Angular Interview Questions & Answers

1. Lifecycle hooks

Hook	Description	Common Use Case
ngOnChanges	Called when @Input() data changes.	React to input property changes.
ngOnInit	Called once after the first ngOnChanges.	Initialize data, fetch APIs.
ngDoCheck	Called on every change detection run.	Custom change detection logic.
ngAfterContentInit	After content (from <ng-content>) is projected.	Interact with projected content.
ngAfterContentChecked	After every check of projected content.	Monitor content changes.
ngAfterViewInit	After component's view is initialized.	Access @ViewChild, DOM manipulations.
ngAfterViewChecked	After every check of the component's view.	Post-view check logic.
ngOnDestroy	Before component is destroyed.	Cleanup: unsubscribe, clear intervals.

📌 Order of Execution

ngOnChanges() → ngOnInit() → ngDoCheck() →

ngAfterContentInit() → ngAfterContentChecked() →

ngAfterViewInit() → ngAfterViewChecked()

📌 Most Commonly Used Hooks

🔷 ngOnInit()

- Used for initialization logic like data fetching or setting default values.

🔷 ngOnChanges(changes: SimpleChanges)

- Called whenever an @Input() bound property changes.

🔷 ngOnDestroy()

- Used for cleanup like **unsubscribing from Observables** or removing event listeners.

🧠 Interview Summary:

- “Angular provides lifecycle hooks like ngOnInit for initialization, ngOnChanges for input changes, and ngOnDestroy for cleanup. I frequently use ngOnInit for setting up data and ngOnDestroy to prevent memory leaks, especially in services using Observables.”

2. Types of Data sharing between components?

- **1. Parent → Child (Using @Input)**
- **How:** Use property binding to pass data from parent to child component.
- **Use Case:** Share simple data (like strings, numbers, or objects) from parent to child.

Example

```
<!-- Parent Component -->
<app-child [message]="parentMessage"></app-child>

// Parent Component TS
export class ParentComponent {
  parentMessage = 'Hello from Parent';
}

// Child Component TS
export class ChildComponent {
  @Input() message!: string;
}
```

2. Child → Parent (Using @Output + EventEmitter)

How: Use @Output with EventEmitter to send data from child to parent component.

Use Case: Notify the parent when an event occurs in the child (e.g., button click).

Example:

```
<!-- Parent Component -->
<app-child (notifyParent)="onNotify($event)"></app-child>
// Parent Component TS
export class ParentComponent {
  onNotify(message: string) {
    console.log(message); // Output: "Child Button Clicked!"
  }
}

// Child Component TS
export class ChildComponent {
  @Output() notifyParent = new EventEmitter<string>();

  sendMessage() {
    this.notifyParent.emit('Child Button Clicked!');
  }
}
```

3. Sibling Components (Using a Shared Service)

How: Use a shared service with a Subject or BehaviorSubject to facilitate communication between sibling components.

Use Case: Send data between two components that don’t have a direct relationship.

Example:

```
// Shared Service
import { Injectable } from '@angular/core';
import { BehaviorSubject } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class SharedService {
  private userSource = new BehaviorSubject<any>(null);
  user$ = this.userSource.asObservable();

  updateUser(user: any) {
    this.userSource.next(user);
  }
}
```

```
// Sibling 1 (Component 1)
export class Sibling1Component {
  constructor(private sharedService: SharedService) {}

  updateUser() {
    const newUser = { name: 'John Doe', age: 30 };
    this.sharedService.updateUser(newUser);
  }
}

// Sibling 2 (Component 2)
export class Sibling2Component {
  user: any;

  constructor(private sharedService: SharedService) {
    this.sharedService.user$.subscribe(data => {
      this.user = data;
    });
  }
}
```

4. Using ViewChild or ViewChildren (Access Child Methods/Properties)

How: Access child component methods and properties directly in the parent.

Use Case: When you need to invoke methods or access properties in a child component.

Example:

```
// Parent Component TS
import { ViewChild } from '@angular/core';
import { ChildComponent } from './child.component';

export class ParentComponent {
  @ViewChild('child') child!: ChildComponent;

  callChildMethod() {
    this.child.childMethod(); // Calls method in child
  }
}

// Child Component TS
export class ChildComponent {
  childMethod() {
    console.log('Child method called!');
  }
}
```

5. Using ContentChild (for Projected Content)

How: Access content passed via <ng-content> inside a component.

Use Case: Interact with content projected into the component's view.

Example:

```
<!-- Parent Component -->
<app-child>
  <div>Projected Content</div>
</app-child>
// Child Component TS
import { ContentChild, ElementRef } from '@angular/core';

export class ChildComponent {
  @ContentChild('content') content!: ElementRef;

  ngAfterContentInit() {
    console.log(this.content.nativeElement.innerText); // Output: Projected Content
  }
}
```

6. Route Parameters / Query Params

How: Share data through the Angular Router, typically via URL parameters.

Use Case: Pass data between components through the route.

Example:

```
// Navigating to a route with parameters
this.router.navigate(['/user', userId]);

// Accessing parameters in the destination component
export class UserComponent {
  userId: string;

  constructor(private route: ActivatedRoute) {
    this.userId = this.route.snapshot.paramMap.get('id')!;
  }
}
```

3.What is Lazy Loading in Angular?

- **Lazy Loading** is a design pattern in Angular that allows you to load modules only when they are needed, rather than loading them all upfront during the initial load.
- **Purpose:** To improve application performance by splitting the app into smaller bundles and loading them on demand.
- **How It Works:** Only the modules required by the current route are loaded, instead of loading the entire application at once.

How to Implement Lazy Loading

1. Create a Feature Module: Generate a module to be lazily loaded.

ng generate module feature --route feature --module app

2. Set Up Routing for Lazy Loaded Module: Configure the route with loadChildren.

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

3. Configure the Feature Module's Routing: Inside the feature module, define its own routes.

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

- 4. Update App Module to Use Lazy Loading:Import the **AppRoutingModule** (which contains lazy loading logic) in AppModule.

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

When to Use Lazy Loading

- **Large Applications:** The app has multiple feature modules, and you want to improve performance by not loading everything at once.
- **Scalability:** The app might grow in the future with additional modules, and lazy loading ensures the app doesn’t get slow.
- **Route-Based Modules:** You have large or isolated parts of your application that users might not need immediately, like admin panels or settings pages.

Why to Use Lazy Loading

- **Improved Performance:** Only load the necessary parts of the application, leading to faster initial loading time.
- **Reduced Bundle Size:** Smaller bundles result in quicker downloads, as modules are loaded on demand.
- **Better User Experience:** Faster initial load time and smoother transitions between modules.

Where to Use Lazy Loading

- **Feature Modules:** For isolated features, such as dashboards, user profiles, or settings, that don’t need to be loaded initially.
- **Admin or Auth Sections:** Load admin or authentication modules only when needed, keeping the initial user experience faster.
- **External Libraries:** For third-party libraries that aren’t essential for the first load of the app.
- **Dynamic Routes:** Routes that are conditionally available based on user interaction or roles.

4. What is a Resolver in Angular?

- A **Resolver** in Angular is a service used to pre-fetch data before the route's component is activated. It resolves the necessary data **before** navigating to the route, ensuring the data is available for the component when it is loaded.
- **Use Case:** Ensures that data required for the route is available immediately when the component is loaded.
- **How It Works:** The resolver fetches the data before the route is activated and allows the component to receive the data without having to wait for it asynchronously after it loads.
- Resolvers are commonly used to:
- **Pre-fetch data:** To load data required by the component before navigation happens, avoiding incomplete UI.
- **Prevent UI Flicker:** Ensures that data is loaded before rendering the component, preventing loading states or flickers.
- **Cleaner Components:** Moves data-fetching logic out of the component's lifecycle methods (like ngOnInit), keeping components focused on UI.

Use **Resolvers** when:

You need data before the component is rendered.

- Example: User profile data needs to be fetched before the user sees their profile page.

You want to avoid loading spinners or incomplete UI.

- Example: Fetching data like user preferences or settings before the component is displayed.

You have complex or shared data fetching logic.

- Example: Fetching large datasets from an API that are necessary across multiple components.

Let's take an example of a **User Profile page**:

Without Resolver:

- The component renders first, and then the data is fetched asynchronously after the component is already displayed, showing loading indicators.

With Resolver:

- The data (e.g., user profile details) is fetched **before** the component is activated, ensuring the user sees the profile page with all the data already available, thus improving user experience.

5.What is an HTTP Interceptor in Angular?

- An **HTTP Interceptor** is a service in Angular that allows you to **intercept and modify HTTP requests and responses** globally before they are sent to the server or before the response is passed to the application. Interceptors can be used to add common headers, handle errors, or modify response data.
- **Purpose:** To centralize HTTP request and response handling logic, making the application more maintainable and efficient.
- **How it works:** When a request is made or a response is received, the interceptor can manipulate them or log the data before it reaches the component or server.

HTTP interceptors are useful in several scenarios:

Authentication and Authorization:

- Automatically attach JWT tokens or other authentication tokens to every HTTP request without needing to add them manually in each request.

Global Error Handling:

- Handle HTTP errors globally, such as 401 (Unauthorized) or 500 (Server Error), in a single place, avoiding repetitive error handling code in each service.

Logging and Monitoring:

- Log HTTP requests or responses for debugging or monitoring, helping with troubleshooting and application insights.

Response Transformation:

- Modify or format response data before it reaches the component. For example, you can standardize the response structure across your app.

6. What is a Provider in Angular?

- A **Provider** in Angular is a way to configure how a service or dependency is created and injected into components or other services. It’s a key part of Angular's **Dependency Injection** system.
- **Purpose:** It tells Angular how to create a service or provide a value.
- **How it works:** Angular uses providers to manage the instantiation of services and inject them wherever needed.

Types of Providers in Angular

1. Class Providers: Tells Angular to create an instance of a class (e.g., a service).

example:

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

2. **Value Providers**: Provides a constant value to inject.

   example:

   { provide: 'API_URL', useValue: 'https://api.com' }

3. **Factory Providers:** Use a factory function to create the service.

   example:

   { provide: MyService, useFactory: () => new MyService() }

4. **Existing Providers:** Alias one service to another.

   example:

   { provide: OldService, useExisting: NewService }

Why Use Providers in Angular?

- **Dependency Injection:** Simplifies service management and makes code more modular and testable.
- **Service Scope:** Control the lifecycle and scope of services (global, module-specific, or component-specific).
- **Reusability:** Allows for easy sharing of services across different parts of the application.

Example of Using Providers

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


@Component({ selector: 'app-home' })
export class HomeComponent {
 constructor(private myService: MyService) {}
}
```

Usage: The MyService is automatically injected into HomeComponent.

7. What is a Pipe in Angular?

- A **Pipe** in Angular is used to transform data in the view. It takes input data, processes it, and returns the transformed data to be displayed.
- **Purpose:** To format or transform data before displaying it in the view.
- **How it Works:** Pipes are used within template expressions to apply transformations like formatting dates, currency, or custom formats.

Types of Pipes in Angular

- There are two main types of pipes in Angular:

1. **Built-in Pipes:**
    - Angular provides several built-in pipes for common transformations.
    - **Examples:**
        - **DatePipe:** Formats dates.
        - **CurrencyPipe:** Formats numbers as currency.
        - **UpperCasePipe:** Converts text to uppercase.
        - **LowerCasePipe:** Converts text to lowercase.
        - **DecimalPipe:** Formats numbers with a specified decimal places.

Example

```
{{ today | date:'short' }}
{{ price | currency:'USD' }}
{{ 'hello' | uppercase }}
```

2. Custom Pipes:

- Custom pipes allow you to create your own transformation logic.
- Useful when you need to format data in a way that Angular doesn't provide out of the box.

Create the Pipe

*Use Angular CLI to generate the custom pi*pe : **ng generate pipe my-custom-pipe**

Implement the transformation logic in the transform method.

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

@Pipe({
  name: 'reverseText'
})
export class ReverseTextPipe implements PipeTransform {
  transform(value: string): string {
    return value.split('').reverse().join('');
  }
}.
```

To use : {{ 'hello' | reverseText }}

8. What Are Directives in Angular?

- **Directives** in Angular are classes that add additional behavior to elements in your Angular applications. They are one of the core building blocks of Angular. Angular provides several built-in directives like:
- **Structural Directives** (e.g., *ngIf, *ngFor) – change the DOM layout by adding/removing elements.
- **Attribute Directives** (e.g., ngClass, ngStyle) – change the appearance or behavior of an element.

Types of Directives

- **Component** – technically a directive with a template.
- **Attribute Directive** – changes the appearance or behavior of an element.
- **Structural Directive** – changes the DOM layout.

- **Creating a Custom Directive in Angular**
- You can create a custom **attribute directive** to change the appearance or behavior of an element.

1. Generate the Directive

Use Angular CLI: **ng generate directive highlight**

2. Directive Code

```
import { Directive, ElementRef, HostListener, Renderer2 } from '@angular/core';
@Directive({
 selector: '[appHighlight]'  // Use this attribute in HTML
})
export class HighlightDirective {
 constructor(private el: ElementRef, private renderer: Renderer2) {}

 @HostListener('mouseenter') onMouseEnter() {
  this.highlight('yellow');
 }
 @HostListener('mouseleave') onMouseLeave() {
  this.highlight('');
 }
 private highlight(color: string) {
  this.renderer.setStyle(this.el.nativeElement, 'backgroundColor', color);
 }}
```

3. Use the Directive in Template

<p appHighlight>Hover over this text to see the highlight directive in action.</p>

- @Directive() – Decorator that marks the class as a directive.
- selector: '[appHighlight]' – The directive will apply to elements with this attribute.
- ElementRef – Access to the DOM element.
- Renderer2 – Safer way to modify the DOM.
- @HostListener() – Listens to DOM events on the host element.

9. What are authentication and authorization techniques in Angular?

In Angular, **authentication** and **authorization** are typically handled using a combination of frontend techniques and backend support (e.g., via REST APIs).

- **1. Authentication**

Authentication verifies **who the user is**. In Angular, this is typically handled using:

✅ **Token-Based Authentication (e.g., JWT)**

- The user logs in via a login form.
- Angular sends credentials (username/password) to the backend.
- If valid, the backend returns a **JWT (JSON Web Token)**.
- The token is stored in **localStorage** or **sessionStorage** on the client.
- On every subsequent HTTP request, the token is sent in the Authorization header.

Example:

// ts

headers: new HttpHeaders().set('Authorization', `Bearer ${token}`)

2. Authorization

Authorization determines **what a user can do or access**.

✅ Role-Based or Permission-Based Access Control

- Once authenticated, the user's roles or permissions (e.g., 'admin', 'user') are embedded in the token or fetched separately.
- Angular uses **route guards** to control access to different routes based on roles.

3. Key Angular Tools Used

🔒 Route Guards

- Angular provides different types of guards:
- CanActivate: Prevents access to a route.
- CanActivateChild: Protects child routes.
- CanLoad: Prevents lazy-loaded modules from loading.
- CanDeactivate: Prevents navigating away from a route.

Example:

```
canActivate(route: ActivatedRouteSnapshot): boolean {
  const role = this.authService.getUserRole();
  return role === 'admin';
}
```

📦 **HttpInterceptor:** Used to automatically attach the JWT token to all outgoing HTTP requests.

```
// ts
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  const token = this.authService.getToken();
  const cloned = req.clone({
    setHeaders: {
      Authorization: `Bearer ${token}`
    }
  });
  return next.handle(cloned);
}
```

4. Best Practices

- Store tokens securely in memory or sessionStorage (avoid localStorage for high-security apps).
- Use refresh tokens to maintain sessions.
- Always validate tokens and permissions **on the server**.
- Protect routes using Angular guards and components using conditional rendering (*ngIf, etc.).

5. Bonus Tip (for Interview)

- "While Angular handles the client-side aspect, I always ensure the backend securely validates all tokens and enforces access control. I also use observables and BehaviorSubjects to manage authentication state reactively across components."

10. What are Observables and BehaviorSubjects in Angular?

In Angular, both **Observables** and **BehaviorSubjects** are part of **RxJS**, a reactive programming library used for handling asynchronous operations such as HTTP requests, user events, or real-time data streams.

🔷 1. Observables

✅ What is it?

An Observable is a data stream that **emits values over time**, and you can **subscribe** to it to react when data is emitted.

✅ Use cases:

- HTTP requests
- User input events
- WebSocket streams

Example:

```
// ts
this.http.get('api/data').subscribe(data => {
  console.log(data);
});
```

✅ **Key properties:**

- Does **not hold a current value**
- Subscribers only get **new values**
- **Cold** by default – starts emitting only when subscribed

- 🔷 **2. BehaviorSubject**

✅ **What is it?**

- A BehaviorSubject is a type of Subject that **stores the latest value**, and **emits it immediately to new subscribers**.

✅ **Use cases:**

- Sharing data across components
- Holding and reacting to application state (e.g., login status)

Example:

```
authStatus = new BehaviorSubject<boolean>(false); // false = not logged in

// Subscribe
this.authStatus.subscribe(isLoggedIn => {
  console.log('User logged in:', isLoggedIn);
});

// Update
this.authStatus.next(true);
```

✅ Key properties:

- Always **holds the latest value**
- **Emits current value to new subscribers**
- Perfect for **state management**

✅ Observables VS Behavior Subject

Feature	Observable	BehaviorSubject
Holds current value	❌ No	✅ Yes
Emits on subscribe	❌ Only new values	✅ Immediately emits last value
Use case	HTTP, events	App state, shared data

✅ Bonus (Interview Tip)

- "I usually use Observables for one-time async operations like HTTP calls, and BehaviorSubjects when I need to manage and share state reactively between components or services."

11. What are Guards in Angular?

Guards in Angular are interfaces that allow us to control **navigation to and from routes**. They are part of Angular's **router module** and help with security, permissions, and conditional routing.

Guards return either:

- true (allow access),
- false (deny access),
- or an Observable/Promise of true/false,
- or a UrlTree (to redirect).

🔐 Types of Route Guards

Guard	Purpose
CanActivate	Checks if a route can be activated (entered)
CanActivateChild	Checks if child routes can be activated
CanDeactivate	Checks if we can leave a route
CanLoad	Checks if a module (lazy-loaded) can be loaded
Resolve	Pre-fetches data before the route is activated

✅ Bonus Interview Tip

"I use guards not only for protecting routes but also for improving UX — for example, by using Resolve to preload data so the component doesn't have to handle spinners or loading states."

12. What are decorators in Angular?

- In Angular, **decorators** are special **functions prefixed with @** that attach metadata to **classes, methods, properties, or parameters**. Angular uses this metadata to understand how to **process and use** these elements at runtime.
- Decorators come from **TypeScript** and play a critical role in Angular's **dependency injection, component system, and module system**.

🧠 Why Are Decorators Important?

- They tell Angular:
- "This class is a component"
- "This property is an input"
- "This method should be triggered on init"
- And so on...

🔍 Examples of Key Decorators

✅ @Component

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

✅ @Injectable

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

✅ @Input and @Output

```
@Input() userData: any;
@Output() onLogin = new EventEmitter<boolean>();
```

✅ @HostListener and @HostBinding

```
@HostListener('mouseenter') onMouseEnter() {
  this.hover = true;
}

@HostBinding('class.active') isActive = false;
```

✅ @NgModule

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

🚀 Best Practices

- Use decorators to **clearly separate concerns** (e.g., component logic vs service logic).
- Remember decorators are just **metadata** – Angular's compiler uses them to generate the runtime code.
- Avoid overusing @ViewChild or @HostBinding — keep components loosely coupled.

✅ Bonus Interview Tip

- "Decorators in Angular are not just syntactic sugar — they define the metadata Angular uses to configure and instantiate components, services, and modules properly. They're fundamental to Angular's architecture."

13. What is State Management in Angular?

State management in Angular refers to the way we **store, track, and share data (state)** across components, services, and modules — especially in **medium to large applications**. It's about ensuring that changes in state are predictable, traceable, and easy to maintain.

- 🧠 **Why is State Management Important?**
- Avoids tightly coupled components
- Keeps UI in sync with data
- Helps manage **shared data** (e.g., user info, cart items, auth status)
- Useful for **debugging and time-travel debugging** in larger apps

📦 Types of State in Angular

Type	Example	Scope
Local state	Form inputs, toggle buttons, etc.	Within component
Shared state	User login status, theme, cart items	Across components
Global state	App-wide settings or data	Entire app

✅ Best Practices

Keep state **centralized** and **immutable** where possible.
Use **selectors** to access only required parts of the state.
Avoid overusing state management for things that can stay local (like UI toggles).
Use **DevTools** (e.g., NgRx Store Devtools) to trace actions and state.

🎯 Bonus Interview Tip

"I start with services and BehaviorSubject for smaller features, and introduce NgRx or NgXs only when the app grows and the state becomes too hard to manage predictably. I also make sure the state is normalized and avoid storing UI-specific data in the global store."

📦 NgRx

- **NgRx** is a **state management library** for Angular, inspired by **Redux**.
- It helps manage **application state** in a **predictable, reactive** way using **RxJS**.

📌 Why Use NgRx?

- Manages **complex shared state** across multiple components.
- Centralizes app state in a **single source of truth (Store)**.
- Enables **time-travel debugging**, undo/redo, and testability.
- Promotes **immutability** and **unidirectional data flow**.

📌 NgRx Flow Summary

Component dispatches Action

↓

Reducer updates Store (pure function)

↓

Selectors read updated state from Store

↓

Effects (optional) handle async operations and dispatch new actions

🧠 Interview Summary:

- "NgRx helps manage global state in Angular using actions, reducers, and effects. I've used it in large applications to maintain predictable state, separate side effects with Effects, and improve scalability. It works well with Angular's reactive nature and leverages RxJS extensively."

Concept	Purpose
Store	Centralized state container
Action	Plain object that describes **what happened**
Reducer	Pure function that handles **state transitions**
Selector	Used to **query state** from the store
Effect	Handles **side effects** (e.g., API calls) using **RxJS observables**

14. What are Standalone Components in Angular?

Standalone components are a new way to define Angular components **without needing to declare them inside an NgModule**. Introduced in **Angular v14**, they simplify Angular's module system by allowing components, directives, and pipes to be **self-contained and directly imported** wherever needed.

🧠 Why Standalone Components?

- ✅ Reduce boilerplate (NgModule declarations)
- ✅ Improve tree-shakability and performance
- ✅ Make components easier to reuse and test
- ✅ Pave the way for a **module-less Angular future**

🛠️ How to Create a Standalone Component

Step 1: Generate using Angular CLI

ng generate component example --standalone

Step 2: Resulting Code

```
@Component({
  standalone: true,
  selector: 'app-example',
  templateUrl: './example.component.html',
  styleUrls: ['./example.component.css'],
  imports: [CommonModule, FormsModule] // Optional
})
export class ExampleComponent { }
```

📦 Using a Standalone Component

Instead of declaring it in a module, you import it directly:

```
import { ExampleComponent } from './example/example.component';

@Component({
  standalone: true,
  imports: [ExampleComponent],
  template: `<app-example></app-example>`
})
export class ParentComponent {}
```

Or use it directly in a route:

```
{
  path: 'example',
  component: ExampleComponent
}
```

🔁 Standalone vs Traditional Components

Feature	Traditional Component	Standalone Component
Declared in NgModule	✅ Yes	❌ No
Reusability	❌ Requires module import	✅ Direct import
Simplifies structure	❌ No	✅ Yes
Tree-shakable	🚫 Less efficient	✅ More efficient

🎯 Bonus Interview Tip

“I’ve used standalone components to streamline onboarding for new devs and to reduce NgModule clutter. They’ve helped us break down features into truly independent, testable units — especially useful in micro frontend-like architectures.”

15. What is a Signal in Angular?

In Angular, a **Signal** is a **reactive primitive** introduced in **Angular v16** as part of the new **reactivity model**. It allows you to declare reactive values that automatically update the DOM or other dependent computations when the value changes — similar to React's state or Vue's reactivity system.

🔧 Why Signals?

- ✅ Simplifies reactive programming without needing RxJS or BehaviorSubject
- ✅ Enables **fine-grained reactivity** and better performance
- ✅ Easier mental model than Observables for component state
- ✅ Great for **state, derived values, and computed logic**

🎯 Bonus Interview Tip

- “Signals let us write more declarative and readable code. I’ve started replacing some of our BehaviorSubject-based state with signal(), especially in components where simpler, local reactivity is enough.”

16. What is SSR (Server-Side Rendering) in Angular?

SSR (Server-Side Rendering) in Angular refers to rendering the application **on the server** (e.g., Node.js) and sending the **fully rendered HTML** to the browser. This is done **before JavaScript loads** on the client, improving **performance, SEO**, and **first contentful paint**.

- Angular provides SSR support through a tool called **Angular Universal**.

🌍 Why Use SSR?

Benefit	Explanation
✅ SEO Friendly	Search engines can index pre-rendered HTML
✅ Faster FCP (First Contentful Paint)	Content is visible before JS loads
✅ Better UX on slow networks	Users see something rendered almost immediately
✅ Social Media Previews	OG tags and meta data work correctly on SSR pages

🧩 Common SSR Challenges

- Handling browser-only APIs (e.g., window, document)
- Lazy-loaded routes may need to be preloaded
- Third-party libraries may not be server-safe
- Requires more complex deployment (Node server)

🎯 Bonus Interview Tip

- “I’ve used Angular Universal to improve SEO and performance for public-facing pages. It helps in apps where initial load speed and search engine visibility matter. I’m also familiar with hybrid rendering setups where we SSR public routes and CSR private dashboards.”

17. How do you optimize an Angular application?

Optimizing an Angular application involves improving **performance, load time, memory usage, and user experience**. I approach optimization from several angles: **build size, rendering performance, network efficiency, and change detection**.

🔷 1. **Optimize Bundle Size**

✅ **Use Lazy Loading**: Load modules only when needed using Angular routing.

Eg:

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

✅ **Tree Shaking**: Angular CLI and Webpack remove unused code by default during production builds.

✅ **Enable Production Mode**:

ng build --configuration=production

✅ **Minification & Uglification**: Automatically handled in production mode.

🔷 2. **Use OnPush Change Detection**

- ✅ Use ChangeDetectionStrategy.OnPush to reduce the number of change detection cycles.

Eg:

```
@Component({
  selector: 'app-optimized',
  templateUrl: './optimized.component.html',
  changeDetection: ChangeDetectionStrategy.OnPush
})
```

Works well with **immutable data** and **observables**.

🔷 3. *Use TrackBy in ngFor*

Improves DOM performance when iterating over lists.

Html

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

ts

```
trackById(index: number, item: any): number {
  return item.id;
}
```

🔷 **4. Lazy Load Images and Components**

- Use native loading="lazy" for images.
- Use NgComponentOutlet or dynamic imports to load components on demand.

🔷 5. Efficient State Management

- Use **RxJS**, **Signals**, or libraries like **NgRx** for reactive state.
- Avoid redundant subscriptions and prefer async pipe.

🔷 6. Preload or Prefetch Resources

- Use Angular's **PreloadAllModules** strategy to speed up navigation:

🔷 7. Avoid Memory Leaks

- Unsubscribe from manual subscriptions (or use takeUntil, AsyncPipe).
- Clean up setTimeout, setInterval, and global event listeners.

🔷 10. Analyze and Audit Performance

- Use ng build --stats-json and tools like **Webpack Bundle Analyzer**:

18.What is the trackBy function in Angular and why is it used in *ngFor?

The trackBy function is used with Angular's *ngFor directive to **optimize rendering performance** of lists. It helps Angular track which items have changed, been added, or removed — **preventing unnecessary DOM re-renders**.

🔍 Why is trackBy Important?

- By default, Angular uses **object identity** to track items. This means if you update an array (e.g., with new references), Angular may re-render **the entire list**, even if only one item changed.
- With trackBy, Angular uses a **custom identifier (like id)** to track items more efficiently.

✅ Bonus Interview Tip

- "Whenever I use *ngFor, I always implement trackBy — especially when rendering large or frequently changing lists. It avoids full DOM refreshes and improves rendering efficiency."

19. What is Dependency Injection in Angular?

Dependency Injection (DI) in Angular is a **design pattern** used to **supply components and services with their dependencies** (like other services or values) rather than creating them manually. Angular has a **built-in DI framework** that makes services reusable, testable, and maintainable.

🧠 Why Use Dependency Injection?

- ✅ Promotes **loose coupling**
- ✅ Improves **testability**
- ✅ Supports **reusability** and **scalability**
- ✅ Simplifies **object creation and lifecycle management**

✅ Bonus Interview Tip

- "I use Angular's DI system to keep my services loosely coupled and testable. By leveraging different provider scopes, I ensure efficient memory use and proper service lifecycle management — especially in feature modules or lazy-loaded routes."

20. How does Angular work?

Angular is a **front-end framework** for building **single-page applications (SPAs)** using **HTML, TypeScript, and components**. It works by creating a dynamic connection between the **view (template)** and the **logic (component)** using a **powerful architecture** built around:

- Components.
- Modules
- Services
- Routing
- Dependency Injection
- Change Detection

📦 High-Level Angular Architecture

main.ts → AppModule → AppComponent → Child Components

↓

Services (via DI)

↓

Templates + Data Binding + Events

↓

DOM Rendering & Change Detection

✅ **Bonus Interview Tip**

- “I think of Angular as a well-organized framework with clear boundaries — components for UI, services for logic, modules for structure, and the DI system to tie everything together. Its ability to compile templates and detect changes efficiently is what powers dynamic single-page apps.”

21. How do you upgrade an Angular application to a newer version?

Upgrading an Angular app involves updating dependencies, configurations, and possibly refactoring deprecated APIs or features. Angular provides a **dedicated upgrade tool** and follows **semantic versioning**, making upgrades predictable.

🛠️ **Step-by-Step Process to Upgrade an Angular App**

🔷 **1. Check Current Version : ng version**

🔷 2. **Use the Angular Update Guide:** Visit: https://update.angular.io

🔷 3. **Update Angular CLI and Core Packages: ng update @angular/cli @angular/core**

🔷 4. **Update Other Dependencies: ng update @angular/material, ng update rxjs etc**

🔷 **5. Run and Test the Application**

🔷 6. **Clean Up and Rebuild: npm install, ng build --configuration=production**

✅ Bonus Interview Tip

- “In my last project, we upgraded from Angular 13 to 16. I used ng update and the Angular Update Guide to manage breaking changes. We had to refactor some RxJS logic and test all lazy-loaded modules. I also set up CI to run tests after each upgrade step to catch regressions early.”

22.Latest features of angular

Angular 15 (Released November 2022)

- **Stable Standalone Components API**: Made the standalone components API stable, allowing for more modular applications without NgModules
- **Directive Composition API**: Introduced a new API for composing directives, enabling more flexible and reusable code.
- **NgOptimizedImage Directive**: Stabilized the image optimization directive, improving image loading performance. Medium
- **Improved Stack Traces**: Enhanced stack traces for better debugging and error tracking. DEV Community

Angular 16 (Released May 2023)

- **Non-Destructive SSR Hydration**: Introduced non-destructive hydration for server-side rendering, reducing flickering and improving user experience. Medium+1Medium+1
- **Vite for Development**: Integrated Vite as the development server, enhancing build times and hot module replacement. Medium+1DEV Community+1
- **Typed Forms Enhancements**: Further improved type safety in forms, reducing potential errors.
- **DestroyRef and takeUntilDestroyed**: Introduced DestroyRef for better lifecycle management and takeUntilDestroyed for cleaner observable subscriptions. Medium
- **Required Component Inputs**: Allowed developers to mark component inputs as required, ensuring necessary data is provided. Medium

23. RxJs Operators

switchMap, forkJoin, concatMap, mergeMap, reduce, filter, and map. These operators are essential in Angular for handling asynchronous operations and transforming data.

✅ 1. switchMap

Purpose: Switch to a new observable when the source emits a new value, and unsubscribe from the previous observable.

Use Case: Useful for scenarios where only the result of the latest observable matters (e.g., HTTP requests, search bar auto-suggestions).

Eg:

```
import { switchMap } from 'rxjs/operators';

source$.pipe(
  switchMap(value => this.someService.getData(value))
).subscribe(data => console.log(data));
```

✅ 2. forkJoin

Purpose: Wait for multiple observables to complete, and then combine their last emitted values into an array or an object.

Use Case: Useful when you need to perform multiple independent HTTP requests and wait for all of them to finish.

```
import { forkJoin } from 'rxjs';


forkJoin([
  this.service1.getData(),
  this.service2.getData(),
]).subscribe(([data1, data2]) => {
  console.log(data1, data2);
});
```

Key Behavior: forkJoin emits only when all observables complete, emitting their last values as an array.

✅ 3. concatMap

Purpose: Map each emitted value from the source observable to an inner observable, but the inner observables are subscribed to one at a time in order.

Use Case: Useful when you need to perform sequential operations, where the next operation depends on the completion of the previous one.

```
import { concatMap } from 'rxjs/operators';


source$.pipe(
  concatMap(value => this.someService.getData(value))
).subscribe(data => console.log(data));
```

Key Behavior: The inner observables are subscribed to one after the other, not concurrently.

✅ 4. mergeMap

Purpose: Map each emitted value from the source observable to an inner observable and merge all of them concurrently.

Use Case: Useful when you want to handle multiple requests or tasks concurrently without waiting for one to finish before starting the next.

```
import { mergeMap } from 'rxjs/operators';


source$.pipe(
  mergeMap(value => this.someService.getData(value))
).subscribe(data => console.log(data));
```

Key Behavior: mergeMap subscribes to multiple inner observables concurrently.

✅ 5. reduce

Purpose: Accumulate values over time (like Array.prototype.reduce), emitting only the final accumulated result when the source observable completes.

Use Case: Useful when you need to combine or sum up values emitted over time (e.g., calculating totals, reducing an array).

```
import { reduce } from 'rxjs/operators';


source$.pipe(
  reduce((acc, value) => acc + value, 0)
).subscribe(total => console.log(total));
```

Key Behavior: Emits only one value — the final accumulated result — when the source observable completes.

✅ 6. filter

Purpose: Filter emitted values based on a condition.

Use Case: Useful when you want to only allow specific items to pass through based on some condition.

```
import { filter } from 'rxjs/operators';


source$.pipe(
  filter(value => value > 10)
).subscribe(filteredValue => console.log(filteredValue));
```

Key Behavior: Emits only the values that pass the given condition.

✅ 7. map

Purpose: Transform emitted values (similar to Array.prototype.map) by applying a function to each value.

Use Case: Useful for transforming data as it flows through an observable pipeline (e.g., converting raw response data into a more usable format).

```
import { map } from 'rxjs/operators';


source$.pipe(
  map(value => value * 2)
).subscribe(transformedValue => console.log(transformedValue));
```

Key Behavior: Transforms each emitted value before passing it downstream.

📚 Summary of Differences

Operator	Description	When to Use
switchMap	Switch to a new observable on every new emission, canceling previous ones.	When only the result of the latest emission matters (e.g., user input).
forkJoin	Wait for all observables to complete and combine their last values.	When you need to wait for multiple HTTP requests to finish.
concatMap	Sequentially process observables in order.	When tasks must be performed one after the other, in sequence.
mergeMap	Merge multiple inner observables concurrently.	When tasks can run concurrently without waiting for each other.
reduce	Accumulate values over time and emit the final result when source completes.	When you need to combine or aggregate emitted values into a final result.
filter	Emit only values that meet a specified condition.	When you want to allow certain values through, based on a condition.
map	Transform each emitted value using a function.	When you need to apply a transformation to each emitted value.

24. What is Zone.js in Angular?

Zone.js is a library used by Angular to **automatically detect asynchronous operations** (like HTTP requests, setTimeout, and event listeners) and **trigger change detection** when those operations complete.

It provides a **execution context** that tracks all async operations and tells Angular when to update the DOM.

🔍 Why is Zone.js Important in Angular?

Angular needs to know **when something changes** (like data from an API) so it can update the view. Instead of manually notifying Angular, Zone.js handles this **automatically**.

🧠 How It Works

Zone.js monkey-patches async APIs like:

- setTimeout
- Promise
- XMLHttpRequest
- DOM events

Angular wraps your application inside a **zone** (called NgZone), which tracks these operations.

When an async task finishes, Zone.js calls Angular's **change detection** to update the UI.

🔁 Example Without and With Zone.js

🔸 Without Zone.js:

You would have to **manually trigger change detection** using ChangeDetectorRef.detectChanges().

🔸 With Zone.js:

Angular knows when an HTTP request completes and updates the DOM **automatically**.

🔧 NgZone vs Zone.js

NgZone is Angular's service that wraps Zone.js.

You can use NgZone.run() and NgZone.runOutsideAngular() to control performance-sensitive tasks.

🧪 Use Cases in Interview Scenarios

Understanding change detection and async behavior

Optimizing performance in high-frequency events (e.g., scroll, resize)

Using zone-less mode for micro-optimizations in large apps

✅ Bonus Interview Tip

“Zone.js plays a critical role in Angular’s automatic change detection. I’ve used NgZone to optimize performance by running expensive operations outside the Angular zone and manually re-entering when needed. For very high-performance apps, I’ve explored disabling Zone.js entirely.”

25. What is the difference between a Component and a Directive in Angular?

In Angular, both **components** and **directives** are used to add behavior to the DOM, but they serve different purposes.

Feature	**Component**	**Directive**
Purpose	Defines **UI elements** with logic and template	Adds **behavior** to existing elements
Decorator Used	@Component	@Directive
Template	Has its own HTML template	Does **not** have a template
Usage	Used to build **UI views** (pages, layouts)	Used to **manipulate DOM**, add logic
Selector	Used as an **element** (e.g. <app-header>)	Used as an **attribute** (e.g. [appHighlight])
Example	A navbar, form, or product list component	A tooltip directive, custom validator

🔧 Code Examples

🔷 Component Example

```
@Component({
 selector: 'app-user-card',
 template: `<div>{{ user.name }}</div>`
})
export class UserCardComponent {
 @Input() user: any;
}
```

🔷 Directive Example

```
@Directive({
 selector: '[appHighlight]'
})
export class HighlightDirective {
 constructor(el: ElementRef) {
  el.nativeElement.style.backgroundColor = 'yellow';
 }
}
```

Usage:

Html

```
<p appHighlight>This text is highlighted</p>
```

✅ Bonus Interview Tip

“I think of a component as a directive with a view. While components define **UI elements**, directives are more about **behavior**. I use custom attribute directives for reusable DOM logic and structural directives to dynamically render elements.”

26. What are the types of forms in Angular and how do they differ?

Angular provides two main types of forms:

Form Type	Description	Also Known As
Template-driven	Form logic is written in the **template (HTML)**	Declarative forms
Reactive	Form logic is written in the **component (TypeScript)**	Model-driven forms

📜 1. Template-Driven Forms

Defined in the HTML template using Angular directives like ngModel, #form="ngForm".
Suitable for **simple forms** with minimal logic.
Uses **FormsModule**.

✅ Example:

```
// html
<form #userForm="ngForm" (ngSubmit)="submitForm(userForm)">
  <input name="username" ngModel required />
</form>

// ts
submitForm(form: NgForm) {
  console.log(form.value);
}
```

- ✅ **Automatic two-way binding**
- ✅ Less boilerplate
- ❌ Harder to unit test
- ❌ Less control over form state

📜 2. Reactive Forms

- **Defined and managed in the component class** using FormGroup, FormControl, and FormBuilder.
- Ideal for **complex or dynamic forms**.
- Uses **ReactiveFormsModule**.

✅ Example:

```
//ts
form = new FormGroup({
  username: new FormControl('', Validators.required)
});

//html
<form [formGroup]="form" (ngSubmit)="submitForm()">
  <input formControlName="username" />
</form>

//  ts
submitForm() {
  console.log(this.form.value);
}
```

- ✅ Better testability
- ✅ Precise control over form validation and structure
- ✅ Easier to handle **dynamic form fields**
- ❌ More verbose for simple forms

✅ Bonus Interview Tip

“I use template-driven forms for quick, simple forms like login or contact forms. For anything more dynamic or complex — like nested or conditional fields — I prefer reactive forms for better structure and control.”

📊 Comparison Table

Feature	Template-Driven	Reactive
Defined in	HTML template	TypeScript class
Module Needed	FormsModule	ReactiveFormsModule
Two-way Binding	Yes (ngModel)	No (manual binding using FormControl)
Validation	Template-based (required, etc.)	Programmatic (Validators.required, etc.)
Form Control Instantiation	Implicit via directives	Explicit via FormControl, FormGroup
Suitable For	Simple forms	Complex, dynamic, and scalable forms
Testing	Less testable	Highly testable

27. What is Change Detection Strategy in Angular?

Change Detection in Angular is the mechanism that keeps the view in sync with the component state. It checks the component tree to detect any changes and updates the DOM accordingly.

Angular offers two main **change detection strategies**:

Strategy	Description
Default	Angular checks **all components** in the tree when any event occurs.
OnPush	Angular checks the component **only if its inputs change**, or manually.

🔄 1. Default Change Detection

This is the **default behavior**.

Angular runs change detection **on every async event**: clicks, HTTP responses, timers, etc.

It checks the **entire component tree** recursively.

✅ Use When:

- Your app has simple data flows.
- You don't need performance optimization.

🚀 2. OnPush Change Detection

Angular will **skip change detection** for the component **unless:**

- An @Input() reference changes.
- You call ChangeDetectorRef.markForCheck() or detectChanges().

Improves performance in **large or deeply nested apps**.

🧪 Real-World Scenario for OnPush

🔶 Scenario:

- You're building a dashboard with many cards (<app-widget-card>), and data is refreshed via WebSockets or external state management.
- If all widgets use Default, every update re-checks **all cards**.
- With OnPush, only the cards receiving new inputs are checked — improving performance drastically.

🧰 Manual Change Detection with OnPush

Use ChangeDetectorRef when needed:

Eg;

```
// ts
constructor(private cdRef: ChangeDetectorRef) {}

update() {
  this.cdRef.markForCheck(); // re-check this component in next cycle
  // or
  this.cdRef.detectChanges(); // run change detection immediately
}
```

⚠️ Pitfall to Watch Out For

If using OnPush, **mutating an object directly won't trigger updates**. You must provide a new reference.

```
//ts
// This won't work with OnPush:
this.user.name = 'John';

// This will:
this.user = { ...this.user, name: 'John' };
```

✅ Bonus Interview Tip

- "I use ChangeDetectionStrategy.OnPush in all presentational components by default. It helps avoid unnecessary re-renders and boosts performance, especially in large-scale apps with many components. I use ChangeDetectorRef when I need to manually trigger updates — like after a setTimeout or third-party event."

28. What is APP_INITIALIZER in Angular?

- APP_INITIALIZER is a special **multi-provider token** in Angular used to **run logic before the application initializes** — typically for **loading configuration**, initializing services, or performing asynchronous tasks (like HTTP requests).
- It **delays app bootstrapping** until the provided functions **complete execution** (including promises or observables).

🔧 Why Use APP_INITIALIZER?

- Load **configuration files** from a server before the app starts.
- Initialize **authentication state**.
- Set up **internationalization (i18n)** or **theme** settings.
- Preload **feature toggles** or **user preferences**.

✅ Bonus Interview Tip

“I’ve used APP_INITIALIZER to load app-wide configuration and environment flags before bootstrapping. It ensures critical data like API base URLs or feature toggles are available app-wide, especially in enterprise apps where runtime config is important.”

29. Error Handling in Angular

1. Using try-catch (for synchronous code)

Example:

```
  try {
    // some logic
      } catch (error) {
       console.error('Error:', error);
     }
```

2. Using RxJS catchError (for HTTP and observables):

Example:

```
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

this.http.get('/api/data').pipe(
  catchError(error => {
    console.error('HTTP Error:', error);
    return throwError(() => error); // or return EMPTY / of()
  })
).subscribe();
```

3. Global Error Handling:

Example: Implement ErrorHandler class

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


@Injectable()
export class GlobalErrorHandler implements ErrorHandler {
  handleError(error: any): void {
    // Log to server or show user-friendly message
    console.error('Global Error:', error);
  }
}
```

- Provide in app module:

```
{ provide: ErrorHandler, useClass: GlobalErrorHandler }
```

4. HTTP Interceptor for centralized API error handling:

```
intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
  return next.handle(req).pipe(
    catchError(error => {
      // Handle HTTP errors
      console.error('Interceptor Error:', error);
      return throwError(() => error);
    })
  );
}
```

30. Dumb vs Smart Components in Angular

1. Smart Component (Container Component)

- Responsible for **handling business logic**, data fetching, and interacting with services.
- Passes data to dumb components via **@Input()** and listens to events via **@Output()**.
- Knows *how* things work.

2. Dumb Component (Presentational Component)

- Focuses on **UI only**.
- Receives data via **@Input()** and emits events using **@Output()**.
- Reusable and easy to test.
- Doesn’t know where data comes from.

31. What is Angular Routing and why is it used?

- Angular Routing enables navigation between different views/components within a **Single Page Application (SPA)**. It allows for dynamic content updates without a page reload.

Routes are configured in the app-routing.module.ts using RouterModule.forRoot(routes). The routes array maps paths to components, defining how navigation works.

```
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
];
```

What is RouterModule and how is it used?

- RouterModule is an Angular module that provides all the necessary functionality for routing (directives like routerLink, services like Router, and others). It's imported in the root module or feature modules.

eg: imports: [RouterModule.forRoot(routes)].

What are route parameters and how can you access them?

- Route parameters allow dynamic values in URLs (e.g., /user/:id). They can be accessed via ActivatedRoute:

Eg:

```
this.route.snapshot.paramMap.get('id’);
```

Use paramMap or queryParamMap to handle route and query parameters.

What is a wildcard route? Why is it used?

A wildcard route ({ path: '**', component: NotFoundComponent }) is used to catch any undefined paths, typically for **404 Not Found** pages or redirection.

Eg:

```
{ path: '**', component: PageNotFoundComponent }
```

What are route guards in Angular? Name some types.

Route guards are used to protect routes based on conditions (e.g., authentication).

- Types of route guards:
  - CanActivate: Checks if a route can be activated.
  - CanDeactivate: Determines if a user can leave the current route.
  - CanLoad: Prevents a module from being loaded if certain conditions are not met.
  - Resolve: Pre-fetches data before activating a route.

How do child routes work in Angular?

- Child routes allow you to define nested routes under a parent route. This is useful for displaying hierarchical components.

Eg:

```
{ path: 'products', children: [
  { path: 'details/:id', component: ProductDetailComponent }
]}
```

How do you reload a component on the same route with different params?

- To reload the component when route parameters change, subscribe to paramMap from ActivatedRoute:

Eg:

```
this.route.paramMap.subscribe(params => {
  this.loadData(params.get('id'));
});
```

32. Host Listening vs Host Binding in Angular

@HostBinding is used to bind a property of a directive or component to a host element's property.

It allows you to **set properties or attributes** on the host element (the element to which the directive or component is attached).

Eg:

```
@HostBinding('class.active') isActive = true;
```

This binds the isActive variable to the active class on the host element. If isActive is true, the active class will be applied to the host element.

Use Case:

- You can dynamically add/remove classes, set inline styles, or even control the host element's attributes.

@HostListener is used to listen to **events** on the host element and trigger a method when the event occurs.

It allows you to **handle events** like clicks, mouse movements, key presses, etc., directly from the directive/component.

Eg:

```
@HostListener('click', ['$event'])
onClick(event: MouseEvent) {
  console.log('Element clicked:', event);
}
```

This listens for the click event on the host element and calls onClick() whenever the event occurs.

Use Case:

- You can listen for native DOM events or custom events on the host element and execute custom logic in response.

Key Takeaways:

- **@HostBinding**: Bind properties or attributes to the host element (class, style, etc.).
- **@HostListener**: Listen for events on the host element and invoke methods in response.

33. Subject, ReplaySubject, and BehaviorSubject in Angular

What is a Subject in RxJS?

Subject is a **multicasting** observable, meaning it can send the same emitted value to multiple subscribers.

- It acts both as an **observer** and an **observable**. This means you can both **emit values** to subscribers and **subscribe** to other observables.

Characteristics:

- **Unicast**: Subscribers only receive values emitted after they subscribe.
- No initial value is provided.

What is a ReplaySubject in RxJS?

ReplaySubject is similar to Subject, but it **replays the last emitted value** (or a set number of values) to new subscribers.

- This means **new subscribers will immediately get the most recent value(s)**, even if they subscribe after the value was emitted.

Characteristics:

- It **remembers** a specific number of emitted values (you can define how many).
- Useful for ensuring new subscribers get the last known state or value.

What is a BehaviorSubject in RxJS?

BehaviorSubject is a type of Subject that **requires an initial value** and always emits the **current value** to new subscribers.

- It remembers the **latest emitted value**, so even if a subscriber subscribes later, it will immediately receive the most recent value.

Characteristics:

- Has an initial value.
- New subscribers always receive the **most recent value** immediately.

Real-world Use Cases:

- **Subject**: Broadcasting events like button clicks or status updates across multiple components.
- **ReplaySubject**: Caching the last few values (e.g., a stream of user activity or logs) to ensure new subscribers always have the latest data.
- **BehaviorSubject**: Managing and sharing application state (e.g., form data, user authentication status) that needs to be available immediately to all subscribers.

34. Differences Between Shared Component and Shared Module?

Feature	Shared Component	Shared Module
Purpose	Reusable, standalone components	A module that encapsulates multiple shared resources
What it Contains	Single UI component or feature (e.g., a button, modal)	Group of components, directives, pipes, and services
Usage	Used directly in templates of other components or modules	Imported into other modules to provide shared resources
Scope	Scope limited to that specific component	Scope is wider, can share multiple resources across the app
Typical Use Case	UI or functionality that needs to be reused across the app	Grouping commonly used components, pipes, or services for reuse

35. Purpose of main.ts in Angular

- main.ts is the **entry point** of an Angular application.
- It’s responsible for **bootstrapping the root module** (usually AppModule) and starting the application in the browser.

Key Responsibilities:

- **Bootstraps Angular** using platformBrowserDynamic().
- Loads the root module (AppModule) to kick off the app.
- Typically enables **production mode** if applicable.

✅ Interview-style Answer

- "main.ts is the entry point of an Angular application. It bootstraps the AppModule using platformBrowserDynamic(), initializes the app in the browser, and optionally enables production mode. It’s the first file that runs when the app loads."

36. Observable vs Subject

An **Observable** is unicast and passive; it emits values only when subscribed to, and each subscriber gets a separate stream. A **Subject** is both an observer and observable — it's multicast and actively emits values using next(), which are shared across all subscribers.

Feature	**Observable**	**Subject**
Type	Only an observable	Both observable and observer
Execution	Unicast (each subscriber gets own stream)	Multicast (shared stream for all)
Emits Values	Emitted by the Observable internally	Emitted manually using next()
Use Case	HTTP requests, async streams	Broadcasting events, state management
Subscription	Independent per subscriber	Shared among all subscribers

- **Miscellaneous Questions**

**** How to check web application score?**

“To evaluate a web app’s performance, I commonly use **Google Lighthouse** and **PageSpeed Insights**. They provide key metrics like **First Contentful Paint**, **Time to Interactive**, and **Core Web Vitals**, helping identify performance bottlenecks on both desktop and mobile.”

Use the **"Performance" tab** in Chrome DevTools to record and analyze frame rendering, scripting time, and layout shifts in real-time.

Google Lighthouse (Built into Chrome DevTools)

Steps:

- Open your web app in **Google Chrome**.
- Press F12 or right-click > **Inspect** to open **DevTools**.
- Go to the **"Lighthouse"** tab.
- Select the desired categories (Performance, SEO, Accessibility, etc.).
- Click **"Analyze page load"**.

Result:
You’ll get a detailed report with scores from **0–100** for:

- Performance
- Accessibility
- Best Practices
- SEO
- PWA (Progressive Web App)

. **Creating package.json using npm

"package.json is created using npm init. It defines the project's metadata, dependencies, scripts, and more. I typically use npm init -y for fast setup during POCs or demos."

. JIT VS AOT

"JIT compiles Angular code in the browser at runtime, suitable for development due to fast rebuilds. AOT compiles code at build time, making it ideal for production with faster startup, smaller bundle size, and earlier error detection."

. Testing in Angular

"Angular testing includes unit, integration, and E2E testing. I use **Jasmine + Karma** for unit tests and **TestBed** for setting up test environments. For E2E, **Cypress** is preferred. I also ensure proper **test coverage** using Angular CLI."