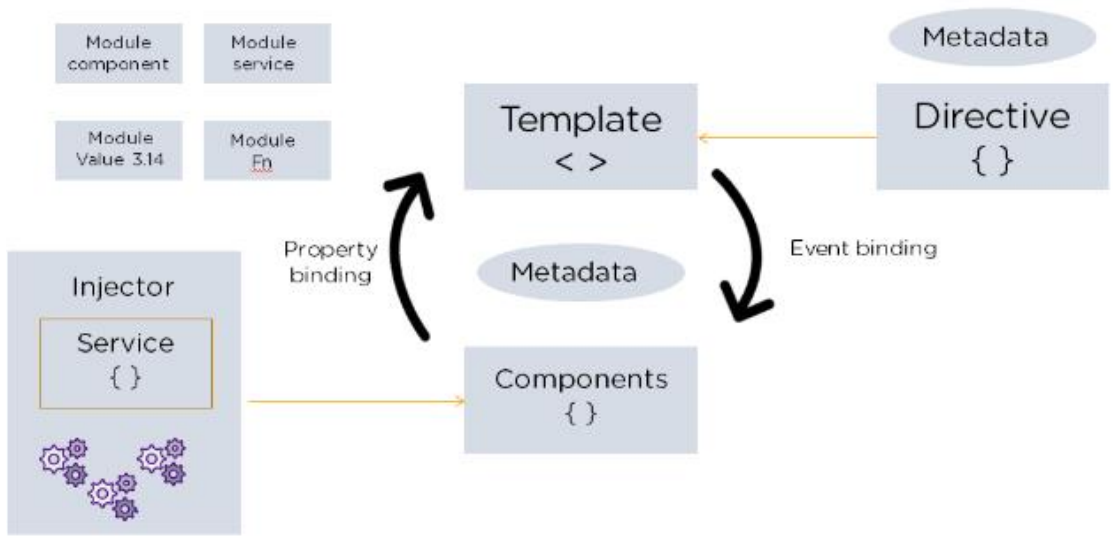
1. What Is Angular?

Angular is an open-source, JavaScript framework written in TypeScript. Google maintains it, and its primary purpose is to develop single-page applications. As a framework, Angular has clear advantages while also providing a standard structure for developers to work with. It enables users to create large applications in a maintainable manner.

2. Angular Architecture



The Eight main building blocks of an Angular application:

1. Modules
2. Components
3. Templates
4. Metadata
5. Data binding
6. Directives
7. Services
8. Dependency injection

Modules: Every Angular app has a root module, conventionally named AppModule, which provides the bootstrap mechanism that launches the application. An app typically contains many functional modules.

```
// app.module.ts
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

If we want to use another custom Angular module, then we need to register that module inside the app.module.ts file. Organizing your code into distinct functional modules helps in managing the development of complex applications, and in designing for re usability.

Components: Every Angular project has at least one component, the root component and root component connects the component hierarchy with a page document object model (DOM). Each component defines the class that contains application data and logic, and it is associated with the HTML template that defines the view to be displayed in a target app.A component controls a patch of screen called a view.
The @Component decorator identifies the class immediately below it as the component and provides the template and related component-specific metadata.

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

Templates: The angular template combines the HTML with Angular markup that can modify HTML elements before they are displayed. Template

directives provide program logic, and binding markup connects your application data and the DOM.
There are two types of data binding.

- Event binding lets your app respond to user input in the target environment by updating your application data.
- Property binding lets you interpolate values that are computed from your application data into the HTML.

```
<div style="text-align:center">
  <h1>
    {{2 | power: 5}}
  </h1>
</div>
```

In the above HTML file, we have used a template. We have also used the pipe inside the template to transform the values to the desired output.

Metadata: Metadata tells Angular how to process a class.It is used to decorate the class so that it can configure the expected behavior of a class. Decorators are the core concept when developing with Angular (versions 2 and above). The user can use metadata to a class to tell Angular app that AppComponent is the component. Metadata can be attached to the TypeScript using the decorator.

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

Here is the @Component decorator, which identifies the class immediately below it as a component class.
The @Component decorator takes a required configuration object with the information Angular needs to create and present the component and its view.
Here are a few of the most useful @Component configuration options:
selector: CSS selector that tells Angular to create and insert an instance of this component where it finds a <app-root>tag which is Parent Component.
templateUrl: module-relative address of this component's HTML template.

providers: array of dependency injection providers for services that the component requires.
The metadata in the @Component tells Angular where to get the major building blocks you specify for the component.
The template, metadata, and component together describe a view.
Apply other metadata decorators in a similar fashion to guide Angular behavior. @Injectable, @Input, and @Output are a few of the more popular decorators.

Data binding: Data binding plays an important role in communication between a template and its component.
Data binding is also important for communication between parent and child components. Angular allows defining communication between a component and the DOM, making it very easy to define interactive applications without worrying about pulling and pushing the data.
From the Component to the DOM
Interpolation: {{ value }}: Interpolation adds the value of the property from the component.
<p>Name: {{ student.name }}</p>
<p>College: {{ student.college }}</p>
Property binding: [property]="value"
With property binding, a value is passed from a component to a specified property, which can often be a simple html attribute.
<input type="text" [value]="student.name" />
<input type="text" [value]="student.college" />
Event binding: (Event)="myFunction($event)"
With Event binding, a value is passed from a child Component to Parent Component,which can often be a simple html attribute.
<input type="text" (myEvent)="onClick($event)" />
Two-Way binding: [(ngModel)]="property"
It is an important fourth form that combines property and event binding in a single notation, using the ngModel directive.
<input [(ngModel)]="hero.name">

Directives: An Angular component isn't more than a directive with the template. When we say that components are the building blocks of Angular applications, we are saying that directives are the building blocks of Angular projects. Let us use built-in Angular directive like ngClass, which is a better example of the existing Angular attribute directive.

```
<p [ngClass]="{'coffee'=true, 'red'=false}">
   Angular 7 Directives Example
</p>
<style>
   .coffee{color: coffee}
   .red{color: red}
</style>
```

Here, based on the [ngClass] directive's value, the text has color. In our example, the text will be coffee because it is true.

Services: For data or logic that isn't associated with a specific view, and that you want to share across components, you create a service class. The @Injectable decorator immediately precedes the service class definition. The decorator provides the metadata that allows your service to be injected into client components as a dependency. Angular distinguishes components from services to increase modularity and reusability. By separating a component's view-related functionality from other kinds of processing, you can make your component classes lean and efficient.

Dependency injection : Dependency injection (DI) lets you keep your component classes lean and efficient. DI does not fetch data from a server, validate the user input, or log directly to the console instead they delegate such tasks to the services. DI is wired into a Angular framework and used everywhere to provide new components with the services or other things they need. Components consume services; that is, you can inject a service into a component, giving the component access to that service class.

3. Difference between Angular and AngularJS

Category	**Angular JS**	**Angular**
Architecture	It supports the Model-View-Controller design. The view processes the information	It uses components and directives. Components are the directives with a

Category	Angular JS	Angular
	available in the model to generate the output.	template.
Written Language	Written in JavaScript.	Written in Microsoft's TypeScript language, which is a superset of ECMAScript 6 (ES6).
Mobile support	It does not support mobile browsers.	Angular is supported by all the popular mobile browsers.
Expression Syntax	**ng-bind** is used to bind data from view to model and vice versa.	Properties enclosed in "()" and "[]" are used to bind data between view and model.
Dependency Injection	It does not use Dependency Injection.	Angular is supported by all the popular mobile browsers.
Routing	AngularJS uses $routeprovider.when() for routing configuration.	Angular uses @Route Config{(…)} for routing configuration.
Structure	It is less manageable in comparison to Angular.	It has a better structure compared to AngularJS, easier to create and maintain large applications but behind in AngularJS in the case of small applications.

4. Difference between component and directive

Component	Directive

Component	Directive
A component is a directive used to shadow DOM to create and encapsulate visual behavior called components. They are typically used to create UI widgets.	A Directive is usually used while adding behavior to an existing DOM element.
For registering a component, we use @Component metadata annotation attributes.	For registering directives, we use the @Directive meta-data annotation attribute.
It is also used to break up the application into smaller components.	It is mainly used to design re-usable components.
Only one component is allowed to be present per DOM element.	Multiple directives can be used in a per DOM element.
@View decorator or template URL template is mandatory in a component.	A Directive does not have View
A component is used to define pipes.	You can't define Pipes in a directive.

5. Data Binding in Angular

Data binding is a core concept in Angular and allows to define communication between a component and the DOM, making it very easy to define interactive applications without worrying about pushing and pulling data. There are four forms of data binding and they differ in the way the data is flowing.

From the Component to the DOM

Interpolation: {{ value }}

This adds the value of a property from the component:

```
<li>Name: {{ user.name }}</li>
<li>Email: {{ user.email }}</li>
```

Property binding: [property]="value"

With property binding, the value is passed from the component to the specified property, which can often be a simple html attribute:
<input type="email" [value]="user.email">
<div [style.background-color]="selectedColor">

From the DOM to the Component
Event binding: (event)="function"
When a specific DOM event happens (eg.: click, change, keyup), call the specified specified method in the component. In the example below, the cookPotato() method from the component is called when the button is clicked:
<button (click)="cookPotato()"></button>

Two-way
Two-way data binding: [(ngModel)]="value"
Using what's called the banana in a box syntax, two-way data binding allows to have the data flow both ways. In this example, the user.email data property is used as the value for the input, but if the user changes the value, the component property gets updated automatically to the new value:
<input type="email" [(ngModel)]="user.email">

6. What are Observables?

An Observable is capable of delivering multiple values over time – it's like streaming. It can be canceled or, in case of errors, easily retried. Numerous Observables can be combined, or there can be a race to have only the first used. The RxJS framework for Observables is a mighty one.
Furthermore, Observables need to have a subscription to start emitting values. Otherwise, they are just a blueprint of code handling future emits. If you create your own "producer" of values, you can react on subscribe and unsubscribe. You can then start, stop, or tear down the emit of values.
The following example creates an observable

```
obs = new Observable((observer) => {
 console.log("Observable starts")
 observer.next("1")
})
```

Now, use subscribe() function for getting continuous updates

```
this.obs.subscribe(val => {
        console.log(val); //next callback
  },   error => {
        console.log('error');  //error callback
  },   () => {
        console.log('Completed');  //complete callback
  });
```

7. What are Promise?

The promise is used to make asynchronous programming. The promise can be used when we want to handle multiple tasks at the same time. By the use of TypeScript promise, we can skip the current operation and move to the next line of the code. Promise provides the feature for asynchronous programming or parallel programming, which allows the number of tasks to be executed simultaneously at the same time. In the coming section, we will discuss more the promise in detail for better understanding.

1. new Promise(function(resolve, reject){
// logic goes here ..
});

In the above line of code, we are using the 'new' keyword to create the instance of the promise. As we already know that this is an object available in TypeScript. Also, it has one inner function, which has two parameters named 'reject' and 'resolve'. Inside this function, we can pass a callback function.

2. Return type: It has two parameters inside the inner function. If the function's response is a success, then it will return 'resolve'; if the response from the function is not successful, it will return 'reject'.

3. States available in promise of Typescript: Promise support several states. These states are used to get the status of the function. We have three different states available in the promise;

reject: If the response from the promise function fails, then the state would be 'reject'.

pending: We the response does not come, and we are waiting for the result, then the state would be 'pending'.

fulfilled: If the response forms the promise in TypeScript is received successfully, then the state would be 'fullfeed'.

```
var mypromise = new Promise((resolve, reject) => {
console.log("Demo to show promise in Typescript !!");
resolve(100);
});
mypromise.then((val) => val + 200)
.then((val) => console.log("Value form the promse function is " + val)
.catch((err) => console.log("inside error block " + err)));
```

8. **Difference between observable and promise**

Observables	**Promises**
Emit multiple values over a period of time.	Emit a single value at a time.
Are lazy: they're not executed until we subscribe to them using the subscribe() method.	Are not lazy: execute immediately after creation.
Have subscriptions that are cancellable using the unsubscribe() method, which stops the listener from receiving further values.	Are not cancellable.
Provide the map for forEach, filter, reduce, retry, and retryWhen operators.	Don't provide any operations.
Deliver errors to the subscribers.	Push errors to the child promises.

Now let's see code snippets / examples of a few operations defined by observables and promises.

Operations	**Observables**	**Promises**

Creation	const obs = new Observable((observer) => {
	observer.next(10);
	}) ;
		const promise = new Promise(() => {
		resolve(10);
		});

Transform	Obs.pipe(map(value) => value * 2);	promise.then((value) => value * 2);

Subscribe	const sub = obs.subscribe((value) => {
	console.log(value)
	});
		promise.then((value) => {
		console.log(value)
		});

Unsubscribe	sub.unsubscribe();	Can't unsubscribe

9. Hot and Cold Observables

Cold Observables : When the data is produced by the Observable itself, we call it a cold Observable,Observables are lazy. Observables are lazy in the sense that they only execute values when something subscribes to it. For each subscriber the Observable starts a new execution, resulting in the fact that the data is not shared. If your Observable produces a lot of different values it can happen that two Observables that subscribe at more or less the same receive two different values. We call this behaviour "**unicasting**". To demonstrate this:

```
import * as Rx from "rxjs";
const observable = Rx.Observable.create((observer) => {
    observer.next(Math.random());
});
// subscription 1
observable.subscribe((data) => {
  console.log(data); // 0.24957144215097515 (random number)
});
// subscription 2
observable.subscribe((data) => {
   console.log(data); // 0.004617340049055896 (random number)
});
```

As you see the data is produced inside the Observable, making it cold. We have two subscriptions which subscribe more or less at the same time. Since the Observable does a new execution for every subscriber and the Observable generates a random number, the data the subscriber receives is different. This is not a bad thing, you just have to be aware of this behaviour.
Of course this behaviour is not always desirable. Luckily it's easy to change this behaviour:

```
import * as Rx from "rxjs";
const random = Math.random()
const observable = Rx.Observable.create((observer) => {
    observer.next(random);
});
// subscription 1
observable.subscribe((data) => {
  console.log(data); // 0.11208711666917925 (random number)
});
// subscription 2
observable.subscribe((data) => {
   console.log(data); // 0.11208711666917925 (random number)
});
```

All we did was moving the data producer out of the Observable. We still have two subscribers and the Observable will still execute two times, but since the data is produced outside the Observable our subscriptions will receive the same data.

Hot Observables: When the data is produced outside the Observable, we call it a hot Observable.hot Observable is able to share data between multiple subscribers. We call this behaviour “**multicasting**”.Generating a random number is not a good real life usecase. A good usecase would be DOM events. Let’s say we’re tracking clicking behaviour and have multiple subscribers do something with the coordinates:

```
import * as Rx from "rxjs";
const observable = Rx.Observable.fromEvent(document, 'click');
// subscription 1
observable.subscribe((event) => {
  console.log(event.clientX); // x position of click
});
// subscription 2
observable.subscribe((event) => {
   console.log(event.clientY); // y position of click
});
```

The data is produced outside of the Observable itself. Which makes it hot, because the data is being created regardless of if there is a subscriber or not. If there is no subscriber when the data is being produced, the data is simply lost.

10. lifecycle hooks in angular

ngOnChanges: Invoked every time there is a change in one of th input properties of the component.
ngOnInit: Invoked when given component has been initialized. This hook is only called once after the first ngOnChanges
ngDoCheck: Invoked when the change detector of the given component is invoked. It allows us to implement our own change detection algorithm for the given component.
ngAfterContentInit: Invoked after Angular performs any content projection into the component’s view (see the previous lecture on Content Projection for more info).

ngAfterContentChecked: Invoked each time the content of the given component has been checked by the change detection mechanism of Angular.
ngAfterViewInit: Invoked when the component's view has been fully initialized.
ngAfterViewChecked: Invoked each time the view of the given component has been checked by the change detection mechanism of Angular.
ngOnDestroy: This method will be invoked just before Angular destroys the component. Use this hook to unsubscribe observables and detach event handlers to avoid memory leaks.

11. Pipes

A pipe takes in data as input and transforms it to the desired output.
Pure pipes: Pure pipes in Angular (which is also default) are executed only when Angular detects a change to the input value.
Impure pipes: An impure pipe is called for every change detection cycle which could slow down your app drastically so be very careful with implementing an impure pipe in Angular.
built-in pipes:
Lowercasepipe
Uppercasepipe
Datepipe
Currencypipe
Jsonpipe
Percentpipe
Decimalpipe
Slicepipe

12. AsyncPipe

Generally, If we are getting the result using observable or promise, We need to use following steps,

1. Subscribe the observable or promise.
2. Wait for a callback.
3. Once a result is received, store the result of the callback in a variable.
4. the last step is to bind that variable on the template.

Using AsyncPipe we can use promises and observables directly in our template, without subscribing the observable and storing the result on an

intermediate property or variable. The AsyncPipe internally subscribes to an Observable or Promise and returns the latest value it has emitted. When a new value is emitted, the AsyncPipe marks the component to be checked for changes.
When the component gets destroyed, the AsyncPipe unsubscribes automatically to avoid potential memory leaks.
Example :

```
<div class="card">
    <div class="card-body">
      <h4 class="card-title">Async Pipe</h4>
      <p ngNonBindable>{{time | async}}</p>
      <p>{{time | async}}</p>
    </div>
</div>
```

time is the observable created in a component.

```
time = new Observable<string>((observer: Subscriber<string>) => {
   setInterval(() => observer.next(new Date().toString()), 1000);
});
```

13. How To Create Custom Pipes?

1. Angular Pipes are the class with @Pipe decorator
   ```
   ng generate pipe
   import { Pipe } from '@angular/core';
   @Pipe({
     name: 'filter'
   })
   export class FilterPipe {
   }
   ```
   name in @Pipe metadata is used to represent this pipe on HTML template.
2. Now implement PipeTransform interface from @angular/core package, which comes with transform() abstract method.
   ```
   import { Pipe, PipeTransform } from '@angular/core';
   @Pipe({
     name: 'filter'
   })
   export class FilterPipe implements PipeTransform {
     transform(value: any, ...args: any[]) {
   ```

```
    throw new Error("Method not implemented.");
  }
}
```

transform() method has one mandatory parameter, which is the input value, and other optional parameters. we can change this method, as per our requirement.

3. Now update the transform() method :

```
import { Pipe, PipeTransform } from '@angular/core';
@Pipe({
  name: 'filter'
})
export class FilterPipe implements PipeTransform {
  transform(array: string[], startWith: string): any {
    let temp: string[] = [];
    temp = array.filter(a => a.startsWith(startWith));
    return temp;
  }
}
```

Here I want to filter string from the list, so input is a string array, and startWith is one parameter.

4. Now our pipe is ready to use, but before that, we need to register it in @NgModule declarations section.

```
import { FilterPipe } from './filter.pipe';
@NgModule({
  declarations: [
    ...
    FilterPipe
  ],
    ...
})
export class AppModule { }
```

PipeTransform is not mandatory to implement. but the transform method is essential to a pipe. The PipeTransform interface defines that method and guides both tooling and the compiler. Technically, it’s optional; Angular looks for and executes the transform method regardless.

14. What is Angular Directive?

The Angular directive helps us to manipulate the DOM. You can change the appearance, behavior, or layout of a DOM element using the Directives. They help you to extend HTML

There are three kinds of directives in Angular:

- Component Directive
- Structural directives
- Attribute directives

Component Directive: Components are special directives in Angular. They are the directive with a template.

Structural Directives: Structural directives can change the DOM layout by adding and removing DOM elements. All structural Directives are preceded by Asterix symbol

ngFor

The ngFor is an Angular structural directive, which repeats a portion of the HTML template once per each item from an iterable list (Collection). The ngFor is similar to ngRepeat in AngularJS

```
<tr *ngFor="let customer of customers;">
   <td>{{customer.customerNo}}</td>
   <td>{{customer.name}}</td>
   <td>{{customer.address}}</td>
   <td>{{customer.city}}</td>
   <td>{{customer.state}}</td>
</tr>
```

ngSwitch

The ngSwitch directive lets you add/remove HTML elements depending on a match expression. ngSwitch directive used along with ngSwitchCase and ngSwitchDefault

```
<div [ngSwitch]="Switch_Expression">
   <div *ngSwitchCase="MatchExpression1”> First Template</div>
   <div *ngSwitchCase="MatchExpression2">Second template</div>
   <div *ngSwitchCase="MatchExpression3">Third Template</div>
   <div *ngSwitchCase="MatchExpression4">Third Template</div>
   <div *ngSwitchDefault?>Default Template</div>
</div>
```

ngIf

The ngIf Directives is used to add or remove HTML elements based on an expression. The expression must return a boolean value. If the expression is false then the element is removed, else the element is inserted

```
<div *ngIf="condition">
  This is shown if condition is true
</div>
```

Attribute Directives: An Attribute or style directive can change the appearance or behavior of an element.

ngModel

The ngModel directive is used the achieve the two-way data binding. We have covered ngModel directive in Data Binding in Angular Tutorial

ngClass

The ngClass is used to add or remove the CSS classes from an HTML element. Using the ngClass one can create dynamic styles in HTML pages

```
<div [ngClass]="'first second'">...</div>
```

ngStyle

ngStyle is used to change the multiple style properties of our HTML elements. We can also bind these properties to values that can be updated by the user or our components.

```
<div [ngStyle]="{'color': 'blue', 'font-size': '24px', 'font-weight': 'bold'}">
  some text
</div>
```

15. How to Create & Use Custom Directive In Angular

```
import { Directive, ElementRef, Input, OnInit } from '@angular/core'
@Directive({
 selector: '[ttClass]',
})
export class ttClassDirective implements OnInit {
 @Input() ttClass: string;
 constructor(private el: ElementRef) {
 }
 ngOnInit() {
   this.el.nativeElement.classList.add(this.ttClass);
 }
}
```

<button [ttClass]="'blue'">Click Me</button>

16. What is Bootstrapping in Angular?

Bootstrapping is a technique of initializing or loading our Angular application.

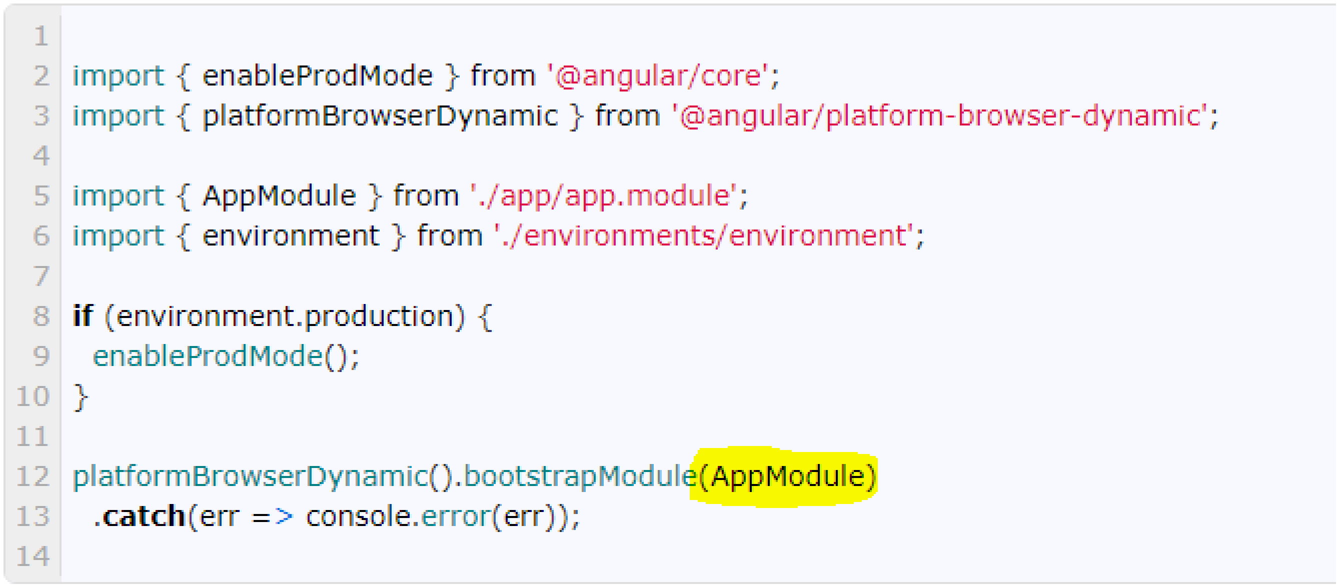
The Angular takes the following steps to load our first view.

- Index.html loads
- Angular, Third-party libraries & Application loads
- Main.ts the application entry point
- Root Module
- Root Component
- Template

Index.html is usually the first page to load, So when index.html is loaded, the Angular core libraries, third-party libraries are loaded.

The Angular finds out the entry point from the configuration file angular.json. This file is located in the root folder of the project. The entry point of our application is main.ts. You will find it under the src folder

The main.ts file is as shown below.



```

import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';

import { AppModule } from './app/app.module';
import { environment } from './environments/environment';

if (environment.production) {
  enableProdMode();
}

platformBrowserDynamic().bootstrapModule(AppModule)
  .catch(err => console.error(err));

```

The angular bootstrapper loads our root module AppModule. The AppModule is located under the folder src/app. The code of our Root module is shown below

```

import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';

import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

The root module must have at least one root component. The root component is loaded, when the module is loaded by the Angular.
In our example, AppComponent is our root component. Hence we import it.

```

import { AppComponent } from './app.component';

```

We use @NgModule class decorator to define a Angular Module and provide metadata about the Modules.

```

@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }

```

The @NgModule has several metadata properties.
Imports: We need to list all the external modules required including other Angular modules, that is used by this Angular Module
Declarations: The Declarations array contains the list of components, directives, & pipes that belong to this Angular Module. We have only one component in our application AppComponent.
Providers: The Providers array, is where we register the services we create. The Angular Dependency injection framework injects these services in components, directives. pipes and other services.
Bootstrap: The component that angular should load, when this Angular Module loads. The component must be part of this module. We want AppComponent load when AppModule loads, hence we list it here.
The Angular reads the bootstrap metadata and loads the AppComponent

17. What is Routing in angular?

Routing basically means navigating between pages. You have seen many sites with links that direct you to a new page. Let us now create a component and see how to use routing with it. In the main parent component app.module.ts, we have to now include the router module as shown below –
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { RouterModule} from '@angular/router';

```
import { AppComponent } from './app.component';
import { NewCmpComponent } from './new-cmp/new-cmp.component';
import { ChangeTextDirective } from './change-text.directive';
import { SqrtPipe } from './app.sqrt';
@NgModule({
  declarations: [
    SqrtPipe,
    AppComponent,
    NewCmpComponent,
    ChangeTextDirective
  ],
  imports: [
    BrowserModule,
    RouterModule.forRoot([
      {
        path: 'new-cmp',
        component: NewCmpComponent
      }
    ])
  ],
  providers: [],
  bootstrap: [AppComponent]
})
export class AppModule { }
import { RouterModule} from '@angular/router'
```

Here, the RouterModule is imported from angular/router. The module is included in the imports as shown below −

```
RouterModule.forRoot([
  {
    path: 'new-cmp',
    component: NewCmpComponent
  }
])
```

RouterModule refers to the forRoot which takes an input as an array, which in turn has the object of the path and the component. Path is the name of the router and component is the name of the class, i.e., the component created.

Let us now see the component created file −
New-cmp.component.ts

```
import { Component, OnInit } from '@angular/core';
@Component({
  selector: 'app-new-cmp',
  templateUrl: './new-cmp.component.html',
  styleUrls: ['./new-cmp.component.css']
})
export class NewCmpComponent implements OnInit {
  newcomponent = "Entered in new component created";
  constructor() {}
  ngOnInit() { }
}
```

The highlighted class is mentioned in the imports of the main module.
New-cmp.component.html

```
<p>
  {{newcomponent}}
</p>
<p>
  new-cmp works!
</p>
```

Now, we need the above content from the html file to be displayed whenever required or clicked from the main module. For this, we need to add the router details in the app.component.html.

```
<h1>Custom Pipe</h1>
<b>Square root of 25 is: {{25 | sqrt}}</b><br/>
<b>Square root of 729 is: {{729 | sqrt}}</b>
<a routerLink = "new-cmp">New component</a>
<router-outlet></router-outlet>
```

In the above code, we have created the anchor link tag and given routerLink as “new-cmp”. This is referred in app.module.ts as the path. When a user clicks new component, the page should display the content. For this, we need the following tag - <router-outlet> </router-outlet>. The above tag ensures that the content in the new-cmp.component.html will be displayed on the page when a user clicks new component.

What is ForRoot?

ForRoot creates a module that configures the root routing module and directives for the app. When you call RouterModule.forRoot(routes), Angular creates an instance of the Router class globally.
Let’s create route for two modules (dashboard and home), which will be loaded normally and create another route for our new LazyModule(profile), here we will specify the path to the module followed by the module’s class name with a hashtag.

```
const routes: Routes = [
  { path: "dashboard", component: DashboardComponent },
  { path: "home", component: HomeComponent },
  {
    path: "profile",
    loadChildren: () => import("./features/profile/profile.module").then(m =>
m.ProfileModule)
  }];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

What is ForChild\ChildRoot?

ForChild creates a module that configures all the directives and the given routes when you call RouterModule.forChild(routes), Angular check Router instance available in the app and register all of these routes with that instance. but does not include the router service. It uses the router service created at the root level.

```
const routes: Routes = [
  { path: 'setting', component: SettingComponent }
];
@NgModule({
  imports: [RouterModule.forChild(routes)],
  exports: [RouterModule]
})
export class ProfileRoutingModule { }
```

When you provide values in eager-loaded modules imported into each other, injectors are going to be merged and you will still have only one instance of the service. This gets more complicated with lazy-loaded modules. Each lazy-loaded module gets its own injector.

When to use ForRoot and ForChild in Angular?
ForRoot is used when a module is “eager” (loads when the application starts). Angular creates a Router instance for all the modules that is going to be injected into the “root” of the modules. When we want to access our providers from any point in the application.
ForChild is used when a module is “lazy” (loads when the module loaded on demand). it has its own injector. specifically, when we want to deliver a provider that is visible only to the “children” modules of our module.

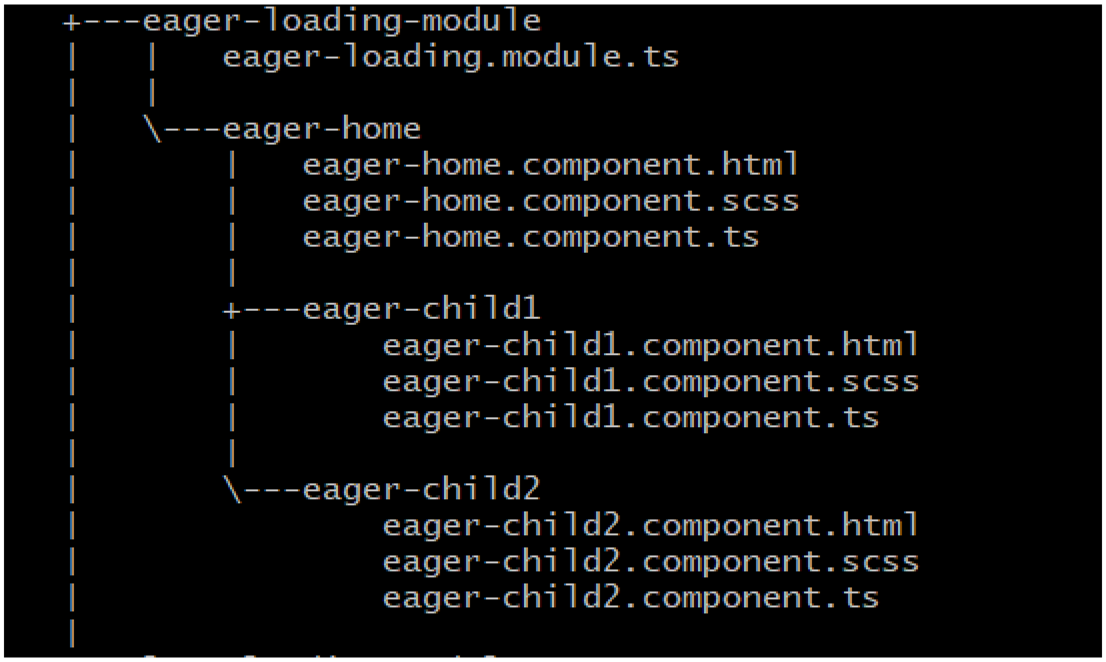
18. Eager Loading, Lazy Loading, and Pre-Loading: What, When, and How?
It introduced a component-based architecture with great benefits like modularity, which allows developers to split the web application into different modules and load them with one of the three module-loading strategies, including Eager Loading, Lazy Loading, and Pre-Loading. In this tutorial, I would like to introduce these three module-loading strategies in Angular with What, When, and How, and at the end of this tutorial, you will be able to implement these three loading strategies in the same web application.
Eager Loading: Feature modules under Eager Loading would be loaded before the application starts. This is the default module-loading strategy.
Lazy Loading: Feature modules under Lazy Loading would be loaded on demand after the application starts. It helps to start application faster.
Pre-Loading: Feature Modules under Pre-Loading would be loaded automatically after the application starts.

What is Eager Loading?
Feature modules under Eager Loading would be loaded before the application starts. This is the default module-loading strategy.

When to use Eager Loading?

Case 1: Small size applications. In this case, it’s not expensive to load all modules before the application starts, and the application will be faster and more responsive to process requests.
Case 2: Core modules and feature modules that are required to start the application. These modules could contain components of the initial page, interceptors (for authentication, authorization, and error handling, etc.), error response components, top-level routing, and localization, etc. We just have to eagerly load these modules to make the application function properly despite the application size.

Eager Loading Implementation
The file structure of the eager loading module is as follow:



```
+---eager-loading-module
|   |   eager-loading.module.ts
|   |
|   \---eager-home
|       |   eager-home.component.html
|       |   eager-home.component.scss
|       |   eager-home.component.ts
|       |
|       +---eager-child1
|       |       eager-child1.component.html
|       |       eager-child1.component.scss
|       |       eager-child1.component.ts
|       |
|       \---eager-child2
|               eager-child2.component.html
|               eager-child2.component.scss
|               eager-child2.component.ts
|
```

In the eager-home.component.html file, I add navigation routes to its two children component. Code shows as follow:

```
<h3 class="eager-title">This is from Eager Loading Module</h3>
<div class="row">
  <div class="col-sm-4">
    <a routerLink="child1">Eager Child 1</a>
  </div>
  <div class="col-sm-4">
    <a routerLink="child2">Eager Child 2</a>
  </div>
```

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

Then we need to declare the eager-home component as well as its two children components in eager-loading.module.ts.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { EagerHomeComponent } from './eager-home/eager-
home.component';
import { EagerChild1Component } from './eager-home/eager-child1/eager-
child1.component';
import { EagerChild2Component } from './eager-home/eager-child2/eager-
child2.component';
import { RouterModule } from '@angular/router';
@NgModule({
  declarations: [
    EagerHomeComponent,
    EagerChild1Component,
    EagerChild2Component
  ],
  imports: [
    CommonModule,
    RouterModule
  ]
})
export class EagerLoadingModule { }
```

To apply Eager Loading strategy, at first, we need to register all components that will be eagerly loaded in the app-routing.module.ts with **forRoot** strategy.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { EagerHomeComponent } from './features/eager-loading-
module/eager-home/eager-home.component';
```

```
import { EagerChild1Component } from './features/eager-loading-
module/eager-home/eager-child1/eager-child1.component';
import { EagerChild2Component } from './features/eager-loading-
module/eager-home/eager-child2/eager-child2.component';
const routes: Routes = [
  {path: '', redirectTo: 'eager-loading', pathMatch: 'full'},
  {path: 'eager-loading', component: EagerHomeComponent, children: [
    {path: '', redirectTo: 'child1', pathMatch: 'full'},
    {path: 'child1', component: EagerChild1Component},
    {path: 'child2', component: EagerChild2Component},
    {path: '**', redirectTo: 'child1'}
  ]},
  {path: '**', redirectTo: 'eager-loading'}
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

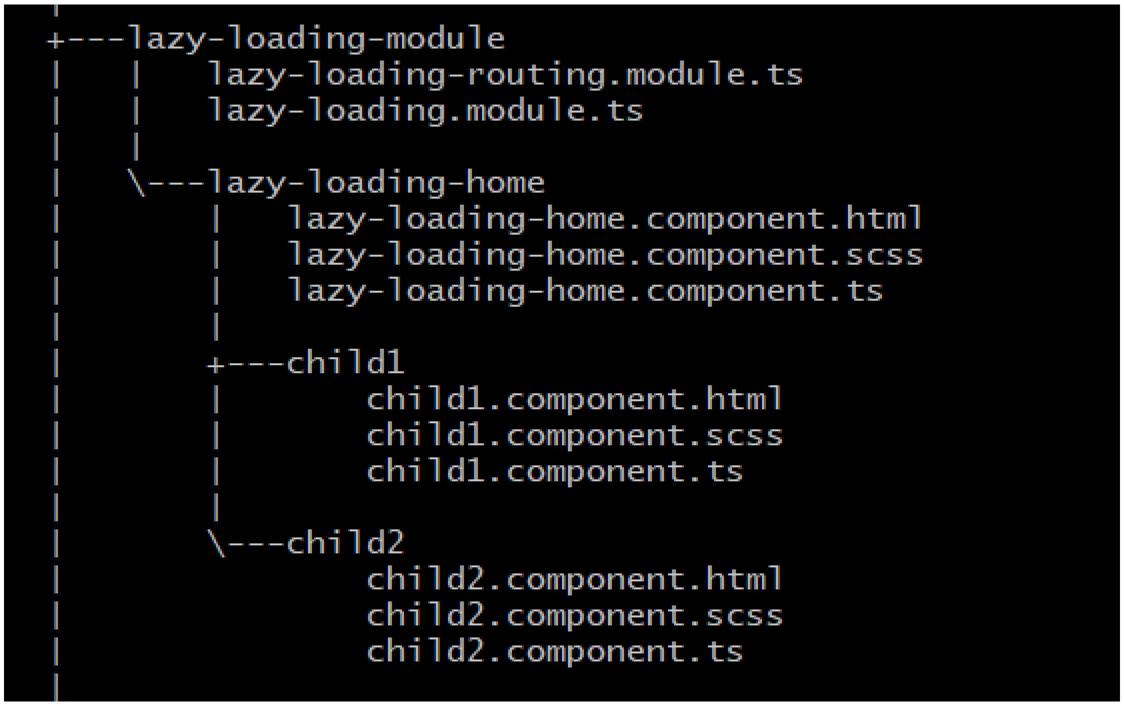
At second, we need to import the EagerLoadingModule in AppModule, so that it will be loaded before the application start.

```
import { BrowserModule } from '@angular/platform-browser';
import { NgModule } from '@angular/core';
import { AppRoutingModule } from './app-routing.module';
import { AppComponent } from './app.component';
import { EagerLoadingModule } from './features/eager-loading-
module/eager-loading.module';
@NgModule({
  declarations: [
    AppComponent
  ],
  imports: [
    BrowserModule,
    AppRoutingModule,
    EagerLoadingModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }
```

What is Lazy Loading?
Feature modules under Lazy Loading would be loaded on demand after the application starts. It helps to start application faster.

When to use Lazy Loading?
The scenario of applying Lazy Loading is relatively simple and straightforward. In a big-size web application, we can lazily load all other modules that are not required when the application starts.

Lazy Loading Implementation
The file structure of lazy loading module is as follow:



```
|
+---lazy-loading-module
|    |   lazy-loading-routing.module.ts
|    |   lazy-loading.module.ts
|    |
|    \---lazy-loading-home
|         |   lazy-loading-home.component.html
|         |   lazy-loading-home.component.scss
|         |   lazy-loading-home.component.ts
|         |
|         +---child1
|         |       child1.component.html
|         |       child1.component.scss
|         |       child1.component.ts
|         |
|         \---child2
|                 child2.component.html
|                 child2.component.scss
|                 child2.component.ts
|
```

At first, we need to create navigation routes in the lazy-loading-home.component.html to its two children component.

```
<h3>This is From Lazy Loading Module</h3>
<div class="row">
  <div class="col-sm-4">
    <a routerLink="child1">Lazy Child1</a>
  </div>
  <div class="col-sm-4">
    <a routerLink="child2">Lazy Child2</a>
  </div>
</div>
<div>
  <router-outlet></router-outlet>
</div>
```

At second, we need to register these lazy loading components in the lazy-loading-routing.module.ts with **forChild** strategy.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { Routes, RouterModule } from '@angular/router';
import { Child1Component } from './lazy-loading-
home/child1/child1.component';
import { Child2Component } from './lazy-loading-
home/child2/child2.component';
import { LazyLoadingHomeComponent } from './lazy-loading-home/lazy-
loading-home.component';
const routes: Routes = [
  {path: '', component: LazyLoadingHomeComponent, children:[
    {path: '', redirectTo: 'child1', pathMatch: 'full'},
    {path: 'child1', component: Child1Component},
    {path: 'child2', component: Child2Component}
  ]},
  {path: '**', redirectTo: ''}
];
@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    RouterModule.forChild(routes)
  ],
  exports:[
    RouterModule
  ]
})
```

```
export class LazyLoadingRoutingModule { }
```

After that, we need to declare all these lazy loading components and import the LazyLoadingModule in the lazy-loading.module.ts.

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { LazyLoadingHomeComponent } from './lazy-loading-home/lazy-
loading-home.component';
import { Child1Component } from './lazy-loading-
home/child1/child1.component';
import { Child2Component } from './lazy-loading-
home/child2/child2.component';
import { LazyLoadingRoutingModule } from './lazy-loading-routing.module';
@NgModule({
  declarations: [
    LazyLoadingHomeComponent,
    Child1Component,
    Child2Component
  ],
  imports: [
    CommonModule,
    LazyLoadingRoutingModule
  ]
})
export class LazyLoadingModule { }
```

So now, the LazyLoadingModule is ready to lazily load. And the very last step is to add a route for this LazyLoadingModule in app-routing.module.ts with **loadChildren** property. And now, the app-routing.module.ts would look like as follow after we added a route for this LazyLoadingModule.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { EagerHomeComponent } from './features/eager-loading-
module/eager-home/eager-home.component';
import { EagerChild1Component } from './features/eager-loading-
module/eager-home/eager-child1/eager-child1.component';
```

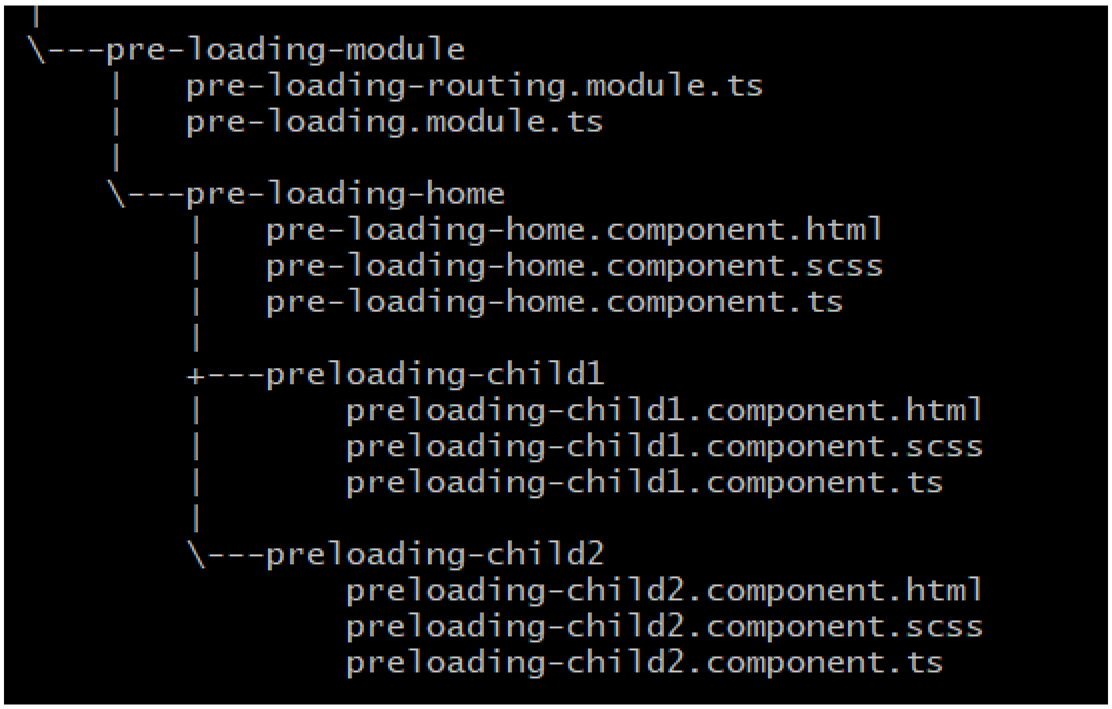
```
import { EagerChild2Component } from './features/eager-loading-
module/eager-home/eager-child2/eager-child2.component';
const routes: Routes = [
  {path: '', redirectTo: 'eager-loading', pathMatch: 'full'},
  {path: 'eager-loading', component: EagerHomeComponent, children: [
    {path: '', redirectTo: 'child1', pathMatch: 'full'},
    {path: 'child1', component: EagerChild1Component},
    {path: 'child2', component: EagerChild2Component},
    {path: '**', redirectTo: 'child1'}
  ]},
  {
    path: 'lazy-loading',
    loadChildren: './features/lazy-loading-module/lazy-
loading.module#LazyLoadingModule'
   },
  {path: '**', redirectTo: 'eager-loading'}
];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```
We should not import this LazyLoadingModule in app.module.ts, otherwise, it will be eagerly loaded instead.

What is Pre-Loading?
Feature Modules under Pre-Loading would be loaded automatically after the application starts.

When to use Pre-Loading?
Compared with Eager Loading and Lazy Loading, Pre-Loading is not so much frequently used in web application development. Based on my understanding of this loading strategy, Pre-Loading would be favorable for two cases though.
Case 1: Medium size application. In this scenario, we can make the application start faster since it will load all other modules later that are not required to run the application. And the application would be more

responsive to process users' requests than applying Lazy Loading strategy since the application will load all these modules after the application started.
Case 2: Some specific modules that users are very likely to use after the application started. In this scenario, we can pre-load these feature modules and still lazy load other modules.

Pre-Loading Implementation
The file structure of Pre-Loading module is as follow:



```
|
\---pre-loading-module
    |   pre-loading-routing.module.ts
    |   pre-loading.module.ts
    |
    \---pre-loading-home
        |   pre-loading-home.component.html
        |   pre-loading-home.component.scss
        |   pre-loading-home.component.ts
        |
        +---preloading-child1
        |       preloading-child1.component.html
        |       preloading-child1.component.scss
        |       preloading-child1.component.ts
        |
        \---preloading-child2
                preloading-child2.component.html
                preloading-child2.component.scss
                preloading-child2.component.ts
```

Actually, Pre-Loading is pretty similar to Lazy Loading, and the only difference is that we need to specify the preloading strategy. In this section, I would like to focus on how to customize pre-loading strategy.
At first, after we added the pre-loading route to app-routing.module.ts, it should look as follow:

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { EagerHomeComponent } from './features/eager-loading-module/eager-home/eager-home.component';
import { EagerChild1Component } from './features/eager-loading-module/eager-home/eager-child1/eager-child1.component';
import { EagerChild2Component } from './features/eager-loading-
module/eager-home/eager-child2/eager-child2.component';
import { CustomPreloadingStrategy } from './core/custom-preloading-
strategy';
const routes: Routes = [
  { path: '', redirectTo: 'eager-loading', pathMatch: 'full' },
  {
    path: 'eager-loading', component: EagerHomeComponent, children: [
      { path: '', redirectTo: 'child1', pathMatch: 'full' },
      { path: 'child1', component: EagerChild1Component },
      { path: 'child2', component: EagerChild2Component },
      { path: '**', redirectTo: 'child1' }
    ]
  },
  {
    path: 'lazy-loading',
    loadChildren: './features/lazy-loading-module/lazy-
loading.module#LazyLoadingModule'
  },
  {
    path: 'pre-loading',
    loadChildren: './features/pre-loading-module/pre-
loading.module#PreLoadingModule',
    data: { applyPreload: true }
  },
  { path: '**', redirectTo: 'eager-loading' }
];
@NgModule({
  imports: [RouterModule.forRoot(routes,
    { preloadingStrategy: CustomPreloadingStrategy }
  )],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

As you can see from above sample code in line bolded: I added an extra property named “**data**” with “**applyPreload**” set to “true”, and I also specified that I will use my own CustomPreloadingStrategy. So now we need to complete two things:

- Implement the **CustomPreloadingStrategy** class using **PreloadingStrategy** interface.
- Provide the CustomPreloadingStrategy class in app.module.ts

Following is my sample code to implement the CustomPreloadingStrategy class:

```
import { PreloadingStrategy, Route } from '@angular/router';
import { Observable, of } from 'rxjs';
export class CustomPreloadingStrategy implements PreloadingStrategy{
   // loadModule could be any function name here
   preload(route: Route, loadModule: Function): Observable<any> {
      return route.data && route.data.applyPreload ? loadModule() :
of(null);
   }
}
```

As you can see from above sample code, this custom preloading strategy will scan each route at root level, and check if that route contains “data” property and “applyPreload” is true. If it is, then it will preload that target module, otherwise, it will do nothing (that means, the Lazy Loading strategy would be applied instead).
Then we need to provide this CustomPreloadingStrategy in the app.module.ts file.

Add Navigation Routes to the Three Modules

So now we have set up everything for these three modules, and we just need to add navigation routes to these three modules in app.component.ts file.

```
<h3 class="title">Eager Loading, Lazy Loading, and Pre-Loading Demo</h3>
<div class="row">
  <div class="col-sm-4">
    <a class="btn btn-primary btn-md" routerLink="eager-loading">Eager
Loading</a>
  </div>
  <div class="col-sm-4">
```

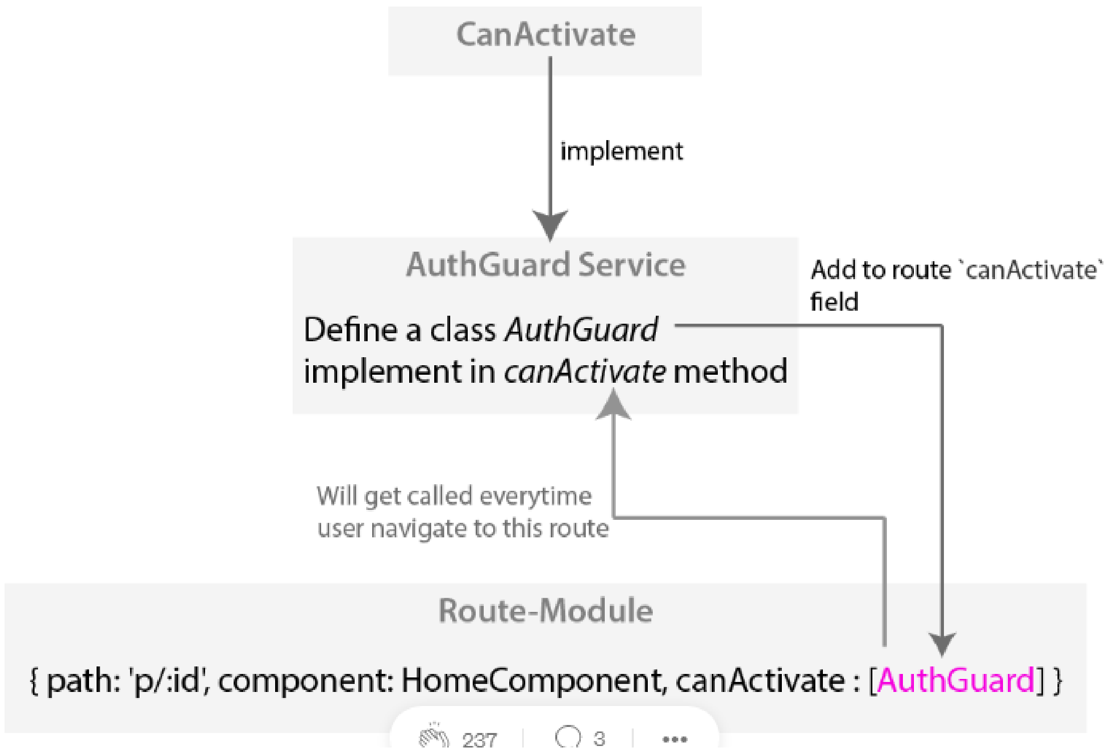
```
    <a class="btn btn-info btn-md" routerLink="lazy-loading">Lazy
Loading</a>
  </div>
  <div class="col-sm-4">
    <a class="btn btn-secondary btn-md" routerLink="pre-loading">Pre
Loading</a>
  </div>
</div>
<div class="container">
  <router-outlet></router-outlet>
</div>
```

19. What is Route Guards?

Angular route guards are interfaces provided by angular which when implemented allow us to control the accessibility of a route based on condition provided in class implementation of that interface.
Five types of route guards are provided by angular :

- CanActivate - Checks to see if a user can visit a route.
- CanActivateChild - Checks to see if a user can visit a routes children.
- CanLoad - Checks to see if a user can route to a module that lazy loaded.
- CanDeactivate - Checks to see if a user can exit a route.
- Resolve - Performs route data retrieval before route activation.

CanActivate: To implementing route guard preventing access to the specific route we use CanActivate interface.
canActivate(route:ActivatedRouteSnapshot, state:RouterStateSnapshot):Observable<boolean | UrlTree> | Promise<boolean | UrlTree> | boolean | UrlTree
Now here we have two options, first is prevent a user from navigating to given route and the other is redirecting the user to some other route e.g. to the login page if the user is unauthorized.


CanActivate
implement
AuthGuard Service
Define a class AuthGuard implement in canActivate method
Add to route `canActivate` field
Will get called everytime user navigate to this route
Route-Module
{ path: 'p/:id', component: HomeComponent, canActivate : [AuthGuard] }
237
3

Now first define a service and create a class AuthGuard (you can use any name you want)

```
// AuthGuard Service
import {CanActivate, ActivatedRouteSnapshot, RouterStateSnapshot,
Router} from '@angular/router';
import { Observable } from 'rxjs/Observable';
export class AuthGuard implements CanActivate {
constructor(private authService: AuthService, private router: Router) {}
canActivate(route: ActivatedRouteSnapshot, state:RouterStateSnapshot):
Observable<boolean> | Promise<boolean> | boolean {
  // return true if you want to navigate, otherwise return false
 }
}
```

Now just use this service in your routes like this

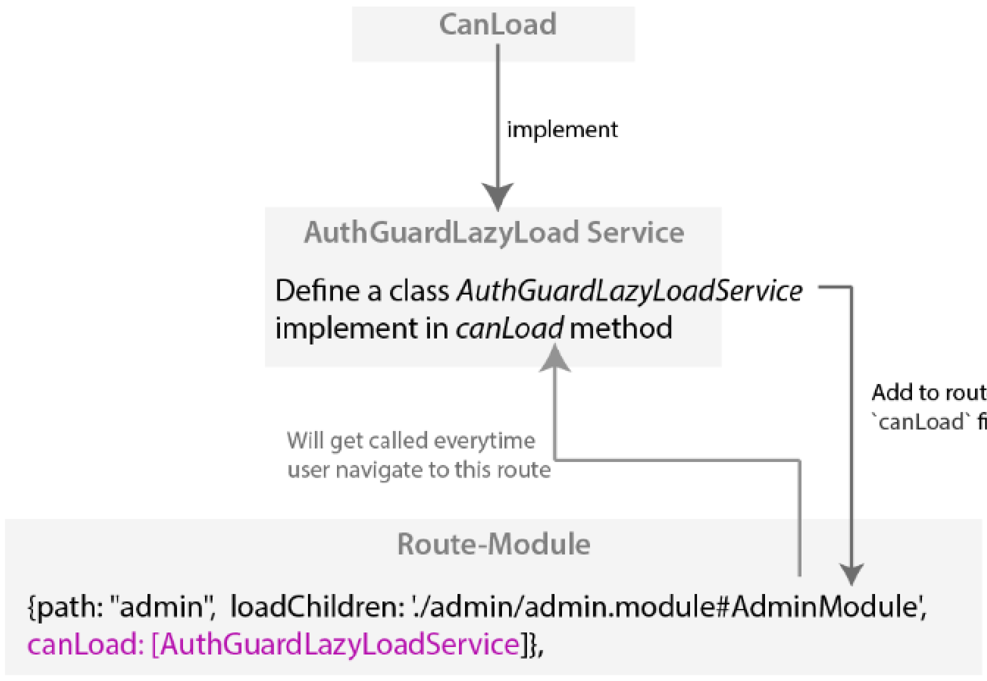
// route-module file
{ path: 'p/:id', component: HomeComponent, **canActivate** : [AuthGuard] }

CanActivateChild: CanActivateChild is almost similar to CanActivate interface, the only difference is CanActivate is used to control the accessibility of the current route but CanActivateChild is used to prevent access to child routes of a given route, so by using this you don't need to add canActive on each child route, in other words, you just need to add canActiveChild to parent route and it will work for child routes as well.

```
// route-module ts file
{
path: 'dashboard',
canActivate: [AuthGuard],
canActivateChild: [AuthGuard],
component: DashboardComponent,
children: [
{ path: ':id', component: InfoComponent},
{ path: ':id/edit', component: EditInfoComponent}
]
}
```

CanLoad: Modules in angular can be loaded all at once or be lazy loaded . By default angular load all modules eagerly. To implement lazy loading we use loadChildren in route definition. The main advantage of lazy loading is that it reduces the loading time of application by downloading only the required modules.

Now here if we want to prevent navigation of an unauthorized user we can use CanActivate Guard, that will do the job but also download the module. Now to control the navigation as well as prevent downloading of that module we can use CanLoad Guard.


CanLoad
implement
AuthGuardLazyLoad Service
Define a class *AuthGuardLazyLoadService* implement in *canLoad* method
Add to route `canLoad` field
Will get called everytime user navigate to this route
Route-Module
{path: "admin", loadChildren: './admin/admin.module#AdminModule',
canLoad: [AuthGuardLazyLoadService]},

canLoad(route:Route,segments:UrlSegment[]):Observable<boolean>|Promise<boolean>|boolean;

The implementation process is the same as that of CanActivate . First, define a service e.g. AuthGuardLazyLoad and just pass to canLoad field of the route definition.

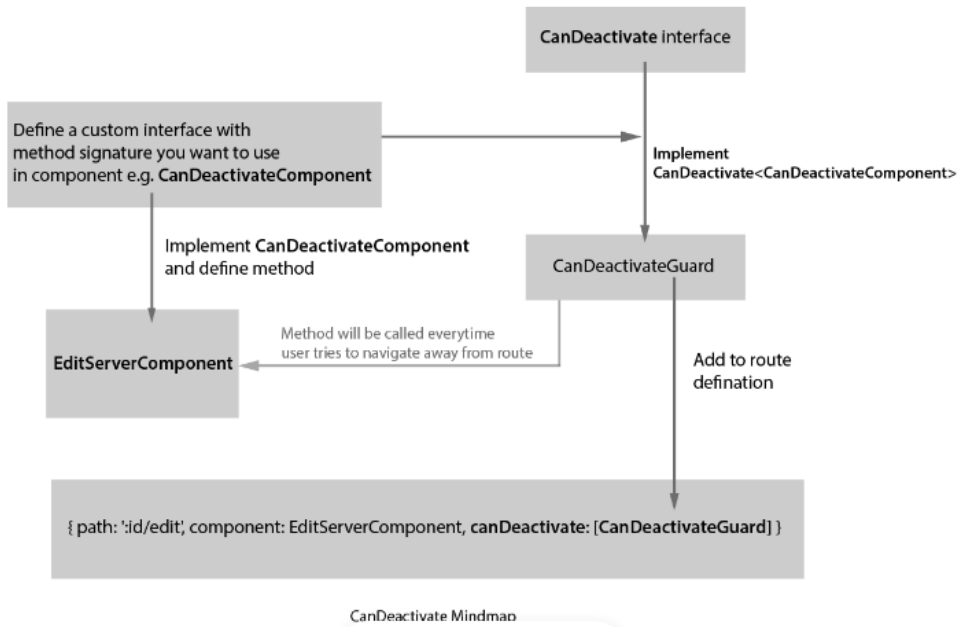
```
import { CanLoad, Route, Router } from '@angular/router';
export class AuthGuardLazyLoad implements CanLoad {
constructor(private router: Router) {
}

canLoad(route:Route,segments:UrlSegment[]):Observable<boolean>|Promise<boolean>|boolean {
 /* return true or false depending on whether you want to load that
module or not
You can also use Observable/Promise resolving boolean value
 */
 }
}
```

Now just pass this service to canLoad field of the route definition, as below :

```
{path: "admin", loadChildren:'./admin/admin.module#AdminModule',
canLoad:[AuthGuardLazyLoad]}
```

CanDeactivate: This route guard is used to keep the user from navigating away from a specific route. This guard can be useful when you want to prevent a user from accidentally navigating away without saving or some other undone tasks.
This route guard is a little bit different in implementation from the above-mentioned routes as it involves defining a method in the component class itself, which gets called whenever the user tries to navigate away from the route
canDeactivate(component: T, currentRoute: ActivatedRouteSnapshot, currentState: RouterStateSnapshot,nextState?: RouterStateSnapshot): Observable<boolean|UrlTree>|Promise<boolean|UrlTree>|boolean|UrlTree;


CanDeactivate interface
Define a custom interface with method signature you want to use in component e.g. CanDeactivateComponent
Implement CanDeactivate<CanDeactivateComponent>
Implement CanDeactivateComponent and define method
CanDeactivateGuard
EditServerComponent
Method will be called everytime user tries to navigate away from route
Add to route defination
{ path: ':id/edit', component: EditServerComponent, canDeactivate: [CanDeactivateGuard] }

CanDeactivate Mindmap

First, create an interface that will be used in component and use it in CanDeactivateGuard service as in below-given code :
// CanDeactivateGuard service
import { Observable } from 'rxjs/Observable';
import { CanDeactivate, ActivatedRouteSnapshot, RouterStateSnapshot } from '@angular/router';
export interface CanComponentDeactivate {
canDeactivate: () => Observable<boolean> | Promise<boolean> | boolean;
}

```
export class CanDeactivateGuard implements
CanDeactivate<CanComponentDeactivate> {
canDeactivate(component:CanComponentDeactivate,currentRoute:Activat
edRouteSnapshot, currentState:RouterStateSnapshot, nextState?:
RouterStateSnapshot): Observable<boolean> | Promise<boolean> |
boolean {
return component.canDeactivate();
}
}
```

Now, you need to implement CanComponentDeactivate interface in component as given in below code :

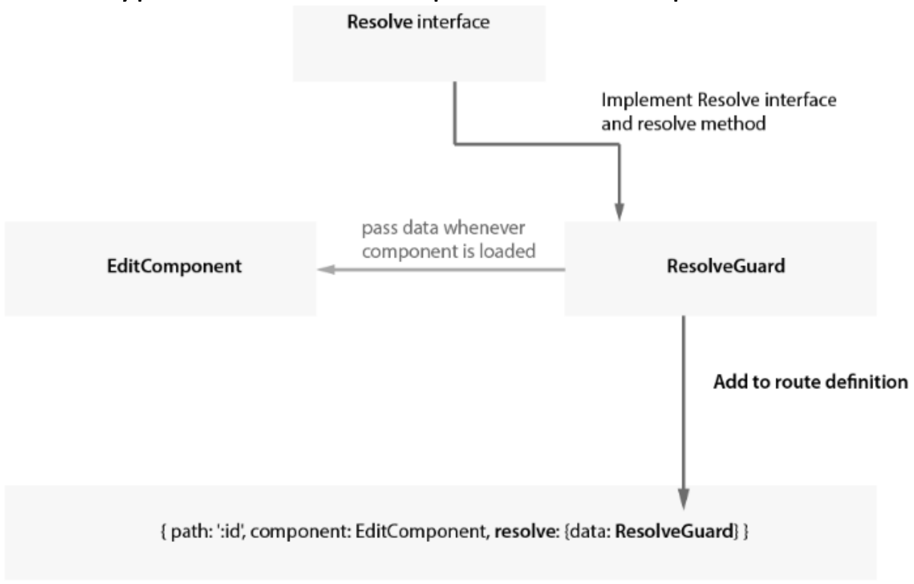
```
// EditComponent component code
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router } from '@angular/router';
import { Observable } from 'rxjs/Observable';
import { CanComponentDeactivate } from './can-deactivate-guard.service';
@Component({
selector: 'app-edit',
templateUrl: './edit.component.html',
styleUrls: ['./edit.component.css']
})
export class EditComponent implements OnInit, CanComponentDeactivate
{
canDeactivate(): Observable<boolean> | Promise<boolean> | boolean {
/* return true or false depends on whether you want user to navigate away
from this route or not. You can also use Observable/Promise resolving to
boolean */
}
}
```

Almost done now, you just need to add this route-guard to your route definition as given below :

```
{ path: ':id/edit', component: EditServerComponent, canDeactivate:
[CanDeactivateGuard] }
```

Resolve Guard: Complex angular applications involve data communication between components, sometimes data is so heavy that it is not possible to

pass data through query params. To handle this situation angular has provided Resolve Guard.
Now what the Resolve guard does is resolving data based on implemented code and pass that data to the component.
resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot): Observable<T> | Promise<T> | T
T is datatype that resolve will pass to the component after resolving data.


Resolve interface
Implement Resolve interface and resolve method
pass data whenever component is loaded
EditComponent
ResolveGuard
Add to route definition
{ path: ':id', component: EditComponent, resolve: {data: ResolveGuard} }

Implementation of Resole guard is quite simple. You just need to define a service and add it to the route definition. As given in code below :

```
// Resolve Guard service
import { Resolve, ActivatedRouteSnapshot, RouterStateSnapshot } from
'@angular/router';
import { Observable } from 'rxjs/Observable';
import { Injectable } from '@angular/core';
import { FetchDataService } from '../FetchData.service';
interface UserName {
id: number;
name: string;
}
@Injectable()
export class ResolveGuard implements Resolve<UserName> {
constructor(private fetchService: FetchDataService) {}
resolve(route: ActivatedRouteSnapshot, state: RouterStateSnapshot):
Observable<UserName> | Promise<UserName> | UserName {
return this.fetchService.getUser(+route.params['id']);
}
}
```
Now add this service to route defination

```
{ path: ':id', component: EditComponent, resolve: {data: ResolveGuard} }
```
as you can see Resolve route is a little bit different from the rest of the route guards as it takes the object as other route guards accept an array.
how to get data in the component?
it's quite simple. Just subscribe data observer in ActivatedRouteSnapshot as given below

```
// Component code
import { Component, OnInit } from '@angular/core';
import { ActivatedRoute, Params, Router, Data } from '@angular/router';
@Component({
selector: 'app-edit',
templateUrl: './edit.component.html',
styleUrls: ['./edit.component.css']
})
export class EditComponent implements OnInit {
userName: {id: number, name: string};
constructor(private route: ActivatedRoute, private router: Router) {
}
ngOnInit() {
this.route.data.subscribe((data: Data) => {
this.userName = data['data'];
});
}
}
```

20. How to detect route change in Angular with examples

Steps to detect route change in Angular application Urls.

- Import Router, Event, NavigationStart, NavigationEnd, NavigationError from ‘@angular/router’.
- And inject router in the constructor.
- Subscribe to the NavigationStart, NavigationEnd, NavigationError events of the router.
- Detect the change in URL route in NavigationStart’s subscribe method.

We can add progress spinner or progress bar whenever a route change detected in Angular applications.We will take similar example as explained in How to get current route URL in Angular, I have created an Angular app which contains three routes. Aboutus,Services and Contactus.

```
import { Component } from '@angular/core';
import { Router, Event, NavigationStart, NavigationEnd, NavigationError}
from '@angular/router';
@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
  styleUrls: ['./app.component.sass']
})
export class AppComponent {
  title = 'detect-route-change';
  currentRoute: string;
  constructor(private router: Router) {
    this.currentRoute = "";
    this.router.events.subscribe((event: Event) => {
        if (event instanceof NavigationStart) {
            // Show progress spinner or progress bar
            console.log('Route change detected');
        }
        if (event instanceof NavigationEnd) {
            // Hide progress spinner or progress bar
            this.currentRoute = event.url;
            console.log(event);
        }
        if (event instanceof NavigationError) {
             // Hide progress spinner or progress bar     // Present error to user
            console.log(event.error);
        }
    });
}
}
```

Lets go deep into the code, we are subscribing to NavigationStart, NavigationEnd events of router.
Whenever there is a change in angular router, first it will call NavigationStart method as we have subscribed to it.
Here we can create a variable which indicates whether we have to show loading indicators such as progress spinner or progress bar.After navigating to the required route, NavigationEnd event will be called.Here we can hide the loading indicators.
I have created a variable called currentRoute, which indicates the current router url value in the component HTML file.In the subscribe event of NavigationEnd, we can change the currentRoute value.

```
<h1>Detect Route Change Angular</h1>
For the Tutorial <a href="https://www.angularjswiki.com/angular/how-to-get-current-route-in-angular/">Visit Angular Wiki</a>
<ul>
  <li routerLinkActive="active">
    <a [routerLink]="['/']">Home</a>
 </li>
  <li routerLinkActive="active">
     <a [routerLink]="['/aboutus']">About Us</a>
  </li>
  <li routerLinkActive="active">
     <a [routerLink]="['/services']">Services</a>
  </li>
  <li routerLinkActive="active">
     <a [routerLink]="['/contactus']">Contact Us</a>
  </li>
</ul>
The current Route Is {{currentRoute}}
<router-outlet></router-outlet>
```

21. What is the activated route?

When routing in any application there comes a point where it makes sense architecturally to share data during navigation. Angular has several tools for sharing data across your apps routing system including Route Guards, Route Resolvers, the Route API and the ActivatedRoute API.

The ActivatedRoute API is an interesting function within Angular as it specifically has to do with the currently activated route in your app and holds a conglomerate of information pertaining to the router.
"A service that is provided to each route component that contains route specific information such as route parameters, static data, resolve data, global query params and the global fragment."
With this data we can handle route specific logic, use route specific state and retrieve the route fragments of the current URL.
In a component, we can access this data by injecting the ActivatedRoute API inside the constructor:
import { ActivatedRoute } from '@angular/router'
constructor(private activatedRoute: ActivatedRoute) {}

What route data is available?
Here is a section from the docs outlining the properties available in the ActivatedRoute API:
snapshot: ActivatedRouteSnapshot - The current snapshot of this route
url: Observable<UrlSegment[]> - An observable of the URL segments matched by this route.
params: Observable<Params> - An observable of the matrix parameters scoped to this route.
queryParams: Observable<Params> - An observable of the query parameters shared by all the routes.
fragment: Observable<string> - An observable of the URL fragment shared by all the routes.
data: Observable<Data> - An observable of the static and resolved data of this route.
outlet: string - The outlet name of the route, a constant.
component: Type<any> | string | null - The component of the route, a constant.
routeConfig: Route | nullRead-Only - The configuration used to match this route.
root: ActivatedRoute Read-Only - The root of the router state.
parent: ActivatedRoute | null Read-Only - The parent of this route in the router state tree.
firstChild: ActivatedRoute | null Read-Only - The first child of this route in the router state tree.

children: ActivatedRoute[]   Read-Only - The children of this route in the router state tree.
pathFromRoot: ActivatedRoute[]   Read-Only - The path from the root of the router state tree to this route.
paramMap: Observable<ParamMap> Read-Only - An Observable that contains a map of the required and optional parameters specific to the route. The map supports retrieving single and multiple values from the same **parameter**.
queryParamMap: Observable<ParamMap> Read-Only - An Observable that contains a map of the query parameters available to all routes. The map supports retrieving single and multiple values from the query parameter.

The snapshot property from the ActivatedRoute contains this ActivatedRouteSnapshot object:

```
interface ActivatedRouteSnapshot {
   routeConfig: Route | null
   url: UrlSegment[]
   params: Params
   queryParams: Params
   fragment: string
   data: Data
   outlet: string
   component: Type<any> | string | null
   root: ActivatedRouteSnapshot
   parent: ActivatedRouteSnapshot | null
   firstChild: ActivatedRouteSnapshot | null
   children: ActivatedRouteSnapshot[]
   pathFromRoot: ActivatedRouteSnapshot[]
   paramMap: ParamMap
   queryParamMap: ParamMap
   toString(): string
}
```

Some properties are duplicated: because they hold the same kind of data but they are not the same types of data. The snapshot properties hold synchronous information that you can access right away; the ActivatedRoute properties that have the same key names hold observable data which is asynchronous.

Summary of data found in ActivatedRoute
Route Tree and Components: The component, root, parent,firstChild, children and pathFromRoot keys comprise the Route Tree which is made up of ActivatedRouteSnapshots that are being loaded in the route, including the components that are part of the route.
Route State: The Route state lies in the data property, which is where Resolver and Observable data are kept. Data put into a Route from the resolver or routeGuard is available here.
URL Information: The url, params, queryParams, fragment, paramMap and queryParamMap keys make up the URL information. The url property only holds the last segment of the URL. For example if your URL path was /home/pangolins the last segment would be pangolins. If you want to access the current URL path you would want to get that from the Router API.
Other Route Metadata: More Metadata is stored in the routeConfig and outlet properties. The routeConfig holds a Route object and the outlet key stores the name of the router outlet being used.

What is the difference between the ActivatedRoute and ActivatedRouteSnapshot?
The ActivatedRoute is the API that holds observable data that you can use to react to events with. You can subscribe to specific observable properties to get data from it asynchronously. The ActivatedRoute also contains ActivatedRouteSnapshot. The ActivatedRouteSnapshot holds the same activated route data but is a static property.
Static activated route snapshot data

```
constructor(private activatedRoute: ActivatedRoute) {
   const activatedRouteSnapshot = activatedRoute.snapshot;
}
```

Observable activated route data
For observable data you must subscribe to a specific observable property on the ActivatedRoute object, for example the data property:

```
constructor(private activatedRoute: ActivatedRoute) {
   activatedRoute.data.subscribe((data) => ...)
}
```

What can you do with this knowledge?

Get URL Params: You can get the URL params using the activatedRoutes query param map, along with its built in get method for retrieving the parameters value. (tip: it is possible for parameters to have casing issues; it's recommended that you normalize keys before getting the value).

```
// www.briebug.com?favoritepangolin=leroyjenkins
ngOnInit() {
   const params = this._activatedRoute.snapshot.queryParamMap;
   const favoritePangolin = params.get('favoritepangolin');
}
```

Get URL Fragment: You may want to get the current URL fragment while the user travels along your page in order to give them a fragment specific user experience. To do that, you can just subscribe to the ActivatedRoute changes.

```
listenForRouteFragment(){
   this.activatedRoute$.fragment.subscribe((fragment: string) => {
      if (fragment === 'youGotPaid') {
         this.makeItRain();
      };
   });
}
```

Pass data and retrieve resolver data: Data can be passed during navigation and retrieved in the component within the activated route. You can include data in the route configuration:

```
const routes: RouterConfig = [
   {
      path: 'pangolins',
      component: cutePangolinComponent,
      data: {
         pangolinArray: [ ... ]
      }
   ];
```

However this approach is not as useful, or as common, as others. You can access this data within the data property of ActivatedRoute but the data is static. A dynamic approach would be much more useful, which is where having a resolver comes into play. With a resolver I can fetch my array of Pangolins from the server. (tip: when data is necessary for rendering a view it can be useful to also use a route guard to make sure the data is available before loading the view).

```
const routes: RouterConfig = [
  {
    path: 'pangolins',
    component: cutePangolinComponent,
    resolve: {
      pangolins: PangolinResolver
    }
  ];
```

Notice how the resolver is inside the resolve property and not the data property. When our data resolves, Angular will take care of putting the data requested inside the data property on the ActivatedRoute. We can then get our data synchronously or asynchronously inside our component.

```
constructor(private activatedRoute: ActivatedRoute) {
  this.pangolins = activatedRoute.snapshot.data['pangolins']
}
```

Asynchronously may be a good option if there is some logic you want to perform on the data.

```
listenForPangolins() {
  this.activatedRoute.data['pangolins'].subscribe((pangolins) => {
    this.getImagesOfPangolins(pangolins)
  })
}
```

Debugging the Router: Logging the router state can be a helpful tool for debugging routes in Angular. The ActivatedRoute holds all the router state you need to see your current route path and history. This can be useful in creating a more textual or visual representation of what your router state is; especially if your app holds a complex spaghetti of routes.

```
constructor(private activatedRoute: ActivatedRoute) {

  // what child routes are being loaded right now?
  const childRouteSnapshots = activatedRoute.snapshot.children;
  console.log('children snapshots: ', childRouteSnapshots);
```

```
    // what is the parent of this route?
    const parentRouteSnapshot = activatedRoute.snapshot.parent;
    console.log('parent route snapshot: ', parentRouteSnapshot);

    // what is the full path of my router?
    const routePath = activatedRoute.snapshot.pathFromRoot;
    console.log('my path from root: ', routePath);

    // I have multiple router outlets. Is this using the correct router outlet?
    const outletName = activatedRoute.snapshot.outlet;
    console.log('current router outlet name: ', outletName);
  }
```

22. What is the Router-Outlet and how to use it?

Router-outlet in Angular works as a placeholder which is used to load the different components dynamically based on the activated component or current route state. Navigation can be done using router-outlet directive and the activated component will take place inside the router-outlet to load its content.

To enable routing, we need to use router-outlet into our HTML template like this.

```
<router-outlet></router-outlet>
{path: 'home', component: HomeComponent }
```

Naming a Router Outlet?

The router outlet can be given a name using the name property as follows:

<router-outlet **name="mainOutlet"**></router-outlet>

Adding Multiple Router-Outlets

You can have multiple outlets in your Angular template:

<router-outlet></router-outlet>
<router-outlet **name="sidebar"**></router-outlet>

The unnamed outlet is the primary outlet in your app and except for the primary outlet, all the other outlets must have a name.

The Route Outlet Property

If you have multiple outlets in your template, you can specify the router outlet where you want to insert the route component using the outlet property of a route as follows:
{path: 'home', component: HomeSidebarComponent, **outlet: "sidebar"** }
In this case, when the home path is matched the HomeSidebarComponent will be inserted in the router outlet named sidebar. This route is called an auxiliary route because it's mapped to the secondary outlet.

23. Setting redirect in Angular Routing

To set up a redirect in an Angular route definition you will have to use redirectTo property.
For redirection you need to configure a route with the following three things-

- path you want to redirect from
- The component you want to redirect to
- A pathMatch value to tell the router how to match the URL.

For example if you want to set a default route to the home page. When the initial URL in the browser is- localhost:4200 it should redirect automatically to localhost:4200/home.
{path: '', redirectTo:'/home', pathMatch: 'full'}
Here **path** you want to redirect from is '' meaning nothing after the base URL.
redirectTo has a value '/home' which tells the Angular router when we visit path localhost:4200 redirect to the /home route.
Here **pathMatch** value is full which means redirection happens only if the full path after the base URL is ''.

Angular routing redirect example

Suppose we have three components Home, Account and Service and the app routing module is as given below.

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { AccountComponent } from './account.component';
import { HomeComponent } from './home.component';
import { ServiceComponent } from './service.component';
const routes: Routes = [
        {path: 'home', component: HomeComponent},
        {path: 'account', component: AccountComponent},
        {path: '', redirectTo:'/home', pathMatch: 'full'},
        {path: 'service', component: ServiceComponent}
  ];
@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule { }
```

As per route definition specified in routing module when URL has /home as path it will be handled by this route definition. HomeComponent is the component that will be called to handle this route.

{path: 'home', component: HomeComponent},

Following route definition sets up a redirect to the above definition for home route if path matches '' (i.e. nothing after the base URL).

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

Different path matching values

In the above route definition pathMatch value is used as 'full' which means value specified with path will be matched fully with the path after the base URL.

Another value for pathMatch is 'prefix' which tells the Angular to check if the path in the URL starts with the value given with path or not.

If you change the route definition to have pathMatch as prefix here-

{path: '', redirectTo:'/home', pathMatch: 'prefix'}

Now every path matches this route definition because technically every path starts with ''.

If you change the pathMatch to 'prefix', as per the route definitions given in the routing module in this example you will never be able to reach ServiceComponent because localhost:4200/service will always be matched by the redirect route. Other two routes above the redirect route will work fine because of the route ordering followed by Angular.

Route order in Angular

Angular uses the first-match wins strategy when matching routes so more specific routes should be placed above less specific routes.

- Routes with a static path should be placed first.
- Then an empty path route which matches the default route.
- The wildcard route should be placed last because it matches every URL.

Wildcard Route is basically used in Angular Application to handle the invalid URLs. Whenever the user enter some invalid URL or if you have deleted some existing URL from your application, then by default 404 page not found error page is displayed. In such scenarios instead of showing the default error page, if you also show a custom error page and this is possible by using the Angular Wildcard Route.

{ **path: '**' , component:CustomerrorComponent** }

24. What is Angular Dependency Injection?

Dependency Injection (DI) is a technique in which a class receives its dependencies from external sources rather than creating them itself. ProductService returns the hard-coded products when getProduct method invoked.

```
product.service.ts
import {Product} from './Product'
export class ProductService{
    public  getProducts() {
      let products:Product[];
      products=[
         new Product(1,'Memory Card',500),
         new Product(1,'Pen Drive',750),
         new Product(1,'Power Bank',100)
      ]
      return products;
   }
}
```

We instantiated the productService directly in our Component as shown below.

```
app.component.ts
import { Component } from '@angular/core';
import { ProductService } from './product.service';
import { Product } from './product';
```

```
@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
})
export class AppComponent {
  products:Product[];
  productService;
  constructor(){
    this.productService=new ProductService();
  }
  getProducts() {
    this.products=this.productService.getProducts();
  }
}
```

The ProductService Instance is local to the Component. The AppComponent is now tightly coupled to the ProductService, This tight coupling brings a lot of Issues.
The ProductService hardcoded in our AppComponent. What if we want to use BetterProductService. We need to change wherever the ProductService is used and rename it to BetterProductService.
What if ProductService depends on another Service. And then we decide to change the service to some other service. Again we need to search and replace the code manually
It is hard to test this Component as it is difficult to provide the Mock for the ProductService. For Instance, what if we wanted to substitute out the implementation of ProductService with MockProductService during testing.
Our Component Class has now tied one particular implementation of ProductService. It will make it difficult to reuse our components.
We would also like to make our ProductService singleton so that we can use it across our application.
How to solve all these problems. Move the creation of ProductService to the constructor the AppComponent class as shown below.

```
export class AppComponent {
  products:Product[];
  constructor(private productService:ProductService) {  }
  getProducts() {
     this.products=this.productService.getProducts();
  }
}
```

Our AppComponent does not create the instance of the ProductService. It just asks for it in its Constructor. The responsibility of Creating the ProductService falls on the creator of the AppComponent.

25. @Injectable and @Inject

@Injectable: The Injectable is a decorator, which you need to add to the consumer of the dependency. This decorator tells angular that it must Inject the constructor arguments via the Angular DI system.

Example of Injectable : We created an example application in the Angular Dependency injection tutorial. It had two services LoggerService & ProductService as shown below.

LoggerService

```
import { Injectable } from '@angular/core';
@Injectable()
export class LoggerService {
 log(message:any) {
   console.log(message);
 }
}
```

ProductService

```
import { Injectable } from '@angular/core';
import {Product} from './Product'
import {LoggerService} from './logger.service'
@Injectable()
export class ProductService{
   constructor(private loggerService: LoggerService) {
      this.loggerService.log("Product Service Constructed");
   }
   public  getProducts() {
      this.loggerService.log("getProducts called");
      let products:Product[];
      products=[
         new Product(1,'Memory Card',500),
         new Product(1,'Pen Drive',750),
         new Product(1,'Power Bank',100)
```

```
        ]
        this.loggerService.log(products);
        return products;
    }
}
```

The **ProductService** has a dependency on the **LoggerService**. Hence it is decorated with the @Injectable decorator. Remove @Injectable() from ProductService and you will get the following error.
Uncaught Error: Can't resolve all parameters for ProductService: (?)
That is because without DI Angular will not know how to inject LoggerService into ProductService.
Remove @Injectable() from LoggerService will not result in any error as the LoggerService do not have any dependency.
The Components & Directives are already decorated with @Component & @Directive decorators. These decorators also tell Angular to use DI, hence you do not need to add the @Injectable().
The injectable decorator also has the ProvidedIn property using which you can specify how Angular should provide the dependency.

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

@Inject: The @Inject() is a constructor parameter decorator, which tells angular to Inject the parameter with the dependency provided in the given token. It is a manual way of injecting the dependency
In the previous example, when we removed the @Injectable decorator from the ProductService we got an error.
We can manually inject the LoggerService by using the @Inject decorator applied to the parameter loggerService as shown below.
The @Inject takes the Injector token as the parameter. The token is used to locate the dependency in the Providers.

```
export class ProductService{
    constructor(@Inject(LoggerService) private loggerService) {
        this.loggerService.log("Product Service Constructed");
    }
```

}

26. Methods to Share Data Between Angular Components? Or How to communicate between the components?

Data Sharing Between Angular Components

- Parent to Child: via Input
- Child to Parent: via Output() and EventEmitter
- Child to Parent: via ViewChild
- Unrelated Components: via a Service

Method1: Parent to Child via @Input

parent component Html

<app-student *ngFor="let teacher of teachers" **[teacher]="teacher" [principle]="principle"**></app-student>

This is how the parent component is supplying the input to the child component

[teacher]="teacher"

[principle]="principle"

Child component have two @input properties.

@Input() teacher: any;

@Input() principle: string;

Method 2: Child to Parent via @Output and EventEmitter

parent component Html

<app-login **(login)="onLogin($event)"**></app-login>

onLogin(user: string) {

this.userName =user;

}

child Component

@Output() login = new EventEmitter<string>();

submit(){

console.log(this.userName)

this.login.emit(this.userName);

}

Where we have imported for EventEmitter, Output which will support us to Emit the event and use the @OutPut.Second thing is

@**Output**() login = new EventEmitter<string>();

So login is going to be the event for the parent component which will receive the emitted event with string type value as mentioned in event emitter(EventEmitter<string>). Now on login button hit we are emitting the event like this
 this.login.**emit**(this.userName);
So let us discuss the parent component which is the parent component here. If you notice the parent component is rendering the login component like below
<app-login **(login)="onLogin($event)"**></app-login>
Here login is @Output which has an event emitted from login component and it is calling a local method called on Login which is setting the user name in header component.
onLogin(user: string) {
 this.userName =user;
 }

Method3: Child to Parent via @ViewChild
 <color-sample [color]="primary" #**primaryColorSample**
(click)="open()"></color-sample>
 @ViewChild(**'primaryColorSample'**) sample: ColorSampleComponent;
by using "sample" you can access any methods, data from child component.

Method 4: Unrelated Components via a Service
private approvalStageMessage = new **BehaviorSubject**('Basic Approval is required!');
ngOnInit() {
 this.appService.approvalStageMessage.**subscribe**(msg => this.message = msg);
}
submit(){
 console.log(this.approvalText)
 this.appService.approvalStageMessage.**next**(this.approvalText)
}

27. what is @HostBinding and @HostListener in angular?

The HostBinding & HostListener are decorators in Angular. HostListener listens to host events, while HostBinding allows us to bind to a property of the host element. The host is an element on which we attach our component or directive. This feature allows us to manipulate the host styles or take some action whenever the user performs some action on the host element.

HostBinding: Host Binding binds a Host element property to a variable in the directive or component
HostBinding Example: The following appHighLight directive, uses the HostBinding on style.border property of the parent element to the border property. Whenever we change the value of the border, the angular will update the border property of the host element

```
import { Directive, HostBinding, OnInit } from '@angular/core'
@Directive({
  selector: '[appHighLight]',
})
export class HighLightDirective implements OnInit {
  @HostBinding('style.border') border: string;
  ngOnInit() {
    this.border="5px solid blue"
  }
}
```

HostListener: HostListener Decorator listens to the DOM event on the host element. It also provides a handler method to run when that event occurs.
HostListener Example: For example, in the following code HostListener listens to the mouseover & mouseleave event. We use the HostListner decorator to decorate functions onMouseOver & onMouseLeave.

```
import { Directive, HostBinding, OnInit, HostListener } from '@angular/core'
@Directive({
  selector: '[appHighLight]',
})
export class HighLightDirective implements OnInit {
  @HostBinding('style.border') border: string;
  ngOnInit() {
  }
  @HostListener('mouseover')
  onMouseOver() {
    this.border = '5px solid green';
    console.log("Mouse over")
```

```
 }
 @HostListener('mouseleave')
 onMouseLeave() {
   this.border = '';
   console.log("Mouse Leave")
 }
}
```

Whenever the mouse is moved over the p element, the mouseover event is captured by the HostListener. It runs the onMouseOver method which we have attached to it. This method adds a green border to the p element using the HostBinding.

28. Angular Decorators

An Angular Decorator is a function, using which we attach metadata to a class, method, accessor, property, or parameter. We apply the decorator using the form @expression, where expression is the name of the decorator.

The Decorators are Typescript features and still not part of the Javascript. It is still in the Proposal stage.

To enable Angular Decorators, we need to add the experimentalDecorators to the tsconfig.json file. The ng new command automatically adds this for us.

```
{
  "compilerOptions": {
    "target": "ES5",
    "experimentalDecorators": true
  }
}
```

29. Map, Filter, Tap operator in Angular Observable

Map operator: The Angular observable Map operator takes an observable source as input. It applies a project function to each of the values emitted by the source observable and transforms it into a new value. It then emits the new value to the subscribers. We use a Map with a Pipe, which allows us to chain multiple operators together. In this guide, we’re going to learn how to use the Map operator with examples like converting the source to upper case, Using Map the Angular HTTP Request, with DOM events, filtering the input data, and using multiple Maps together, etc.

Filter Operator: The Filter Operator in Angular filters the items emitted by the source Observable by using a condition (predicate). It emits only those values, which satisfies the condition and ignores the rest.

Tap operator: The Angular Tap RxJs operator returns an observable that is identical to the source. It does not modify the stream in any way. Tap operator is useful for logging the value, debugging the stream for the correct values, or perform any other side effects.

30. SwitchMap, MergeMap, concatMap, ExhaustMap in Angular

SwitchMap: The Angular SwitchMap maps each value from the source observable into an inner observable, subscribes to it, and then starts emitting the values from it. It creates a new inner observable for every value it receives from the Source. Whenever it creates a new inner observable it unsubscribes from all the previously created inner observables. Basically it switches to the newest observable discarding all other.

```
let srcObservable= of(1,2,3,4)
let innerObservable= of('A','B','C','D')

srcObservable.pipe(switchMap( val => {
   console.log('Source value '+val)
   console.log('starting new observable')
   return innerObservable
  })
).subscribe(ret=> {
  console.log('Recd ' + ret);
})
```

MergeMap: The Angular MergeMap maps each value from the source observable into an inner observable, subscribes to it, and then starts emitting the values from it replacing the original value. It creates a new inner observable for every value it receives from the Source. Unlike SwitchMap, MergeMap does not cancel any of its inner observables. It merges the values from all of its inner observables and emits the values back into the stream.

```
let srcObservable= of(1,2,3,4)
let innerObservable= of('A','B','C','D')

srcObservable.pipe(mergeMap( val => {
    console.log('Source value '+val)
    console.log('starting new observable')
    return innerObservable
  })
).subscribe(ret=> {
  console.log('Recd ' + ret);
});
```

MergeMap Vs Map: The map operators emit value as observable. The MergeMap creates an inner observable, subscribes to it, and emits its value as observable.

concatMap: The Angular ConcatMap maps each value from the source observable into an inner observable, subscribes to it, and then starts emitting the values from it replacing the original value. It creates a new inner observable for every value it receives from the Source. It merges the values from all of its inner observables in the order in which they are subscribed and emits the values back into the stream. Unlike SwitchMap, ConcatMap does not cancel any of its inner observables. It is Similar to MergeMap except for one difference that it maintains the order of its inner observables.

```
let srcObservable= of(1,2,3,4)
let innerObservable= of('A','B','C','D')

srcObservable.pipe(concatMap( val => {
    console.log('Source value '+val)
    console.log('starting new observable')
    return innerObservable
  })
).subscribe(ret=> {
  console.log('Recd ' + ret);
})
```

ConcatMap Vs Map: The map operators emit value as observable. The ConcatMap creates an inner observable, subscribes to it, and emits its value as observable. It emits the value in the order in which it creates the observable.

ExhaustMap: The Angular ExhaustMap maps each value from the source observable into an inner observable, subscribes to it. It then starts emitting the values from it replacing the original value. It then waits for the inner observable to finish. If it receives any new values before the completion of the inner observable it ignores it. It receives a new value after completion of the inner observable, then it creates a new inner observable. The whole process repeats itself until the source observable is completes

```
let srcObservable= of(1,2,3,4)
let innerObservable= of('A','B','C','D')

srcObservable.pipe(exhaustMap( val => {
   console.log('Source value '+val)
   console.log('starting new observable')
   return innerObservable
 })
).subscribe(ret=> {
  console.log('Recd ' + ret);
})
```

31. DebounceTime & Debounce in Angular

DebounceTime & Debounce are the Angular RxJs Operators. Both emit values from the source observable, only after a certain amount of time has elapsed since the last value. Both emit only the latest value and discard any intermediate values. In this tutorial, we will learn how to use both DebounceTime & Debounce with examples.

Use Case of Debounce Operators

The typeahead/autocomplete fields are one of the most common use cases for Debounce Operators.

As the user types in the typeahead field, we need to listen to it and send an HTTP request to the back end to get a list of possible values. If we send HTTP requests for every keystroke, we end up making numerous unneeded calls to the server.

By using the Debounce Operators, we wait until the user pauses typing before sending an HTTP Request. This will eliminates unnecessary HTTP requests.
DebounceTime: The Debouncetime emits the last received value from the source observable after a specified amount of time has elapsed without any other value appearing on the source Observable
How it works

- First, we assign a timeout duration (dueTime) to the Debouncetime operator.
- The Debouncetime operator starts counting time after it receives a value.
- If the source observable emits a value before the timeout duration, then counting is reset to zero & started again.
- When the timeout duration elapses the operator emits the last value and the counting stops.
- Counting starts again when the operators receive another value.

DebounceTime Example: The following DebounceTime example shows how to use the operator to listen for input field changes

```
import { Component, VERSION } from "@angular/core";
import { FormControl, FormGroup } from "@angular/forms";
import { Observable, Subscription } from "rxjs";
import { debounceTime } from "rxjs/operators";
@Component({
  selector: "my-app",
  template: ` <form [formGroup]="mform">Name: <input
formControlName="name" /></form>`,
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  mform: FormGroup = new FormGroup({
    name: new FormControl()
  });
  obs:Subscription;
  ngOnInit() {

this.obs=this.mform.valueChanges.pipe(debounceTime(500)).subscribe(dat
a => console.log(data));
```

```
  }
  ngOnDestroy() {
    this.obs.unsubscribe();
  }
}
```
Sending HTTP GET Request: The following example shows how you can send an HTTP GET Request using switchMap & debounceTime.

```
 ngOnInit() {

this.obs=this.mform.valueChanges.pipe(debounceTime(500),switchMap(id
=> {
          console.log(id)
          return this.http.get(url)
        })
      ).subscribe(data => console.log(data));
  }
```

Debounce: The Debounce operator is similar to the DebounceTime operator, but another observable will provide the time span. By Using the Debounce, we can set the time span dynamically.

Debounce Example: The following example works the same as the previous example. Instead of using the hardcoded time span, we create a new observable using the interval operator.

```
import { Component, VERSION } from "@angular/core";
import { FormControl, FormGroup } from "@angular/forms";
import { debounce } from "rxjs/operators";
import { interval, Subscription } from "rxjs";
@Component({
  selector: "my-app",
  template: `<form [formGroup]="mform">Name: <input
formControlName="name" /></form> `,
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  mform: FormGroup = new FormGroup({
    name: new FormControl()
  });
  obs:Subscription
  ngOnInit() {
    this.obs=this.mform.valueChanges.pipe(debounce(() =>
interval(500))).subscribe(data => console.log(data));
  }
  ngOnDestroy() {
    this.obs.unsubscribe()
  }
}
```

You can also customize the delay. In the following example, we increase the delay by 100ms after every keystroke.

```
 delay = 500;
  obs: Subscription;
  ngOnInit() {
    this.obs = this.mform.valueChanges.pipe(debounce(() => {
          this.delay = this.delay + 100;
          console.log(this.delay);
          return interval(this.delay);
        })
      ).subscribe(data => console.log(data));
  }
```

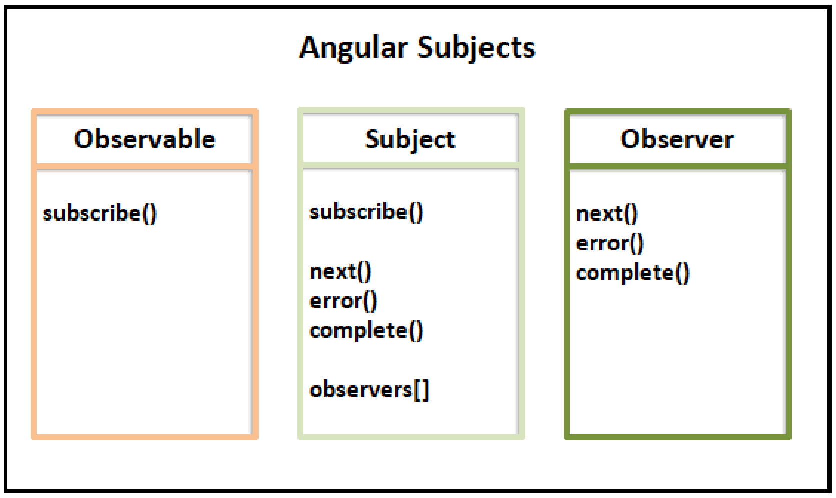
32. Subjects in Angular

The Subjects are special observable which acts as both observer & observable. They allow us to emit new values to the observable stream using the next method. All the subscribers, who subscribe to the subject will receive the same instance of the subject & hence the same values.

Subjects: A Subject is a special type of Observable that allows values to be multicasted to many Observers. The subjects are also observers because they can subscribe to another observable and get value from it, which it will multicast to all of its subscribers.

Basically, a subject can act as both observable & an observer.

How does Subjects work: Subjects implement both subscribe method and next, error & complete methods. It also maintains a collection of observers[]



An Observer can subscribe to the Subject and receive value from it. Subject adds them to its collection observers. Whenever there is a value in the stream it notifies all of its Observers.
The Subject also implements the next, error & complete methods. Hence it can subscribe to another observable and receive values from it.
Creating a Subject in Angular: The following code shows how to create a subject in Angular.

```
import { Component, VERSION } from "@angular/core";
import { Subject } from "rxjs";
@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  subject$ = new Subject();
  ngOnInit() {
    this.subject$.subscribe(val => {
      console.log(val);
    });
    this.subject$.next("1");
    this.subject$.next("2");
    this.subject$.complete();
  }
}
```

The code that creates a subject.

```
subject$ = new Subject();
```

We subscribe to it just like any other observable.

```
this.subject$.subscribe(val => {
  console.log(val);
});
```

The subjects can emit values. You can use the next method to emit the value to its subscribers. Call the complete & error method to raise complete & error notifications.

```
this.subject$.next("1");
this.subject$.next("2");
this.subject$.complete();
//this.subject$.error("error")
```

Subjects are Multicast: Another important distinction between observable & subject is that subjects are multicast.
More than one subscriber can subscribe to a subject. They will share the same instance of the observable. This means that all of them receive the same event when the subject emits it. Multiple observers of an observable, on the other hand, will receive a separate instance of the observable.
Multicast vs Unicast: The Subjects are multicast.
Consider the following example, where we have an observable and subject defined. The observable generates a random number and emits it.

```
import { Component, VERSION } from "@angular/core";
import { from, Observable, of, Subject } from "rxjs";
@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  observable$ = new Observable<number>(subscriber => {
    subscriber.next(Math.floor(Math.random() * 200) + 1);
  });
  subject$ = new Subject();
  ngOnInit() {
    this.observable$.subscribe(val => {
      console.log("Obs1 :" + val);
    });
    this.observable$.subscribe(val => {
      console.log("Obs2 :" + val);
    });
    this.subject$.subscribe(val => {
      console.log("Sub1 " + val);
    });
    this.subject$.subscribe(val => {
      console.log("Sub2 " + val);
    });
    this.subject$.next(Math.floor(Math.random() * 200) + 1);
  }
}
```

We have two subscribers of the observable. Both these subscribers will get different values. This is because observable on subscription creates a separate instance of the producer. Hence each one will get a different random number

```
this.observable$.subscribe(val => {
  console.log("Obs1 :" + val);
});
this.observable$.subscribe(val => {
  console.log("Obs2 :" + val);
});
```

Next, we create two subscribers to the subject. The subject emits the value using the random number. Here both subscribets gets the same value.

```
this.subject$.subscribe(val => {
  console.log("Sub1 " + val);
});
this.subject$.subscribe(val => {
  console.log("Sub2 " + val);
});
this.subject$.next(Math.floor(Math.random() * 200) + 1);
```

33. BehaviorSubject, ReplaySubject & AsyncSubject in Angular

BehaviorSubject: BehaviorSubject requires an initial value and stores the current value and emits it to the new subscribers.

```
import { Component, VERSION } from "@angular/core";
import { BehaviorSubject, Subject } from "rxjs";
@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  subject$ = new BehaviorSubject("0");
  ngOnInit() {
    this.subject$.subscribe(val => {
      console.log("Sub1 " + val);
    });
    this.subject$.next("1");
    this.subject$.next("2");
    this.subject$.subscribe(val => {
      console.log("sub2 " + val);
    });
    this.subject$.next("3");
    this.subject$.complete();
  }
}
```

We create a new BehaviorSubject providing it an initial value or seed value. The BehaviorSubject stores the initial value.

```
    subject$ = new BehaviorSubject("0");
```

As soon as the first subscriber subscribes to it, the BehaviorSubject emits the stored value. i.e. 0

```
    this.subject$.subscribe(val => {
      console.log("Sub1 " + val);
    });
```

We emit two more values. The BehaviorSubject stores the last value emitted i.e. 2

```
    this.subject$.next("1");
    this.subject$.next("2");
```

Now, Subscriber2 subscribes to it. It immediately receives the last value stored i.e. 2

```
    this.subject$.subscribe(val => {
```

```
    console.log("sub2 " + val);
  });
```

ReplaySubject: ReplaySubject replays old values to new subscribers when they first subscribe.
The ReplaySubject will store every value it emits in a buffer. It will emit them to the new subscribers in the order it received them. You can configure the buffer using the arguments bufferSize and windowTime
bufferSize: No of items that ReplaySubject will keep in its buffer. It defaults to infinity.
windowTime: The amount of time to keep the value in the buffer. Defaults to infinity.

```
import { Component, VERSION } from "@angular/core";
import { ReplaySubject, Subject } from "rxjs";
@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  subject$ = new ReplaySubject();
  ngOnInit() {
    this.subject$.next("1");
    this.subject$.next("2");
    this.subject$.subscribe(
      val => console.log("Sub1 " + val),
      err => console.error("Sub1 " + err),
      () => console.log("Sub1 Complete")
    );
    this.subject$.next("3");
    this.subject$.next("4");
    this.subject$.subscribe(val => {
      console.log("sub2 " + val);
    });
    this.subject$.next("5");
    this.subject$.complete();
    this.subject$.error("err");
    this.subject$.next("6");
    this.subject$.subscribe(
      val => {
        console.log("sub3 " + val);
      },
      err => console.error("sub3 " + err),
      () => console.log("Complete")
    );
  }
}
```

First, we create a ReplaySubject

```
    subject$ = new ReplaySubject();
```

ReplaySubject emits two values. It will also store these in a buffer.

```
    this.subject$.next("1");
    this.subject$.next("2");
```

We subscribe to it. The observer will receive 1 & 2 from the buffer

```
    this.subject$.subscribe(
      val => console.log("Sub1 " + val),
      err => console.error("Sub1 " + err),
      () => console.log("Sub1 Complete")
    );
```

We subscribe again after emitting two more values. The new subscriber will also receive all the previous values.

```
    this.subject$.next("3");
    this.subject$.next("4");
    this.subject$.subscribe(val => {
      console.log("sub2 " + val);
    });
```

We emit one more value & complete. All the subscribers will receive complete. They will not receive any further values or notifcations.

```
    this.subject$.next("5");
    this.subject$.complete();
```

We now fire an error notification and a value. None of the previous subscribers will receive this as they are already closed.

```
    this.subject$.error("err");
    this.subject$.next("6");
```

Now, we subscribe again. The subscriber will receive all the values up to Complete. But will not receive the Complete notification, instead, it will receive the Error notification.

```
  this.subject$.subscribe(
    val => {
      console.log("sub3 " + val);
    },
    err => console.error("sub3 " + err),
    () => console.log("Complete")
  );
```

AsyncSubject: AsyncSubject only emits the latest value only when it completes. If it errors out then it will emit an error, but will not emit any values.

```
import { Component, VERSION } from "@angular/core";
import { AsyncSubject, Subject } from "rxjs";
@Component({
  selector: "my-app",
  templateUrl: "./app.component.html",
  styleUrls: ["./app.component.css"]
})
export class AppComponent {
  subject$ = new AsyncSubject();
  ngOnInit() {
    this.subject$.next("1");
    this.subject$.next("2");
    this.subject$.subscribe(
      val => console.log("Sub1 " + val),
      err => console.error("Sub1 " + err),
      () => console.log("Sub1 Complete")
    );
    this.subject$.next("3");
    this.subject$.next("4");
    this.subject$.subscribe(val => {
      console.log("sub2 " + val);
    });
    this.subject$.next("5");
```

```
        this.subject$.complete();
        this.subject$.error("err");
        this.subject$.next("6");
        this.subject$.subscribe(
          val => console.log("Sub3 " + val),
          err => console.error("sub3 " + err),
          () => console.log("Sub3 Complete")
        );
      }
    }
```

In the above example, all the subscribers will receive the value 5 including those who subscribe after the complete event.
But if the AsyncSubject errors out, then all subscribers will receive an error notification and no value.

34. View Encapsulation in Angular

View Encapsulation in Angular defines how the styles defined in the template affects the other parts of the application. The angular uses three strategies, while rendering the view **ViewEncapsulation.Emulated, ViewEncapsulation.ShadowDOMand ViewEncapsulation.None** This article describes what is View Encapsulation using an example and how it is implemented in angular. We also learn what is shadow dom in Angular.
The Angular allows us the specify the component specific styles. This is done by specifying either inline style styles: or using the external style sheet styleUrls: in the @Component decorator or @directive decorator.

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

What is Encapsulation ?

One of the fundamental concept in object oriented programming (OOP) is Encapsulation. It defines the idea that all the data and methods that operate on that data are kept private in a single unit (or class). It is like hiding the implementation detail from the outside world. The consumer of encapsulated object know that it works.

What is View Encapsulation in Angular ?
The View Encapsulation in Angular is a strategy which determines how angular hides (encapsulates) the styles defined in the component from bleeding over to the the other parts of the application.
The following three strategies provided by the Angular to determine how styles are applied.

- ViewEncapsulation.None
- ViewEncapsulation.Emulated
- ViewEncapsulation.ShadowDOM

The viewEncapsulation Native is deprecated since Angular version 6.0.8, and is replaced by viewEncapsulation ShadowDom
Adding View Encapsulation to components: The Encapsulation methods are added using the encapsulation metadata of the @Component decorator as shown below

```
@Component({
  template: `<p>Using Emulator</p>`,
  styles: ['p { color:red}'],
  encapsulation: ViewEncapsulation.Emulated     //This is default
//encapsulation: ViewEncapsulation.None
//encapsulation: ViewEncapsulation.ShadowDOM
})
```

ViewEncapsulation.None
The ViewEncapsulation.None is used, when we do not want any encapsulation. When you use this, the styles defined in one component affects the elements of the other components. Now, let us look at ViewEncapsulation.None does.
Create a new component ViewNoneComponent and add the following code.

```
import { Component,ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-none',
  template: `<p>I am not encapsulated and in blue
          (ViewEncapsulation.None) </p>`,
  styles: ['p { color:blue}'],
  encapsulation: ViewEncapsulation.None
})
export class ViewNoneComponent {
```

}
We have added encapsulation: ViewEncapsulation.None. We have also defined the inline style p { color:blue}
Do not forget to import & declare the component in AppModule. You also need to add the <app-none></app-none> selector in app.component.html

The important points are

- The styles defined in the component affect the other components
- The global styles affect the element styles in the component

ViewEncapsulation.Emulated

In a HTML page, we can easily add a id or a class to the element to increase the specificity of the CSS rules so that the CSS rules do not interfere with each other.. The ViewEncapsulation.Emulated strategy in angular adds the unique HTML attributes to the component CSS styles and to the markup so as to achieve the encapsulation. This is not true encapsulation. The Angular Emulates the encapsulation, Hence the name Emulated.
If you do not specify encapsulations in components, the angular uses the ViewEncapsulation.Emulated strategy
Create a new component in Angular app and name it as ViewEmulatedComponent. as shown below

```
import { Component,ViewEncapsulation } from '@angular/core';
 @Component({
  selector: 'app-emulated',
  template: `<p>Using Emulator</p>`,
  styles: ['p { color:red}'],
  encapsulation: ViewEncapsulation.Emulated
})
export class ViewEmulatedComponent {
}
```

We have not added any id in the above component. Only change this component has with one ViewNoneComponent was encapsulation mode, which is set to ViewEncapsulation.Emulated
Now, you can see that the style does not spill out to other components, when you use emulated mode. i.e because angular adds **_ngcontent-c#** attributes to the emulated components and makes necessary changes in the generated styles

The important points are

- This strategy isolates the component styles. They do not bleed out to other components.
- The global styles may affect the element styles in the component
- The Angular adds the attributes to the styles and mark up

ViewEncapsulation.ShadowDOM: The Shadow DOM is a scoped sub-tree of the DOM. It is attached to a element (called shadow host) of the DOM tree. The shadow dom do not appear as child node of the shadow host, when you traverse the main DOM.
The browser keeps the shadow DOM separate from the main DOM. The rendering of the Shadow dom and the main DOM happens separately. The browser flattens them together before displaying it to the user. The feature, state & style of the Shadow DOM stays private and not affected by the main DOM. Hence it achieves the true encapsulation.
The Shadow DOM is part of the Web Components standard. Not all browsers support shadow dom. Use google chrome for the following examples
To create shadow dom in angular , all we need to do is to add the ViewEncapsulation.ShadowDom as the encapsulation strategy.
Create a new component ViewShadowdomComponent and add the following code

```
import { Component,ViewEncapsulation } from '@angular/core';
@Component({
  selector: 'app-shadowdom',
  template: `<p>I am encapsulated inside a Shadow DOM
ViewEncapsulation.ShadowDom</p>`,
  styles: ['p { color:brown}'],
  encapsulation: ViewEncapsulation.ShadowDom
})
export class ViewShadowdomComponent {
 }
```

Shadow dom terminology: The app-shadowdom is the CSS selector in the ViewShadowdomComponent. We used it in our app-component.html. The Angular renders component as shadow dom and attaches it to the app-shadowdom selector. Hence, we call the element as Shadow host
The Shadow DOM starts from #shadow-root element. Hence, we call this element as shadow root. The Angular injects the component into the shadow root.

The Shadow boundary starts from the #shadow-root. The browser encapsulates everything inside this element including the node #shadow-root
The shadow dom archives the true encapsulation. It truly isolates the component from the styles from the other parts of the app.
The styles from the parent component & sibling components are still injected into the shadow dom. but that is an angular feature. The angular wants the component to share the parent & sibling styles. Without this the component may look out of place with the other component

The important points are

- The shadow dom achieves the true encapsulation
- The parent and sibling styles still affect the component. but that is angular's implementation of shadow dom
- The shadow dom not yet supported in all the browsers. You can check it from the this link.

35.

36.