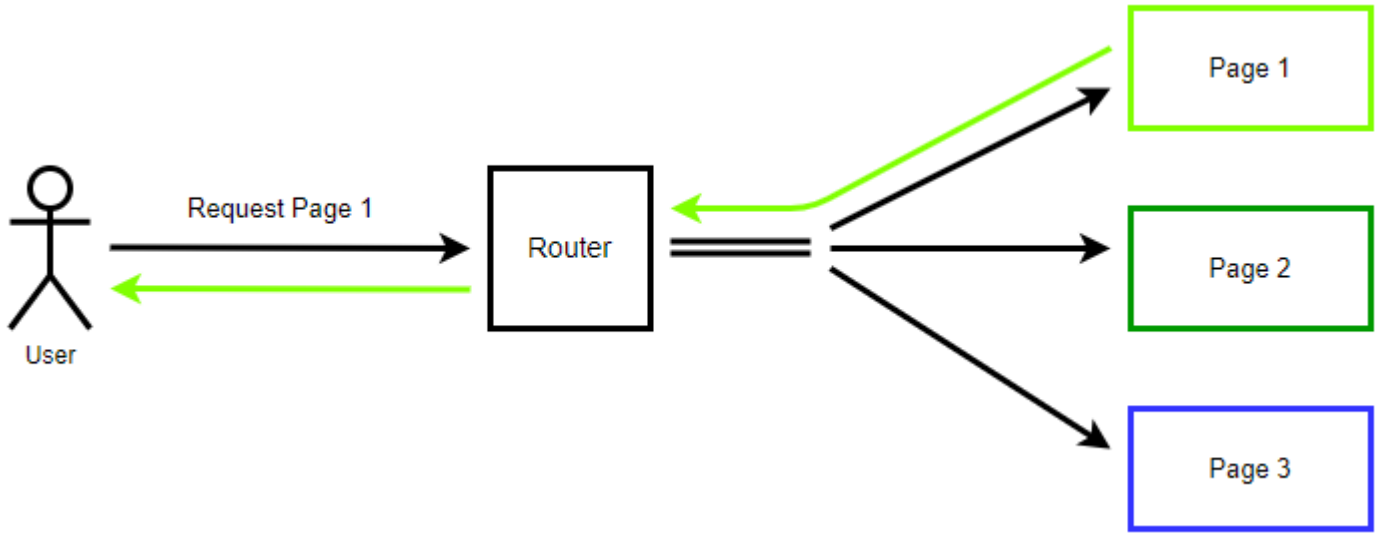


Routing in Angular

What is routing in Angular?

Routing means navigating between different pages.

Routing in Angular allows us to move from one part of the application to another part, or from one view to another view.



In a single-page application, we change what the user sees by showing and hiding parts of the display according to that specific component, rather than going out to the server to load a whole new page.

It makes the Angular applications faster than usual applications.

Routing is an important part of this behaviours that SPA exhibits.

How to do routing ?

1. Create Separate Module for Routing

2. Import Router Module ,Routes In your AppRouting Module

```
import { Routes,RouterModules} from '@angular.router'
```

3. Create a Routes Config

```
const appRoutes :Routes =[
    {path:'home',component :HomeComponent},
    {path:'user-list',component:UserListComponent},
    {path:' ',redirectTo:'/home',pathMatch:'full'}
];
```

4. Call RouterModule.forRoot() and give it the Routes config

5. Export this module into you **RootModule**

```
@NgModule({
      imports :[RouterModule.forRoot(appRoutes)] ,
      exports :[RouterModule]
})
```

6. Place a **<route-outlet></router-outlet>** tag in your template where you to perform it.

7. Place links that will take your user to those Routes and use **routerLink** attribute to give them links.

THE ROUTER-OUTLET

The Router-Outlet is a directive that's available from the router library where the Router inserts the component that gets matched based on the current browser's URL.

You can add multiple outlets in your Angular application which enables you to implement advanced routing scenarios.

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

Any component that gets matched by the Router will render it as a sibling of the Router outlet.

ROUTES AND PATHS

- Routes are definitions (objects) comprised from at least a path and a component (or a redirectTo path) attributes.

- The path refers to the part of the URL that determines a unique view that should be displayed, and component refers to the Angular component that needs to be associated with a path.

- Based on a route definition that we provide (via a static RouterModule.forRoot(routes) method), the Router is able to navigate the user to a specific view.
- Each Route maps a URL path to a component.
- The path can be empty which denotes the default path of an application and it's usually the start of the application.

ROUTES AND PATHS

- The path can take a wildcard string (**). The router will select this route if the requested URL doesn’t match any paths for the defined routes. This can be used for displaying a “Not Found” view or redirecting to a specific view if no match is found.

- **{ path: 'contacts', component: ContactListComponent}**

- If this route definition is provided to the Router configuration, the router will render ContactListComponent when the browser URL for the web application becomes /contacts.

ROUTE MATCHING STRATEGIES

- The Angular Router provides different route matching strategies.
- The default strategy is simply checking if the current browser's URL is prefixed with the path.
- For example our previous route:

```
{ path:  'contacts', component:  ContactListComponent}
```

- Could be also written as:

```
{ path:  'contacts', pathMatch: 'prefix', component:  ContactListComponent}
```

- The patchMath attribute specifies the matching strategy. In this case, it's prefix which is the default.

ROUTE MATCHING STRATEGIES

- The second matching strategy is **full**. When it's specified for a route, the router will check if the the path is exactly **equal** to the path of the current browser's URL:

  { path: 'contacts',pathMatch: 'full', component: ContactListComponent}

ROUTE PARAMS

- Creating routes with parameters is a common feature in web apps.
- Angular Router allows you to access parameters in different ways:

  A. Using the ActivatedRoute service,

  B. Using the ParamMap observable available starting with v4.

- You can create a route parameter using the colon syntax.

{ path: 'contacts/:id', component: ContactDetailComponent}

NAVIGATION DIRECTIVE

The Angular Router provides the routerLink directive to create navigation links.

This directive takes the path associated with the component to navigate to.

For example:

<a [routerLink]="'/contacts'">Contacts

Angular Services

The Angular Services are the piece of code or logic that are used to perform some specific task. A service can contain a value or function or combination of both.

The Services in angular are injected into the application using the dependency injection mechanism.

Why do we need a service in Angular?

Whenever you need to reuse the same data and logic across multiple components of your application, then you need to go for angular service.

The logic or data is implemented in a services and the service can be used across multiple components of your angular application. So, services is a mechanism used to share the functionality between the components.

Without Angular Services, you would have to repeat the same logic or code in multiple components wherever you want this code or logic.

Step1**. Creating Angular Service**

```
ng generate service Student
```

The angular service is composed of three things.

- You need to create an export class
- You need to decorate that class with @Injectable decorator
- You need to import the Injectable decorator from the angular core library.

The syntax to create angular service is

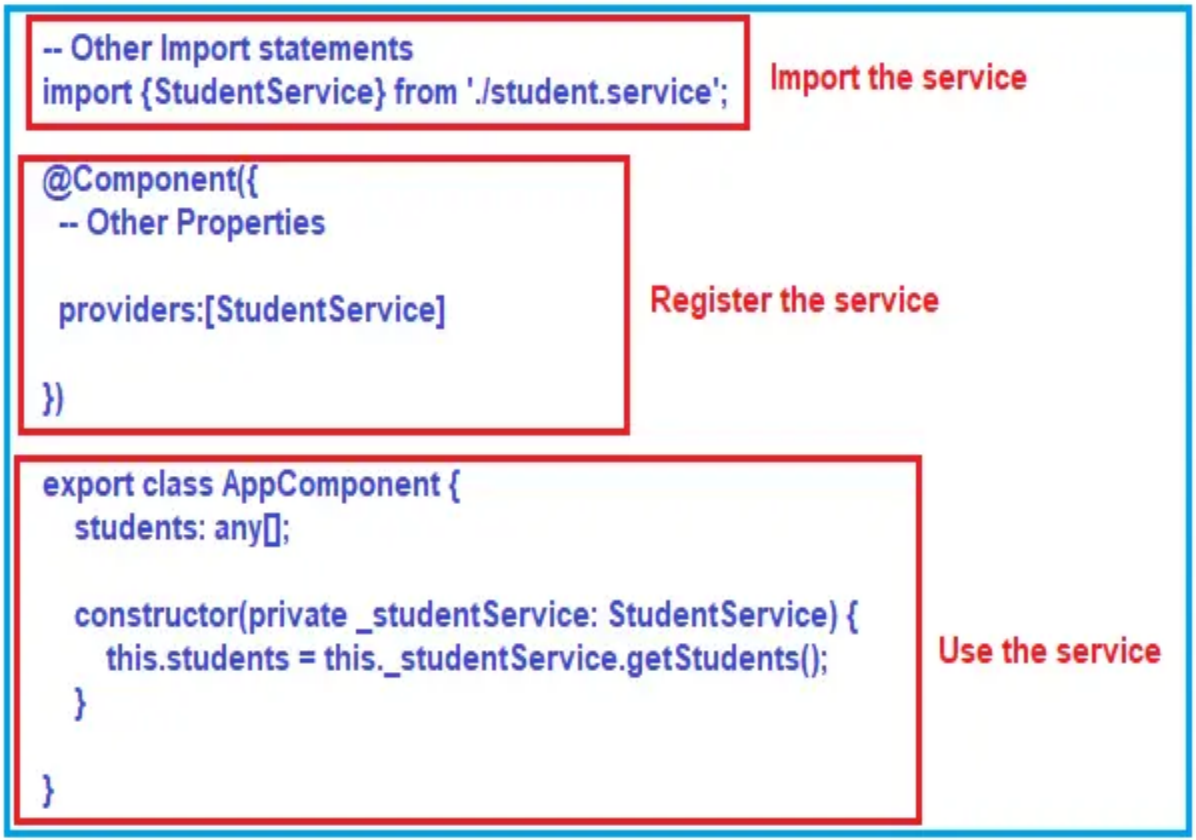
```
import { Injectable } from '@angular/core';

@Injectable()

export class ServiceName {

   //Method and Properties

}
```

```
import { Injectable } from '@angular/core';
@Injectable()
export class StudentService {
  getStudents(): any[] {
    return [
     {
       ID: 'std101', FirstName: 'Preety', LastName: 'Tiwary',
       Branch: 'CSE', DOB: '29/02/1988', Gender: 'Female'
    },
     {
       ID: 'std103', FirstName: 'Priyanka', LastName: 'Dewangan',
       Branch: 'CSE', DOB: '24/07/1992', Gender: 'Female'
    },
    {
       ID: 'std104', FirstName: 'Hina', LastName: 'Sharma',
       Branch: 'ETC', DOB: '19/08/1990', Gender: 'Female'
    },
    {
       ID: 'std105', FirstName: 'Sambit', LastName: 'Satapathy',
       Branch: 'CSE', DOB: '12/94/1991', Gender: 'Male'
    }
    ];
  }
}
```

Note: The @Injectable() decorator in angular is used to inject other dependencies into the service. At the moment our service does not have any other dependencies, so, yo can remove the @Injectable() decorator and the service should works. However, the Angular Team recommends to always use @Injectable()

Step2: Using the Service in Angular

It is a three step process.
a. Import the service
b. register the service
c. use the service.



Using ngOnInit() life cycle hook to call the getStudents() service Method

```
import { Component} from '@angular/core';
import {StudentService} from './student.service';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers:[StudentService]
})
export class AppComponent {
    students: any[];

    constructor(private _studentService: StudentService) { }

    ngOnInit() {
        this.students = this._studentService.getStudents();
    }
}
```

Difference between constructor and ngOnInit

Whenever you create an instance of a class, the class constructor is automatically called. Like other programming languages, the class constructor in angular is also used to initialize the members of the class and it's sub classes.

The ngOnInit is a life cycle hook method provided by Angular which is called after the constructor and is generally used to perform tasks related to Angular bindings.

For example, ngOnInit is the right place to call a service method to fetch data from a remote server.

We can also do the same using a class constructor, but the general rule of thumb is, tasks that are time consuming should use ngOnInit instead of the constructor. As fetching data from a remote server is time consuming, the better place for calling the service method is ngOnInit.

Dependency Injection in Angular

In software engineering, dependency injection is a technique whereby one object supplies the dependencies of another object. In other words, we can say that, it is a coding pattern in which classes receives their dependencies from external sources rather than creating them itself.

Dependency Injection is the heart of Angular Applications. The Dependency Injection in Angular is a combination of two terms i.e. Dependency and Injection

Dependency: Dependency is an object or service that is going to be used by another object.

Injections: It is a process of passing the dependency object to the dependent object. It creates a new instance of the class along with its require dependencies.

```
import { Injectable } from '@angular/core';
@Injectable()
export class StudentService {
  getTitle()
  {
    return "Dependency Injection in Angular";
  }
  getStudents(): any[] {
    return [
      {
        ID: 'std101', FirstName: 'Preety', LastName: 'Tiwary',
        Branch: 'CSE', DOB: '29/02/1988', Gender: 'Female'
    },
    {
        ID: 'std103', FirstName: 'Priyanka', LastName: 'Dewangan',
        Branch: 'CSE', DOB: '24/07/1992', Gender: 'Female'
    },
    ];
  }
}
```

```
import { Component} from '@angular/core';
import {StudentService} from './student.service';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],
  providers:[StudentService]
})
export class AppComponent {
    students: any[];
    pageTitle: string;
    private _studentService: StudentService;
    constructor(studentService: StudentService) {
        this._studentService = studentService;
     }
    ngOnInit() {
        this.students = this._studentService.getStudents();
        this.pageTitle = this._studentService.getTitle();
    }
}
```

Who is creating and providing the instance to the constructor?

The answer is Angular Injector. When an instance of AppComponent class is created, the angular injector creates an instance of the StudentService class and provides it to the AppComponent constructor.

The constructor then assigns that instance to the private field _studentService. We then use this private field _studentService to call the StudentService methods getStudents() and getTitle()

How does the angular injector knows about StudentService?

For the Angular injector to be able to create and provide an instance of StudentService , first we need to register the StudentService with the Angular Injector.

We register a service with the angular injector by using the providers property of @Component decorator or @NgModule decorator.

At Component Level:

```
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.css'],

  //Registering Dependency at Component Level
  providers:[StudentService]

})
```

At Module Level:

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

  //Register Dependency at App Level
  providers: [StudentService],

  bootstrap: [AppComponent]
})
export class AppModule { }
```

Why should we use dependency injection in Angular

```
export class Computer {
  private processor: Processor;

  constructor() {
    this.processor = new Processor();
  }
}
```

```
export class Processor {

  constructor() {
  }
}
```

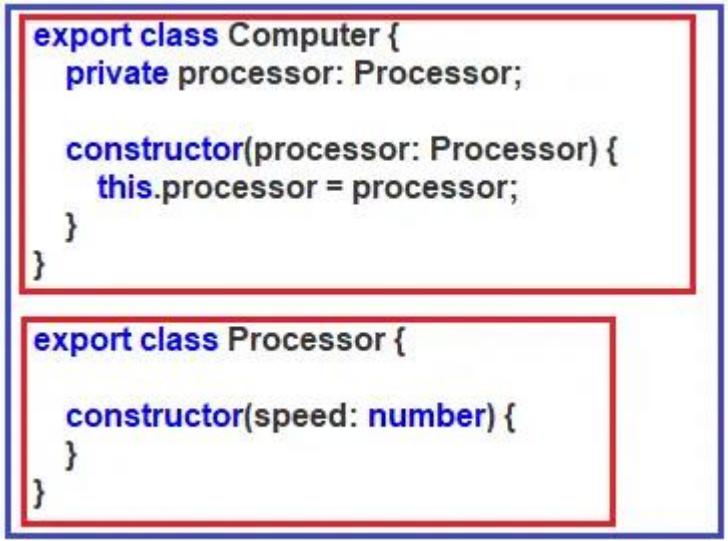
Problems

1. This code is difficult to maintain over time

2. Instances of dependencies created by a class that needs those dependencies are local to the class and cannot share data and logic.

3. Hard to unit test

```
export class Processor {

  constructor(speed: number) {
  }
}
```



Instances of dependencies created by a class that needs those dependencies are local to the class and cannot share data and logic.

Advantages of Dependency Injection in Angular?

1. Create applications that are easy to write and maintain over time as the application evolves

2. Easy to share data and functionality as the angular injector provides a Singleton i.e a single instance of the service

3. Easy to write and maintain unit tests as the dependencies can be mocked