**Angular**

1. **What is Angular?**
   * Angular is a TypeScript-based open-source web application framework developed by Google. It is used for building dynamic single-page applications (SPAs) with features such as two-way data binding, dependency injection, and a modular architecture.
2. **What are the key features of Angular?**
   * + Component-based architecture
     + Two-way data binding
     + Dependency injection
     + Directives
     + Angular CLI
     + RxJS for reactive programming
     + Modular architecture with NgModules
3. **Explain the concept of data binding in Angular.**
   * Data binding is the synchronization of data between the model and the view. Angular supports several types of data binding:
     + **Interpolation**: {{ value }}
     + **Property binding**: [property]="value"
     + **Event binding**: (event)="handler()"
     + **Two-way data binding**: [(ngModel)]="value"
4. **What are Angular components and how do you create one?**
   * Components are the building blocks of an Angular application. They encapsulate the HTML, CSS, and TypeScript code for a part of the UI. To create a component, you use the Angular CLI command ng generate component component-name or ng g c component-name.
5. **What is the purpose of Angular modules (NgModules)?**
   * NgModules are a way to organize an Angular application into cohesive blocks of functionality. Each module can contain components, directives, pipes, and services. Modules help in lazy loading, separation of concerns, and dependency management.
6. **How does Angular handle dependency injection?**
   * Angular uses a hierarchical dependency injection system where dependencies are provided by injectors. Services and other dependencies are registered in Angular’s injector tree, and Angular injects them where needed. You can provide dependencies at the root level or at the component/module level.
7. **What is the difference between @Input() and @Output() in Angular?**
   * @Input() is used to pass data from a parent component to a child component.
   * @Output() is used to emit events from a child component to a parent component.

// Parent component

<app-child [inputProperty]="parentData" (outputEvent)="handleEvent($event)"></app-child>

// Child component

@Input() inputProperty: any;

@Output() outputEvent = new EventEmitter<any>();

1. **Explain Angular’s lifecycle hooks.**
   * Lifecycle hooks are methods that allow you to tap into specific points in a component’s lifecycle. Common lifecycle hooks include:
     + ngOnInit(): Called once after the first ngOnChanges()
     + ngOnChanges(): Called when an input property changes
     + ngOnDestroy(): Called just before the component is destroyed
     + ngAfterViewInit(): Called after Angular initializes the component’s views and child views
2. **What is Angular’s Change Detection mechanism?**
   * Angular’s Change Detection mechanism keeps the view in sync with the model. Angular uses a strategy called “Zone.js” to detect changes. When a change occurs, Angular runs a change detection cycle to update the DOM. There are two change detection strategies:
     + **Default**: Checks all components.
     + **OnPush**: Only checks components when the input properties change or when an event occurs in the component.
3. **What are Angular Directives and how do they differ from Components?**
   * Directives are classes that add behavior to elements in your application. There are three types:
     + **Structural Directives**: Change the DOM layout (e.g., \*ngIf, \*ngFor)
     + **Attribute Directives**: Change the appearance or behavior of an element (e.g., ngClass, ngStyle)
     + **Component Directives**: These are directives with a template (essentially a component)
4. **How do you implement lazy loading in Angular?**
   * Lazy loading is used to load modules only when they are needed. You can implement it by setting up lazy-loaded routes in your routing module.

const routes: Routes = [

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

];

1. **What is the purpose of Angular’s Router and how do you configure it?**
   * Angular Router enables navigation between views or pages in a single-page application. You configure it using the RouterModule.forRoot(routes) method in the root module or RouterModule.forChild(routes) in feature modules.

const routes: Routes = [

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

{ path: 'about', component: AboutComponent }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule { }

1. **How do you handle HTTP requests in Angular?**
   * Angular provides the HttpClient module to handle HTTP requests. You use HttpClient to perform GET, POST, PUT, DELETE operations, and more.

@Injectable({

providedIn: 'root'

})

export class DataService {

constructor(private http: HttpClient) { }

getData() {

return this.http.get('https://api.example.com/data');

}

}

1. **What is Angular’s Reactive Forms and how does it differ from Template-driven Forms?**
   * Reactive Forms are more flexible and scalable than Template-driven Forms. They are built using the FormControl, FormGroup, and FormArray classes. Reactive Forms provide more control and are suitable for complex scenarios.

@Component({

selector: 'app-reactive-form',

templateUrl: './reactive-form.component.html'

})

export class ReactiveFormComponent implements OnInit {

form: FormGroup;

constructor(private fb: FormBuilder) { }

ngOnInit() {

this.form = this.fb.group({

name: ['', Validators.required],

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

});

}

onSubmit() {

console.log(this.form.value);

}

}

1. **Explain the concept of Angular Services and how they are used in dependency injection.**
   * Services are classes that provide specific functionality and can be injected into components or other services using Angular’s dependency injection system. They are used to share data and logic across the application. Services are typically provided in the root injector or specific module injectors.

@Injectable({

providedIn: 'root'

})

export class UserService {

constructor(private http: HttpClient) { }

getUser(id: number) {

return this.http.get(`https://api.example.com/users/${id}`);

}

}

1. **What is NgRx and how does it fit into Angular applications?**
   * NgRx is a library for managing state in Angular applications using the Redux pattern. It provides a centralized store for managing application state and facilitates state changes through actions and reducers. NgRx helps maintain predictable state management and side effects in complex applications.

// Action

export const loadItems = createAction('[Items] Load Items');

// Reducer

export const itemsReducer = createReducer(

initialState,

on(loadItems, state => ({ ...state, loading: true }))

);

// Selector

export const selectItems = createSelector(

(state: AppState) => state.items,

items => items

);

1. **How do you optimize Angular application performance?**
   * Optimization techniques include:
     + **Lazy loading** modules and routes
     + **Change detection strategy**: Use OnPush where applicable
     + **TrackBy** functions in \*ngFor to reduce DOM manipulations
     + **AOT Compilation**: Use Ahead-of-Time compilation for faster rendering
     + **Tree shaking**: Remove unused code during build
     + **Code splitting**: Break the application into smaller bundles
     + **Caching** and **debouncing** HTTP requests
2. **What is Angular Universal and why would you use it?**
   * Angular Universal is a server-side rendering (SSR) tool for Angular applications. It allows rendering of Angular applications on the server, which improves performance and SEO. It generates static HTML on the server and sends it to the client, where Angular takes over for dynamic interactions.
3. **How do you implement internationalization (i18n) in Angular?**
   * Angular provides support for internationalization using the @angular/localize package. You use the Angular CLI to extract translation strings, translate them, and then build the application for different locales.

ng extract-i18n

import { LOCALE\_ID, NgModule } from '@angular/core';

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

import localeFr from '@angular/common/locales/fr';

registerLocaleData(localeFr);

@NgModule({

providers: [{ provide: LOCALE\_ID, useValue: 'fr' }]

})

export class AppModule { }

1. **What is Angular Ivy and what are its benefits?**
   * Angular Ivy is the new rendering engine introduced in Angular 9. It provides improved build times, smaller bundle sizes, and better debugging capabilities. Ivy’s benefits include:
     + Faster compilation
     + Improved tree-shaking
     + Better runtime performance
     + More efficient code generation
2. **What is Angular’s Dependency Injection (DI) system and how does it work?**

* Angular’s DI system is a design pattern used to implement IoC (Inversion of Control). It allows you to inject dependencies (services, components, etc.) into a class without manually creating instances. Angular creates and manages these dependencies using providers, which are defined in modules or components. Services are typically provided in the providers array of an NgModule or component, and Angular’s injector system resolves them when needed.

@Injectable({

providedIn: 'root'

})

export class MyService {

constructor(private http: HttpClient) {}

}

1. **What is Angular's Ahead-of-Time (AOT) compilation and how does it differ from Just-in-Time (JIT) compilation?**
   * **AOT Compilation**: Compiles Angular HTML and TypeScript code into JavaScript code during the build process before the browser downloads and runs the code. This results in faster rendering and fewer runtime errors. It also helps in smaller bundle sizes and better security.
   * **JIT Compilation**: Compiles the code in the browser at runtime. This is useful during development for faster build times but results in larger bundle sizes and slower rendering in production.

ng build --prod --aot

1. **How does Angular handle route guards and what are their types?**

* Route guards are used to control navigation in Angular applications. They can prevent unauthorized users from accessing routes or redirect users based on certain conditions. There are several types of route guards:
  + **CanActivate**: Determines if a route can be activated.
  + **CanDeactivate**: Determines if a route can be deactivated.
  + **Resolve**: Fetches data before a route is activated.
  + **CanLoad**: Determines if a module can be loaded lazily.

@Injectable({

providedIn: 'root'

})

export class AuthGuard implements CanActivate {

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

canActivate(

route: ActivatedRouteSnapshot,

state: RouterStateSnapshot

): boolean {

if (this.authService.isAuthenticated()) {

return true;

} else {

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

return false;

}

}

}

1. **Explain the concept of ngZone and its role in Angular applications.**

* NgZone is a service that provides the ability to execute code inside or outside of Angular’s zone. It helps manage change detection and allows you to run code in the Angular zone (which triggers change detection) or outside of it (which does not). This is useful for optimizing performance when dealing with long-running tasks or third-party libraries.

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

constructor(private ngZone: NgZone) {}

runOutsideAngular() {

this.ngZone.runOutsideAngular(() => {

// Code running outside Angular's zone

setTimeout(() => {

// Code running after timeout

this.ngZone.run(() => {

// Code running back inside Angular's zone

});

}, 1000);

});

}

1. **What is Angular’s Change Detection Strategy and how does OnPush work?**

* Angular’s Change Detection Strategy determines how and when Angular checks for changes in the application’s data. The two main strategies are:
  + **Default**: Angular checks all components and their child components.
  + **OnPush**: Angular only checks components when their input properties change, or an event occurs within the component. This can improve performance by reducing the number of checks Angular performs.

@Component({

selector: 'app-example',

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

changeDetection: ChangeDetectionStrategy.OnPush

})

export class ExampleComponent {

@Input() data: any;

}

1. **How do you implement custom Angular decorators and provide an example?**

* Custom decorators in Angular can be used to add metadata to classes or properties. You create a custom decorator function and use it to annotate your classes or properties.

function CustomDecorator() {

return function (target: any, key: string) {

console.log(`Custom decorator applied to ${key}`);

};

}

@Component({

selector: 'app-example',

templateUrl: './example.component.html'

})

export class ExampleComponent {

@CustomDecorator()

myProperty: string;

}

1. **What are Angular Elements and how do you use them to create custom web components?**

* Angular Elements allows you to create custom web components from Angular components. You use Angular Elements to package Angular components as native web components, which can be used in any web application.

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

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

import { createCustomElement } from '@angular/elements';

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

@NgModule({

declarations: [AppComponent],

imports: [BrowserModule],

entryComponents: [AppComponent]

})

export class AppModule {

constructor(private injector: Injector) {

const element = createCustomElement(AppComponent, { injector });

customElements.define('app-element', element);

}

ngDoBootstrap() {}

}

1. **What is the purpose of Angular’s ng-template and how does it differ from a regular HTML template?**

* ng-template is a directive that allows you to define a template that is not rendered immediately. It is used for deferred rendering and can be conditionally included in the DOM using structural directives like \*ngIf and \*ngFor. Unlike regular HTML templates, ng-template content is not part of the DOM until it is used.

html

<ng-template #loading>

<p>Loading...</p>

</ng-template>

<ng-container \*ngIf="isLoading; else content">

<ng-container \*ngTemplateOutlet="loading"></ng-container>

</ng-container>

<ng-template #content>

<p>Data loaded successfully!</p>

</ng-template>

1. **How do you implement custom pipes in Angular?**

* Custom pipes are used to transform data in Angular templates. You create a custom pipe by implementing the PipeTransform interface and decorating the class with the @Pipe decorator.

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

@Pipe({

name: 'reverse'

})

export class ReversePipe implements PipeTransform {

transform(value: string): string {

return value.split('').reverse().join('');

}

}

html

<p>{{ 'hello' | reverse }}</p> <!-- Output: 'olleh' -->

1. **What are Angular’s Router Events and how can you use them?**

* Angular’s Router Events allow you to listen to various events in the routing lifecycle, such as navigation start, navigation end, and route activation. You use the Router service to subscribe to these events and handle them accordingly.

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

constructor(private router: Router) {

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

if (event instanceof NavigationStart) {

console.log('Navigation started');

}

if (event instanceof NavigationEnd) {

console.log('Navigation ended');

}

});

}

1. **How do you use Angular’s HttpInterceptor to modify HTTP requests and responses?**

* HttpInterceptor allows you to intercept and modify HTTP requests and responses. You create an interceptor by implementing the HttpInterceptor interface and providing it in the application module.

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 clonedReq = req.clone({

headers: req.headers.set('Authorization', 'Bearer your-token')

});

return next.handle(clonedReq);

}

}

@NgModule({

providers: [

{ provide: HTTP\_INTERCEPTORS, useClass: AuthInterceptor, multi: true }

]

})

export class AppModule {}

1. **What is Angular’s ChangeDetectorRef and how can you use it to manually trigger change detection?**

* ChangeDetectorRef is a service that provides methods to control the change detection mechanism manually. It is useful when you need to trigger change detection outside the Angular zone or for performance optimizations.

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

constructor(private cdr: ChangeDetectorRef) {}

someMethod() {

// Manually trigger change detection

this.cdr.detectChanges();

}

1. **What is Angular’s Renderer2 and how do you use it?**

* Renderer2 is a service that provides an abstraction for manipulating the DOM in a platform-independent way. It helps ensure that the DOM operations are safe and consistent across different platforms.

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

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

changeBackgroundColor() {

this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', 'blue');

}

1. **What are Angular’s Validators and how do you create custom validators?**

* Validators are used to perform form validation. Angular provides built-in validators such as required, minLength, maxLength, and pattern. You can also create custom validators to perform more complex validation logic.

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

export function forbiddenNameValidator(forbiddenName: string): ValidatorFn {

return (control: AbstractControl): { [key: string]: any } | null => {

const forbidden = control.value.indexOf(forbiddenName) !== -1;

return forbidden ? { 'forbiddenName': { value: control.value } } : null;

};

}

1. **How do you handle Angular module dependencies and circular dependencies?**

* To handle module dependencies, you should organize your application into feature modules and use Angular’s dependency injection to manage them. Circular dependencies can occur when two or more modules depend on each other. To avoid circular dependencies, ensure that:
  + Services and shared components are placed in shared or core modules.
  + Use dependency inversion and interface-based programming to decouple modules.

@NgModule({

imports: [CommonModule],

declarations: [SharedComponent],

exports: [SharedComponent]

})

export class SharedModule {}

1. **What is the role of @NgModule in Angular applications?**

* The @NgModule decorator defines an Angular module. It provides metadata about the module, including declarations (components, directives, pipes), imports (other modules), exports (components, directives, pipes available to other modules), providers (services), and bootstrap (root component). @NgModule helps Angular organize the application into cohesive blocks of functionality.

@NgModule({

declarations: [AppComponent, OtherComponent],

imports: [BrowserModule, FormsModule],

providers: [AppService],

bootstrap: [AppComponent]

})

export class AppModule { }

1. **What is the difference between ngOnInit and constructor in Angular components?**
   * **Constructor**: It is a TypeScript feature and is used for initializing class members and injecting dependencies. It is called before the component is fully initialized.
   * **ngOnInit**: It is an Angular lifecycle hook that is called after Angular has initialized all data-bound properties of a component. It is a good place to perform additional initialization tasks that require the component’s inputs to be set.

constructor(private service: MyService) {}

ngOnInit() {

this.service.loadData().subscribe(data => this.data = data);

}

1. **Explain the concept of Angular's Content Projection.**

* Content Projection allows you to insert content into a component from outside the component's template. Angular provides the <ng-content> directive to achieve this. This technique is useful for creating reusable and configurable components.

<!-- parent.component.html -->

<app-child>

<p>Projected Content</p>

</app-child>

<!-- child.component.html -->

<ng-content></ng-content>

1. **What are Angular Decorators and how do they work?**

* Angular Decorators are functions that modify or enhance classes, properties, methods, or parameters. They are used to define Angular components, modules, directives, and services. Decorators are prefixed with @ and are applied to classes or class members to add metadata.

@Component({

selector: 'app-example',

template: '<h1>Hello</h1>'

})

export class ExampleComponent {}

1. **What are Angular Pipes and how do they differ from filters in AngularJS?**

* **Pipes**: They are used to transform data in templates. Pipes take an input value and return a transformed value. Angular provides built-in pipes like DatePipe, CurrencyPipe, and DecimalPipe. Custom pipes can also be created for specific transformations.
* **Filters in AngularJS**: They were used to format data before displaying it in views, but Angular moved to the concept of pipes in Angular (2+), providing a more powerful and flexible data transformation mechanism.

@Pipe({

name: 'uppercase'

})

export class UppercasePipe implements PipeTransform {

transform(value: string): string {

return value.toUpperCase();

}

}

1. **How do you manage state in an Angular application?**

* State management in Angular applications can be achieved through various methods:
  + **Services**: Use Angular services to share state between components.
  + **NgRx**: A state management library based on the Redux pattern, providing a store, actions, reducers, and selectors.
  + **BehaviorSubject**: Use BehaviorSubject from RxJS for simple state management where state needs to be shared and observed.

import { BehaviorSubject } from 'rxjs';

@Injectable({

providedIn: 'root'

})

export class StateService {

private state = new BehaviorSubject<any>(initialState);

state$ = this.state.asObservable();

updateState(newState: any) {

this.state.next(newState);

}

}

1. **What is Angular’s Lazy Loading and how does it benefit an application?**

* Lazy Loading is a technique where modules are loaded only when they are needed, rather than at the initial application load. It helps in reducing the initial bundle size, improving application load time, and enhancing performance.

const routes: Routes = [

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

];

1. **What is the Angular CLI and what are some of its common commands?**

* Angular CLI (Command Line Interface) is a powerful tool used to initialize, develop, scaffold, and maintain Angular applications. It simplifies tasks like project setup, development, and deployment. Common commands include:
  + ng new: Create a new Angular project.
  + ng serve: Build and serve the application in a development environment.
  + ng build: Build the application for production.
  + ng generate: Generate Angular components, services, modules, etc.
  + ng test: Run unit tests.

ng new my-angular-app

ng serve

ng build --prod

ng generate component my-component

1. **How do you handle errors in Angular applications?**

* Error handling in Angular can be achieved in several ways:
  + **Global Error Handling**: Implement ErrorHandler to catch and handle errors globally.
  + **HttpInterceptor**: Use interceptors to handle HTTP errors centrally.
  + **Component-Level Handling**: Use try-catch blocks and error messages within individual components.

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

@Injectable()

export class GlobalErrorHandler implements ErrorHandler {

handleError(error: any): void {

// Handle error

console.error('An error occurred:', error.message);

}

}

1. **What is Angular’s Router and how does it handle navigation?**

* Angular Router is a module that allows navigation between different views or pages in an Angular application. It uses routes, defined in the RouterModule, to map URL paths to components. Navigation can be handled programmatically using the Router service or through router links in templates.

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

constructor(private router: Router) {}

navigateToHome() {

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

}

1. **Explain the purpose of Angular's @ViewChild and @ContentChild decorators.**
   * **@ViewChild**: Used to access and interact with child components, directives, or DOM elements within the same view.
   * **@ContentChild**: Used to access and interact with content projected into a component (i.e., content passed into ng-content).

@Component({

selector: 'app-parent',

template: `<app-child #child></app-child>`

})

export class ParentComponent {

@ViewChild('child') childComponent: ChildComponent;

}

1. **What are Angular’s modules and how do they contribute to application architecture?**

* Angular modules, defined by @NgModule, are used to group related components, directives, pipes, and services into cohesive blocks of functionality. They help in organizing the application into functional areas, enabling lazy loading, and managing dependencies.

@NgModule({

declarations: [MyComponent],

imports: [CommonModule],

exports: [MyComponent]

})

export class SharedModule { }

1. **How do you handle form validation in Angular?**

* Form validation in Angular can be handled using:
  + **Reactive Forms**: Define form controls and validators in the component class using FormBuilder, FormGroup, and FormControl.
  + **Template-driven Forms**: Use Angular directives like ngModel and built-in validators in the template.

this.form = this.fb.group({

name: ['', Validators.required],

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

});

1. **What are Angular’s built-in validators and how do you use them?**

* Angular provides several built-in validators:
  + Validators.required: Ensures the form control is not empty.
  + Validators.minLength(length): Ensures the form control value has at least the specified length.
  + Validators.maxLength(length): Ensures the form control value does not exceed the specified length.
  + Validators.pattern(pattern): Ensures the form control value matches a specified pattern.

this.form = this.fb.group({

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

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

});

1. **What is Angular’s @HostListener and @HostBinding and how are they used?**
   * **@HostListener**: Decorator to listen to DOM events on the host element of a directive or component.
   * **@HostBinding**: Decorator to bind a property of the host element to a property of the directive or component.

@Directive({

selector: '[appHover]'

})

export class HoverDirective {

@HostBinding('class.hover') isHovered = false;

@HostListener('mouseover') onMouseOver() {

this.isHovered = true;

}

@HostListener('mouseleave') onMouseLeave() {

this.isHovered = false;

}

}

1. **What are Angular’s Renderer2 and ElementRef and when should you use them?**
   * **Renderer2**: A service that provides a way to manipulate the DOM safely and in a platform-independent manner. It is used to interact with the DOM without directly accessing native DOM APIs.
   * **ElementRef**: A wrapper around a native DOM element that allows you to directly access and manipulate the element.

@Component({

selector: 'app-example',

templateUrl: './example.component.html'

})

export class ExampleComponent {

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

changeBackgroundColor() {

this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', 'blue');

}

}

1. **Explain the concept of Angular’s Reactive Forms and how they differ from Template-driven Forms.**
   * **Reactive Forms**: Provide a model-driven approach to handling form inputs, allowing you to create forms programmatically. They offer more control and flexibility, support for complex validation, and better testability. Forms are defined in the component class using FormGroup, FormControl, and FormBuilder.

this.form = this.fb.group({

name: ['', Validators.required],

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

});

* + **Template-driven Forms**: Use Angular directives to manage form inputs and validation in the template. They are simpler to use for basic forms but provide less control and are less suitable for complex forms.

<form #form="ngForm" (ngSubmit)="onSubmit(form)">

<input name="name" ngModel required>

<input name="email" ngModel required email>

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

</form>

1. **How does Angular’s Change Detection work and how can you optimize it?**

* Angular’s Change Detection mechanism checks for changes in the application’s state and updates the view accordingly. The default change detection strategy checks all components in the component tree. You can optimize change detection by:
  + Using the OnPush change detection strategy to limit checks to components with @Input changes or events.
  + Detaching the change detector for components that do not need frequent updates.
  + Using ChangeDetectorRef to manually trigger change detection when necessary.

@Component({

selector: 'app-example',

changeDetection: ChangeDetectionStrategy.OnPush

})

export class ExampleComponent {

constructor(private cdr: ChangeDetectorRef) {}

updateData() {

// Update data and trigger change detection manually

this.cdr.detectChanges();

}

}

1. **How do you implement lazy loading in Angular?**

* Lazy loading in Angular allows you to load modules only when they are needed, reducing the initial load time of the application. This is achieved using the loadChildren property in the route configuration.

const routes: Routes = [

{

path: 'feature',

loadChildren: () => import('./feature/feature.module').then(m => m.FeatureModule)

}

];

1. **What are Angular's @ViewChild and @ContentChild decorators used for?**
   * **@ViewChild**: Used to get a reference to a child component or directive within the same view. It allows interaction with the child component directly.

@Component({

selector: 'app-parent',

template: `<app-child #child></app-child>`

})

export class ParentComponent {

@ViewChild('child') childComponent: ChildComponent;

}

* + **@ContentChild**: Used to get a reference to a projected content (content passed via ng-content) in the component. It allows interaction with elements projected into the component.

@Component({

selector: 'app-host',

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

})

export class HostComponent {

@ContentChild(ChildComponent) childComponent: ChildComponent;

}

1. **How does Angular handle form validation and what are the key differences between reactive and template-driven forms?**
   * **Reactive Forms**: Use a model-driven approach to manage form inputs and validations. You define the form structure and validations in the component class, offering more control and flexibility.

this.form = this.fb.group({

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

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

});

* + **Template-driven Forms**: Use Angular directives to create and manage forms directly in the template. They are simpler and suitable for basic forms but provide less control compared to reactive forms.

<form #form="ngForm" (ngSubmit)="onSubmit(form)">

<input name="name" ngModel required minlength="3">

<input name="email" ngModel required email>

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

</form>

1. **What are Angular's AsyncPipe and Observable? How are they used in Angular applications?**
   * **AsyncPipe**: A built-in pipe that subscribes to an Observable or Promise and automatically updates the view when the data changes. It simplifies the subscription management.

<div \*ngIf="data$ | async as data">

{{ data }}

</div>

* + **Observable**: A core part of RxJS, Observables are used to handle asynchronous data streams and events. They can be subscribed to, and their values can be transformed, combined, and manipulated.

import { Observable } from 'rxjs';

this.data$: Observable<any> = this.dataService.getData();

1. **How does Angular’s Change Detection Strategy OnPush work and when would you use it?**

* The OnPush change detection strategy optimizes performance by checking a component’s view only when its input properties change, or when an event occurs within the component. It reduces the number of checks Angular performs, which can be beneficial for performance in large applications.

@Component({

selector: 'app-example',

changeDetection: ChangeDetectionStrategy.OnPush

})

export class ExampleComponent {

@Input() data: any;

}

1. **How do you handle routing and navigation in Angular applications?**

* Routing in Angular is managed by the RouterModule, which allows for configuring routes and handling navigation between different views. It uses route definitions to map URL paths to components and provides navigation methods to programmatically navigate between routes.

const routes: Routes = [

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

{ path: 'about', component: AboutComponent }

];

@NgModule({

imports: [RouterModule.forRoot(routes)],

exports: [RouterModule]

})

export class AppRoutingModule {}

typescript

Copy code

constructor(private router: Router) {}

navigateToAbout() {

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

}