**Beginner Level**

1. **Create a basic Angular app** with a simple "Hello World" message.
2. **Create a component** that displays a list of items (using \*ngFor).
3. **Use data binding** to display a value in the HTML from the component.
4. **Use event binding** to create a button that changes a value in the component when clicked.
5. **Create an Angular service** to manage data and inject it into a component.
6. **Create an Angular form** with one text input and a submit button.
7. **Implement two-way data binding** with [(ngModel)] on an input field.
8. **Use ngIf directive** to conditionally display a list of items.
9. **Use ngFor directive** to iterate over an array of objects and display their properties.
10. **Create a component with lifecycle hooks**: ngOnInit, ngOnChanges.
11. **Set up routing** to navigate between two components.
12. **Add a simple CSS class** to a component element using ngClass.
13. **Create a button** that toggles a class (for example, changing color on click).
14. **Implement a basic Angular pipe** to transform data (e.g., uppercase).
15. **Create a custom Angular directive** that changes the background color of an element.
16. **Use Angular CLI** to generate components, services, and modules.
17. **Create an Angular app with Bootstrap** or a CSS framework for styling.
18. **Use Angular Material** to implement a button and a form.
19. **Create an input field** that validates the user input as they type.
20. **Create a toggle feature** that switches between two views (e.g., a login and a register form).

**Intermediate Level**

1. **Create an observable in a service** and subscribe to it in a component.
2. **Use RxJS operators** (e.g., map, filter, debounceTime) in Angular components.
3. **Implement a reactive form** with validation in Angular.
4. **Use FormBuilder** to build a reactive form.
5. **Create a dynamic form** based on an array of form fields.
6. **Use Angular HTTPClient** to make an API request and display the result.
7. **Handle HTTP errors** and display user-friendly error messages.
8. **Create a custom form validator** that checks for a valid email address.
9. **Use Angular's built-in validators** like required, minLength, and maxLength.
10. **Implement lazy loading** for routing modules in Angular.
11. **Use Angular interceptors** to add headers or handle errors in HTTP requests.
12. **Create a modal** using Angular components for a dialog box.
13. **Pass data between parent and child components** using Input and Output decorators.
14. **Implement a basic authentication mechanism** with a login form and guards.
15. **Create a service with localStorage** to persist user preferences.
16. **Handle HTTP responses with async/await** syntax in Angular.
17. **Create an Angular pipe** for date formatting.
18. **Implement routing guards** to protect routes from unauthorized users.
19. **Use Angular router's queryParams** to pass data between components.
20. **Create a search bar** that filters data dynamically.
21. **Use the ngModelChange event** for real-time form validation.
22. **Add a loading spinner** while waiting for an HTTP request to complete.
23. **Create a custom event emitter** to send events from a child to a parent component.
24. **Implement pagination** for a list of data.
25. **Create a date range picker** with Angular forms.
26. **Set up Angular internationalization (i18n)** for language support.
27. **Use Angular animations** to animate elements in and out of the view.
28. **Implement role-based access control** (RBAC) for different users.
29. **Create a dropdown component** that supports multi-select.
30. **Implement custom Angular pipes** for filtering and sorting lists.
31. **Use ngSwitch directive** to switch between multiple views based on a condition.
32. **Implement a tooltip component** using Angular Material.
33. **Create a breadcrumb navigation component** for better UX.
34. **Create a breadcrumb routing system** in Angular.
35. **Write unit tests for a service** using Jasmine and TestBed.
36. **Write unit tests for a component** with mock dependencies.
37. **Mock HTTP requests in Angular unit tests** using HttpClientTestingModule.
38. **Use Angular's dependency injection** to inject a service into a component.
39. **Implement a custom filter pipe** that sorts or filters data.
40. **Use ngModel to create a two-way binding** for form controls in a form group.
41. **Create a reusable button component** that emits an event when clicked.
42. **Set up Angular Universal (SSR)** for server-side rendering.
43. **Add Google Maps to an Angular project** using a wrapper library.
44. **Create a date picker component** with Angular Material.
45. **Implement drag-and-drop functionality** in an Angular app using Angular Material.
46. **Create a file upload component** that shows progress.
47. \*\*Use \*\***ng-content** to project content into a child component.
48. **Set up an Angular project with environment-specific configurations**.
49. **Optimize performance with trackBy in ngFor** for better rendering.
50. **Implement infinite scrolling** for long lists of data.
51. **Use Angular’s Change Detection strategy** to optimize performance.
52. **Create a multi-step form wizard** using Angular forms.
53. **Use Angular’s lazy loading** for large modules to improve the load time.
54. **Create a reusable modal component** to show alerts and confirmations.

**Advanced Level**

1. **Create a state management solution** using NgRx or Akita.
2. **Optimize bundle size using Angular CLI's build optimizations**.
3. **Create a custom Angular validator** that checks for complex patterns.
4. **Build a progressive web app (PWA)** with Angular and Service Workers.
5. **Integrate Firebase Authentication** for user login and registration.
6. **Create a real-time chat application** with Angular and WebSockets.
7. **Implement a custom backend API call** with retry logic and exponential backoff.
8. **Use Angular’s dynamic component loading** with a factory resolver.
9. **Create a global error handler** to catch and handle application errors.
10. **Implement offline support** in an Angular app with Service Workers and IndexedDB.
11. **Use Angular CLI to create a custom schematic**.
12. **Implement JWT (JSON Web Tokens)** for authentication in an Angular app.
13. **Create a deep linking solution** with the Angular router for bookmarkable routes.
14. **Use the Angular ChangeDetectionStrategy.OnPush** for performance optimization.
15. **Use custom pipes for deep object comparison**.
16. **Build an Angular application that consumes GraphQL API**.
17. **Implement a custom Angular module** with lazy-loaded components.
18. **Create a complex form** with multiple nested form groups using Angular forms.
19. **Build a multi-language (i18n) Angular app** that changes the language dynamically.
20. **Implement a micro-frontends architecture** using Angular elements and web components.
21. **Use Web Workers** in an Angular app for performance improvement.
22. **Integrate with an external third-party API** (e.g., Stripe, Google Maps).
23. **Implement server-side pagination** for a large dataset in Angular.
24. **Create an Angular app with multi-theme support**.
25. **Implement advanced routing** with child routes and nested views.
26. **Write integration tests** using Protractor or Cypress for an Angular app.

### \*\*Beginner Level\*\*

---

**### 1. \*\*Create a basic Angular app with a simple "Hello World" message\*\***

```bash

ng new hello-world-app

cd hello-world-app

ng serve

```

In `src/app/app.component.html`:

```html

<h1>Hello World</h1>

```

**### 2. \*\*Create a component that displays a list of items (using \*ngFor)\*\***

```bash

ng generate component item-list

```

In `src/app/item-list/item-list.component.ts`:

```typescript

export class ItemListComponent {

items = ['Item 1', 'Item 2', 'Item 3'];

}

```

In `src/app/item-list/item-list.component.html`:

```html

<ul>

<li \*ngFor="let item of items">{{ item }}</li>

</ul>

```

**### 3. \*\*Use data binding to display a value in the HTML from the component\*\***

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

message = "Hello, Angular!";

}

```

In `src/app/app.component.html`:

```html

<h1>{{ message }}</h1>

```

### 4. \*\***Use event binding to create a button that changes a value in the component when clicked**\*\*

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

count = 0;

increment() {

this.count++;

}

}

```

In `src/app/app.component.html`:

```html

<p>{{ count }}</p>

<button (click)="increment()">Increment</button>

```

### 5. \*\***Create an Angular service to manage data and inject it into a component**\*\*

```bash

ng generate service data

```

In `src/app/data.service.ts`:

```typescript

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

@Injectable({

providedIn: 'root'

})

export class DataService {

getData() {

return ['Data 1', 'Data 2', 'Data 3'];

}

}

```

In `src/app/app.component.ts`:

```typescript

import { Component, OnInit } from '@angular/core';

import { DataService } from './data.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

})

export class AppComponent implements OnInit {

data: string[] = [];

constructor(private dataService: DataService) {}

ngOnInit() {

this.data = this.dataService.getData();

}

}

```

In `src/app/app.component.html`:

```html

<ul>

<li \*ngFor="let item of data">{{ item }}</li>

</ul>

```

### 6. \*\***Create an Angular form with one text input and a submit button**\*\*

In `src/app/app.component.ts`:

```typescript

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

import { FormBuilder, FormGroup } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

})

export class AppComponent {

form: FormGroup;

constructor(private fb: FormBuilder) {

this.form = this.fb.group({

name: ['']

});

}

onSubmit() {

console.log(this.form.value);

}

}

```

In `src/app/app.component.html`:

```html

<form [formGroup]="form" (ngSubmit)="onSubmit()">

<input formControlName="name" placeholder="Enter your name" />

<button type="submit">Submit</button>

</form>

```

### 7. \*\***Implement two-way data binding with `[(ngModel)]` on an input field**\*\*

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

name = '';

}

```

In `src/app/app.component.html`:

```html

<input [(ngModel)]="name" placeholder="Enter your name" />

<p>Your name is: {{ name }}</p>

```

### 8. \*\***Use ngIf directive to conditionally display a list of items**\*\*

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

showItems = true;

items = ['Item 1', 'Item 2', 'Item 3'];

}

```

In `src/app/app.component.html`:

```html

<button (click)="showItems = !showItems">Toggle Items</button>

<ul \*ngIf="showItems">

<li \*ngFor="let item of items">{{ item }}</li>

</ul>

```

### 9. \*\***Use ngFor directive to iterate over an array of objects and display their properties**\*\*

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

people = [

{ name: 'John', age: 30 },

{ name: 'Jane', age: 25 },

{ name: 'Smith', age: 40 }

];

}

```

In `src/app/app.component.html`:

```html

<ul>

<li \*ngFor="let person of people">{{ person.name }} - {{ person.age }} years old</li>

</ul>

```

### 10. \*\*Create a component with lifecycle hooks: ngOnInit, ngOnChanges\*\*

In `src/app/lifecycle/lifecycle.component.ts`:

```typescript

import { Component, OnInit, OnChanges, SimpleChanges, Input } from '@angular/core';

@Component({

selector: 'app-lifecycle',

templateUrl: './lifecycle.component.html',

})

export class LifecycleComponent implements OnInit, OnChanges {

@Input() data: string;

ngOnInit() {

console.log('ngOnInit called');

}

ngOnChanges(changes: SimpleChanges) {

console.log('ngOnChanges called', changes);

}

}

```

In `src/app/lifecycle/lifecycle.component.html`:

```html

<p>{{ data }}</p>

```

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

data = 'Initial data';

changeData() {

this.data = 'New data';

}

}

```

In `src/app/app.component.html`:

```html

<app-lifecycle [data]="data"></app-lifecycle>

<button (click)="changeData()">Change Data</button>

```

### 11. \*\***Set up routing to navigate between two components**\*\*

First, you need to create a routing module:

```bash

ng generate module app-routing --flat --module=app

```

In `src/app/app-routing.module.ts`:

```typescript

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

import { AboutComponent } from './about/about.component';

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'about', component: AboutComponent },

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule],

})

export class AppRoutingModule {}

```

Generate components for `Home` and `About`:

```bash

ng generate component home

ng generate component about

```

In `src/app/home/home.component.html`:

```html

<h2>Home Component</h2>

<a routerLink="/about">Go to About</a>

```

In `src/app/about/about.component.html`:

```html

<h2>About Component</h2>

<a routerLink="/">Go to Home</a>

```

In `src/app/app.component.html`:

```html

<router-outlet></router-outlet>

```

### 12. \*\***Add a simple CSS class to a component element using ngClass**\*\*

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

isActive = true;

}

```

In `src/app/app.component.html`:

```html

<div [ngClass]="{'active': isActive}">

This div is dynamically styled

</div>

<button (click)="isActive = !isActive">Toggle Class</button>

```

In `src/app/app.component.css`:

```css

.active {

color: green;

font-weight: bold;

}

```

### 13. \*\***Create a button that toggles a class (for example, changing color on click**)\*\*

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

isBlue = false;

toggleColor() {

this.isBlue = !this.isBlue;

}

}

```

In `src/app/app.component.html`:

```html

<button [ngClass]="{'blue': isBlue}" (click)="toggleColor()">Toggle Color</button>

```

In `src/app/app.component.css`:

```css

.blue {

background-color: blue;

color: white;

}

```

### 14. \*\***Implement a basic Angular pipe to transform data (e.g., uppercase)**\*\*

First, generate a custom pipe:

```bash

ng generate pipe uppercase

```

In `src/app/uppercase.pipe.ts`:

```typescript

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'uppercase'

})

export class UppercasePipe implements PipeTransform {

transform(value: string): string {

return value.toUpperCase();

}

}

```

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

text = 'hello world';

}

```

In `src/app/app.component.html`:

```html

<p>{{ text | uppercase }}</p>

```

### 15. \*\***Create a custom Angular directive that changes the background color of an element**\*\*

Generate a directive:

```bash

ng generate directive backgroundColor

```

In `src/app/background-color.directive.ts`:

```typescript

import { Directive, ElementRef, Renderer2 } from '@angular/core';

@Directive({

selector: '[appBackgroundColor]'

})

export class BackgroundColorDirective {

constructor(private el: ElementRef, private renderer: Renderer2) {

this.renderer.setStyle(this.el.nativeElement, 'background-color', 'yellow');

}

}

```

In `src/app/app.component.html`:

```html

<p appBackgroundColor> This paragraph has a background color. </p>

```

### 16. \*\***Use Angular CLI to generate components, services, and modules**\*\*

You can generate a component, service, or module using the Angular CLI:

- \*\*Component\*\*:

```bash

ng generate component my-component

```

- \*\*Service\*\*:

```bash

ng generate service my-service

```

- \*\*Module\*\*:

```bash

ng generate module my-module

```

### 17. \*\***Create an Angular app with Bootstrap or a CSS framework for styling**\*\*

Install Bootstrap:

```bash

npm install bootstrap

```

In `angular.json`, add Bootstrap’s CSS to the styles array:

```json

"styles": [

"src/styles.css",

"node\_modules/bootstrap/dist/css/bootstrap.min.css"

],

```

Then you can use Bootstrap classes in your components:

In `src/app/app.component.html`:

```html

<button class="btn btn-primary">Bootstrap Button</button>

```

### 18. \*\***Use Angular Material to implement a button and a form**\*\*

First, install Angular Material:

```bash

ng add @angular/material

```

In `src/app/app.module.ts`, import the necessary Angular Material modules:

```typescript

import { MatButtonModule } from '@angular/material/button';

import { MatInputModule } from '@angular/material/input';

import { MatFormFieldModule } from '@angular/material/form-field';

@NgModule({

imports: [

MatButtonModule,

MatInputModule,

MatFormFieldModule,

// other modules

],

})

export class AppModule {}

```

In `src/app/app.component.html`:

```html

<mat-form-field>

<input matInput placeholder="Enter your name">

</mat-form-field>

<button mat-raised-button>Submit</button>

```

### 19. \*\***Create an input field that validates the user input as they type**\*\*

In `src/app/app.component.ts`:

```typescript

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

import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

})

export class AppComponent {

form = new FormGroup({

name: new FormControl('', [Validators.required, Validators.minLength(3)])

});

}

```

In `src/app/app.component.html`:

```html

<form [formGroup]="form">

<input formControlName="name" placeholder="Enter your name">

<button [disabled]="form.invalid">Submit</button>

</form>

```

### 20. \*\***Create a toggle feature that switches between two views (e.g., a login and a register form**)\*\*

In `src/app/app.component.ts`:

```typescript

export class AppComponent {

showLogin = true;

toggleView() {

this.showLogin = !this.showLogin;

}

}

```

In `src/app/app.component.html`:

```html

<button (click)="toggleView()">Switch View</button>

<div \*ngIf="showLogin; else register">

<h2>Login</h2>

<form>

<!-- Login Form -->

</form>

</div>

<ng-template #register>

<h2>Register</h2>

<form>

<!-- Register Form -->

</form>

</ng-template>

```

### \*\***Intermediate Level**\*\*

---

### \*\*21. **Create an observable in a service and subscribe to it in a component**\*\*

Generate service:

```bash

ng generate service data

```

\*\*data.service.ts\*\*

```typescript

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

import { Observable, of } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class DataService {

getData(): Observable<string[]> {

return of(['Apple', 'Banana', 'Cherry']);

}

}

```

\*\*app.component.ts\*\*

```typescript

import { Component, OnInit } from '@angular/core';

import { DataService } from './data.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

})

export class AppComponent implements OnInit {

fruits: string[] = [];

constructor(private dataService: DataService) {}

ngOnInit() {

this.dataService.getData().subscribe(data => {

this.fruits = data;

});

}

}

```

\*\*app.component.html\*\*

```html

<ul>

<li \*ngFor="let fruit of fruits">{{ fruit }}</li>

</ul>

```

---

### \*\*22. **Use RxJS operators (e.g., map, filter, debounceTime**)\*\*

Example: filter users whose names contain "a"

```typescript

import { Component, OnInit } from '@angular/core';

import { of } from 'rxjs';

import { filter, map } from 'rxjs/operators';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

})

export class AppComponent implements OnInit {

users = [];

ngOnInit() {

of(['Alice', 'Bob', 'Charlie', 'David'])

.pipe(

map(users => users.filter(name => name.includes('a')))

)

.subscribe(filtered => this.users = filtered);

}

}

```

---

### \*\*23**. Implement a reactive form with validation in Angular**\*\*

\*\*app.module.ts\*\*

```ts

import { ReactiveFormsModule } from '@angular/forms';

@NgModule({

imports: [ReactiveFormsModule]

})

```

\*\*app.component.ts\*\*

```ts

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

import { FormGroup, FormControl, Validators } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent {

form = new FormGroup({

email: new FormControl('', [Validators.required, Validators.email])

});

submit() {

console.log(this.form.value);

}

}

```

\*\*app.component.html\*\*

```html

<form [formGroup]="form" (ngSubmit)="submit()">

<input formControlName="email" placeholder="Email">

<div \*ngIf="form.get('email')?.invalid && form.get('email')?.touched">

Invalid Email

</div>

<button type="submit" [disabled]="form.invalid">Submit</button>

</form>

```

---

### \*\*24. **Use FormBuilder to build a reactive form**\*\*

\*\*app.component.ts\*\*

```ts

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent {

form: FormGroup;

constructor(private fb: FormBuilder) {

this.form = this.fb.group({

username: ['', Validators.required],

password: ['', [Validators.required, Validators.minLength(6)]]

});

}

}

```

---

### \*\*25**. Create a dynamic form based on an array of form fields**\*\*

\*\*app.component.ts\*\*

```ts

fields = ['firstName', 'lastName', 'email'];

form: FormGroup;

constructor(private fb: FormBuilder) {

const controls = this.fields.reduce((acc, field) => {

acc[field] = new FormControl('');

return acc;

}, {} as any);

this.form = this.fb.group(controls);

}

```

\*\*app.component.html\*\*

```html

<form [formGroup]="form">

<div \*ngFor="let field of fields">

<input [formControlName]="field" [placeholder]="field">

</div>

<button (click)="submit()">Submit</button>

</form>

```

---

### \*\*26. **Use Angular HttpClient to make an API request and display the result**\*\*

\*\*data.service.ts\*\*

```ts

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

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

import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })

export class DataService {

constructor(private http: HttpClient) {}

getUsers(): Observable<any> {

return this.http.get('https://jsonplaceholder.typicode.com/users');

}

}

```

\*\*app.component.ts\*\*

```ts

import { Component, OnInit } from '@angular/core';

import { DataService } from './data.service';

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent implements OnInit {

users: any[] = [];

constructor(private dataService: DataService) {}

ngOnInit() {

this.dataService.getUsers().subscribe(data => this.users = data);

}

}

```

---

### \*\*27. **Handle HTTP errors and display user-friendly error messages**\*\*

\*\*data.service.ts\*\*

```ts

import { catchError } from 'rxjs/operators';

import { throwError } from 'rxjs';

getUsers(): Observable<any> {

return this.http.get('https://api.invalid-url.com')

.pipe(

catchError(error => {

console.error('API error:', error);

return throwError(() => new Error('Failed to fetch data'));

})

);

}

```

\*\*app.component.ts\*\*

```ts

errorMsg = '';

ngOnInit() {

this.dataService.getUsers().subscribe({

next: data => this.users = data,

error: err => this.errorMsg = err.message

});

}

```

\*\*app.component.html\*\*

```html

<div \*ngIf="errorMsg" class="alert alert-danger">{{ errorMsg }}</div>

```

---

### \*\*28. **Create a custom form validator that checks for a valid email address**\*\*

\*\*custom-email.validator.ts\*\*

```ts

import { AbstractControl, ValidationErrors } from '@angular/forms';

export function CustomEmailValidator(control: AbstractControl): ValidationErrors | null {

const value = control.value;

const isValid = /^[^\s@]+@[^\s@]+\.[^\s@]+$/.test(value);

return isValid ? null : { invalidEmail: true };

}

```

\*\*app.component.ts\*\*

```ts

import { CustomEmailValidator } from './custom-email.validator';

this.form = this.fb.group({

email: ['', CustomEmailValidator]

});

```

---

### \*\*29. **Use Angular’s built-in validators like `required`, `minLength`, and `maxLength**`\*\*

```ts

this.form = this.fb.group({

username: ['', [Validators.required, Validators.minLength(3), Validators.maxLength(10)]]

});

```

```html

<input formControlName="username">

<div \*ngIf="form.get('username')?.errors?.['minlength']">Too short!</div>

```

---

### \*\*30**. Implemented lazy loading for routing modules in Angular**\*\*

\*\*Step 1: Generate a module with routing\*\*

```bash

ng generate module admin --route admin --module app

```

Angular automatically sets up lazy loading in `app-routing.module.ts`:

```ts

const routes: Routes = [

{ path: 'admin', loadChildren: () => import('./admin/admin.module').then(m => m.AdminModule) }

];

```

\*\*admin-routing.module.ts\*\* (auto-generated):

```ts

const routes: Routes = [

{ path: '', component: AdminComponent }

];

```

### \*\*31. **Use Angular interceptors to add headers or handle errors in HTTP requests**\*\*

\*\*Step 1:\*\* Create an interceptor:

```bash

ng generate interceptor auth

```

\*\*auth.interceptor.ts\*\*

```ts

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

import {

HttpInterceptor,

HttpRequest,

HttpHandler,

HttpEvent

} from '@angular/common/http';

import { Observable } from 'rxjs';

@Injectable()

export class AuthInterceptor implements HttpInterceptor {

intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {

const cloned = req.clone({

setHeaders: {

Authorization: `Bearer fake-jwt-token`

}

});

return next.handle(cloned);

}

}

```

\*\*app.module.ts\*\*

```ts

import { HTTP\_INTERCEPTORS } from '@angular/common/http';

providers: [

{

provide: HTTP\_INTERCEPTORS,

useClass: AuthInterceptor,

multi: true

}

]

```

---

### \*\*32. **Create a modal using Angular components for a dialog box**\*\*

Using \*\*Angular Material\*\*:

\*\*Install:\*\*

```bash

ng add @angular/material

```

\*\*Generate modal component:\*\*

```bash

ng generate component my-modal

```

\*\*my-modal.component.ts\*\*

```ts

import { MatDialogRef } from '@angular/material/dialog';

constructor(public dialogRef: MatDialogRef<MyModalComponent>) {}

close() {

this.dialogRef.close();

}

```

\*\*app.component.ts\*\*

```ts

import { MatDialog } from '@angular/material/dialog';

import { MyModalComponent } from './my-modal/my-modal.component';

constructor(private dialog: MatDialog) {}

openDialog() {

this.dialog.open(MyModalComponent);

}

```

\*\*app.component.html\*\*

```html

<button mat-button (click)="openDialog()">Open Modal</button>

```

---

### \*\*33. **Pass data between parent and child components using Input and Output decorators**\*\*

\*\*child.component.ts\*\*

```ts

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-child',

template: `

<p>{{ message }}</p>

<button (click)="sendData()">Send to Parent</button>

`

})

export class ChildComponent {

@Input() message: string;

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

sendData() {

this.messageChange.emit('Data from Child!');

}

}

```

\*\*app.component.html\*\*

```html

<app-child [message]="parentMessage" (messageChange)="updateMessage($event)"></app-child>

```

\*\*app.component.ts\*\*

```ts

parentMessage = 'Hello from Parent';

updateMessage(newMsg: string) {

this.parentMessage = newMsg;

}

```

---

### \*\*34. **Implement a basic authentication mechanism with a login form and guards**\*\*

\*\*auth.service.ts\*\*

```ts

@Injectable({ providedIn: 'root' })

export class AuthService {

isLoggedIn = false;

login(username: string, password: string): boolean {

if (username === 'admin' && password === 'admin') {

this.isLoggedIn = true;

return true;

}

return false;

}

}

```

\*\*auth.guard.ts\*\*

```ts

@Injectable({ providedIn: 'root' })

export class AuthGuard implements CanActivate {

constructor(private auth: AuthService, private router: Router) {}

canActivate(): boolean {

if (!this.auth.isLoggedIn) {

this.router.navigate(['/login']);

return false;

}

return true;

}

}

```

\*\*app-routing.module.ts\*\*

```ts

{ path: 'dashboard', component: DashboardComponent, canActivate: [AuthGuard] }

```

---

### \*\*35. **Create a service with localStorage to persist user preferences**\*\*

\*\*local-storage.service.ts\*\*

```ts

@Injectable({ providedIn: 'root' })

export class LocalStorageService {

save(key: string, value: string): void {

localStorage.setItem(key, value);

}

get(key: string): string | null {

return localStorage.getItem(key);

}

clear(key: string): void {

localStorage.removeItem(key);

}

}

```

---

### \*\*36**. Handle HTTP responses with async/await syntax in Angular**\*\*

```ts

async getData() {

try {

const data = await this.http.get<any>('https://jsonplaceholder.typicode.com/posts').toPromise();

console.log(data);

} catch (error) {

console.error('Error:', error);

}

}

```

> 📝 Note: `toPromise()` is deprecated in Angular 16+. Use `firstValueFrom()` from `rxjs`.

---

### \*\*37. **Create an Angular pipe for date formatting**\*\*

\*\*Generate pipe:\*\*

```bash

ng generate pipe formatDate

```

\*\*format-date.pipe.ts\*\*

```ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'formatDate'

})

export class FormatDatePipe implements PipeTransform {

transform(value: Date | string): string {

return new Date(value).toLocaleDateString('en-US');

}

}

```

\*\*app.component.html\*\*

```html

<p>{{ today | formatDate }}</p>

```

---

### \*\*38. **Implement routing guards to protect routes from unauthorized users**\*\*

Use the `AuthGuard` example from \*\*step 34\*\*, and apply it to routes you want to protect:

```ts

{ path: 'admin', component: AdminComponent, canActivate: [AuthGuard] }

```

---

### \*\*39. **Use Angular router's queryParams to pass data between components**\*\*

\*\*Component A (with link):\*\*

```html

<a [routerLink]="['/details']" [queryParams]="{ id: 42 }">View Details</a>

```

\*\*Component B (receiving):\*\*

```ts

constructor(private route: ActivatedRoute) {}

ngOnInit() {

this.route.queryParams.subscribe(params => {

console.log(params['id']); // 42

});

}

```

---

### \*\*40**. Create a search bar that filters data dynamically**\*\*

\*\*app.component.ts\*\*

```ts

items = ['Apple', 'Banana', 'Cherry'];

filteredItems = [...this.items];

searchText = '';

filterItems() {

this.filteredItems = this.items.filter(item =>

item.toLowerCase().includes(this.searchText.toLowerCase())

);

}

```

\*\*app.component.html\*\*

```html

<input [(ngModel)]="searchText" (input)="filterItems()" placeholder="Search..." />

<ul>

<li \*ngFor="let item of filteredItems">{{ item }}</li>

</ul>

```

---

### \*\*41**. Use the ngModelChange event for real-time form validation**\*\*

\*\*app.component.ts\*\*

```ts

export class AppComponent {

email: string = '';

validateEmail() {

console.log('Validating email:', this.email);

// You can add more logic to validate the email

}

}

```

\*\*app.component.html\*\*

```html

<input [(ngModel)]="email" (ngModelChange)="validateEmail()" placeholder="Email" />

```

---

### \*\*42. **Add a loading spinner while waiting for an HTTP request to complete**\*\*

\*\*app.component.ts\*\*

```ts

export class AppComponent {

isLoading = false;

constructor(private http: HttpClient) {}

fetchData() {

this.isLoading = true;

this.http.get('https://jsonplaceholder.typicode.com/posts')

.subscribe(data => {

console.log(data);

this.isLoading = false;

});

}

}

```

\*\*app.component.html\*\*

```html

<button (click)="fetchData()">Fetch Data</button>

<div \*ngIf="isLoading">Loading...</div>

```

---

### \*\*43. **Create a custom event emitter to send events from a child to a parent component**\*\*

\*\*child.component.ts\*\*

```ts

import { Component, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-child',

template: `<button (click)="sendToParent()">Send to Parent</button>`

})

export class ChildComponent {

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

sendToParent() {

this.customEvent.emit('Hello from Child!');

}

}

```

\*\*app.component.html\*\*

```html

<app-child (customEvent)="receiveFromChild($event)"></app-child>

<p>{{ messageFromChild }}</p>

```

\*\*app.component.ts\*\*

```ts

export class AppComponent {

messageFromChild: string;

receiveFromChild(message: string) {

this.messageFromChild = message;

}

}

```

---

### \*\*44. **Implement pagination for a list of data**\*\*

\*\*app.component.ts\*\*

```ts

export class AppComponent {

data = Array.from({ length: 100 }, (\_, i) => `Item ${i + 1}`);

currentPage = 1;

itemsPerPage = 10;

get paginatedData() {

const start = (this.currentPage - 1) \* this.itemsPerPage;

const end = start + this.itemsPerPage;

return this.data.slice(start, end);

}

nextPage() {

if (this.currentPage \* this.itemsPerPage < this.data.length) {

this.currentPage++;

}

}

prevPage() {

if (this.currentPage > 1) {

this.currentPage--;

}

}

}

```

\*\*app.component.html\*\*

```html

<ul>

<li \*ngFor="let item of paginatedData">{{ item }}</li>

</ul>

<button (click)="prevPage()">Previous</button>

<button (click)="nextPage()">Next</button>

```

---

### \*\*45. **Create a date range picker with Angular forms**\*\*

\*\*app.component.ts\*\*

```ts

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

import { FormGroup, FormBuilder } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent {

dateRangeForm: FormGroup;

constructor(private fb: FormBuilder) {

this.dateRangeForm = this.fb.group({

startDate: [''],

endDate: ['']

});

}

submit() {

console.log(this.dateRangeForm.value);

}

}

```

\*\*app.component.html\*\*

```html

<form [formGroup]="dateRangeForm" (ngSubmit)="submit()">

<input type="date" formControlName="startDate">

<input type="date" formControlName="endDate">

<button type="submit">Submit</button>

</form>

```

---

### \*\*46**. Set up Angular internationalization (i18n) for language support**\*\*

\*\*Step 1:\*\* Install Angular i18n:

```bash

ng add @angular/localize

```

\*\*Step 2:\*\* Modify `app.component.ts` to include multiple translations.

\*\*app.component.ts\*\*

```ts

export class AppComponent {

title = 'Angular Internationalization';

}

```

\*\*app.component.html\*\*

```html

<h1>{{ 'title' | translate }}</h1>

```

\*\*Step 3:\*\* Generate translation files using Angular CLI:

```bash

ng extract-i18n

```

---

### \*\*47. **Use Angular animations to animate elements in and out of the view**\*\*

\*\*app.component.ts\*\*

```ts

import { trigger, transition, style, animate } from '@angular/animations';

@Component({

selector: 'app-root',

templateUrl: './app.component.html',

animations: [

trigger('fade', [

transition(':enter', [

style({ opacity: 0 }),

animate('0.5s', style({ opacity: 1 }))

]),

transition(':leave', [

animate('0.5s', style({ opacity: 0 }))

])

])

]

})

export class AppComponent {

showElement = false;

toggle() {

this.showElement = !this.showElement;

}

}

```

\*\*app.component.html\*\*

```html

<button (click)="toggle()">Toggle Element</button>

<div \*ngIf="showElement" @fade>Content fades in/out</div>

```

---

### \*\*48. **Implement role-based access control (RBAC) for different users**\*\*

\*\*auth.guard.ts\*\*

```ts

@Injectable({ providedIn: 'root' })

export class AuthGuard implements CanActivate {

constructor(private authService: AuthService, private router: Router) {}

canActivate(): boolean {

if (this.authService.userRole !== 'admin') {

this.router.navigate(['/unauthorized']);

return false;

}

return true;

}

}

```

---

### \*\*49**. Create a dropdown component that supports multi-select**\*\*

\*\*dropdown.component.ts\*\*

```ts

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

@Component({

selector: 'app-dropdown',

templateUrl: './dropdown.component.html'

})

export class DropdownComponent {

options = ['Option 1', 'Option 2', 'Option 3'];

selectedOptions: string[] = [];

toggleSelection(option: string) {

const index = this.selectedOptions.indexOf(option);

if (index > -1) {

this.selectedOptions.splice(index, 1);

} else {

this.selectedOptions.push(option);

}

}

}

```

\*\*dropdown.component.html\*\*

```html

<ul>

<li \*ngFor="let option of options">

<input type="checkbox" [checked]="selectedOptions.includes(option)" (change)="toggleSelection(option)" />

{{ option }}

</li>

</ul>

```

---

### \*\*50. **Implement custom Angular pipes for filtering and sorting lists**\*\*

\*\*filter.pipe.ts\*\*

```ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'filter'

})

export class FilterPipe implements PipeTransform {

transform(value: string[], term: string): string[] {

if (!term) return value;

return value.filter(item => item.toLowerCase().includes(term.toLowerCase()));

}

}

```

\*\*app.component.ts\*\*

```ts

items = ['Apple', 'Banana', 'Cherry', 'Date', 'Elderberry'];

term = '';

```

\*\*app.component.html\*\*

```html

<input [(ngModel)]="term" placeholder="Search">

<ul>

<li \*ngFor="let item of items | filter:term">{{ item }}</li>

</ul>

```

### \*\*51. **Use ngSwitch directive to switch between multiple views based on a condition**\*\*

\*\*app.component.ts\*\*

```ts

export class AppComponent {

currentView = 'home';

}

```

\*\*app.component.html\*\*

```html

<div [ngSwitch]="currentView">

<div \*ngSwitchCase="'home'">Home View</div>

<div \*ngSwitchCase="'profile'">Profile View</div>

<div \*ngSwitchDefault>Default View</div>

</div>

<button (click)="currentView = 'home'">Home</button>

<button (click)="currentView = 'profile'">Profile</button>

<button (click)="currentView = 'other'">Other</button>

```

---

### \*\*52. **Implement a tooltip component using Angular Material**\*\*

\*\*Step 1:\*\* Install Angular Material if not already done.

```bash

ng add @angular/material

```

\*\*app.component.html\*\*

```html

<button mat-raised-button matTooltip="This is a tooltip" aria-label="Button that shows a tooltip">

Hover over me

</button>

```

---

### \*\*53. **Create a breadcrumb navigation component for better UX**\*\*

\*\*breadcrumb.component.ts\*\*

```ts

import { Component, Input } from '@angular/core';

@Component({

selector: 'app-breadcrumb',

template: `

<nav aria-label="breadcrumb">

<ol class="breadcrumb">

<li \*ngFor="let item of breadcrumb" class="breadcrumb-item">

<a [routerLink]="item.link">{{ item.label }}</a>

</li>

</ol>

</nav>

`

})

export class BreadcrumbComponent {

@Input() breadcrumb: { label: string; link: string }[] = [];

}

```

\*\*app.component.html\*\*

```html

<app-breadcrumb [breadcrumb]="[{label: 'Home', link: '/'}, {label: 'Products', link: '/products'}]"></app-breadcrumb>

```

---

### \*\*54**. Create a breadcrumb routing system in Angular**\*\*

\*\*app-routing.module.ts\*\*

```ts

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'products', component: ProductsComponent }

];

```

\*\*breadcrumb.component.ts (Updated for dynamic routing)\*\*

```ts

import { ActivatedRoute, NavigationEnd, Router } from '@angular/router';

@Component({

selector: 'app-breadcrumb',

template: `

<nav aria-label="breadcrumb">

<ol class="breadcrumb">

<li \*ngFor="let item of breadcrumb" class="breadcrumb-item">

<a [routerLink]="item.link">{{ item.label }}</a>

</li>

</ol>

</nav>

`

})

export class BreadcrumbComponent {

breadcrumb: { label: string; link: string }[] = [];

constructor(private router: Router, private activatedRoute: ActivatedRoute) {

this.router.events.subscribe(event => {

if (event instanceof NavigationEnd) {

this.updateBreadcrumb();

}

});

}

private updateBreadcrumb() {

let route = this.activatedRoute.root;

this.breadcrumb = [];

while (route) {

const routeSnapshot = route.snapshot;

if (routeSnapshot.data['breadcrumb']) {

this.breadcrumb.push({ label: routeSnapshot.data['breadcrumb'], link: routeSnapshot.url.join('/') });

}

route = route.firstChild;

}

}

}

```

---

### \*\*55. **Write unit tests for a service using Jasmine and TestBed**\*\*

\*\*auth.service.ts\*\*

```ts

export class AuthService {

private isAuthenticated = false;

login(username: string, password: string): boolean {

if (username === 'admin' && password === 'admin') {

this.isAuthenticated = true;

return true;

}

return false;

}

getAuthenticationStatus(): boolean {

return this.isAuthenticated;

}

}

```

\*\*auth.service.spec.ts\*\*

```ts

import { TestBed } from '@angular/core/testing';

import { AuthService } from './auth.service';

describe('AuthService', () => {

let service: AuthService;

beforeEach(() => {

TestBed.configureTestingModule({});

service = TestBed.inject(AuthService);

});

it('should be created', () => {

expect(service).toBeTruthy();

});

it('should authenticate with correct credentials', () => {

expect(service.login('admin', 'admin')).toBeTrue();

expect(service.getAuthenticationStatus()).toBeTrue();

});

it('should not authenticate with incorrect credentials', () => {

expect(service.login('user', 'wrong')).toBeFalse();

expect(service.getAuthenticationStatus()).toBeFalse();

});

});

```

---

### \*\*56. **Write unit tests for a component with mock dependencies**\*\*

\*\*app.component.ts\*\*

```ts

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

import { AuthService } from './auth.service';

@Component({

selector: 'app-root',

template: '<div>{{ message }}</div>'

})

export class AppComponent {

message: string;

constructor(private authService: AuthService) {}

ngOnInit() {

this.message = this.authService.getAuthenticationStatus() ? 'Welcome' : 'Please log in';

}

}

```

\*\*app.component.spec.ts\*\*

```ts

import { ComponentFixture, TestBed } from '@angular/core/testing';

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

import { AuthService } from './auth.service';

import { of } from 'rxjs';

class MockAuthService {

getAuthenticationStatus() {

return true;

}

}

describe('AppComponent', () => {

let component: AppComponent;

let fixture: ComponentFixture<AppComponent>;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [AppComponent],

providers: [{ provide: AuthService, useClass: MockAuthService }]

});

fixture = TestBed.createComponent(AppComponent);

component = fixture.componentInstance;

fixture.detectChanges();

});

it('should display welcome message when authenticated', () => {

expect(component.message).toBe('Welcome');

});

});

```

---

### \*\*57. **Mock HTTP requests in Angular unit tests using HttpClientTestingModule**\*\*

\*\*app.component.ts\*\*

```ts

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

import { Component, OnInit } from '@angular/core';

@Component({

selector: 'app-root',

template: `<div>{{ data }}</div>`

})

export class AppComponent implements OnInit {

data: any;

constructor(private http: HttpClient) {}

ngOnInit() {

this.http.get('https://api.example.com/data').subscribe(response => {

this.data = response;

});

}

}

```

\*\*app.component.spec.ts\*\*

```ts

import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';

import { ComponentFixture, TestBed } from '@angular/core/testing';

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

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

describe('AppComponent', () => {

let component: AppComponent;

let fixture: ComponentFixture<AppComponent>;

let httpMock: HttpTestingController;

beforeEach(() => {

TestBed.configureTestingModule({

declarations: [AppComponent],

imports: [HttpClientTestingModule]

});

fixture = TestBed.createComponent(AppComponent);

component = fixture.componentInstance;

httpMock = TestBed.inject(HttpTestingController);

fixture.detectChanges();

});

it('should fetch data from API', () => {

const mockData = { name: 'Angular' };

component.ngOnInit();

const req = httpMock.expectOne('https://api.example.com/data');

expect(req.request.method).toBe('GET');

req.flush(mockData);

fixture.detectChanges();

expect(component.data).toEqual(mockData);

});

});

```

---

### \*\*58. **Use Angular's dependency injection to inject a service into a component**\*\*

\*\*app.component.ts\*\*

```ts

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

import { DataService } from './data.service';

@Component({

selector: 'app-root',

template: '<div>{{ data }}</div>'

})

export class AppComponent {

data: string;

constructor(private dataService: DataService) {}

ngOnInit() {

this.data = this.dataService.getData();

}

}

```

\*\*data.service.ts\*\*

```ts

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

@Injectable({ providedIn: 'root' })

export class DataService {

getData() {

return 'Hello, world!';

}

}

```

---

### \*\*59. **Implement a custom filter pipe that sorts or filters data**\*\*

\*\*sort.pipe.ts\*\*

```ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'sort'

})

export class SortPipe implements PipeTransform {

transform(value: string[], ...args: unknown[]): string[] {

return value.sort();

}

}

```

\*\*app.component.ts\*\*

```ts

export class AppComponent {

items = ['Banana', 'Apple', 'Cherry'];

}

```

\*\*app.component.html\*\*

```html

<ul>

<li \*ngFor="let item of items | sort">{{ item }}</li>

</ul>

```

---

### \*\*60. **Use ngModel to create a two-way binding for form controls in a form group**\*\*

\*\*app.component.ts\*\*

```ts

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

import { FormGroup, FormBuilder } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent {

form: FormGroup;

constructor(private fb: FormBuilder) {

this.form = this.fb.group({

name: ['']

});

}

}

```

\*\*app.component.html\*\*

```html

<form [formGroup]="form">

<input formControlName="name" [(ngModel)]="form.value.name" placeholder="Enter name" />

</form>

<p>{{ form.value.name }}</p>

```

### \*\*61. **Create a reusable button component that emits an event when clicked**\*\*

\*\*button.component.ts\*\*

```ts

import { Component, EventEmitter, Output } from '@angular/core';

@Component({

selector: 'app-button',

template: `<button (click)="handleClick()">Click me</button>`

})

export class ButtonComponent {

@Output() buttonClicked = new EventEmitter<void>();

handleClick() {

this.buttonClicked.emit();

}

}

```

\*\*app.component.html\*\*

```html

<app-button (buttonClicked)="onButtonClicked()"></app-button>

```

\*\*app.component.ts\*\*

```ts

export class AppComponent {

onButtonClicked() {

console.log('Button clicked!');

}

}

```

---

### \*\*62. **Set up Angular Universal (SSR) for server-side rendering**\*\*

\*\*Step 1:\*\* Set up Angular Universal.

```bash

ng add @nguniversal/express-engine

```

\*\*Step 2:\*\* Run the server-side rendering build.

```bash

npm run build:ssr

npm run serve:ssr

```

Angular Universal generates the necessary code and configuration to enable server-side rendering of your Angular app.

---

### \*\*63. Add Google Maps to an Angular project using a wrapper library\*\*

\*\*Step 1:\*\* Install the wrapper library.

```bash

npm install @agm/core

```

\*\*app.module.ts\*\*

```ts

import { AgmCoreModule } from '@agm/core';

@NgModule({

imports: [AgmCoreModule.forRoot({ apiKey: 'YOUR\_GOOGLE\_MAPS\_API\_KEY' })],

...

})

export class AppModule {}

```

\*\*app.component.html\*\*

```html

<agm-map [latitude]="lat" [longitude]="lng" [zoom]="zoom">

<agm-marker [latitude]="lat" [longitude]="lng"></agm-marker>

</agm-map>

```

\*\*app.component.ts\*\*

```ts

export class AppComponent {

lat = 51.678418;

lng = 7.809007;

zoom = 8;

}

```

---

### \*\*64. **Create a date picker component with Angular Material**\*\*

\*\*Step 1:\*\* Install Angular Material.

```bash

ng add @angular/material

```

\*\*app.module.ts\*\*

```ts

import { MatDatepickerModule } from '@angular/material/datepicker';

import { MatNativeDateModule } from '@angular/material/core';

@NgModule({

imports: [MatDatepickerModule, MatNativeDateModule],

})

export class AppModule {}

```

\*\*app.component.html\*\*

```html

<mat-form-field>

<input matInput [matDatepicker]="picker" placeholder="Choose a date">

<mat-datepicker #picker></mat-datepicker>

</mat-form-field>

```

---

### \*\*65. **Implement drag-and-drop functionality in an Angular app using Angular Material**\*\*

\*\*Step 1:\*\* Install Angular Material.

```bash

ng add @angular/material

```

\*\*app.module.ts\*\*

```ts

import { DragDropModule } from '@angular/cdk/drag-drop';

@NgModule({

imports: [DragDropModule],

})

export class AppModule {}

```

\*\*app.component.html\*\*

```html

<div cdkDropList (cdkDropListDropped)="onDrop($event)">

<div \*ngFor="let item of items" cdkDrag>{{ item }}</div>

</div>

```

\*\*app.component.ts\*\*

```ts

export class AppComponent {

items = ['Item 1', 'Item 2', 'Item 3'];

onDrop(event: CdkDragDrop<string[]>) {

const previousIndex = this.items.findIndex(d => d === event.item.data);

moveItemInArray(this.items, previousIndex, event.currentIndex);

}

}

```

---

### \*\*66. **Create a file upload component that shows progress**\*\*

\*\*file-upload.component.ts\*\*

```ts

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

import { HttpClient, HttpEventType, HttpHeaders, HttpRequest } from '@angular/common/http';

@Component({

selector: 'app-file-upload',

template: `

<input type="file" (change)="onFileSelected($event)" />

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

<div>Upload Progress: {{ progress }}%</div>

</div>

`

})

export class FileUploadComponent {

progress: number | null = null;

constructor(private http: HttpClient) {}

onFileSelected(event: any) {

const file = event.target.files[0];

if (file) {

this.uploadFile(file);

}

}

uploadFile(file: File) {

const formData = new FormData();

formData.append('file', file);

const uploadReq = new HttpRequest('POST', '/upload', formData, {

headers: new HttpHeaders(),

reportProgress: true

});

this.http.request(uploadReq).subscribe(event => {

switch (event.type) {

case HttpEventType.UploadProgress:

if (event.total) {

this.progress = Math.round((100 \* event.loaded) / event.total);

}

break;

case HttpEventType.Response:

console.log('File uploaded successfully');

break;

}

});

}

}

```

---

### \*\*67**. Use ng-content to project content into a child component**\*\*

\*\*child.component.ts\*\*

```ts

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

@Component({

selector: 'app-child',

template: `<div><ng-content></ng-content></div>`

})

export class ChildComponent {}

```

\*\*app.component.html\*\*

```html

<app-child>

<p>This is projected content</p>

</app-child>

```

---

### \*\*68**. Set up an Angular project with environment-specific configurations**\*\*

\*\*Step 1:\*\* Modify `angular.json` to include environment-specific configurations.

```json

"configurations": {

"production": {

"fileReplacements": [

{

"replace": "src/environments/environment.ts",

"with": "src/environments/environment.prod.ts"

}

]

}

}

```

\*\*src/environments/environment.ts\*\*

```ts

export const environment = {

production: false,

apiUrl: 'http://localhost:3000'

};

```

\*\*src/environments/environment.prod.ts\*\*

```ts

export const environment = {

production: true,

apiUrl: 'https://api.example.com'

};

```

\*\*app.component.ts\*\*

```ts

import { environment } from '../environments/environment';

export class AppComponent {

apiUrl = environment.apiUrl;

}

```

---

### \*\*69**. Optimize performance with trackBy in ngFor for better rendering**\*\*

\*\*app.component.ts\*\*

```ts

export class AppComponent {

items = [{ id: 1, name: 'Item 1' }, { id: 2, name: 'Item 2' }];

trackById(index: number, item: any): number {

return item.id;

}

}

```

\*\*app.component.html\*\*

```html

<ul>

<li \*ngFor="let item of items; trackBy: trackById">{{ item.name }}</li>

</ul>

```

---

### \*\*70. **Implement infinite scrolling for long lists of data**\*\*

\*\*Step 1:\*\* Install Angular CDK.

```bash

npm install @angular/cdk

```

\*\*app.component.html\*\*

```html

<div (scrolled)="loadData()" class="scroll-container">

<div \*ngFor="let item of items">{{ item }}</div>

</div>

```

\*\*app.component.ts\*\*

```ts

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

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent {

items = Array(20).fill(0).map((\_, i) => `Item ${i + 1}`);

loadData() {

const nextItems = Array(20).fill(0).map((\_, i) => `Item ${i + 21}`);

this.items.push(...nextItems);

}

}

```

\*\*app.component.css\*\*

```css

.scroll-container {

height: 400px;

overflow-y: scroll;

}

```

### \*\*71. **Use Angular’s ChangeDetectionStrategy.OnPush for performance optimization**\*\*

\*\*app.component.ts\*\*

```ts

import { Component, ChangeDetectionStrategy } from '@angular/core';

@Component({

selector: 'app-root',

changeDetection: ChangeDetectionStrategy.OnPush,

template: `

<div>

<p>{{ message }}</p>

<button (click)="changeMessage()">Change Message</button>

</div>

`

})

export class AppComponent {

message = 'Hello, world!';

changeMessage() {

this.message = 'Message changed!';

}

}

```

\*\*Explanation:\*\*

Using `ChangeDetectionStrategy.OnPush` tells Angular to check for changes only when specific events occur (such as an input change or an event). This improves performance for large applications by reducing the number of change detection cycles.

---

### \*\*72**. Create a multi-step form wizard using Angular forms**\*\*

\*\*app.component.html\*\*

```html

<form (ngSubmit)="onSubmit()" [formGroup]="form">

<div \*ngIf="step === 1">

<label>Name:</label>

<input formControlName="name" />

<button (click)="nextStep()">Next</button>

</div>

<div \*ngIf="step === 2">

<label>Email:</label>

<input formControlName="email" />

<button (click)="nextStep()">Next</button>

</div>

<div \*ngIf="step === 3">

<label>Age:</label>

<input formControlName="age" />

<button type="submit">Submit</button>

</div>

</form>

```

\*\*app.component.ts\*\*

```ts

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

import { FormBuilder, FormGroup, Validators } from '@angular/forms';

@Component({

selector: 'app-root',

templateUrl: './app.component.html'

})

export class AppComponent {

form: FormGroup;

step = 1;

constructor(private fb: FormBuilder) {

this.form = this.fb.group({

name: ['', Validators.required],

email: ['', [Validators.required, Validators.email]],

age: ['', Validators.required]

});

}

nextStep() {

if (this.form.valid) {

this.step++;

}

}

onSubmit() {

console.log(this.form.value);

}

}

```

---

### \*\*73**. Use Angular’s lazy loading for large modules to improve the load time**\*\*

\*\*Step 1:\*\* Create a feature module and configure lazy loading.

\*\*app-routing.module.ts\*\*

```ts

import { NgModule } from '@angular/core';

import { RouterModule, Routes } from '@angular/router';

import { HomeComponent } from './home/home.component';

const routes: Routes = [

{ path: '', component: HomeComponent },

{ path: 'feature', loadChildren: () => import('./feature/feature.module').then(m => m.FeatureModule) }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule {}

```

\*\*feature.module.ts\*\*

```ts

import { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';

import { FeatureComponent } from './feature.component';

@NgModule({

declarations: [FeatureComponent],

imports: [CommonModule]

})

export class FeatureModule {}

```

\*\*Explanation:\*\*

This will make the `FeatureModule` lazy-loaded, meaning it will only be loaded when the user navigates to the `feature` route, reducing the initial load time.

---

### \*\*74. **Create a reusable modal component to show alerts and confirmations**\*\*

\*\*modal.component.ts\*\*

```ts

import { Component, Input, Output, EventEmitter } from '@angular/core';

@Component({

selector: 'app-modal',

template: `

<div class="modal" \*ngIf="isVisible">

<div class="modal-content">

<p>{{ message }}</p>

<button (click)="closeModal()">Close</button>

</div>

</div>

`,

styleUrls: ['./modal.component.css']

})

export class ModalComponent {

@Input() isVisible: boolean = false;

@Input() message: string = '';

@Output() closed = new EventEmitter<void>();

closeModal() {

this.closed.emit();

}

}

```

\*\*app.component.html\*\*

```html

<app-modal [isVisible]="modalVisible" [message]="'Are you sure?'"

(closed)="modalVisible = false"></app-modal>

<button (click)="modalVisible = true">Show Modal</button>

```

---

### \*\*75. **Create a state management solution using NgRx or Akita**\*\*

\*\*Step 1:\*\* Install NgRx.

```bash

npm install @ngrx/store

```

\*\*app.module.ts\*\*

```ts

import { StoreModule } from '@ngrx/store';

@NgModule({

imports: [StoreModule.forRoot({ count: counterReducer })],

...

})

export class AppModule {}

```

\*\*counter.actions.ts\*\*

```ts

import { createAction } from '@ngrx/store';

export const increment = createAction('[Counter] Increment');

export const decrement = createAction('[Counter] Decrement');

```

\*\*counter.reducer.ts\*\*

```ts

import { createReducer, on } from '@ngrx/store';

import { increment, decrement } from './counter.actions';

export const initialState = 0;

const \_counterReducer = createReducer(

initialState,

on(increment, state => state + 1),

on(decrement, state => state - 1)

);

export function counterReducer(state: number = initialState, action: any) {

return \_counterReducer(state, action);

}

```

\*\*Explanation:\*\*

NgRx provides a powerful state management solution based on Redux principles. It involves creating actions, reducers, and selectors to manage and access the app's state.

---

### \*\*76. **Optimize bundle size using Angular CLI's build optimizations**\*\*

\*\*Step 1:\*\* Build your app with optimizations enabled.

```bash

ng build --prod

```

\*\*Explanation:\*\*

The `--prod` flag enables several optimizations like tree-shaking, minification, and Ahead-of-Time (AOT) compilation, which reduce the bundle size and improve the app’s performance.

---

### \*\*77**. Create a custom Angular validator that checks for complex patterns**\*\*

\*\*pattern-validator.directive.ts\*\*

```ts

import { Directive } from '@angular/core';

import { AbstractControl, ValidationErrors, NG\_VALIDATORS, Validator } from '@angular/forms';

@Directive({

selector: '[appPatternValidator]',

providers: [{ provide: NG\_VALIDATORS, useExisting: PatternValidatorDirective, multi: true }]

})

export class PatternValidatorDirective implements Validator {

validate(control: AbstractControl): ValidationErrors | null {

const pattern = /^[A-Za-z0-9]+$/; // alphanumeric pattern

if (!pattern.test(control.value)) {

return { invalidPattern: true };

}

return null;

}

}

```

\*\*app.component.html\*\*

```html

<input formControlName="username" appPatternValidator />

```

---

### \*\*78. **Build a progressive web app (PWA) with Angular and Service Workers**\*\*

\*\*Step 1:\*\* Add PWA support to the Angular project.

```bash

ng add @angular/pwa

```

\*\*Step 2:\*\* Build and serve your app as a PWA.

```bash

ng build --prod

```

\*\*Explanation:\*\*

Angular PWA adds service workers to cache assets and provide offline functionality, improving the app experience on unreliable networks.

---

### \*\*79. **Integrate Firebase Authentication for user login and registration**\*\*

\*\*Step 1:\*\* Install Firebase and Firebase Authentication.

```bash

npm install firebase @angular/fire

```

\*\*Step 2:\*\* Set up Firebase Authentication in the project.

\*\*app.module.ts\*\*

```ts

import { AngularFireModule } from '@angular/fire';

import { AngularFireAuthModule } from '@angular/fire/auth';

@NgModule({

imports: [

AngularFireModule.initializeApp(environment.firebase),

AngularFireAuthModule

],

})

export class AppModule {}

```

\*\*app.component.ts\*\*

```ts

import { AngularFireAuth } from '@angular/fire/auth';

export class AppComponent {

constructor(private afAuth: AngularFireAuth) {}

login(email: string, password: string) {

this.afAuth.signInWithEmailAndPassword(email, password)

.then(user => console.log('User logged in', user))

.catch(error => console.error(error));

}

register(email: string, password: string) {

this.afAuth.createUserWithEmailAndPassword(email, password)

.then(user => console.log('User registered', user))

.catch(error => console.error(error));

}

}

```

---

### \*\*80. **Create a real-time chat application with Angular and WebSockets**\*\*

\*\*Step 1:\*\* Set up WebSocket in Angular.

\*\*chat.service.ts\*\*

```ts

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

import { WebSocketSubject } from 'rxjs/webSocket';

@Injectable({

providedIn: 'root'

})

export class ChatService {

private socket: WebSocketSubject<any>;

constructor() {

this.socket = new WebSocketSubject('ws://localhost:8080');

}

sendMessage(message: string) {

this.socket.next({ message });

}

getMessages() {

return this.socket.asObservable();

}

}

```

\*\*app.component.ts\*\*

```ts

export class AppComponent {

messages: string[] = [];

constructor(private chatService: ChatService) {

chatService.getMessages().subscribe(message => {

this.messages.push(message);

});

}

sendMessage(message: string) {

this.chatService.sendMessage(message);

}

}

```

### \*\*81**. Implement a custom backend API call with retry logic and exponential backoff**\*\*

\*\*Step 1:\*\* Create a service to call the backend API.

\*\*api.service.ts\*\*

```ts

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

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

import { Observable, throwError } from 'rxjs';

import { catchError, retryWhen, delay, scan } from 'rxjs/operators';

@Injectable({

providedIn: 'root',

})

export class ApiService {

private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

constructor(private http: HttpClient) {}

getPosts(): Observable<any> {

return this.http.get(this.apiUrl).pipe(

retryWhen(errors =>

errors.pipe(

scan((retryCount, error) => {

if (retryCount >= 3) {

throw error;

}

return retryCount + 1;

}, 0),

delay(1000)

)

),

catchError(error => {

console.error('API error:', error);

return throwError('Something went wrong');

})

);

}

}

```

\*\*Explanation:\*\*

This service uses the `retryWhen` operator to retry an API call up to 3 times with an exponential backoff (delay). If all retries fail, the error is thrown.

---

### \*\*82. **Use Angular’s dynamic component loading with a factory resolver**\*\*

\*\*Step 1:\*\* Create the components to load dynamically.

\*\*dynamic.component.ts\*\*

```ts

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

@Component({

selector: 'app-dynamic',

template: `<h1>Dynamic Component Loaded!</h1>`

})

export class DynamicComponent {}

```

\*\*Step 2:\*\* Dynamically load the component.

\*\*app.component.ts\*\*

```ts

import { Component, ViewChild, ViewContainerRef, ComponentFactoryResolver, OnInit } from '@angular/core';

import { DynamicComponent } from './dynamic.component';

@Component({

selector: 'app-root',

template: `<div #container></div> <button (click)="loadDynamicComponent()">Load Component</button>`

})

export class AppComponent implements OnInit {

@ViewChild('container', { read: ViewContainerRef }) container!: ViewContainerRef;

constructor(private resolver: ComponentFactoryResolver) {}

ngOnInit(): void {}

loadDynamicComponent() {

const factory = this.resolver.resolveComponentFactory(DynamicComponent);

this.container.createComponent(factory);

}

}

```

\*\*Explanation:\*\*

Using `ComponentFactoryResolver`, Angular can dynamically load the `DynamicComponent` into the DOM when a button is clicked.

---

### \*\*83**. Create a global error handler to catch and handle application errors**\*\*

\*\*Step 1:\*\* Create a global error handler service.

\*\*error-handler.service.ts\*\*

```ts

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

import { ErrorHandler } from '@angular/core';

@Injectable()

export class GlobalErrorHandlerService implements ErrorHandler {

handleError(error: any): void {

console.error('Global Error:', error);

}

}

```

\*\*Step 2:\*\* Provide the error handler in the app module.

\*\*app.module.ts\*\*

```ts

import { NgModule } from '@angular/core';

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

import { GlobalErrorHandlerService } from './error-handler.service';

@NgModule({

declarations: [],

imports: [BrowserModule],

providers: [{ provide: ErrorHandler, useClass: GlobalErrorHandlerService }],

bootstrap: []

})

export class AppModule {}

```

\*\*Explanation:\*\*

This error handler service will catch and log errors globally in your Angular application.

---

### \*\*84. **Implement offline support in an Angular app with Service Workers and IndexedDB**\*\*

\*\*Step 1:\*\* Install Angular PWA and enable service workers.

```bash

ng add @angular/pwa

```

\*\*Step 2:\*\* Enable IndexedDB for offline data storage (this is optional for caching data).

\*\*offline.service.ts\*\*

```ts

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

import { openDB } from 'idb';

@Injectable({

providedIn: 'root',

})

export class OfflineService {

dbPromise = openDB('my-database', 1, {

upgrade(db) {

db.createObjectStore('posts', { keyPath: 'id' });

},

});

async savePost(post: any) {

const db = await this.dbPromise;

await db.put('posts', post);

}

async getPosts() {

const db = await this.dbPromise;

return db.getAll('posts');

}

}

```

\*\*Explanation:\*\*

By using Angular PWA and IndexedDB, this service enables your app to cache data for offline use. When the user is offline, data is saved locally and later synchronized with the backend.

---

### \*\*85. **Use Angular CLI to create a custom schematic**\*\*

\*\*Step 1:\*\* Install Angular CLI and Schematics.

```bash

npm install -g @angular-devkit/schematics-cli

```

\*\*Step 2:\*\* Create a new schematic.

```bash

schematics blank --name=my-schematic

```

\*\*Explanation:\*\*

This creates a custom schematic that you can use to automate the creation of components, services, or other parts of your Angular project.

---

### \*\*86. **Implement JWT (JSON Web Tokens) for authentication in an Angular app**\*\*

\*\*Step 1:\*\* Install dependencies.

```bash

npm install @auth0/angular-jwt

```

\*\*Step 2:\*\* Set up JWT authentication.

\*\*auth.service.ts\*\*

```ts

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

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

import { Observable } from 'rxjs';

import { environment } from '../environments/environment';

@Injectable({

providedIn: 'root',

})

export class AuthService {

private apiUrl = `${environment.apiUrl}/auth`;

constructor(private http: HttpClient) {}

login(username: string, password: string): Observable<any> {

return this.http.post<any>(`${this.apiUrl}/login`, { username, password });

}

setToken(token: string) {

localStorage.setItem('jwt', token);

}

getToken() {

return localStorage.getItem('jwt');

}

}

```

\*\*Step 3:\*\* Use the JWT in API requests.

\*\*app.component.ts\*\*

```ts

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

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

import { AuthService } from './auth.service';

@Component({

selector: 'app-root',

template: `

<button (click)="login()">Login</button>

`,

})

export class AppComponent {

constructor(private authService: AuthService, private http: HttpClient) {}

login() {

this.authService.login('username', 'password').subscribe((response) => {

this.authService.setToken(response.token);

});

}

}

```

---

### \*\*87**. Create a deep linking solution with the Angular router for bookmarkable routes\***\*

\*\*app-routing.module.ts\*\*

```ts

const routes: Routes = [

{ path: 'product/:id', component: ProductComponent },

];

```

\*\*product.component.ts\*\*

```ts

import { Component, OnInit } from '@angular/core';

import { ActivatedRoute } from '@angular/router';

@Component({

selector: 'app-product',

template: `<h1>Product ID: {{ productId }}</h1>`

})

export class ProductComponent implements OnInit {

productId!: string;

constructor(private route: ActivatedRoute) {}

ngOnInit(): void {

this.productId = this.route.snapshot.paramMap.get('id')!;

}

}

```

\*\*Explanation:\*\*

Deep linking allows users to directly navigate to a specific route or view in an Angular app using URLs. Here, we are using route parameters to load specific data.

---

### \*\*88. **Use the Angular ChangeDetectionStrategy.OnPush for performance optimization**\*\*

\*\*app.component.ts\*\*

```ts

import { Component, ChangeDetectionStrategy } from '@angular/core';

@Component({

selector: 'app-root',

changeDetection: ChangeDetectionStrategy.OnPush,

template: `

<div>

<p>{{ message }}</p>

<button (click)="changeMessage()">Change Message</button>

</div>

`

})

export class AppComponent {

message = 'Hello, world!';

changeMessage() {

this.message = 'Message changed!';

}

}

```

---

### \*\*89. **Use custom pipes for deep object comparison**\*\*

\*\*deep-equal.pipe.ts\*\*

```ts

import { Pipe, PipeTransform } from '@angular/core';

@Pipe({

name: 'deepEqual'

})

export class DeepEqualPipe implements PipeTransform {

transform(value: any, comparisonObj: any): boolean {

return JSON.stringify(value) === JSON.stringify(comparisonObj);

}

}

```

\*\*app.component.html\*\*

```html

<div \*ngIf="obj1 | deepEqual: obj2">

Objects are equal!

</div>

```

\*\*Explanation:\*\*

This pipe compares two objects deeply and returns `true` if they are the same and `false` otherwise. It serializes both objects to JSON strings before comparing them.

---

### \*\*90. **Build an Angular application that consumes GraphQL API**\*\*

\*\*Step 1:\*\* Install Apollo Client for Angular.

```bash

npm install @apollo/client @angular/graphql

```

\*\*Step 2:\*\* Create the GraphQL service.

\*\*graphql.service.ts\*\*

```ts

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

import { Apollo, gql } from 'apollo-angular';

@Injectable({

providedIn: 'root',

})

export class GraphQLService {

constructor(private apollo: Apollo) {}

getPosts() {

const GET\_POSTS = gql`

query {

posts {

title

body

}

}

`;

return this.apollo.watchQuery({ query: GET\_POSTS }).valueChanges;

}

}

```

\*\*Step 3:\*\* Use the service in the component.

\*\*app.component.ts\*\*

```ts

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

import { GraphQLService } from './graphql.service';

@Component({

selector: 'app-root',

template: `<ul><li \*ngFor="let post of posts">{{ post.title }}</li></ul>`

})

export class AppComponent {

posts: any[] = [];

constructor(private graphqlService: GraphQLService) {}

ngOnInit() {

this.graphqlService.getPosts().subscribe((response: any) => {

this.posts = response.data.posts;

});

}

}

```

### \*\*91. **Implement a custom Angular module with lazy-loaded components**\*\*

\*\*Step 1:\*\* Create a module and a component that will be lazy-loaded.

\*\*lazy.module.ts\*\*

```ts

import { NgModule } from '@angular/core';

import { CommonModule } from '@angular/common';

import { LazyComponent } from './lazy.component';

@NgModule({

declarations: [LazyComponent],

imports: [CommonModule]

})

export class LazyModule {}

```

\*\*lazy.component.ts\*\*

```ts

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

@Component({

selector: 'app-lazy',

template: `<h2>Lazy Loaded Component</h2>`

})

export class LazyComponent {}

```

\*\*Step 2:\*\* Set up lazy loading in your routing module.

\*\*app-routing.module.ts\*\*

```ts

const routes: Routes = [

{ path: 'lazy', loadChildren: () => import('./lazy/lazy.module').then(m => m.LazyModule) }

];

```

\*\*Explanation:\*\*

This example demonstrates how to create a lazy-loaded Angular module. The `LazyComponent` will only be loaded when the user navigates to the `/lazy` path.

---

### \*\*92. **Create a complex form with multiple nested form groups using Angular forms**\*\*

\*\*Step 1:\*\* Create a form with nested form groups.

\*\*app.component.ts\*\*

```ts

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

import { FormGroup, FormBuilder, Validators } from '@angular/forms';

@Component({

selector: 'app-root',

template: `

<form [formGroup]="profileForm" (ngSubmit)="onSubmit()">

<div formGroupName="userInfo">

<label for="name">Name:</label>

<input id="name" formControlName="name">

<label for="email">Email:</label>

<input id="email" formControlName="email">

</div>

<div formGroupName="address">

<label for="city">City:</label>

<input id="city" formControlName="city">

<label for="postalCode">Postal Code:</label>

<input id="postalCode" formControlName="postalCode">

</div>

<button type="submit">Submit</button>

</form>

`

})

export class AppComponent {

profileForm: FormGroup;

constructor(private fb: FormBuilder) {

this.profileForm = this.fb.group({

userInfo: this.fb.group({

name: ['', Validators.required],

email: ['', [Validators.required, Validators.email]],

}),

address: this.fb.group({

city: ['', Validators.required],

postalCode: ['', Validators.required],

})

});

}

onSubmit() {

console.log(this.profileForm.value);

}

}

```

\*\*Explanation:\*\*

This example uses Angular's `FormGroup` and `FormBuilder` to create a complex form with nested groups for user information and address. The form has validation as well.

---

### \*\*93. **Build a multi-language (i18n) Angular app that changes the language dynamically**\*\*

\*\*Step 1:\*\* Install and configure Angular i18n.

```bash

ng add @angular/localize

```

\*\*Step 2:\*\* Set up the i18n configuration and translation files.

\*\*en.json\*\*

```json

{

"hello": "Hello"

}

```

\*\*fr.json\*\*

```json

{

"hello": "Bonjour"

}

```

\*\*Step 3:\*\* Set up dynamic language switching.

\*\*app.component.ts\*\*

```ts

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

import { TranslateService } from '@ngx-translate/core';

@Component({

selector: 'app-root',

template: `

<div>{{ 'hello' | translate }}</div>

<button (click)="switchLanguage('en')">English</button>

<button (click)="switchLanguage('fr')">French</button>

`

})

export class AppComponent {

constructor(private translate: TranslateService) {

this.translate.setDefaultLang('en');

}

switchLanguage(lang: string) {

this.translate.use(lang);

}

}

```

\*\*Explanation:\*\*

This example uses Angular i18n (via `@ngx-translate/core`) to switch between English and French dynamically. The translation files contain the text for each language, and the `TranslateService` manages language switching.

---

### \*\*94. **Implement a micro-frontends architecture using Angular elements and web components**\*\*

\*\*Step 1:\*\* Create an Angular element as a custom web component.

\*\*microfrontend.component.ts\*\*

```ts

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

@Component({

selector: 'app-microfrontend',

template: `<h2>Microfrontend Component</h2>`

})

export class MicroFrontendComponent {}

```

\*\*Step 2:\*\* Convert the Angular component to a web component.

\*\*main.ts\*\*

```ts

import { enableProdMode } from '@angular/core';

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

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

import { MicroFrontendComponent } from './app/microfrontend.component';

platformBrowserDynamic().bootstrapModule(AppModule)

.catch(err => console.error(err));

customElements.define('microfrontend-element', MicroFrontendComponent);

```

\*\*Explanation:\*\*

This demonstrates how to use Angular elements to create a custom web component (`microfrontend-element`). Micro-frontends are small, independently deployable pieces of an app, and Angular elements allow Angular components to be used as standalone web components.

---

### \*\*95. **Use Web Workers in an Angular app for performance improvement**\*\*

\*\*Step 1:\*\* Create a Web Worker.

```ts

// worker.ts

self.addEventListener('message', (event) => {

const data = event.data;

self.postMessage(`Hello ${data.name}`);

});

```

\*\*Step 2:\*\* Set up the Angular component to communicate with the worker.

\*\*app.component.ts\*\*

```ts

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

@Component({

selector: 'app-root',

template: `<button (click)="sendMessage()">Send Message</button><div>{{ message }}</div>`

})

export class AppComponent {

message = '';

worker: Worker;

constructor() {

this.worker = new Worker(new URL('./worker', import.meta.url));

this.worker.onmessage = (event) => {

this.message = event.data;

};

}

sendMessage() {

this.worker.postMessage({ name: 'Angular' });

}

}

```

\*\*Explanation:\*\*

This example uses a Web Worker to perform background tasks without blocking the main thread, improving app performance.

---

### \*\*96. **Integrate with an external third-party API (e.g., Stripe, Google Maps**)\*\*

\*\*Step 1:\*\* Install Stripe package.

```bash

npm install @stripe/stripe-js

```

\*\*Step 2:\*\* Set up Stripe in your Angular component.

\*\*stripe.component.ts\*\*

```ts

import { Component, OnInit } from '@angular/core';

import { loadStripe } from '@stripe/stripe-js';

@Component({

selector: 'app-stripe',

template: `<button (click)="pay()">Pay with Stripe</button>`

})

export class StripeComponent implements OnInit {

stripe: any;

ngOnInit(): void {

loadStripe('your-public-key-here').then((stripe) => {

this.stripe = stripe;

});

}

pay() {

// Implement Stripe payment logic

}

}

```

\*\*Explanation:\*\*

This example demonstrates how to integrate Stripe’s JavaScript library with Angular to process payments. You can initialize Stripe with your public key and use it to handle payments.

---

### \*\*97**. Implement server-side pagination for a large dataset in Angular**\*\*

\*\*Step 1:\*\* Set up server-side pagination logic.

\*\*api.service.ts\*\*

```ts

getPosts(page: number, limit: number): Observable<any> {

return this.http.get(`https://jsonplaceholder.typicode.com/posts?\_page=${page}&\_limit=${limit}`);

}

```

\*\*Step 2:\*\* Create a component to display paginated data.

\*\*app.component.ts\*\*

```ts

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

import { ApiService } from './api.service';

@Component({

selector: 'app-root',

template: `

<div \*ngFor="let post of posts">

<h3>{{ post.title }}</h3>

</div>

<button (click)="loadPosts()">Next</button>

`

})

export class AppComponent {

posts: any[] = [];

currentPage = 1;

constructor(private apiService: ApiService) {}

loadPosts() {

this.apiService.getPosts(this.currentPage, 10).subscribe((data) => {

this.posts = data;

this.currentPage++;

});

}

}

```

\*\*Explanation:\*\*

This example demonstrates server-side pagination, where the client requests specific pages of data from the server (with limits), improving performance for large datasets.

---

### \*\*98. **Create an Angular app with multi-theme support**\*\*

\*\*Step 1:\*\* Define two themes.

\*\*styles.scss\*\*

```scss

/\* Default Theme \*/

:root {

--primary-color: blue;

}

/\* Dark Theme \*/

.dark-theme {

--primary-color: black;

--text-color: white;

}

body {

background-color: var(--primary-color);

color: var(--text-color);

}

```

\*\*Step 2:\*\* Switch themes dynamically.

\*\*app.component.ts\*\*

```ts

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

@Component({

selector: 'app-root',

template: `

<button (click)="toggleTheme()">Toggle Theme</button>

`

})

export class AppComponent {

isDark = false;

toggleTheme() {

this.isDark = !this.isDark;

if (this.isDark) {

document.body.classList.add('dark-theme');

} else {

document.body.classList.remove('dark-theme');

}

}

}

```

\*\*Explanation:\*\*

This example shows how to toggle between multiple themes in an Angular app using CSS custom properties (CSS variables).

---

### \*\*99**. Implement advanced routing with child routes and nested views**\*\*

\*\*app-routing.module.ts\*\*

```ts

const routes: Routes = [

{

path: 'parent',

component: ParentComponent,

children: [

{ path: 'child', component: ChildComponent }

]

}

];

```

\*\*Explanation:\*\*

In this example, the parent route has a nested child route. When you visit `/parent/child`, Angular will display both the `ParentComponent` and the `ChildComponent` in a nested view.

---

### \*\*100. **Write integration tests using Protractor or Cypress for an Angular app**\*\*

\*\*Step 1:\*\* Set up Cypress.

```bash

npm install cypress --save-dev

```

\*\*Step 2:\*\* Write a test.

\*\*cypress/integration/app.spec.ts\*\*

```ts

describe('Angular App', () => {

it('should display the title', () => {

cy.visit('/');

cy.contains('Welcome');

});

});

```

**What is RxJS?**

Explanation: RxJS is a library for reactive programming using Observables, making it easy to compose asynchronous or callback-based code. It allows you to work with asynchronous events and data streams.

**What is an Observable?**

Explanation: An Observable is a core concept in RxJS. It represents a collection of future values or events. Observables can emit multiple values over time and can be subscribed to by observers.

**What is a Subject in RxJS?**

Explanation: A Subject is both an observer and an observable. It allows values to be multicasted to many Observers. Unlike an Observable, a Subject can emit values to observers even if they subscribe after the value is emitted.

**What is the difference between Observable and Observer?**

Explanation:

Observable is a producer of values (can emit multiple values over time).

Observer is a consumer of values (it listens to or subscribes to an Observable and reacts to the emitted values).

**Explain the difference between map() and flatMap() (or mergeMap() in RxJS)?**

Explanation:

map(): Transforms values by applying a function to each emitted item.

flatMap() (or mergeMap()): Flattens the emitted values if the result is an observable. It’s useful when dealing with Observables of Observables.

**Intermediate Level**

**What is the take() operator in RxJS?**

Explanation: The take() operator allows you to limit the number of emissions an observable can send to an observer. Once the specified number of emissions is reached, the observable completes.

**What is the difference between concatMap() and switchMap()?**

Explanation:

concatMap(): Maintains the order of emissions. It waits for the current observable to complete before moving on to the next one.

switchMap(): Cancels the previous inner observable if a new value arrives. It switches to the new inner observable, which is useful in scenarios like user input autocomplete where you don’t need results from previous queries.

**What is debounceTime() and when would you use it?**

Explanation: debounceTime() is used to delay emissions from the observable by the specified time. It’s useful when you want to wait for a "pause" in a stream, like typing in a search box, to avoid triggering actions too frequently.

**What is combineLatest() used for in RxJS?**

Explanation: combineLatest() combines the latest emissions from multiple observables. It emits values whenever any of the source observables emits a value, using the most recent value from each observable.

**What is catchError() in RxJS?**

Explanation: catchError() is used to handle errors that may occur in an observable stream. You can catch errors and provide a fallback value or recover from the error by returning a new observable.

**Advanced Level**

**What is the difference between concat() and merge() in RxJS?**

Explanation:

concat(): Sequentially merges multiple observables, ensuring that each observable completes before the next one starts.

merge(): Merges multiple observables concurrently. Emissions from the source observables are interleaved based on when they emit.

**What are observeOn() and subscribeOn() in RxJS?**

Explanation:

observeOn(): Specifies the scheduler on which the subscription will observe the emitted items. It's useful for changing the execution context of the observable.

subscribeOn(): Specifies the scheduler for subscribing to the observable. It’s useful to control when the subscription happens (i.e., synchronously or asynchronously).

**Angular Interview Questions for 4 Years of Experience**

**Basic to Intermediate Level**

**What is Angular and how is it different from AngularJS?**

Explanation: Angular is a platform and framework for building client-side applications with HTML, CSS, and TypeScript/JavaScript. AngularJS is the first version (Angular 1.x) based on JavaScript, whereas Angular (2+) is a complete rewrite, based on TypeScript, and follows component-based architecture.

**What are Angular Components, and what role do they play in an Angular application?**

Explanation: Components are the fundamental building blocks of an Angular application. They consist of a TypeScript class, HTML template, and CSS styles. A component controls a part of the user interface and defines behavior with associated data and methods.

**What is a service in Angular? How do you create and inject a service?**

Explanation: Services in Angular are used to provide reusable business logic, data handling, or HTTP calls. You can create a service using @Injectable() decorator and inject it into components or other services using Angular's dependency injection system.

**What is Dependency Injection (DI) in Angular?**

Explanation: DI is a design pattern used to implement IoC (Inversion of Control). Angular’s DI system allows you to inject services and other dependencies into components, directives, and other services to reduce tight coupling and enhance modularity.

**Dependency Injection** is a way for one class to get the objects (dependencies) it needs from the outside, rather than creating them itself.

In Angular, this means we can "inject" things like services into components to keep everything more **modular**, **testable**, and **easy to manage**.

**What is ngOnInit() and when is it used in Angular components?**

Explanation: ngOnInit() is a lifecycle hook in Angular that is called after the component’s constructor is executed and after the first ngOnChanges(). It is commonly used for initializing data in components.

**What is Angular CLI and what are its common commands?**

Explanation: Angular CLI is a command-line interface for automating common Angular development tasks, like creating components, services, modules, and running tests. Some common commands include ng new, ng serve, ng generate, and ng test.

**How does Angular handle data binding? What are the different types of data binding?**

Explanation: Angular supports various types of data binding:

**Interpolation** ({{ value }}): Binding data from the component to the view.

**Property Binding** ([property]="value"): Binding an element's property to a component property.

**Event Binding** ((event)="method()"): Binding events from the view to the component's method.

**Two-way Binding** ([(ngModel)]="value"): Combination of property and event binding.

**What are directives in Angular? Explain the difference between structural and attribute directives.**

Explanation: Directives are special markers in Angular templates that modify the behavior or appearance of elements.

Structural Directives: Change the DOM structure, e.g., \*ngIf, \*ngFor.

Attribute Directives: Change the behavior or appearance of an element, e.g., ngClass, ngStyle.

**What is the purpose of Angular modules (NgModule)?**

Explanation: NgModule is a decorator used to define Angular modules. Modules help organize the application into cohesive blocks of functionality and manage dependencies by declaring components, directives, services, and other modules.

 Groups related **components**, **directives**, **pipes**, and **services**.

 Helps organize your app into **functional units** (feature modules).

 Makes the app **scalable** and **maintainable**.

 Can **import** other modules and **export** pieces for sharing.

**Advanced Level**

**Explain the Angular Change Detection mechanism.**

Explanation: **Angular Change Detection** is the mechanism that updates the **view** whenever the **component's data changes**. It runs after events like user actions, HTTP responses, or timers, and it checks the component tree to see if any data-bound properties have changed. If changes are detected, Angular updates the DOM accordingly.

Angular provides two change detection strategies:

1. **Default**:
   * Checks all components from top to bottom of the tree.
   * Ensures updates but can impact performance in large apps.
2. **OnPush**:
   * Only checks the component when its @Input() properties change or when an event occurs inside the component.
   * Great for optimizing performance in larger applications.

**What are Angular Observables, and how do they relate to the RxJS library?**

Explanation: Angular uses Observables (from RxJS) for handling asynchronous operations, such as HTTP requests or events. Observables provide a more flexible and functional way of handling async operations and allow composition, transformation, and error handling of streams of data.

**How do you optimize performance in Angular applications?**

**Lazy Loading**: Load feature modules only when required, instead of loading everything upfront.

**Ahead of Time Compilation (AOT):** Precompiles Angular templates and components during the build, reducing the runtime overhead.

**Change Detection Strategy:** Use OnPush change detection to avoid unnecessary checks.

**TrackBy in ngFor**: Use trackBy with ngFor to optimize DOM rendering when iterating over large lists.

Avoiding Expensive Computations in Templates: Avoid putting logic-heavy expressions directly in templates.

**What is Angular Router, and how do you handle routing in an Angular application?**

**Explanation**: Angular Router is a powerful tool used to navigate between different views (components) in a single-page application (SPA). You define routes in a module using RouterModule, and routerLink is used to navigate to different components. Angular Router also provides route guards, lazy loading, and parameter handling.

**What are Route Guards in Angular, and what types are available?**

**Explanation**: Route guards are used to control navigation in Angular. They can prevent or allow navigation based on certain conditions. The types of route guards include:

**CanActivate**: Determines if a route can be activated.

**CanActivateChild**: Determines if a child route can be activated.

**CanDeactivate**: Prevents navigation away from a route (e.g., unsaved data).

**Resolve**: The **Resolve** guard is used to **pre-fetch data before a route is activated**.

This means Angular waits to load the route **until the required data is ready**, ensuring components have everything they need when they are rendered.

**What is a store in Angular? How do you manage state in Angular applications?**

**Explanation**: State management is crucial for large applications. Angular doesn’t have a built-in state management solution, but there are third-party libraries like NgRx (inspired by Redux) or Akita that help manage state in a more structured way. They allow centralized state and actions to maintain data consistency.

**What are Angular Pipes, and how do you create a custom pipe?**

**Explanation**: Pipes are used for transforming data in templates. They can format dates, currency, or perform custom transformations. You can create a custom pipe by implementing the PipeTransform interface and decorating the class with @Pipe().

**What is the difference between ngOnInit() and constructor() in Angular components?**

**Explanation**:

**constructor():** Used for dependency injection and initializing basic values.

**ngOnInit():** Called after Angular has initialized the component's input properties. It’s where you typically perform data initialization or fetch data for the component.

**Explain the role of HttpClientModule in Angular.**

Explanation: The HttpClientModule is used for making HTTP requests to backend services. It provides methods like get(), post(), put(), and delete() for handling requests and returns an observable that allows handling the response or errors in a reactive way.

**How would you handle forms in Angular?**

Explanation: Angular supports both template-driven and reactive forms:

Template-driven forms are simple to use and rely on Angular’s directives like ngModel to handle form controls.

Reactive forms are more powerful and scalable, allowing you to define the form model in TypeScript and control form validation, state, and handling dynamically.

**What is ngrx and why would you use it in an Angular application?**

Explanation: NgRx is a reactive state management library for Angular applications, based on the Redux pattern. It helps manage global state, side effects (using effects), and store data in a single place, allowing components to reactively subscribe to state changes. It is suitable for large applications with complex state management needs.

**How do you perform unit testing in Angular?**

Explanation: Angular provides testing utilities through Jasmine and Karma for unit testing components, services, and pipes. You use TestBed to configure modules, and then you can test the component's logic and DOM using Angular's testing utilities.

**What is the purpose of trackBy in ngFor in Angular?**

Explanation: trackBy is used to improve performance in ngFor loops. It tells Angular how to track items in the list, preventing unnecessary DOM manipulations by keeping track of the identity of each item based on a unique identifier (e.g., an ID).

Miscellaneous

What is the Angular lifecycle, and what are the key lifecycle hooks?

Explanation: Angular provides various lifecycle hooks, including:

ngOnChanges(): Called when an input property changes.

ngOnInit(): Called once the component is initialized.

ngDoCheck(): Called during each change detection cycle.

ngAfterViewInit(): Called after the component’s view has been initialized.

ngOnDestroy(): Called just before the component is destroyed.

**1. Encapsulation**

Meaning: "Encapsulation" comes from the word "capsule". Just like a medicine is enclosed in a capsule to protect it and make it easier to take, encapsulation in OOP is about bundling the data (attributes) and methods (functions) inside a class and protecting the internal details.

Example:

Think of a television remote. The remote has buttons like "Power", "Volume Up", and "Channel Change". But, you don't need to know the internal workings of how the buttons control the TV. The remote encapsulates all that complexity for you, and you just use the simple buttons to interact with it. **Key Point**: You don’t directly mess with the internals; you interact with a simple interface.

**2. Inheritance**

Meaning: "Inheritance" is like passing down traits or features from parents to children. A child can inherit characteristics like hair color, height, etc., from their parents. In OOP, a child class can inherit properties and methods from a parent class, making it easier to reuse code and add new features.

**Example:**

Imagine a general vehicle class that has properties like wheels and engine. Now, a Car class can inherit these features from the vehicle class, but also add its own specific features like airConditioning. Similarly, a Bike class can also inherit from vehicle but add features like kickstand.

Key Point: Inheritance is like a child inheriting traits from their parents but also adding their own unique features.

**3. Polymorphism**

Meaning: "Polymorphism" means "many forms". It’s like how one action can have different behaviors based on the object that is performing it. In OOP, this allows you to use the same method name for different types of objects, and the behavior will change based on the object that calls it.

**Example:**

Imagine you have a "Play" button on different devices like a CD player, TV, or Radio. When you press "Play", each device will do something different:

CD Player: Starts playing a CD.

TV: Starts playing a show.

Radio: Starts playing a station.

Although the "Play" button is the same, each device (object) responds to it differently. This is polymorphism: the same action (method) has different results depending on the object.

**Key Point:** Polymorphism is like one button ("Play") that works differently on different devices (CD, TV, Radio).

**4. Abstraction**

Meaning: "Abstraction" means hiding unnecessary details and showing only the essential parts. It’s like using a smartphone — you don’t need to know how the phone's processor works, you just need to know how to use it (e.g., swipe to unlock).

Example:

Think of a coffee machine. You don’t need to know the details of how it boils water, grinds beans, or mixes the ingredients. All you need to do is press a button, and it makes the coffee. The machine hides all the internal complexity from you and provides a simple interface for making coffee.

Key Point: Abstraction is like using a coffee machine — you don’t need to know the internal mechanics, just how to use it.

**Recap of Key Terms:**

Encapsulation: Hiding complexity and only exposing necessary parts (like using a remote without knowing how it works inside).

Inheritance: Inheriting traits or features from a parent (like a child inheriting traits from parents).

Polymorphism: One action (like "Play") can have many different behaviors based on the object (CD, TV, Radio).

Abstraction: Hiding the details and showing only what's necessary (like making coffee with a machine without knowing how it works inside). need word file download

**1. JavaScript Fundamentals**

**var, let, and const:**

**var:** Variables declared using var are function-scoped, meaning they are accessible within the function they're defined in, even if they're declared outside a block. They also get hoisted to the top of the scope.

**let:** Variables declared with let are block-scoped, meaning they are only available within the block they’re defined in (e.g., inside a loop or conditional). They are also hoisted but are not accessible until the point of declaration (this is known as the "temporal dead zone").

**const:** Variables declared with const are also block-scoped but are immutable. You can’t reassign a const variable once it's assigned, but the contents of objects or arrays declared with const can still be modified.

**Hoisting:** In JavaScript, hoisting refers to the behavior where variable and function declarations are moved to the top of their scope during the compilation phase. However, only declarations (not initializations) are hoisted. Variables declared with var are hoisted and initialized with undefined, while let and const are hoisted but not initialized, causing a reference error if accessed before declaration.

**null vs undefined:**

* **null** is an assigned value representing "no value" or "empty." It's an object in JavaScript (historically).
* **undefined** means a variable has been declared but hasn't been assigned a value yet.

**Closures:** A closure is a function that retains access to the variables from its lexical scope, even after the outer function has finished executing. This allows for data encapsulation and private variables.

**bind(), call(), and apply():**

* These methods are used to change the context (this) of a function.
* call() and apply() are similar but differ in how they pass arguments. call() takes arguments as a comma-separated list, while apply() takes an array of arguments.
* bind() returns a new function with a fixed this value and specified parameters.

**2. Asynchronous JavaScript**

**Callbacks, Promises, Async/Await:**

* **Callbacks** are functions passed as arguments to other functions and executed later, typically used in asynchronous operations. The problem with callbacks is "callback hell," where multiple nested callbacks become difficult to manage.
* **Promises** provide a cleaner way to handle asynchronous operations. They represent the eventual completion (or failure) of an asynchronous operation and its resulting value.
* **Async/Await:** Introduced in ES2017, async functions return a promise, and within these functions, you can use await to pause execution until the promise is resolved.

**Event Loop:** JavaScript is single-threaded, meaning it can execute one operation at a time. The event loop allows JavaScript to handle asynchronous operations by queuing them in the message queue and processing them after the current execution stack is empty.

**Error Handling in Promises:** If a promise is rejected, it will propagate the error down the chain unless you handle it with .catch() or use try/catch in async/await functions.

**3. ES6+ Features**

**Destructuring:** Destructuring allows you to unpack values from arrays or properties from objects into distinct variables. This improves readability and reduces boilerplate code.

const person = { name: 'John', age: 30 };

const { name, age } = person; // Destructuring

**Spread Operator and Rest Parameters:**

The spread operator (...) allows you to unpack values from an array or object into a new array or object.

Rest parameters allow you to collect remaining arguments into a single array.

const arr = [1, 2, 3];

const arr2 = [...arr, 4, 5]; **// Spread**

function sum(...numbers) { return numbers.reduce((a, b) => a + b); **} // Rest**

**Modules:** ES6 modules allow you to export code from one file and import it into another, making your code more modular and maintainable. You use export to export variables or functions and import to bring them into other files.

// math.js

export function add(a, b) { return a + b; }

// main.js

import { add } from './math.js';

**4. Object-Oriented JavaScript**

Prototypal Inheritance: JavaScript is prototype-based, meaning that objects can directly inherit from other objects. Each object has a prototype, which it inherits properties from.

let animal = { eats: true };

let rabbit = Object.create(animal);

console.log(rabbit.eats); // true

**Class-based vs Prototypal Inheritance:** In ES6, JavaScript introduced the class syntax to make working with prototypal inheritance more structured. However, under the hood, JavaScript classes are still based on prototypes.

**Getters and Setters:** These are special methods used to access or set the properties of an object. Getters retrieve a property’s value, and setters modify the property’s value. They're often used for data encapsulation.

**5. Performance and Optimization**

**Debouncing and Throttling:**

* **Debouncing** is the practice of ensuring that a function is only executed after a certain amount of time has passed since it was last called (e.g., waiting for a user to stop typing before running a search query).
* **Throttling** ensures that a function is only executed at most once every specified interval, preventing excessive calls to resource-intensive functions, such as window resizing or scrolling.

**Lazy Loading**: Lazy loading is a design pattern that postpones the loading of resources (like images, scripts) until they are needed, improving the performance of an application, particularly for large websites.

**Memoization:** Memoization is a technique used to cache the results of expensive function calls and return the cached result when the same inputs occur again, significantly improving performance in some cases.

**6. Advanced Concepts**

**Higher-Order Functions:** A higher-order function is a function that either takes one or more functions as arguments or returns a function as its result. Functions like map(), reduce(), and filter() are higher-order functions.

**Currying**: Currying is a functional programming technique where a function is broken down into a series of functions that each take a single argument. This allows for partial application of arguments.

function add(a) {

return function(b) {

return a + b; }}

const add5 = add(5);

console.log(add5(10)); // 15

Functional Programming: This is a programming paradigm where functions are treated as first-class citizens. It emphasizes immutability, avoiding side effects, and using higher-order functions for composition and abstraction.

**7. Testing and Debugging**

* **Unit Testing:** Unit testing involves testing individual functions or components of a program to ensure they work as expected. Popular JavaScript testing frameworks include Jest, Mocha, and Jasmine.
* **Debugging**: Tools like Chrome DevTools allow you to step through your code, inspect variables, and track down issues. Techniques like console.log and breakpoints help diagnose problems in your code.

**8. Working with the DOM**

* **Event Delegation:** This is the practice of attaching event listeners to a parent element instead of each child element. This is especially useful for dynamically added elements, as the listener will apply to them as well.
* **DOM Manipulation:** JavaScript allows you to dynamically change the content and structure of a webpage by manipulating the DOM. You can modify elements using properties like innerHTML, textContent, and value.

**9. Frameworks and Libraries**

**React, Angular, Vue.js:** These are front-end libraries and frameworks used to build dynamic and responsive web applications. They differ in terms of design philosophy, structure, and the way they handle data and state management.

**State Management:** Frameworks like React use libraries like Redux or Context API for managing the application's state. In Vue.js, Vuex is used for similar purposes.

**10. Version Control**

**Git**: Git is a version control system used to track changes in code and collaborate with other developers. Branching allows you to work on separate features or bug fixes without affecting the main codebase, while merging brings those changes back together.

Merge Conflicts: These occur when changes to the same part of a file are made in different branches. Conflicts need to be resolved manually by reviewing the code and deciding which changes to keep.

**Basic Level**

**What is NgRx?**

**Answer:** NgRx is a state management library for Angular applications, based on the Redux pattern. It helps manage application state in a reactive way using a unidirectional data flow. It utilizes the store to centralize state management, making it easier to handle complex state changes in a predictable manner.

**What are the core concepts of NgRx?**

**Answer:** The core concepts of NgRx include:

Store: Centralized state container for the application.

Actions: Dispatched events that describe a change to the application state.

Reducers: Functions that handle state transitions based on dispatched actions

Selectors: Functions to query slices of state from the store.

Effects: Manage side effects (e.g., HTTP requests, local storage access) by listening to actions and dispatching new actions.

**What is the purpose of a Store in NgRx?**

**Answer:** The Store in NgRx holds the entire state of the application. It is a single source of truth for the application's state. The store is immutable, and updates are done through dispatched actions that trigger reducers to produce a new state. This makes the application state predictable and debuggable.

**What are Actions in NgRx?**

**Answer:** Actions are payloads of information that describe something that happened in the application. They are dispatched to the store, which causes a change in the state. Actions are typically plain objects with a type property and an optional payload.

**What are Reducers in NgRx?**

**Answer:** Reducers are pure functions that specify how the application state should change in response to a dispatched action. Reducers take the current state and an action as arguments, and return a new state based on the action type. Reducers do not mutate the state but return a new state object.

**What are Selectors in NgRx?**

**Answer:** Selectors are functions that allow you to query and retrieve slices of state from the store. They are used to optimize the process of accessing state by memoizing the results, ensuring that the component only re-renders when the relevant part of the state changes.

**What is the role of Effects in NgRx?**

**Answer:** Effects manage side effects (such as API calls, HTTP requests, or other async tasks) in NgRx. They listen for specific actions, perform some side-effect, and then dispatch new actions in response. Effects provide a way to separate side-effect logic from components and reducers.

**How do you perform state updates using NgRx?**

**Answer:** To update the state, you follow these steps:

Dispatch an Action: A component or service dispatches an action that describes the intended change. Action is processed by Reducer: The action is received by a reducer function, which creates a new state based on the current state and the action’s payload.

State is updated: The store updates the state with the new value returned by the reducer.

**Intermediate Level**

**How do you handle async operations like HTTP requests in NgRx?**

**Answer:** Async operations like HTTP requests are handled using Effects in NgRx. Effects listen for actions related to async processes, perform the HTTP request, and dispatch new actions based on the result (success or failure). For example:

Dispatch an action to trigger the HTTP request.

The effect catches that action, performs the HTTP request, and dispatches a success or failure action.

The reducer processes the success or failure actions and updates the state accordingly.

**What is the purpose of the StoreModule and EffectsModule in NgRx?**

**Answer:**

StoreModule: Configures the NgRx store in your application by importing StoreModule.forRoot() at the root of your app and StoreModule.forFeature() for feature modules. It establishes the state management structure.

EffectsModule: Configures and registers effects in the app. You import EffectsModule.forRoot() to register global effects and EffectsModule.forFeature() to register module-specific effects.

**What are the benefits of using NgRx over traditional services for state management in Angular**

**Answer:** NgRx provides a centralized, predictable state management system that allows you to:

Easily manage complex state and handle side effects.

Debug the application state using time-travel debugging tools like Redux DevTools.

Ensure consistency of state throughout the application with a single source of truth.

Handle side effects outside components, making components simpler and more focused.

**Can you explain the difference between the ngrx/store and ngrx/data libraries?**

**Answer:**

ngrx/store is the core NgRx library for state management, providing a reactive store, actions, reducers, selectors, and effects.

ngrx/data is built on top of ngrx/store and simplifies the management of entities. It automatically handles entity CRUD operations and provides utility functions for managing collections of data (e.g., pagination, caching).

**How does the ngrx/store module differ from Redux?**

**Answer:**

NgRx is designed specifically for Angular and integrates seamlessly with Angular's reactive programming paradigm (using RxJS and observables).

Redux is a more generic state management library that can be used with any JavaScript framework (not specific to Angular).

NgRx has built-in support for Angular’s dependency injection system and provides decorators to reduce boilerplate code.

**What are the advantages and disadvantages of using NgRx for state management in Angular?**

Answer:

* **Advantages:**

Predictable state management with a single source of truth.

Debugging support with Redux DevTools.

Separation of concerns (business logic in reducers, side effects in effects).

* **Disadvantages:**

Boilerplate code for actions, reducers, and effects.

Learning curve for developers unfamiliar with Redux or reactive programming.

May add overhead for small or simple applications.

**Advanced Level**

**How do you manage a complex state using NgRx?**

**Answer**: For complex state management, it's recommended to:

Break the state into feature modules and use StoreModule.forFeature() for each module.

Use entity adapters to normalize large or complex datasets into simpler, flat structures

Use selectors to extract data from nested structures and to ensure efficient re-rendering.

**How do you test NgRx actions, reducers, and effects?**

**Answer:**

Actions: Test actions by checking if they are correctly dispatched with the right type and payload.

Reducers: Use Jasmine or Jest to test reducers by dispatching actions and verifying that the state is updated correctly.

Effects: Test effects using EffectsTestingModule, simulating action dispatches and verifying if new actions are dispatched based on side effects.

**What are some performance optimizations you can apply when using NgRx?**

**Answer:**

Use selectors to efficiently select data from the store. Memoization ensures components only re-render when relevant state changes.

Normalize the state using entity adapters for managing large collections of data.

Avoid unnecessary re-renders by using take(1) or shareReplay() in observables to cache the result.

Use ngrx/store-devtools selectively in production.

**What are the potential pitfalls or challenges when using NgRx in large-scale applications?**

**Answer:** Challenges include:

The complexity of managing a large number of actions, reducers, and effects.

The learning curve, especially for developers unfamiliar with reactive programming.

Boilerplate code, which can make the development process cumbersome for smaller projects.

**What is the difference between StoreModule.forRoot() and StoreModule.forFeature()?**

**Answer:**

StoreModule.forRoot() is used to configure the root store and is called once in the main AppModule.

StoreModule.forFeature() is used to configure feature-specific slices of state within a specific module.

**Can you explain the concept of entity adapters in NgRx?**

**Answer:** EntityAdapter is a utility provided by NgRx to simplify managing collections of entities (e.g., users, products) in state. It normalizes the data into a flat structure, making it easier to perform CRUD operations and avoid unnecessary state updates.

**What are "action creators" in NgRx, and how do they help in simplifying action dispatching?**

**Answer:** Action creators are functions that return action objects. They help simplify the dispatch process by reducing the boilerplate code needed to create actions manually. Action creators are typically created using the createAction function from NgRx.

**Array Question code**

**Question 1: Group Passed/Failed Students**

**✅ Question 1: Group Passed/Failed Students**

const students = [

{ name: "Alice", marks: 85 },

{ name: "Bob", marks: 40 },

{ name: "Charlie", marks: 60 },

{ name: "David", marks: 55 },

{ name: "Eve", marks: 91 }

];

**// Output:**

// [{ passed: ["Alice", "Charlie", "Eve"], failed: ["Bob", "David"] }]

**// your code**

let res = [

{'passed': students.filter(p => p.marks >=60).map(m=> m.name)},

{'Failed': students.filter(p => p.marks < 60).map(m=> m.name)}

]

console.log(res);

**✅ Question 2: Extract Names with Specific Role**

const team = [

{ id: 1, name: "Anna", role: "dev" },

{ id: 2, name: "Ben", role: "designer" },

{ id: 3, name: "Cara", role: "dev" },

{ id: 4, name: "Dan", role: "manager" }

];

// Output:

// ["Anna", "Cara"]

**// your code**

let teams = team.filter(p => p.role == 'dev').map(m=> m.name)

**✅ Question 3: Count Votes**

const votes = ["yes", "no", "yes", "yes", "no", "yes", "no"];

// Output: // { yes: 4, no: 3 }

**// your code**

const result = { yes: 0, no: 0 };

votes.forEach(vote => result[vote]++);

console.log(result); // { yes: 4, no: 3 }

**✅ Question 4: Find First Active User**

const users = [

{ id: 1, name: "Alex", active: false },

{ id: 2, name: "Brian", active: false },

{ id: 3, name: "Clara", active: true },

{ id: 4, name: "Diana", active: true }

];

// Output: "Clara"

**// your code**

// let res = users.find(e=>e.active == true)?.name;

// let res2 = users.filter(e=>e.active == true).map(e=>e.name)[0];

**✅ Question 5: Group Orders by Status**

const orders = [

{ id: 101, status: "delivered" },

{ id: 102, status: "pending" },

{ id: 103, status: "delivered" },

{ id: 104, status: "cancelled" },

{ id: 105, status: "pending" }

];

**// Output:**

// {

// delivered: [101, 103],

// pending: [102, 105],

// cancelled: [104]

// }

**// your code**

let **res** ={

delivered: orders.filter(e=> e.status == 'delivered').map(m=>m.id),

pending: orders.filter(e=> e.status == 'pending').map(m=>m.id),

cancelled: orders.filter(e=> e.status == 'cancelled').map(m=>m.id)

}

You got it! Here's the full list of **solutions for Questions 1–20**, neatly organized.

## ✅ **Solutions for Q1–Q5**

### **Q1: Group Passed/Failed Students**

const students = [

{ name: "Alice", marks: 85 },

{ name: "Bob", marks: 40 },

{ name: "Charlie", marks: 60 },

{ name: "David", marks: 55 },

{ name: "Eve", marks: 91 }

];

const result = [{

passed: students.filter(s => s.marks >= 60).map(s => s.name),

failed: students.filter(s => s.marks < 60).map(s => s.name)

}];

console.log(result);

### **Q2: Extract Names with Specific Role**

const team = [

{ id: 1, name: "Anna", role: "dev" },

{ id: 2, name: "Ben", role: "designer" },

{ id: 3, name: "Cara", role: "dev" },

{ id: 4, name: "Dan", role: "manager" }

];

const devs = team.filter(t => t.role === "dev").map(t => t.name);

console.log(devs);

### **Q3: Count Votes**

const votes = ["yes", "no", "yes", "yes", "no", "yes", "no"];

const result = votes.reduce((acc, vote) => {

acc[vote] = (acc[vote] || 0) + 1;

return acc;

}, {});

console.log(result);

### **Q4: Find First Active User**

const users = [

{ id: 1, name: "Alex", active: false },

{ id: 2, name: "Brian", active: false },

{ id: 3, name: "Clara", active: true },

{ id: 4, name: "Diana", active: true }

];

const name = users.find(u => u.active)?.name;

console.log(name);

### **Q5: Group Orders by Status**

const orders = [

{ id: 101, status: "delivered" },

{ id: 102, status: "pending" },

{ id: 103, status: "delivered" },

{ id: 104, status: "cancelled" },

{ id: 105, status: "pending" }

];

const grouped = orders.reduce((acc, order) => {

acc[order.status] = acc[order.status] || [];

acc[order.status].push(order.id);

return acc;

}, {});

console.log(grouped);

## ✅ **Solutions for Q6–Q12**

### **Q6: Group by Minor/Adult**

const people = [

{ name: "Tom", age: 12 },

{ name: "Jerry", age: 18 },

{ name: "Anna", age: 15 },

{ name: "Mike", age: 22 }

];

const result = {

minor: people.filter(p => p.age < 18).map(p => p.name),

adult: people.filter(p => p.age >= 18).map(p => p.name)

};

console.log(result);

### **Q7: Extract All Unique Tags**

const posts = [

{ title: "Post 1", tags: ["react", "js"] },

{ title: "Post 2", tags: ["node", "js"] },

{ title: "Post 3", tags: ["html", "css", "react"] }

];

const tags = [...new Set(posts.flatMap(p => p.tags))];

console.log(tags);

### **Q8: Convert Array to ID-Based Object**

const users = [

{ id: 1, name: "Sam" },

{ id: 2, name: "Alex" },

{ id: 3, name: "John" }

];

const result = users.reduce((acc, user) => {

acc[user.id] = user.name;

return acc;

}, {});

console.log(result);

### **Q9: Find Most Frequent Element**

const items = ["a", "b", "a", "c", "a", "b", "b"];

const freq = items.reduce((acc, item) => {

acc[item] = (acc[item] || 0) + 1;

return acc;

}, {});

const mostFrequent = Object.keys(freq).reduce((a, b) => freq[a] > freq[b] ? a : b);

console.log(mostFrequent);

### **Q10: Total Price with Discounts**

const cart = [

{ item: "Shoes", price: 100, discount: 10 },

{ item: "Shirt", price: 60, discount: 5 },

{ item: "Hat", price: 30, discount: 0 }

];

const total = cart.reduce((sum, item) => sum + (item.price - item.discount), 0);

console.log(total);

### **Q11: Find User by Email (Case Insensitive)**

const users = [

{ id: 1, email: "john@example.com" },

{ id: 2, email: "alice@example.com" },

{ id: 3, email: "BOB@EXAMPLE.COM" }

];

const search = "bob@example.com";

const found = users.find(u => u.email.toLowerCase() === search.toLowerCase());

console.log(found);

### **Q12: Flatten Array of Arrays**

const input = [[1, 2], [3, 4], [5]];

const result = input.flat();

console.log(result);

## ✅ **Solutions for Q13–Q20**

### **Q13: Find the Oldest Person**

const people = [

{ name: "Ava", age: 25 },

{ name: "Ben", age: 38 },

{ name: "Cara", age: 31 }

];

const oldest = people.reduce((max, p) => p.age > max.age ? p : max);

console.log(oldest);

### **Q14: Get Emails of Active Users**

const users = [

{ name: "John", email: "john@example.com", active: true },

{ name: "Jill", email: "jill@example.com", active: false },

{ name: "Jake", email: "jake@example.com", active: true }

];

const emails = users.filter(u => u.active).map(u => u.email);

console.log(emails);

### **Q15: Sum Salaries by Department**

const employees = [

{ name: "A", salary: 1000, dept: "HR" },

{ name: "B", salary: 1500, dept: "IT" },

{ name: "C", salary: 1200, dept: "HR" },

{ name: "D", salary: 2500, dept: "IT" }

];

const result = employees.reduce((acc, emp) => {

acc[emp.dept] = (acc[emp.dept] || 0) + emp.salary;

return acc;

}, {});

console.log(result);

### **Q16: Get Top 3 Scores**

const scores = [45, 87, 32, 99, 66, 75];

const top3 = scores.sort((a, b) => b - a).slice(0, 3);

console.log(top3);

### **Q17: Check All Verified**

const users = [

{ name: "Alice", verified: true },

{ name: "Bob", verified: true },

{ name: "Charlie", verified: false }

];

**const allVerified = users.every(u => u.verified);**

**console.log(allVerified);**

### **Q18: Create Sentence from Words**

const words = ["JavaScript", "is", "fun"];

const sentence = words.join(" ");

console.log(sentence);

### **Q19: Remove Duplicates from Array**

const arr = [1, 2, 2, 3, 4, 4, 5];

const unique = [...new Set(arr)];

console.log(unique);

### **Q20: Capitalize All Names**

const names = ["john", "ALICE", "mArK"];

const capitalized = names.map(name =>

name.charAt(0).toUpperCase() + name.slice(1).toLowerCase()

);

**1. How do you find the length of a string?**

**Example:**  
"Hello World".length

**Output:**  
11

**2. How do you convert a string to uppercase?Example:**  
"hello".toUpperCase()

**Output:**  
"HELLO"

**3. How do you check if a string includes a word?**

**Example:**  
"JavaScript is cool".includes("cool")

**Output:**  
true

**4. How do you find the position of a substring?**

**Example:**  
"Frontend Developer".indexOf("Developer")

**Output:**  
9

**5. How do you extract part of a string?**

**Example:**  
"JavaScript".slice(4, 10)

**Output:**  
"Script"

**6. How do you remove whitespace from both ends of a string?**

**Example:**  
" trimmed ".trim()

**Output:**  
"trimmed"

**7. How do you replace a word in a string?**

**Example:**  
"I like cats".replace("cats", "dogs")

**Output:**  
"I like dogs"

**8. How do you split a string into an array?**

**Example:**  
"a,b,c".split(",")

**Output:**  
["a", "b", "c"]

**9. Is "Hello" equal to "hello"?**

**Example:**  
"Hello" === "hello"

**Output:**  
false

**10. Check if a string starts with a certain word**

**Example:**  
"OpenAI GPT".startsWith("Open")

**Output:**  
true

Awesome! Here are some **advanced JavaScript string interview examples** — again, just **example inputs and outputs**, no code, nice and clean:

**11. Check if a string is a palindrome**

(A palindrome reads the same backward and forward)

**Example:**  
Input: "racecar"  
Output: true

**Example:**

let poli = 'racecarA'

function polidram(str){

let out = poli.split('').reverse().join('')

return str === out

}  
Input: "hello"  
Output: false

**12. Find the first non-repeating character**

**Example:**

function firstNonRepeatingChar(str) {

for (let char of str) {

if (str.indexOf(char) === str.lastIndexOf(char)) {

return char;

}

}

return '';

}

Input: "aabbcddfe"  
Output: "c"

**13. Remove duplicate characters from a string**

**Example:**

function duplic(str) {

let resu = ''

for (let char of str){

if(!resu.includes(char)){

resu +=char

}

}

return resu;

}  
Input: "banana"  
Output: "ban"

**14. Count number of vowels in a string**

**Example:**

function vowels(str){

vowels = 'aeiousAEIOUS'

count = 0

for (let char of str){

if(vowels.includes(char)){

count ++;

}

}

return count

}  
Input: "ChatGPT is amazing"  
Output: 6  
*(a, A, i, a, i, and another a)*

**15. Check if two strings are anagrams**

(Anagrams = same letters in different order)

**Example:**  
Input: "listen" and "silent"  
Output: true

**Example:**  
Input: "hello" and "world"  
Output: false

**16. Capitalize the first letter of each word**

**Example:**

let cp = 'hello world from chatgpt'

let cap = cp.split(' ').map(e=> e.charAt(0).toUpperCase() + e.slice(1)).join(' ');  
Input: "hello world from chatgpt"  
Output: "Hello World From Chatgpt"

**17. Reverse the words in a sentence**

**Example:**

let rev = 'JavaScript is fun'

let revR = rev.split(' ').reverse().join(' ')  
Input: "JavaScript is fun"  
Output: "fun is JavaScript

### 🎯 **Mock Interview: Angular (Infosys Style)**

**~~Want me to simulate a mock interview with Angular questions for Infosys-style interviews? Or focus on a specific area like forms, services, routing, or performance?~~**

z

#### 🧠 Question 1:

**"Can you explain the difference between Template-Driven and Reactive Forms in Angular? When would you use one over the other?"**

**Answer**

**Template-driven forms** use Angular directives like [(ngModel)] for two-way data binding and are generally suitable for simpler forms. The form logic is mostly handled in the HTML template. They're easier to use for small-scale or simple forms where you don’t need complex validations.

**Reactive forms**, on the other hand, are built using classes like FormControl, FormGroup, and FormBuilder in the component class. They provide more **control, scalability, and testability**, making them ideal for larger, dynamic, or complex forms where validation and control logic need to be programmatic.

You could optionally mention:

* Template-driven forms rely on **FormsModule**
* Reactive forms use **ReactiveFormsModule**

**🔄 Question 2:**

**👉 How does Angular handle component communication? Name at least three ways components can share data."**

Angular offers several ways for components to communicate:

1. **@Input()** – for parent-to-child communication, passing data from the parent into a child component.
2. **@Output() with EventEmitter** – for child-to-parent communication, allowing a child to emit events to its parent.
3. **Shared Service with Observables or BehaviorSubject** – a common service acts as a mediator for communication between unrelated (sibling or distant) components.
4. **Signals (Angular 17+)** – Signals can be used to create reactive data that components can subscribe to, and they help with fine-grained reactivity.
5. **NgRx or RxJS State Management** (optional advanced) – for large-scale apps, especially when multiple components need to react to shared data.

**🔄 Question 3:**

**What are the main building blocks of an Angular application?**

The main building blocks of an Angular application include:

1. **Modules (NgModules)** – These are containers for different parts of the app, like components and services. Every Angular app has at least a root module (AppModule).
2. **Components** – These are the core building blocks of UI in Angular. Each component contains a TypeScript class, an HTML template, and optional CSS styles.
3. **Templates** – Define the HTML structure and how Angular renders it using directives and data binding.
4. **Directives** – Instructions in the DOM. There are structural directives (like \*ngIf, \*ngFor) and attribute directives (like ngClass, ngStyle).
5. **Services and Dependency Injection** – Services contain business logic and reusable code. DI makes it easy to inject them where needed.
6. **Routing** – The Angular Router enables navigation between views/components.
7. **Pipes** – Used to transform data in templates (e.g., date, currency, or custom pipes).
8. **Forms** – Angular supports both Template-driven and Reactive forms for handling user input and validation.

**🔄 Question 4:**

**What is the difference between a component and a directive in Angular?  
Your turn!**

A **component** is a special type of directive that includes a template. It controls a section of the UI and is the main building block of Angular applications. Each component has:

* A **TypeScript class** (logic)
* An **HTML template**
* Optional **styles** and **metadata**

A **directive**, on the other hand, is used to add behavior to existing DOM elements. There are two types:

1. **Attribute Directives** – Modify the appearance or behavior of an element, e.g., ngClass, ngStyle, ngModel.
2. **Structural Directives** – Change the structure of the DOM, e.g., \*ngIf, \*ngFor.

**🔄 Question 5:**

**What is change detection in Angular, and how can you optimize it for performance?**

Change detection in Angular is the process where Angular checks if data in your app has changed, and then updates the DOM to match. It happens automatically whenever events like user input or HTTP responses occur.

To **optimize change detection for better performance**, here are some simple techniques:

1. **Use OnPush strategy** – This tells Angular to check a component only when its inputs change, not every time anything changes in the app.
2. **Use trackBy in \*ngFor** – This helps Angular track items in a list by ID, so it doesn’t re-render the whole list.
3. **Avoid calling functions in the template** – Call them in the component and store the result, to avoid running them every time change detection happens.
4. **Detach change detection when not needed** – You can manually stop Angular from checking certain components if they don’t need updates.
5. **Debounce or throttle inputs** – Especially for search bars or typing, use RxJS to limit how often Angular checks for changes.

Angular Guards and How They Work for Route Protection.

**🔄 Question 5:**

### **What are Angular Guards and How Do They Work for Route Protection?**

**Answer:**

In Angular, **guards** are used to control access to routes. They help decide if a user can visit a specific page (route) or not.

### Types of Guards:

1. **CanActivate**: Checks if a user can enter a route. For example, checking if they are logged in before allowing access.
2. **CanActivateChild**: Protects child routes under a parent route.
3. **CanDeactivate**: Prevents leaving a page (e.g., unsaved form warning).
4. **Resolve**: Fetches data before entering a route.
5. **CanLoad**: Prevents loading a module (used for lazy-loaded modules).

**🔄 Question 6**

### How Do Guards Work for Route Protection?

* Guards are added to routes in the Angular router configuration.
* When a user tries to navigate to a route, the guard checks if they can access it.
  + If the guard returns true, they can enter the route.
  + If the guard returns false, they are blocked from accessing the route.

**🔄 Question 7:**

**What is the purpose of the *ngOnInit()* lifecycle hook in Angular?**

**ngOnInit** is indeed called after the constructor but **only once** during the lifecycle of a component.

While ngOnInit() is called when a component is created, it **won’t be triggered by jus t any route change** unless a new component is instantiated (such as with lazy loading or route reuse strategies).

**Key Points**:

* It runs after the **constructor** and **before the view is displayed**.
* It’s a good place to call methods or make API requests.
* **ngOnInit** will not be called on every route change; only when the component is created, unless Angular re-creates the component (e.g., in a route change or lazy loading).

**🔄 Question 8:**  
**How would you implement custom validation in Reactive Forms?**

In Reactive Forms, you can implement custom validation by creating a validator function that returns either null (if valid) or an object (if invalid).

For example, to validate a mobile number using a custom regex:

function mobileValidator(control: AbstractControl): ValidationErrors | null {

const mobileRegex = /^[6-9]\d{9}$/;

const valid = mobileRegex.test(control.value);

return valid ? null : { invalidMobile: true };

}

**🔄 Question 9:**

**👉 How would you implement lazy loading in Angular for a feature module?**

Lazy loading in Angular helps improve performance by loading feature modules **only when they're needed** (e.g., on route navigation).

For **module-based lazy loading**:

1. Create a feature module (e.g., AdminModule) and its routing module.
2. In app-routing.module.ts, use:

**✅ Next Question – Real-World Scenario:**

👉 *You're working on a dashboard with multiple components like charts, tables, and filters. The performance is starting to drop. What steps would you take to optimize your Angular application’s performance?*

 **Modularize the UI** using smaller, reusable components (using selectors). This makes the dashboard more maintainable and each component loads only what it needs.

 Use \*ngFor with a **trackBy function** to avoid unnecessary DOM re-renders when the data updates.

 Apply **ChangeDetectionStrategy.OnPush** to components that don't rely on deep object mutation, so Angular only checks them when inputs change.

 **Lazy load** heavy modules (like charts or reports) so they only load when needed.

 Optimize large data tables using **pagination**, **virtual scrolling**, or **lazy data loading**.

 Cache or debounce API calls when filters or inputs are used.

 For micro frontend architecture, I might split parts of the dashboard into independent apps or modules loaded via routing or Webpack Module Federation (if needed at enterprise scale).

👉 How would you share data between two unrelated components (not parent-child) in Angular?

To share data between two unrelated components, I would create a **shared service**. The service would expose an **Observable** (often via a BehaviorSubject or Subject) to allow components to subscribe to it and get real-time updates when the data changes.

**👉 What’s the difference between *ngIf* and *ngFor* in Angular, and how does each affect performance?**

You’ve got the basic idea — **ngIf** and **ngFor** are used for different purposes:

* **ngIf** is for conditionally rendering elements.
* **ngFor** is for iterating over arrays or collections.

However, we can elaborate more on the **performance implications** and more detailed behavior of both directives:

1. **ngIf**:
   * Adds or removes the DOM element based on the condition.
   * Can **create** or **destroy** the element each time the condition changes.
   * **Performance impact**: If ngIf toggles frequently, it can cause Angular to create and destroy elements, which might not be efficient in some cases.
2. **ngFor**:
   * Iterates over an array and renders an element for each item.
   * Uses **trackBy** to improve performance by avoiding unnecessary re-renders of unchanged elements.
   * **Performance impact**: Without trackBy, Angular re-renders every item in the list whenever the array changes. Using trackBy optimizes this by tracking items uniquely and preventing unnecessary re-renders.

**👉 Question 9: How do you handle error handling in Angular applications, especially when dealing with HTTP requests?**

**In Angular, we handle errors in HTTP requests by using the HttpClient module along with RxJS operators like catchError. Here's the approach:**

1. **Make HTTP Request**: Using Angular’s HttpClient service.
2. **Error Handling**: Use the catchError operator in the Observable chain to catch any errors that occur during the HTTP request.
3. **Handling Error Logic**: We can either show an error message to the user, log the error, or retry the request based on the error type.

👉 **Question 10:** Can you explain the concept of ***"Dependency Injection"*** in Angular? How does it work and why is it useful?

### **What is Dependency Injection (DI)?**

**Dependency Injection** is a design pattern used to implement **Inversion of Control (IoC)**, where the control of creating and managing objects is transferred to an external system (in this case, Angular).

In simpler terms, **DI** allows you to inject **dependencies** (services, other components, etc.) into a class rather than hard-coding them within that class. This makes the class more flexible, testable, and decoupled from its dependencies.

### **How DI Works in Angular?**

1. **Services** in Angular are typically the dependencies that are injected into components, directives, or other services.
2. Angular uses an **Injector** (a built-in service) to manage these dependencies. The injector looks for the dependencies that need to be provided to a class (like a component or service) and injects them.

**Steps:**

* You define a **service** or dependency.
* You **inject** this service into the class (typically a component or another service) that needs it.
* Angular’s **Injector** takes care of providing the correct instance of the service.