Angular 14 - Overview

WrapPixel

What is Angular?

Angular is a development platform, built on TypeScript. As a platform, Angular includes:

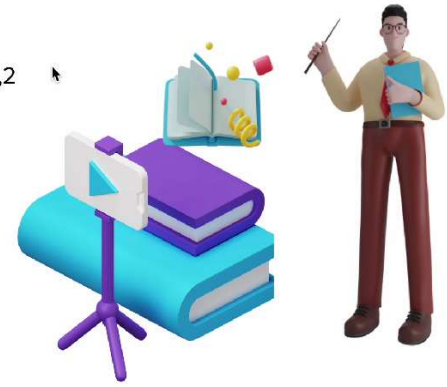
- A component-based framework for building scalable web applications
- A collection of well-integrated libraries that cover a wide variety of features, including routing, forms management,client-server communication, and more
- A suite of developer tools to help you develop, build, test, and update your code

Current Stable Version?

- 14
- Previous versions: 13, 12,11, 10,9, 8, 7,6,5,4,2

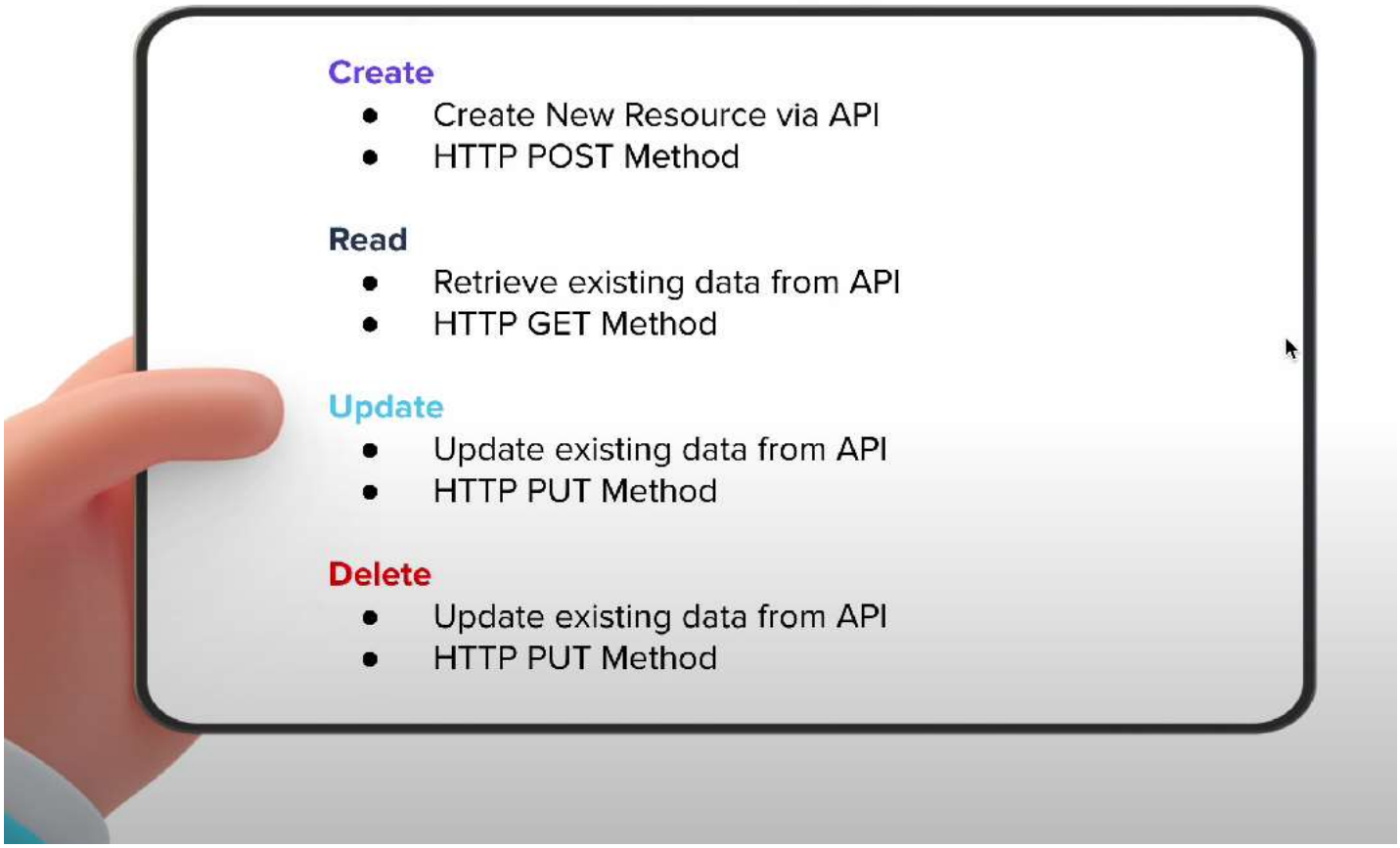
Important concepts to learn

1. Components
    - Standalone Components
2. Services
3. Directives
4. Pipes
5. Templates
6. Forms
7. Routing
8. Lazy Loading



ARC Tutorials

CRUD - Overview



REST - Overview

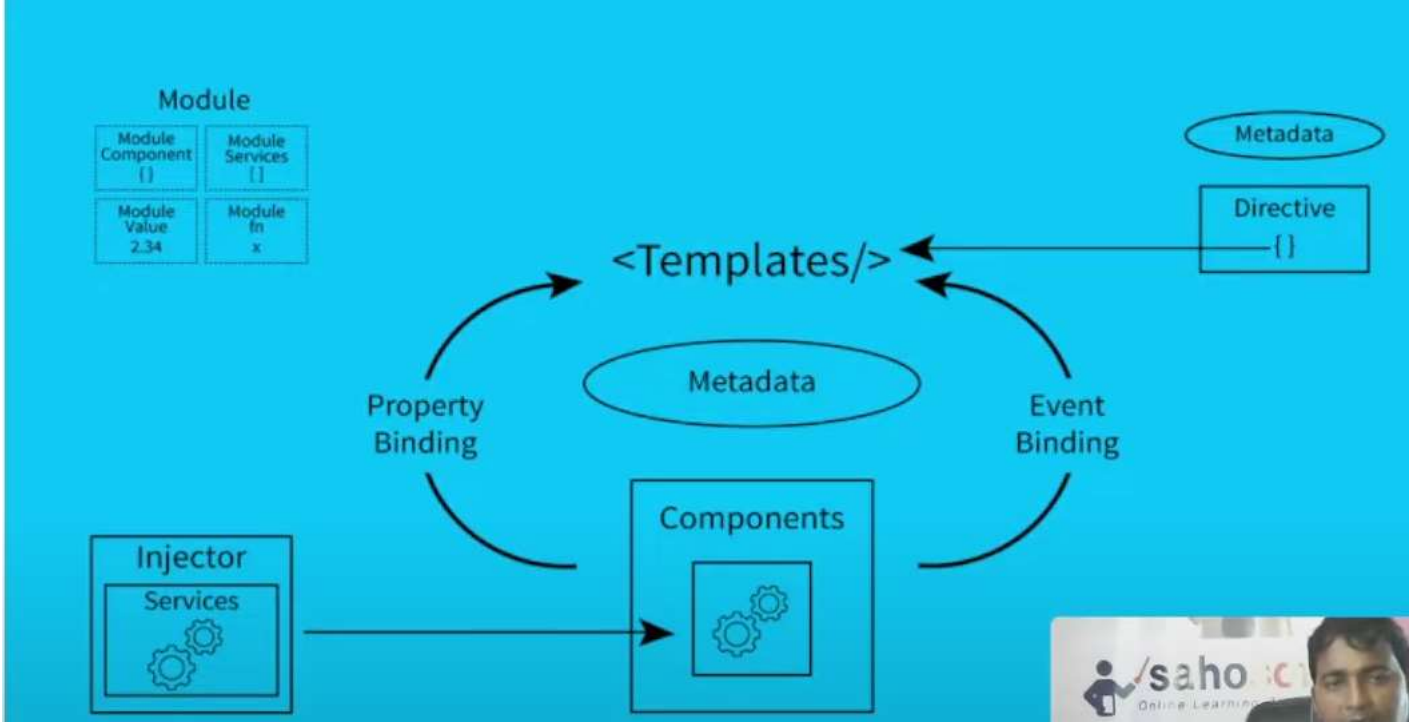
A REST API (also known as RESTful API) is **an application programming interface (API or web API) that conforms to the constraints of REST architectural style and allows for interaction with RESTful web services**

- 1xx: Informational – Communicates transfer protocol-level information.
- 2xx: Success – Indicates that the client's request was accepted successfully.
- 3xx: Redirection – Indicates that the client must take some additional action in order to complete their request.
- 4xx: Client Error – This category of error status codes points the finger at clients.
- 5xx: Server Error – The server takes responsibility for these error status codes

HTTP Methods - Overview

HTTP Verb	CRUD	Entire Collection (e.g. /customers)	Specific Item (e.g. /customers/{id})
POST	Create	201 (Created), 'Location' header with link to /customers/{id} containing new ID.	404 (Not Found), 409 (Conflict) if resource already exists..
GET	Read	200 (OK), list of customers. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single customer. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method Not Allowed), unless you want to update/replace every resource in the entire collection.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
PATCH	Update/Modify	405 (Method Not Allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method Not Allowed), unless you want to delete the whole collection—not often desirable.	200 (OK). 404 (Not Found), if ID not found or invalid.

Architecture:



Angular 14 Tutorial

Modules

Module in Angular refers to a place where you can group the components, directives, pipes, and services, which are related to the application.
In case you are developing a website, the header, footer, left, center and the right section become part of a module.

```
import { TestcompComponent } from './testcomp/testcomp.component';
import { EmployeeComponent } from './employee/employee.component';

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

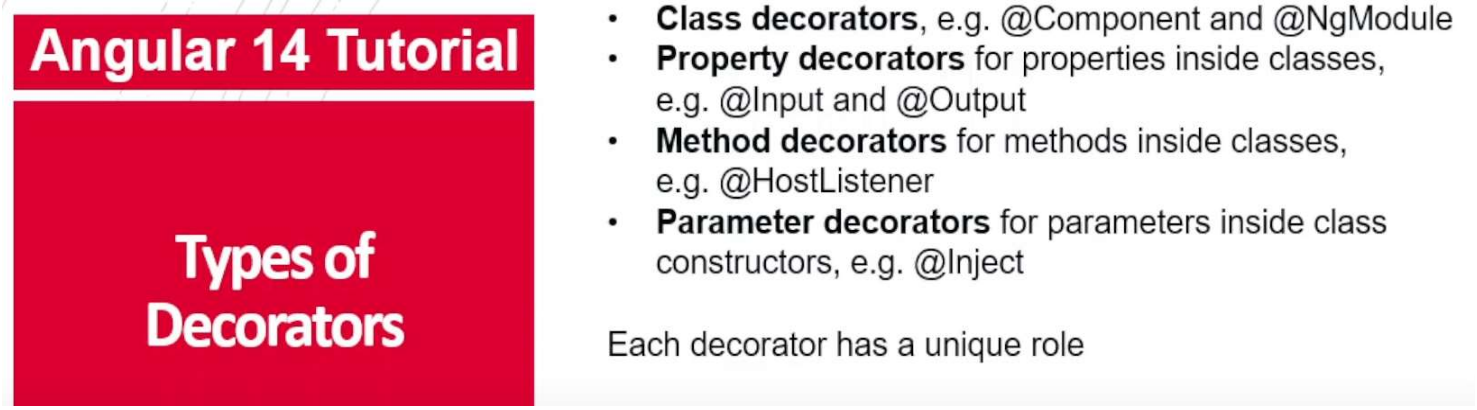
Here's how each property in the @NgModule decorator functions:

1. **Declarations**: This array holds all the components, directives, and pipes that belong to this module. Any component or directive that you want to use in this module must be declared here.
2. **Imports**: This array includes all the other modules that this module depends on. You must import any module that contains components, directives, or pipes that you want to use within your module.
3. **Providers:** This array is for services that the module provides. You can register services that can be injected throughout the module. If a service is declared here, it will be available for any component within this module.
4. **Exports:** This array holds the modules, components, directives, or pipes that you want to make available to other modules. If you want to use components or directives from this module in another module, you must export them.
5. **BootStrap:** This array specifies the root component of the application, which Angular will bootstrap when the application starts. It typically includes only one component, which is the main component (often called AppComponent) that serves as the entry point of your application.

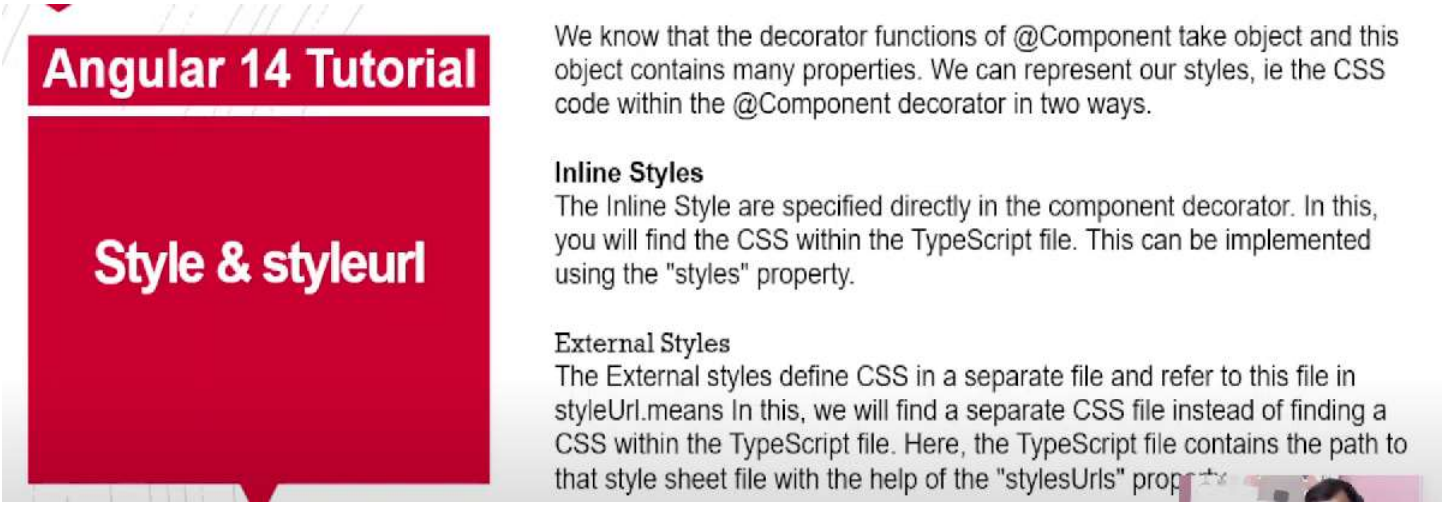
- **Decorators** are a feature of TypeScript and are implemented as functions. The name of the decorator starts with @ symbol following by brackets and arguments, since decorators are just functions in TypeScipt.
- Decorators are simply functions that return functions. These functions supply metadata to Angular about a particular class, property, value, or method…
- **Decorators are invoked at runtime.**
- Decorators allow you to execute functions. For example @Component executes the Component function imported from Angular 14.

- @NgModule() to define modules…
- @Component() to define compents…
- @Injectable() to define services…
- @Input() and @Output() to define properties…that send and receive data from the dom.

There are many built-in decorators available in Angular…and many properties on each decorator



>> Decorators gets automatically called.



>>> **Use Style:** writing inline code in component.

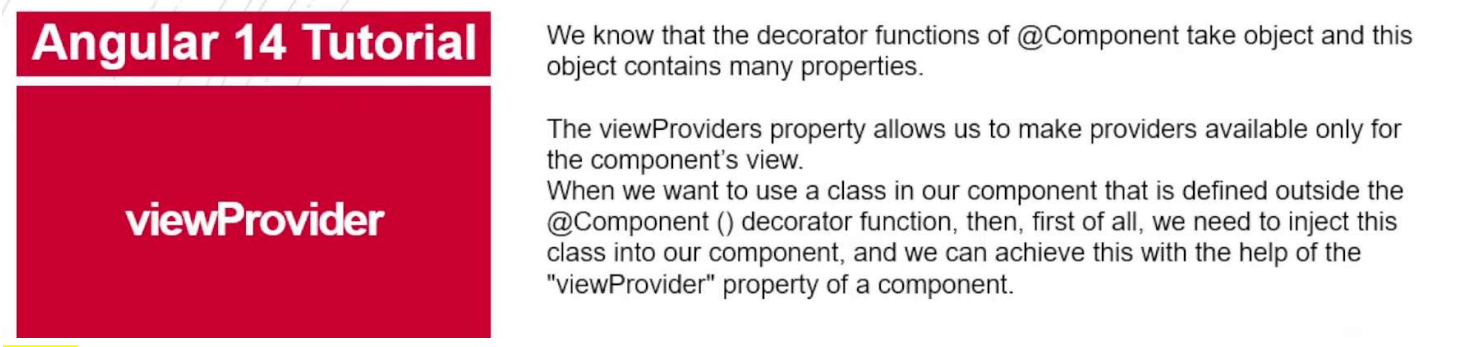
```
@Component({
  selector: 'app-student',
  templateUrl: './student.component.html',
 // styleUrls: ['./student.component.css']
 styles:['h1{background-color:pink}']
})
```

>>>> **Use Styleurl:** writing external css file address in component.

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

or using multiple file:

```
@Component({
  selector: 'app-student',
  templateUrl: './student.component.html',
 /styles:['h1{background-color:pink}'],
 styleUrls: ['./student.component.css','./test.css'],
})
```



Note: The viewProviders array is used to define services/classes that are only needed by a specific component and its child components. This ensures that these services are instantiated at the component level rather than at the module level, which

helps to prevent unnecessary reloading and provides better encapsulation of the service's state. Using viewProviders is beneficial for optimizing performance and managing dependencies more effectively, especially when the service is only relevant to that particular component.

Example Scenario

Suppose you have a simple application with a parent component called AppComponent and a child component called ChildComponent. The child component needs a service for managing some state or functionality, but this service should only be used by ChildComponent and its descendants. For this, we can use viewProviders.==> i.e immediate child component ddeclared in template of parent component will have access to the service , other components declared inside template won't be able to access this service , if viewProvider is set in the parent component.
Others using service in this case will throw errror on use: "No provider for ChildService!."
This is because: OtherComponent is a sibling to ChildComponent and does not have access to the instance of ChildService provided in AppComponent."

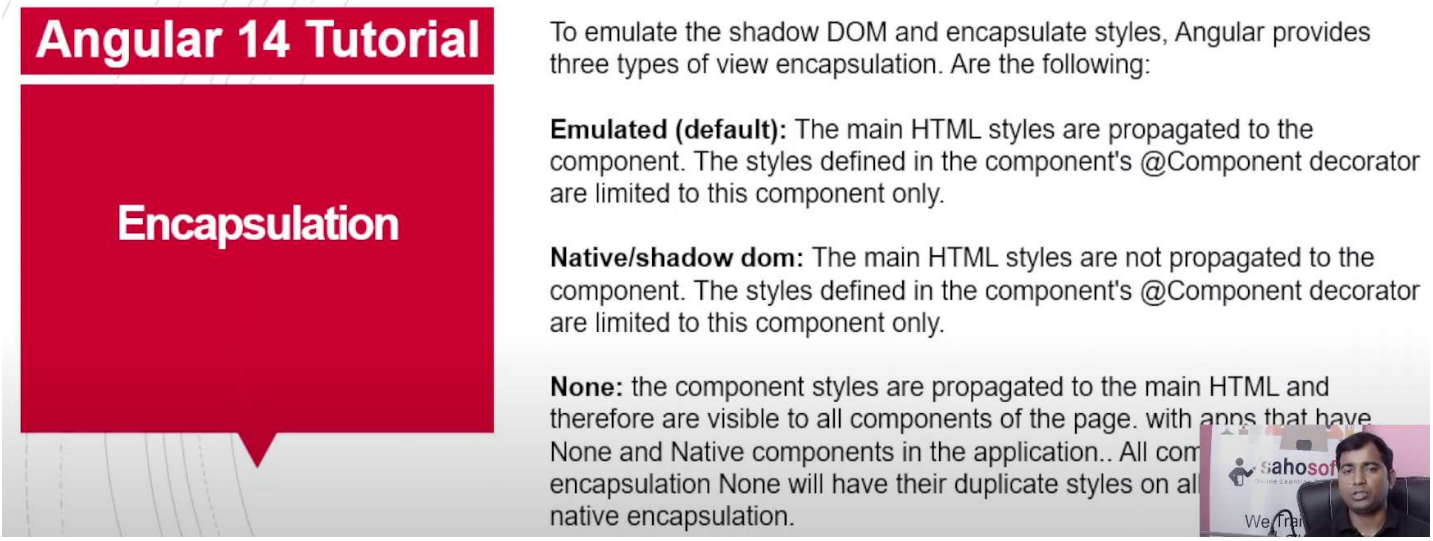
```
// app.component.ts
import { Component } from '@angular/core';
import { ChildService } from './child.service';

@Component({
  selector: 'app-root',
  template: `
    <h1>Parent Component</h1>
    <app-child></app-child>    <!-- Child component -->
    <app-other></app-other>    <!-- Sibling component -->
  `,
  viewProviders: [ChildService]  // Providing the service only to the ChildComponent
})
export class AppComponent {}
```

==========

View encapsulation

It determines how styles defined in a component are scoped and how they affect the rest of the application.



Types:

>> ViewEncapsulation.Emulated(default): Means Style encapsulation **i.e parent css** not going to **child.** No Shadow DOM **.** This is the default encapsulation method in Angular. Styles are scoped to the component and do not leak out. Angular adds unique attributes to the component's styles and elements, ensuring that they do not affect other components**.**

Example: Imagine you have a shirt that has a special pattern only for you. Even if you wear it in a crowd, that pattern won't be seen by others; it's just for you.

```
  `,
  styles: [`
    .child {
      color: red; /* This style will only apply to this child component */
    }
  `],
  encapsulation: ViewEncapsulation.Emulated, // Default encapsulation
})
export class ChildComponent {}
```

>> ViewEncapsulation.Native/Shadow dom: **DEPRECATED**

>> ViewEncapsulation.None: Styles defined in a component are applied globally. This means they can affect any child component or element in the application.

Note: Say in child component if given this(**encapsulation: ViewEncapsulation.None**) then no encapsulation will be there and styles of parent component will override style of child component.

```
@Component({
  selector: 'app-child',
  template: `
    <div class="child">
      This is a child component!
    </div>
  `,
  styles: [`
    .child {
      color: red; /* This will be overridden by the parent style */
    }
  `],
  encapsulation: ViewEncapsulation.None, // No encapsulation
})
export class ChildComponent {}
```

==========

Provider at Module and Component level

It mwans we can provide metadata service at module level making it singleton and at component level as well for encapsulation.

Angular 14 Tutorial

Providers Property

providers metadata is available in the @NgModule() decorator and @Component() decorator. provider's metadata is used to configure the provider.

A provider provides concrete and runtime version of a dependency value. The injector injects the objects provided by provider into components and services. Therefore, it is necessary to configure a service with the provider, otherwise the injector will not be able to inject it.

The injector injects the objects provided by provider into components and services. Only those classes configured by providers are available for dependency injection (DI).

directive ngIf and else and then

html Copy code

```
<ng-template #thenBlock>Then Block Content</ng-template>
<ng-template #elseBlock>Else Block Content</ng-template>
<div *ngIf="condition; then thenBlock; else elseBlock"></div>
```

```
<h1>
  Welcome to {{ title }}!
</h1>
<h2>Using Ngif</h2>
<input type="radio" name="rdio1" (click)="changevalue(true)">True
<input type="radio" name="rdio1" (click)="changevalue(False)">False
<div *ngIf="isvalid">
  This is Valid data
</div>
<div *ngIf="!isvalid">
  This is not Valid data
</div>

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

```
export class AppComponent {
  title = 'SahoSoft Solutions !';
  isvalid:boolean=true;
  changevalue(valid){
    this.isvalid=valid;
  }
}
```

directive ngSwitch

used as property binding:

```
Enter Your Name : <input type="text" #txtname (keyup)="0" />

<div [ngSwitch]="txtname.value">
  <div *ngSwitchCase="'Ram'">Hello Ram</div>
  <div *ngSwitchCase="'Mohan'">Hello Mohan</div>
  <div *ngSwitchCase="'Sohan'">Hello Sohan</div>
  <div *ngSwitchCase="'Rohit'">Hello Rohit</div>
  <div *ngSwitchCase="'Soni'">Hello Soni</div>
  <div *ngSwitchDefault>No Data Matched</div>

</div>
```

```
export class AppComponent {
  title = 'myangularapp';
  name="Ram";
```

directive ngfor

used as property binding:

```
<h1>
  Welcome to {{ title }}
</h1>
<h2>Using NgFor </h2>
<ul>
  <li *ngFor="let student of Students ; let ii=index ; let f=first; let l = last ; let ev=even;let od=odd">
  {{i+1}} - {{student.name}} - {{f}} - {{l}}  - {{ev}} - {{od}}
  </li>
</ul>

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

Using NgFor

- 1 - Rahul Kumar - true - false - true - false
- 2 - Ajeet Kumar - false - false - false - true
- 3 - Rohan Kumar - false - false - true - false
- 4 - Mahesh Kumar - false - false - false - true
- 5 - Chandan Kumar - false - false - true - false
- 6 - Soni Kumari - false - true - false - true

TrackBy with ngfor

The use of trackBy it's a performance optimization and is usually not needed by default, it's in principle only needed if running into performance issues.
Suppose we have some data coming from some API request into the collection and we need to change the data over the web page using ngFor directive. In this case, Angular 14 will remove all the DOM elements that associated with the data and will create them again in the DOM tree, even the same data is coming (as shown in below screenshot). That means a lot of Dom manipulations will happen if a large amount of data coming from the API.

The solution is, we can use trackBy function, which will be helpful for Angular to track the items which have been added or removed.
TrackBy function will take two arguments first is index and second is current item and we can return the unique identifier as a return argument.

```
trackByStudentID(index: number, student: any): string {
  return student.studentID;
}
```

>>>> Used to bind data say manipulated data on screen i.e say transformed data to be binded . Say some data comes comes api loads on screen i.e whole dome rendered , NOw if we add or remove some data then whole dom will again be rerendered and time/bandwidth consumption will be there so to save it and optimize it TrackBy *ngFor comes for rescue.
SO what here **trackby does is instead of loading whole data again along with updated , newly added one , it only loads the updated one keeping the older as it is** .....it makes ux more appropriate and optimized.
eg: can be use on button(Show more.)
Eg: First we created / say some data is coming. named students array of object.
Then we say created another data to be added or shown along with this data on button click:

```
export class AppComponent {
  title = 'SahoSoft Solutions !';
students:any[];
constructor(){
  this.students=[
    {
      studentid:1,
      name:'Chandan',
      gender:'Male',
      age:27,
      Course:'MCA'

    },
    {
     studentid:1,
     name:'Chandan',
     gender:'Male',
     age:27,
     Course:'MCA'

    },
  ];
```

```
trackbystudentid(index:number,student:any):string{
  return student.studentid;
}
```

Also create a method below this new method to track this.student.
here in the getmorestudents() we will put all a=our old data inside like along with some new objects:
getmorestudents(){ this.students =[{},{},.....]}
new we will bind first our old data on screen then on button click will bind this changed data on screen.. So here on useing trackById tracking by id will only bind the changed updated new added data leave the old as it is.

```
  <h2>Track by example</h2>

  <table border="1">
    <tr *ngFor="let s of students; let i=index;trackBy:trackbystudentid">
<td>{{i+1}}</td>
<td>{{s.studentid}}</td>
<td>{{s.name}}</td>
<td>{{s.gender}}</td>
<td>{{s.age}}</td>
<td>{{s.Course}}</td>

</tr>
  </table>
  <br/>
  <button (click)="getmorestudents()">Get more students</button>
<router-outlet></router-outlet>
```

Grouping with ngfor

Binding data in group wise to display on screen in groupwise:

using the *ngFor directive allows you to iterate over an array or list to display items dynamically in your template. If you want to group items based on certain criteria (for example, grouping a list of products by category), you can achieve this by transforming your data structure before passing it to the template.

```
  <h1>
    Welcome to {{ title }}
  </h1>
  <h2>Grouping by example</h2>
<ul *ngFor="let group of Countrydetails">
  <li>{{group.country}}</li>
<ul>
  <li *ngFor="let person of group.people">
{{person.name}}
  </li>
</ul>
</ul>

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

```
Countrydetails:any[]=[
  {
    'country': 'India',
    'people':[
      {
        "name":"Ajeet Kumar"
      },
      {
        "name":"Chandan Singh"
      },
      {
        "name":"Mohan Singh"
      }
    ]

  },
  {
    'country': 'UK',
    'people':[
```

Output:

Grouping by example

- India
  - Ajeet Kumar
  - Chandan Singh
  - Mohan Singh
- UK
  - ABC
  - XYZ
  - PQR

ngStyle

Angular 14 Tutorial

ngStyle

The ngStyle directive lets you set a given DOM elements style properties.

```
<div [ngStyle]="{'background-color' : green}"></div>
```

ngStyle becomes much more useful when the value is dynamic.

```
<div [ngStyle]="{'background-color' :
person.country==='UK' ? 'green' : 'red'}"></div>
```

```
people:any[]=[
  {
    "name":"Ajeet Kumar",
    "Country":'India'
  },
  {
    "name":"Chandan Singh",
    "Country":'India'
  },
  {
    "name":"ABC",
    "Country":'UK'
  },
  {
    "name":"XYZ",
    "Country":'USA'
  },
  {
    "name":"Ajeet Kumar",
    "Country":'India'
  },
  {
    "name":"PQR",
    "Country":'UK'
  },
  {
    "name":"mnp",
    "Country":'USA'
  },
];
getcolor(Country){
  switch(Country)
  {
    case 'India':
    return 'green';
    case 'UK':
    return 'blue';
    case 'USA':
    return 'red';
```

```
<h1>
  Welcome to {{ title }}
</h1>
<h2>ngStyle Example</h2>
<ul *ngFor="let person of people">
  <li [ngStyle]="{'color': getcolor(person.Country)}">
{{person.name}}- ({{person.Country}})
  </li>
</ul>

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

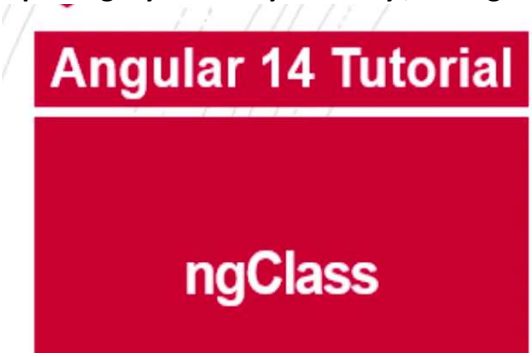
then in html: using ngstyle to bind dynamic color based on country name case. Output:

ngStyle Example

- Ajeet Kumar- (India)
- Chandan Singh- (India)
- ABC- (UK)
- XYZ- (USA)
- cook tyson- (USA)
- Rahul Kumar- (India)
- PQR- (UK)
- mnp- (USA)

ngClass

Updating style class dynamically , adding multiple classes.



ngClass directive allows us to set the CSS class dynamically for a DOM element.

- ✓ ngClass with string
- ✓ ngClass with array
- ✓ ngClass with object
- ✓ ngClass with component method

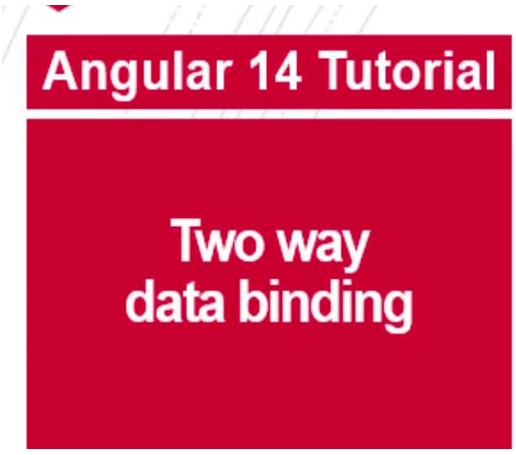
Used as property binding:

1. As String: `<div [ngClass]="active ? 'active' : 'inactive'">Toggle Class</div>`

2. As Array: `<div [ngClass]="['base-class', active ? 'active' : 'inactive']">Toggle Class</div>`

3. As object: `<div [ngClass]="{ 'active': active, 'inactive': !active }">Toggle Class</div>`

2way binding with and without ngModel



In simple words, two-way data binding is a combination of both Property Binding and Event Binding.

```
<input [value]='data1' (input)='data = $event.target.value'>
```

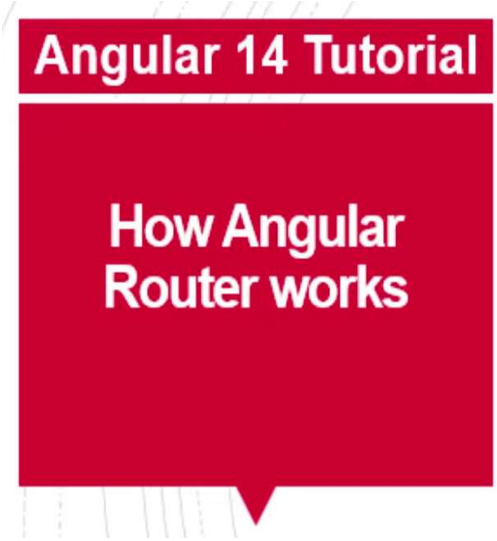
Binding using [(ngModel)] directive

ngModel directive which combines the square brackets of property binding with the parentheses of event binding in a single notation.

```
<input [(ngModel)] ='data'>
```

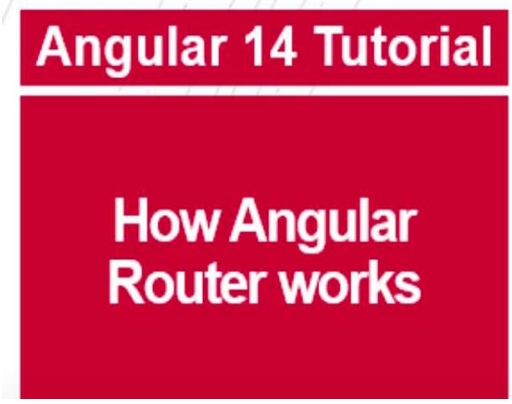
```
  <h1>
    Welcome to {{ title }}
  </h1>
Enter Your Name: <input [value]='data' (input)='data=$event.target.value' ><br/>
Your Name : {{data}}
<input [(ngModel)]='data1'>
Entered value : {{data1}}
<router-outlet></router-outlet>
```

Routing :



When a user navigates to a page, Angular Router performs the following steps in order:
Every time a link is clicked or the browser URL changes, Angular router makes sure your application reacts accordingly.
To accomplish that, Angular router performs the following 7 steps in order:

Step 1 - Parse the URL:
In step 1 of the routing process, Angular router takes the browser URL and parses it as a URL tree.
A URL tree is a data structure that will later help Angular router identify the router state tree in step 3



To parse the URL, Angular uses the following conventions:

- ✓ / - slashes divide URL segments
- ✓ () - parentheses specify secondary routes
- ✓ : - a colon specifies a named router outlet
- ✓ ; - a semicolon specifies a matrix parameter
- ✓ ? - a question mark separates the query string parameters
- ✓ # - a hashtag specifies the fragment
- ✓ // - a double slash separates multiple secondary routes

Angular 14 Tutorial

How Angular Router works

Step 2 – Redirect

Before Angular router uses the URL tree to create a router state, it checks to see if any redirects should be applied. **There are 2 kinds of redirect:**

local redirect: when redirectTo does not start with a slash. replaces a single URL segment

Example: { path: 'one', redirectTo: 'two' }

absolute redirect: when redirectTo starts with a slash. replaces the entire URL

Example: { path: 'one', redirectTo: '/two' }

Only one redirect is applied!

Angular 14 Tutorial

How Angular Router works

Step 3 - Identify the router state

Angular router traverses the URL tree and matches the URL segments against the paths configured in the router configuration.
If a URL segment matches the path of a route, the route's child routes are matched against the remaining URL segments until all URL segments are matched.
If no complete match is found, the router backtracks to find a match in the next sibling route

Angular 14 Tutorial

How Angular Router works

Step 4 - Guard - run guards

At the moment, *any* user can navigate *anywhere* in the application *anytime*. That's not always the right thing to do.
Perhaps the user is not authorized to navigate to the target component.
Maybe the user must login (*authenticate*) first.
Maybe you should fetch some data before you display the target component.
You might want to save pending changes before leaving a component.
You might ask the user if it's OK to discard pending changes rather than save them.
You can add *guards* to the route configuration to handle these scenarios.

Angular 14 Tutorial

How Angular Router works

Step 5 - Resolve - run resolvers
it resolves the required data for the router state.

Step 6- Activate
it activates the Angular components to display the page.

Step 7 – Manage
Finally, when the new router state has been displayed to the screen, Angular router listens for URL changes and state changes. it manages navigation and repeats the process when a new URL is requested.

Router Outlet

* **Router outlet** is a dynamic component that the router uses to displays views based on router navigations.
* Router outlet is a *Routing component*. An Angular component with a RouterOutlet that displays views based on router navigations.
* The role of **<router-outlet>** is to mark where the router displays a view. (This is the location where Angular will insert the component we want to route to on the view)
* The <router-outlet> tells the router where to display routed views.
* The RouterOutlet is one of the router directives that became available to the AppComponent because AppModule imports AppRoutingModule which exported RouterModule.

>>> In side .html where wants ato land/load all routes components:

```
<nav [ngClass] = '"menu"'>
  <a routerLink="/home" routerLinkActive="active-link">Home</a> |
  <a routerLink="/add-book" routerLinkActive="active-link">Add Book</a> |
  <a routerLink="/manage-book" routerLinkActive="active-link">Manage Book</a>
</nav>
<router-outlet></router-outlet>
```

Clientside And Serverside dynamic routing(uses event binding with condition):

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

<h3>Router Link Example</h3>
<a [routerLink]="['/student']">Student</a>-----
<a [routerLink]="['/studentdetails']">Studentdetails</a><br/>
<button (click)="student()" >show student</button>
<div>
<router-outlet></router-outlet>
</div>
```

```
export class AppComponent {
  title = 'SahoSoft Solutions !';
constructor(private router:Router){}
student(){

  this.router.navigate(['/student']);
}
}
```

REDIRECTING:

app-routing.module.ts × app.component.html

```
import { NgModule } from '@angular/core';
import { Routes, RouterModule } from '@angular/router';
import { StudentComponent } from './student/student.component';
import { StudentdetailsComponent } from './studentdetails/studentdetails.component';

const routes: Routes = [
  {
    path:'',redirectTo:'student', pathMatch:'full'
  },
  {
    path:'student',component:StudentComponent
  },
  {
    path:'studentdetails',component:StudentdetailsComponent
  }
];

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

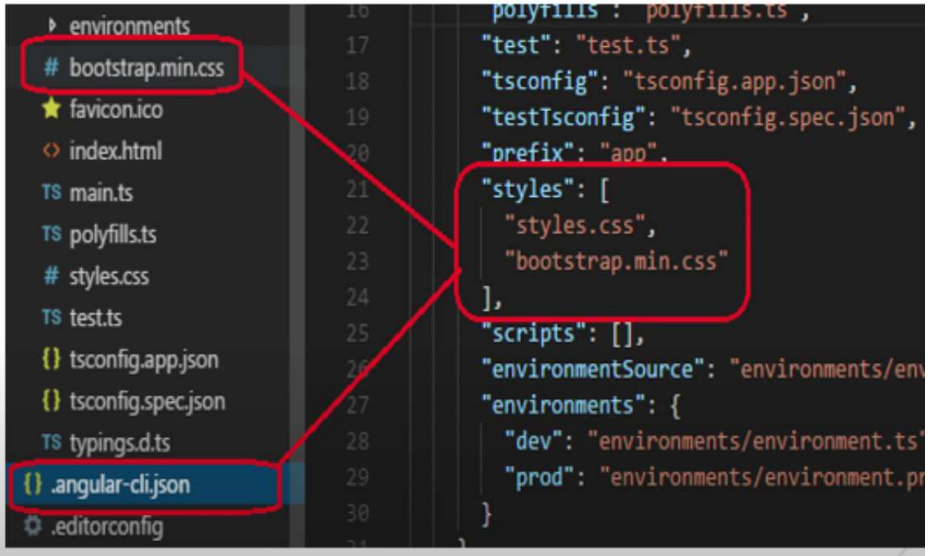
Using Bootstrap with Angular :

>> Step1: Install bootstrap: npm install bootstrap --save
>> Step2: Give path of min.css to angular.json file earlier known as angular.cli.json:

```
            "src/assets"
          ],
          "styles": [
            "src/styles.css",
            "../node_modules/bootstrap/dist/css/bootstrap.min.css"
          ],
          "scripts": []
```

Angular 14 Tutorial

Bootstrap



Now Name of Angular-cli.json file changed to Angular.json

Using Angular Material:

Angular 14 Tutorial

Angular Material

Angular Material is a UI component library for Angular developers. Angular Material's reusable UI components help in constructing attractive, consistent, and functional web pages and web applications while adhering to modern web design principles like browser portability, device independence, and graceful degradation.

Optimized for Angular
Built by the Angular team to integrate seamlessly with Angular.

Versatile
Themable, for when you need to stay on brand or just have a favorite color.
Accessible and internationalized so that all users are welcome.

Angular 14 Tutorial

Angular Material

Responsive Design
Angular Material is by design very minimal and flat.
It is designed considering the fact that it is much easier to add new CSS rules than to overwrite existing CSS rules.
It supports shadows and bold colors.
The colors and shades remain uniform across various platforms and devices.

Angular 14 Tutorial

Angular Material

Extensible

Angular Material has in-built responsive designing so that the website created using Angular Material will redesign itself as per the device size. Angular Material classes are created in such a way that the website can fit any screen size.
The websites created using Angular Material are fully compatible with PC, tablets, and mobile devices.

INSTALLATION:

npm install --save @angular/material @angular/cdk @angular/animations

Explanation of the Packages

1. `@angular/material`: This package contains the Angular Material components that you can use in your application.
2. `@angular/cdk`: The Angular Component Dev Kit (CDK) is a set of tools that can be used to create custom components that follow the Material Design specifications. Many Angular Material components depend on the CDK for functionalities like layout, accessibility, and more.
3. `@angular/animations`: This package is required for animations within Angular Material components. It allows you to create smooth transitions and animations, enhancing the user experience.

Step 3: Configure Angular Material

After installing, you still need to configure Angular Material in your application.

1. **Import Browser Animations**: Open `app.module.ts` and import `BrowserAnimationsModule`.
2. **Import Material Modules**: Import the specific Material modules you want to use. Here's an example:

ANOTHER INSTALLATION WAY:

Using ng add @angular/material is a quick and easy way to set up Angular Material, as it automatically configures your application for you, including adding required styles and setting up Angular animations.

>>> If want to apply differennt themes then can import those theme in style.css:

app.module.ts | app.component.html | app.component.ts | package.json | styles.css

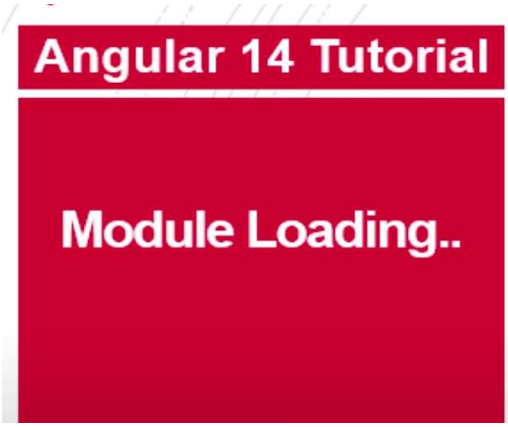
```
/* @import '~@angular/material/prebuilt-themes/Indigo-pink.css';      */
/* @import '~@angular/material/prebuilt-themes/Pink-bluegrey.css';    */
@import '~@angular/material/prebuilt-themes/Deeppurple-amber.css';
/* @import '~@angular/material/prebuilt-themes/Purple-green.css';     */
```

>> MOdify color of a material component and modify add content:

```
<mat-toolbar color="accent">
<span>Sahosft Solutions</span>
</mat-toolbar>
<mat-card>
  <mat-card-title>
    Angular Material component with Angular 5
  </mat-card-title>
  <mat-card-content>
    <form class="example-form">
      <mat-form-field class="example-full-width">
        <input matInput placeholder="First Name" >
      </mat-form-field>

      <mat-form-field class="example-full-width">
        <textarea matInput placeholder="Last Name"></textarea>
      </mat-form-field>
    </form>
  </mat-card-content>
</mat-card>
```

Important:



An effective loading strategy is key to a successful single-page application. A module can be loaded eagerly, lazily and preloaded.

Eager loading: In an eager loading scenario, all of the modules and functions are loaded on application startup. The root module is always eagerly loaded, and in some cases you will use eager loading for additional features.

Lazy loading is loading modules on demand.

Preloading is loading modules in background just after application starts.

1. Eager Loading

Definition: Eager loading is the default module loading strategy in Angular. When the application starts, all the modules are loaded immediately, regardless of whether they are needed right away or not.

```typescript
@NgModule({
  imports: [
    CommonModule,
    FormsModule,
    ReactiveFormsModule,
    // other modules
  ],
  declarations: [MyComponent],
})
export class MyModule {}
```

2. Lazy Loading

Definition: Lazy loading is a design pattern in Angular that allows you to load modules on demand, meaning that a module is only loaded when the user navigates to a route that requires it. This reduces the initial load time of the application. Loads with loadChildren property.

How to Implement Lazy Loading:

1. Create a Feature Module:

```bash
ng generate module feature --routing
```

2. Define Routes in the Feature Module:

```typescript
// feature-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { FeatureComponent } from './feature.component';

const routes: Routes = [{ path: '', component: FeatureComponent }];

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

3. **Configure the App Routing Module:**

typescript Copy code

```typescript
// app-routing.module.ts
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';

const routes: Routes = [
  {
    path: 'feature',
    loadChildren: () =>
      import('./feature/feature.module').then((m) => m.FeatureModule),
  },
];

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

3. Preloading Loading

Definition: Preloading strategy allows you to load certain lazy-loaded modules in the background after the application has been loaded. This helps in making the application responsive and faster when navigating to routes that require these modules..

1. In preloading, feature modules are loaded in background asynchronously. In preloading, modules start loading just after application starts.

2. When we hit the application, first AppModule and modules imported by it, will be loaded eagerly. Just after that modules configured for preloading is loaded asynchronously.

3. Preloading is useful to load those features which are in high probability to be visited by user just after loading the application.

4. To configure preloading, angular provides preloadingStrategy property which is used with RouterModule.forRoot in routing module. Find the code snippet.

5. To configure preloading features modules, first we will configure them for lazy loading and then using Angular in-built PreloadAllModules strategy, we enable to load all lazy loading into preloading modules.
6. Using PreloadAllModules strategy, all modules configured by loadChildren property will be preloaded. The modules configured by loadChildren property will be either lazily loaded or preloaded but not both. To preload only selective modules, we need to use custom preloading strategy.
7. We can create custom preloading strategy. For this we need to create a service by implementing Angular PreloadingStrategy interface and override its preload method and then configure this service with preloadingStrategy property in routing module. To select a module for custom preloading we need to use dataproperty in route configuration. data can be configured as data: { preload: true } for selective feature module preloading.

How to Implement Preloading:

1. Configure the Router with a Preloading Strategy:

typescript Copy code

```typescript
import { NgModule } from '@angular/core';
import { RouterModule, Routes, PreloadAllModules } from '@angular/router';

const routes: Routes = [
  {
    path: 'feature',
    loadChildren: () =>
      import('./feature/feature.module').then((m) => m.FeatureModule),
  },
];

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

Type of Loading	Description	Use Case
Eager Loading	Loads all modules at startup.	Small applications.
Lazy Loading	Loads modules on demand when the user navigates to a route.	Large applications for performance.
Preloading	Loads lazy-loaded modules in the background after the app is initialized.	Improves user experience.

Custom Preloading in Angular 14

Custom preloading strategies allow to preload selective modules. We can also customize to preload modules after a certain delay once application starts. To create a custom preloading strategy, Angular provides **PreloadingStrategyclass** with a preload method. We need to create a service by implementing **PreloadingStrategy** and then overriding its preload method. To enable custom preloading strategies, we need to configure our preloading strategy service with **RouterModule.forRoot** using preloadingStrategy property. We will also configure our preloading strategy service into providers metadata in application routing module.

Step-1: Use data property in route configuration. data is a property of Route interface. data provides additional data to component via ActivatedRoute. The data property can be used in route configuration as following.

```
const routes: Routes = [
{
path: 'country',
loadChildren: 'app/country/country.module#CountryModule',
data: { preload: true } },
];
```

whereEver there will be true that will be preloaded else not.

Step-2: Create a service by implementing PreloadingStrategy class and override its preload method. Here we are creating a custom preloading strategy with name **CustomPreloadingStrategy**.

```
@Injectable()
export class CustomPreloadingStrategy implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    if (route.data && route.data['preload']) {
      return load();
    } else {
      return Observable.of(null);
    }
  }
}
```

Step-3: Suppose we have following route configuration.

```
const routes: Routes = [
    {
        path: 'person',
        loadChildren: 'app/person/person.module#PersonModule',
        data: { preload: true }
    },
    {
        path: 'address',
        loadChildren: 'app/address/address.module#AddressModule'
    }
    ------
];
```

Step-4: To enable our custom preloading strategy CustomPreloadingStrategy, we need to assign it to preloadingStrategy property in RouterModule.forRoot method in application routing module.

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

Step-5: Configure our custom preloading strategy CustomPreloadingStrategy with providers metadata in application routing module.

```
@NgModule({
  providers: [ CustomPreloadingStrategy ],
  ------
})
export class AppRoutingModule { }
```

==========

Most Important:

Angular 14 Tutorial

Observables and Subscribe

Observable belongs to **RxJS** library. To perform asynchronous programming in Angular application we can use either Observable or Promise. When we send and receive data over HTTP, we need to deal it asynchronously because fetching data over HTTP may take time. Observable is subscribed by using async pipe or by using subscribe method.

Observable is a class of RxJS library. **RxJS** is ReactiveX library for JavaScript that performs reactive programming. Observable represents any set of values over any amount of time. Observable plays most basic role in reactive programming with RxJS. Some methods of Observable class are subscribe, map, mergeMap, switchMap, exhaustMap, debounceTime, of, retry, catch, throw etc.

Angular HttpClient performs HTTP requests for the given URL. HttpClient works with Observable. Find some of its methods.

```
get(url: string, options: {...}): Observable<any>

post(url: string, body: any | null, options: {...}): Observable<any>

put(url: string, body: any | null, options: {...}): Observable<any>

delete(url: string, options: {...}): Observable<any>
```

The http class provide the get() method to getting a resource, Post() for creating it, Put for updating it, delete for deleting resource and head for getting meta data regarding resource.

```
getBooksFromStore(): Observable<Book[]> {
 return this.http.get<Book[]>(this.bookUrl);
}
--------------------------------
getsoftBooks() {
this.bookService.getBooksFromStore().subscribe(books => this.softBooks =
books);
}
```

Observables and Subscribe with web API in Angular 14

Why Use In-Memory Web API in Angular Development

The **In-Memory Web API** is a module that simulates a backend API, allowing you to perform CRUD operations on mock data. Here are several reasons to use it:

1. **Rapid Prototyping**:
   - It allows developers to quickly prototype and test applications without needing a real backend server. You can focus on building the frontend while the in-memory API handles requests and responses.
2. **Testing**:
   - When developing applications, it's crucial to have a reliable way to test data interactions. The in-memory Web API provides a controlled environment to simulate various API responses (success, error, empty data) without r[illegible]ing an external service.
3. **No Dependency on Backend**:
   - You can develop your application independently from the backend. This is especially useful when the backend is still under development or if the development team is separate.
4. **Easy Data Manipulation**:
   - You can easily modify the mock data directly in your Angular application. It allows for quick changes without needing to modify a database or backend service.
5. **Reduced Configuration**:
   - Setting up a real backend can be time-consuming and involves configuration. The in-memory Web API is quick to set up, requiring minimal configuration.

Step1:

```
Windows PowerShell
Copyright (C) Microsoft Corporation. All rights reserved.

PS F:\projects\angularapi> npm i angular-in-memory-web-api@0.5.3 --save
```

Step 2: Create a Data Model

Create a simple model for the data. For example, let's say we're working with posts.

// post.model.ts

export interface Post { id: number; title: string; body: string;}

Step 3: Create an In-Memory Data Service

Create a new service to provide in-memory data:.

ng generate service in-memory-data

Step 4: implement the service:

```
// in-memory-data.service.ts
import { InMemoryWebApiModule } from 'angular-in-memory-web-api';
import { Injectable } from '@angular/core';
import { Post } from './post.model'; // Import the Post model
import { InMemoryDbService } from 'angular-in-memory-web-api';

@Injectable({
  providedIn: 'root',
})
export class InMemoryDataService implements InMemoryDbService {
  createDb() {
    const posts: Post[] = [
      { id: 1, title: 'Post 1', body: 'This is the body of post 1' },
      { id: 2, title: 'Post 2', body: 'This is the body of post 2' },
      { id: 3, title: 'Post 3', body: 'This is the body of post 3' },
    ];
    return { posts };
  }
}
```

Step 5: Configure the App Module

Import HttpClientModule and InMemoryWebApiModule in your app.module.ts and configure the in-memory data service.

```
// app.module.ts
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';
import { InMemoryWebApiModule } from 'angular-in-memory-web-api';
import { InMemoryDataService } from './in-memory-data.service'; // Import the data service
import { AppComponent } from './app.component';
import { DataService } from './data.service'; // Import your data service

@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    HttpClientModule,
    InMemoryWebApiModule.forRoot(InMemoryDataService), // Set up the in-memory Web API
  ],
  providers: [DataService],
  bootstrap: [AppComponent],
})
export class AppModule {}
```

Copy code

Step 6: Create a Data Service for API Calls
Create a new service to handle HTTP requests to the in-memory API.
ng g s data

Implement the service to fetch data from the in-memory API:

```
// data.service.ts
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';
import { Post } from './post.model'; // Import the Post model

@Injectable({
  providedIn: 'root',
})
export class DataService {
  private apiUrl = 'api/posts'; // API URL for the in-memory Web API

  constructor(private http: HttpClient) {}

  // Method to get posts
  getPosts(): Observable<Post[]> {
    return this.http.get<Post[]>(this.apiUrl);
  }
}
```

Step 7: Use the Data Service in a Component

Now, use the DataService in your main application component to fetch and display the posts.

```
// app.component.ts
import { Component, OnInit } from '@angular/core';
import { DataService } from './data.service';
import { Post } from './post.model';

@Component({
  selector: 'app-root',
  templateUrl: './app.component.html',
})
export class AppComponent implements OnInit {
  posts: Post[] = []; // Array to hold the posts
  error: string | null = null; // Error handling

  constructor(private dataService: DataService) {}

  ngOnInit(): void {
    this.fetchPosts();
  }
```

```
  fetchPosts(): void {
    this.dataService.getPosts().subscribe(
      (data) => {
        this.posts = data; // Assign the received data to the posts array
      },
      (error) => {
        this.error = 'Failed to fetch posts'; // Handle error
        console.error('Error fetching posts:', error);
      }
    );
  }
}
```

Step 8: Update the Component Template

Now, update the template file (app.component.html) to display the fetched posts.

```
<!-- app.component.html -->
<h1>Posts</h1>
<div *ngIf="error" class="error">{{ error }}</div>
<ul>
  <li *ngFor="let post of posts">
    <h2>{{ post.title }}</h2>
    <p>{{ post.body }}</p>
  </li>
</ul>
```

Observables + async pipe + *ngFor in Angular 14

Angular async pipe subscribes to Observable and returns its last emitted value.

Step-1: We will create a method to fetch data over HTTP using Angular HttpClient in our service, suppose in BookService

```
getBooksFromStore(): Observable<Book[]> {
 return this.http.get<Book[]>(this.bookUrl);
}
```

Method will return Observable<Book[]>

Step-2: In our component we will create a property.

```
allBooks : Observable<Book[]>
```

Now we will call the service method and assign value to the allBooks property in our component.

```
getBooks() {
 this.allBooks = this.bookService.getBooksFromStore();
}
```

Call the above method in ngOnInit.

```
ngOnInit() {
  this.getBooks();
 }
```

Step-3: Now we will subscribe our Observable variable i.e. allBooks using async pipe. We need to keep in mind that HTTP hit will take place only when we subscribe our Observable variable. In our HTML template we will subscribe our Observable variable i.e. allBooks using async pipe with ngFor.

```
<ul>
  <li *ngFor="let book of allBooks | async" >
  Id: {{book.id}}, Name: {{book.name}}, Category:
{{book.category}}
 </li>
</ul>
```

==========

HTTPCLIENT

The new HttpClient service is included in the HTTP Client Module that used to initiate HTTP request and responses in angular apps. The HttpClientModule is a replacement of HttpModule. HttpClient also gives us advanced functionality like the ability to listen for progress events and **interceptors** to modify requests or responses.

Before using the HttpClient, you must need to import the Angular HttpClientModule and the HttpClientModule is imported from *@angular/common/http*.

You must import HttpClientModule after BrowserModule in your angular apps.

Need to imported HttpClientModule from @angular/common/http in your app module and it must be import HttpClientModule after BrowserModule in your angular apps.

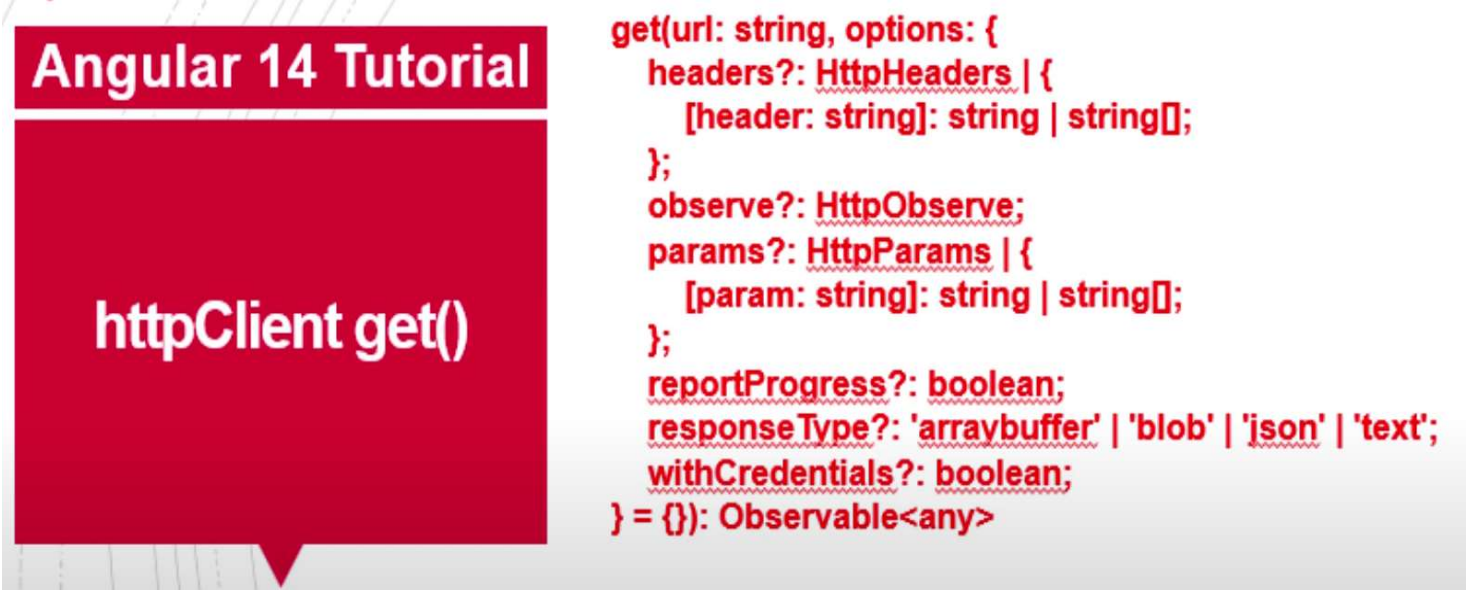
After imported HttpClientModule into the AppModule, you can inject the HttpClient into your created service.

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

@Injectable()
export class CustomerService {

  //Inject HttpClient into your components or services
  constructor(private httpClient: HttpClient) { }

}
```



headers: It is of HttpHeaders types. It sets headers for the http GET request.

observe: It defines whether we want complete response or body only or events only. We need to assign values for observe property such as **response** for complete response, **body** for response with body and **events** for response with events.

params: It is of HttpParams type. It sets parameters in URL for http GET request.
The response type of HttpClient.get is Observable i.e. provided by RxJS library. Observable is a representation of any set of values over any amount of time.

Question: Difference in event binding and @HostListener?
Sol.

Feature	Event Binding	@HostListener
Scope	Limited to the element in the template	Works on the host element of a directive/component
Usage	Directly in the template	Inside the component or directive class
Encapsulation	Logic is in the template	Logic is encapsulated within the class
Reuse	Harder to reuse logic in different contexts	Easy to reuse in different templates
Complexity	Simpler for basic event handling	More powerful for complex event handling

Conclusion

While both Event Binding and @HostListener can be used to handle events, choosing the right approach depends on your needs:

- **Use Event Binding** when you need a simple and direct way to handle events for specific elements.
- **Use @HostListener** for encapsulated, reusable components or directives that require internal event handling without cluttering the template.

>>>>>>>>>>UseCases:

@HostListener is a decorator that allows you to listen to events on the host element of a directive or component. It provides more flexibility and is especially useful in the following scenarios:

When to Use @HostListener

- **Encapsulating Logic**: Use it to encapsulate event logic inside the component or directive, keeping the template cleaner and separating concerns.
- **Reusable Components**: Use it in directives and components where you want to manage the host element's events and behaviors without exposing too much logic to the template.
- **Complex Event Handling**: If the event handling logic requires interacting with component properties or methods without cluttering the template with too many handlers.

```
@HostListener('click',['$event'])
show(){
  alert("hello");
}
```

Event Binding is the straightforward way to listen to events on elements in your component's template. You typically use this when you want to handle events directly associated with a specific DOM element in the template.

When to Use Event Binding

- **Simplicity**: Use it for straightforward event handling where you want to capture events like clicks, mouse movements, or keyboard inputs directly on a template element.
- **Specific Elements**: Use it when you only need to handle events for specific elements, as it keeps the code more readable and direct.

==========

Question: How to use event Emitter using services. Give eg:

Sol.

Summary

1. **Create a service** with `EventEmitter` to emit events.
2. **Use the service in a sender component** to emit notifications.
3. **Use the service in a receiver component** to listen for notifications.
4. **Include both components** in your application's template.

Code:

Using **EventEmitter** with services in Angular 14 can be a powerful way to communicate between components and services, especially when you need to share data or notify components of changes in the application state. Here's how to set it up step by step.

Step 1: Create a Service

First, you need to create a service where you can use `EventEmitter` to emit events.

Generate the service:

You can generate a service using the Angular CLI:

bash Copy code

```
ng generate service myEvent
```

`my-event.service.ts`:

Here's an example of a simple service that uses `EventEmitter`:

typescript Copy code

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

@Injectable({
  providedIn: 'root'
})
export class MyEventService {
  // Create a new EventEmitter
  public notify: EventEmitter<string> = new EventEmitter<string>();

  constructor() {}
```

```
  // Method to emit an event
  emitNotification(message: string) {
    this.notify.emit(message);
  }
}
```

Step 2: Use the Service in a Component to Emit Events

Next, you will need a component that will use this service to emit notifications.

Generate a component:

You can generate a component using the Angular CLI:

bash Copy code

```
ng generate component sender
```

`sender.component.ts`:

Here's how you might use the service in a component to emit an event:

typescript Copy code

```
import { Component } from '@angular/core';
import { MyEventService } from '../my-event.service';

@Component({
  selector: 'app-sender',
  templateUrl: './sender.component.html'
})
export class SenderComponent {

  constructor(private myEventService: MyEventService) {}

  sendNotification() {
    this.myEventService.emitNotification('Hello from Sender Component!');
  }
}
```

`sender.component.html`:

html Copy code

```
<button (click)="sendNotification()">Send Notification</button>
```

>>>>>>>>>

Step 3: Use the Service in Another Component to Listen for Events

Now, create another component that will listen for events emitted by the service.

Generate a component:

You can generate another component using the Angular CLI:

bash Copy code

```
ng generate component receiver
```

`receiver.component.ts`:

Here's how to use the service in another component to listen for events:

typescript Copy code

```typescript
import { Component, OnInit } from '@angular/core';
import { MyEventService } from '../my-event.service';

@Component({
  selector: 'app-receiver',
  templateUrl: './receiver.component.html'
})
export class ReceiverComponent implements OnInit {
  message: string = '';

  constructor(private myEventService: MyEventService) {}

  ngOnInit() {
    // Subscribe to the EventEmitter in the service
    this.myEventService.notify.subscribe((message: string) => {
      this.message = message; // Update the message when an event is received
    });
  }
}
```

`receiver.component.html`:

html Copy code

```
<p>Received Message: {{ message }}</p>
```

Step 4: Use the Components in Your App

Finally, you need to use the sender and receiver components in your main application component or another parent component.

`app.component.html`:

html Copy code

```
<h1>EventEmitter with Services Example</h1>
<app-sender></app-sender>
<app-receiver></app-receiver>
```

==========

==========

Installation:

Step 3: Install Angular CLI

The Angular Command Line Interface (CLI) is the tool you'll use to create and manage Angular applications. To install the Angular CLI globally, run the following command:

bash Copy code

```
npm install -g @angular/cli@14
```

Step 4: Verify Angular CLI Installation

After the installation is complete, you can verify that Angular CLI has been installed correctly and check its version with the following command:

bash Copy code

```
ng version
```

>>> node support: v14.17.2.3
>>> npm v6.14.13

>>>>>>>>>>>>>>>>>>>>>>>To create project:

To create a new Angular project using Angular CLI, you can use the following command:

```bash
ng new my-angular-app
```

Replace `my-angular-app` with the desired name for your project. You will be prompted to choose whether to add Angular routing and to select a stylesheet format (CSS, SCSS, etc.).

Step 6: Navigate to Your Project Directory

Once the project has been created, navigate to the project directory:

```bash
cd my-angular-app
```

>>>>>>>>>>>>>>>>>>>>To run orpject in dev:

To run your Angular application locally, use the following command:

```bash
ng serve
```

This will start a local development server, and you can access your application in your web browser at `http://localhost:4200`.

==========