ANGULAR BASICS ANSWERS

---

1. What is Angular?

Answer:

- **Angular** is a **TypeScript-based open-source** front-end framework
- Developed and maintained by **Google**
- Used for building **Single Page Applications (SPAs)**
- Provides a complete solution with routing, forms, HTTP client, and more
- Uses **component-based architecture**
- Supports **two-way data binding**, **dependency injection**, and **modular development**
- Current versions are called **Angular** (2+), distinct from AngularJS (1.x)

Theoretical Keywords:

TypeScript, **SPA**, **Component-based**, **Google**, **Two-way binding**,
Dependency Injection, **Modular architecture**, **Cross-platform**

Example:

```
// app.component.ts - Basic Angular Component
import { Component } from '@angular/core';

@Component({
  selector: 'app-root',
  template: `
    <h1>{{ title }}</h1>
    <p>Welcome to Angular!</p>
  `,
  styles: [`h1 { color: blue; }`]
})
export class AppComponent {
  title = 'My Angular App';
}
```

---

2. Difference between AngularJS and Angular

Answer:

Feature	AngularJS (1.x)	Angular (2+)
Language	JavaScript	TypeScript
Architecture	MVC-based	Component-based
Mobile Support	Not built for mobile	Mobile-friendly
Data Binding	Two-way with $scope	Two-way with [(ngModel)]
Dependency Injection	Basic	Hierarchical DI
CLI	None	Angular CLI
Performance	Slower (dirty checking)	Faster (change detection)
Routing	ngRoute	@angular/router

Theoretical Keywords:

MVC vs Components, **TypeScript vs JavaScript**, **Performance**,
Dirty checking vs Change detection, **Mobile support**, **CLI**

---

3. What are components?

Answer:

- **Components** are the **building blocks** of Angular applications
- Each component has its own **template**, **styles**, and **logic**
- Defined using `@Component` decorator
- Contains: **selector**, **template/templateUrl**, **styles/styleUrls**
- Components can be **nested** to create complex UIs
- Encapsulates view and behavior for a **specific UI element**

Theoretical Keywords:

Building blocks, **@Component decorator**, **Encapsulation**,
Template, **Selector**, **View + Logic**, **Reusable**

Example:

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

@Component({
  selector: 'app-user-card',
  templateUrl: './user-card.component.html',
  styleUrls: ['./user-card.component.css']
})
export class UserCardComponent {
  userName: string = 'John Doe';
  userEmail: string = 'john@example.com';

  onEdit(): void {
    console.log('Editing user...');
  }
}
```

---

4. What is a module?

Answer:

- **Module** is a **container** that groups related components, services, pipes, and directives
- Defined using `@NgModule` decorator
- Every Angular app has at least one module: **AppModule** (root module)
- Modules help organize code into **cohesive blocks** of functionality
- Can be **eagerly** or **lazily** loaded
- Types: Root module, Feature modules, Shared modules, Core modules

Theoretical Keywords:

@NgModule, **Container**, **Organization**, **Lazy loading**,
Feature modules, **Declarations**, **Imports**, **Exports**

Example:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { UserModule } from './user/user.module';

@NgModule({
  declarations: [AppComponent],  // Components, directives, pipes
  imports: [BrowserModule, UserModule],  // Other modules
  providers: [],  // Services
  bootstrap: [AppComponent]  // Root component
})
export class AppModule { }
```

---

5. What is a service?

Answer:

- **Service** is a class that contains **business logic** and **data operations**
- Used for **code reusability** and **separation of concerns**
- Defined using `@Injectable` decorator
- Injected into components via **Dependency Injection**
- Typically used for: API calls, data sharing, utility functions
- Can be **singleton** (providedIn: 'root') or **scoped** to module/component

Theoretical Keywords:

@Injectable, **Business logic**, **Dependency Injection**, **Singleton**,
Reusability, **Separation of concerns**, **providedIn**

Example:

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

@Injectable({
  providedIn: 'root'  // Singleton service
})
export class UserService {
  private apiUrl = '/api/users';

  constructor(private http: HttpClient) {}

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>(this.apiUrl);
  }

  getUserById(id: number): Observable<User> {
    return this.http.get<User>(`${this.apiUrl}/${id}`);
```

```
  }
}
```

---

6. What is a directive?

Answer:

- **Directives** are classes that **modify DOM elements** or component behavior
- Three types of directives:
  1. **Component directives**: Components with templates
  2. **Structural directives**: Change DOM layout (`*ngIf`, `*ngFor`)
  3. **Attribute directives**: Change appearance/behavior (`ngClass`, `ngStyle`)
- Defined using `@Directive` decorator
- Can create **custom directives** for specific functionality

Theoretical Keywords:

DOM manipulation, **Structural directives**, **Attribute directives**,
@Directive decorator, **Custom directives**, **Behavior modification**

Example:

```
// Custom attribute directive
import { Directive, ElementRef, HostListener } from '@angular/core';

@Directive({
  selector: '[appHighlight]'
})
export class HighlightDirective {
  constructor(private el: ElementRef) {}

  @HostListener('mouseenter')
  onMouseEnter(): void {
    this.el.nativeElement.style.backgroundColor = 'yellow';
  }

  @HostListener('mouseleave')
  onMouseLeave(): void {
    this.el.nativeElement.style.backgroundColor = '';
  }
}

// Usage: <p appHighlight>Hover over me!</p>
```

---

7. What is data binding?

Answer:

- **Data binding** is the **synchronization** between component and view
- Allows **automatic update** of view when data changes and vice versa
- Angular supports **four types** of data binding
- Eliminates need for manual DOM manipulation
- Makes code more **declarative** and **maintainable**

Theoretical Keywords:

Synchronization, **Component-View connection**, **Automatic updates**,
Declarative UI, **One-way**, **Two-way binding**

---

8. Types of data binding in Angular

Answer:

Angular has **four types** of data binding:

Type	Syntax	Direction	Description
Interpolation	`{{ value }}`	Component → View	Display component data
Property Binding	`[property]="value"`	Component → View	Bind to element property
Event Binding	`(event)="handler()"`	View → Component	Listen to DOM events
Two-way Binding	`[(ngModel)]="value"`	Both ways	Sync data both directions

Theoretical Keywords:

Interpolation, **Property binding**, **Event binding**, **Two-way binding**,
ngModel, **One-way vs Two-way**, **Data flow**

Example:

```
<!-- 1. Interpolation: Component → View -->
<h1>{{ title }}</h1>
<p>User: {{ user.name }}</p>

<!-- 2. Property Binding: Component → View -->
<img [src]="imageUrl">
<button [disabled]="isDisabled">Click</button>
<div [ngClass]="{'active': isActive}"></div>

<!-- 3. Event Binding: View → Component -->
<button (click)="onClick()">Click Me</button>
<input (input)="onInput($event)">
<form (submit)="onSubmit()">

<!-- 4. Two-way Binding: Both Directions -->
<input [(ngModel)]="username">
<!-- Equivalent to: -->
<input [ngModel]="username" (ngModelChange)="username = $event">
```

---

9. What is interpolation?

Answer:

- **Interpolation** displays component data in the template
- Uses **double curly braces** syntax: `{{ expression }}`
- Evaluates the expression and converts to **string**
- Supports **expressions**, **method calls**, and **pipes**
- One-way binding: **Component → View**
- Cannot use: assignments, new, chaining, increment/decrement

Theoretical Keywords:

Double curly braces, **Expression evaluation**, **String conversion**, **One-way binding**, **Template expressions**, **Pipes support**

Example:

```
@Component({
  selector: 'app-demo',
  template: `
    <!-- Simple interpolation -->
    <h1>{{ title }}</h1>

    <!-- Object property -->
    <p>Name: {{ user.name }}</p>

    <!-- Expression -->
    <p>Total: {{ price * quantity }}</p>

    <!-- Method call -->
    <p>Greeting: {{ getGreeting() }}</p>

    <!-- With pipe -->
    <p>Date: {{ today | date:'fullDate' }}</p>
    <p>Price: {{ price | currency:'USD' }}</p>

    <!-- Ternary operator -->
    <p>Status: {{ isActive ? 'Active' : 'Inactive' }}</p>
  `
})
export class DemoComponent {
  title = 'Welcome';
  user = { name: 'John', age: 25 };
  price = 99.99;
  quantity = 2;
  today = new Date();
  isActive = true;

  getGreeting(): string {
    return `Hello, ${this.user.name}!`;
  }
}
```

---

10. What are templates?

Answer:

- **Templates** define the **view** of a component
- Written in **HTML** with Angular-specific syntax
- Can be **inline** (template) or **external** (templateUrl)
- Contains: HTML, directives, bindings, pipes
- Angular compiles templates into **JavaScript** (AOT/JIT)
- Templates are **type-checked** with TypeScript in strict mode

Theoretical Keywords:

View definition, **HTML + Angular syntax**, **Inline vs external**, **Compilation**, **Type checking**, **Template syntax**

Example:

```
// Inline template
@Component({
  selector: 'app-inline',
  template: `
    <div class="container">
      <h1>{{ title }}</h1>
      <ul>
        <li *ngFor="let item of items">{{ item }}</li>
      </ul>
    </div>
  `
})
export class InlineComponent { }

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

```
<!-- external.component.html -->
<div class="container">
  <header>
    <h1>{{ title }}</h1>
  </header>

  <main>
    <ng-container *ngIf="isLoaded; else loading">
      <app-content [data]="contentData"></app-content>
    </ng-container>

    <ng-template #loading>
      <p>Loading...</p>
    </ng-template>
  </main>
</div>
```

---

11. What is the purpose of `@NgModule`?

Answer:

- **@NgModule** is a decorator that defines an **Angular module**
- Configures the **compilation context** for components
- Has several important properties:
  - **declarations**: Components, directives, pipes in this module
  - **imports**: Other modules whose exports are needed
  - **exports**: Components/directives/pipes available to other modules
  - **providers**: Services available for injection
  - **bootstrap**: Root component (only in AppModule)

Theoretical Keywords:

Module decorator, **Compilation context**, **declarations**, **imports**, **exports**, **providers**, **bootstrap**

Example:

```
import { NgModule } from '@angular/core';
import { CommonModule } from '@angular/common';
import { FormsModule, ReactiveFormsModule } from '@angular/forms';

// Components
import { UserListComponent } from './user-list/user-list.component';
import { UserDetailComponent } from './user-detail/user-detail.component';
import { UserFormComponent } from './user-form/user-form.component';

// Services
import { UserService } from './services/user.service';

// Pipes
import { UserFilterPipe } from './pipes/user-filter.pipe';

@NgModule({
  // What belongs to this module
  declarations: [
    UserListComponent,
    UserDetailComponent,
    UserFormComponent,
    UserFilterPipe
```

```
  ],

  // What this module needs from other modules
  imports: [
    CommonModule,
    FormsModule,
    ReactiveFormsModule
  ],

  // What this module exposes to other modules
  exports: [
    UserListComponent,
    UserDetailComponent
  ],

  // Services scoped to this module
  providers: [
    UserService
  ]
})
export class UserModule { }
```

COMPONENTS AND TEMPLATES ANSWERS

---

12. What is a component in Angular?

Answer:

- A **component** is a **TypeScript class** decorated with `@Component`
- Controls a **patch of screen** called a view
- Consists of three parts:
    1. **Class** (.ts): Logic and data
    2. **Template** (.html): View/UI
    3. **Styles** (.css/.scss): Component-specific styles
- Components are **reusable** and **composable**
- Has its own **lifecycle** managed by Angular

Theoretical Keywords:

@Component decorator, **View controller**, **Class + Template + Styles**, **Encapsulation**, **Reusability**, **Lifecycle hooks**

Example:

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

@Component({
  selector: 'app-product-card',  // HTML tag name
  templateUrl: './product-card.component.html',
  styleUrls: ['./product-card.component.scss'],
  // Or inline:
  // template: `<div>{{ product.name }}</div>`,
  // styles: [`.card { padding: 10px; }`]
})
export class ProductCardComponent implements OnInit {
  // Properties (data)
  product: Product = {
    id: 1,
    name: 'Laptop',
    price: 999.99
  };

  isExpanded = false;

  // Lifecycle hook
  ngOnInit(): void {
    console.log('Component initialized');
  }

  // Methods (behavior)
  toggleExpand(): void {
    this.isExpanded = !this.isExpanded;
  }

  addToCart(): void {
    console.log('Added to cart:', this.product);
  }
}
```

---

13. Difference between component and directive

Answer:

Feature	Component	Directive
Decorator	`@Component`	`@Directive`
Template	Required (has view)	No template
Purpose	Build UI blocks	Modify existing elements
Selector	Element selector	Attribute/class selector
Usage	`<app-header>`	`<div appHighlight>`
Shadow DOM	Optional encapsulation	No encapsulation

Theoretical Keywords:

Template presence, **UI building vs DOM modification**, **Element vs Attribute selector**, **View encapsulation**

Example:

```
// COMPONENT - Has its own view
@Component({
  selector: 'app-button',
  template: `
    <button [class]="buttonClass" (click)="onClick()">
      <ng-content></ng-content>
    </button>
  `,
  styles: [`button { padding: 10px 20px; }`]
})
export class ButtonComponent {
  @Input() buttonClass = 'primary';
  @Output() buttonClick = new EventEmitter<void>();

  onClick(): void {
    this.buttonClick.emit();
  }
}

// DIRECTIVE - Modifies existing elements
@Directive({
  selector: '[appTooltip]'
})
export class TooltipDirective {
  @Input() appTooltip = '';

  constructor(private el: ElementRef) {}

  @HostListener('mouseenter')
  showTooltip(): void {
    // Add tooltip to existing element
  }

  @HostListener('mouseleave')
  hideTooltip(): void {
    // Remove tooltip
  }
}

// Usage:
// <app-button>Click Me</app-button>  <!-- Component -->
// <span appTooltip="Help text">Hover me</span>  <!-- Directive -->
```

---

14. How to pass data from parent to child component?

Answer:

- Use **`@Input()` decorator** in child component
- Parent passes data through **property binding**
- Data flows **one-way**: Parent → Child
- Can use **setter** or **ngOnChanges** to react to input changes
- Supports any data type: primitives, objects, arrays

Theoretical Keywords:

@Input decorator, **Property binding**, **One-way data flow**, **Parent-child communication**, **Input setter**, **ngOnChanges**

Example:

```
// child.component.ts
import { Component, Input, OnChanges, SimpleChanges } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <div class="child">
      <h3>{{ title }}</h3>
      <p>User: {{ user?.name }}</p>
      <ul>
```

```
        <li *ngFor="let item of items">{{ item }}</li>
      </ul>
    </div>
  `
})
export class ChildComponent implements OnChanges {
  // Simple input
  @Input() title: string = '';

  // Object input
  @Input() user: User | null = null;

  // Array input
  @Input() items: string[] = [];

  // Input with setter (react to changes)
  private _count = 0;
  @Input()
  set count(value: number) {
    this._count = value;
    console.log('Count changed to:', value);
  }
  get count(): number {
    return this._count;
  }

  // React to changes using lifecycle hook
  ngOnChanges(changes: SimpleChanges): void {
    if (changes['user']) {
      console.log('User changed:', changes['user'].currentValue);
    }
  }
}

// parent.component.ts
@Component({
  selector: 'app-parent',
  template: `
    <app-child
      [title]="pageTitle"
      [user]="currentUser"
      [items]="menuItems"
      [count]="counter">
    </app-child>

    <button (click)="counter = counter + 1">Increment</button>
  `
})
export class ParentComponent {
  pageTitle = 'Welcome Page';
  currentUser = { id: 1, name: 'John' };
  menuItems = ['Home', 'About', 'Contact'];
  counter = 0;
}
```

---

15. How to pass data from child to parent component?

Answer:

- Use **`@Output()` decorator** with **EventEmitter**
- Child **emits events** that parent listens to
- Parent uses **event binding** to receive data
- Data flows: **Child → Parent**
- Can pass any data type with the event

Theoretical Keywords:

@Output decorator, **EventEmitter**, **Event binding**,
Child-parent communication, **Custom events**, **emit()**

Example:

```
// child.component.ts
import { Component, Output, EventEmitter } from '@angular/core';

@Component({
  selector: 'app-child',
  template: `
    <div class="child">
      <input [(ngModel)]="inputValue" placeholder="Enter text">
      <button (click)="sendData()">Send to Parent</button>
      <button (click)="notifyParent()">Notify</button>

      <div class="item-list">
        <div *ngFor="let item of items" (click)="selectItem(item)">
          {{ item.name }}
        </div>
      </div>
    </div>
  `
```

```
})
export class ChildComponent {
  inputValue = '';
  items = [
    { id: 1, name: 'Item 1' },
    { id: 2, name: 'Item 2' }
  ];

  // Output with data
  @Output() dataSubmit = new EventEmitter<string>();

  // Output without data (just notification)
  @Output() notify = new EventEmitter<void>();

  // Output with object
  @Output() itemSelected = new EventEmitter<{ id: number; name: string }>();

  sendData(): void {
    this.dataSubmit.emit(this.inputValue);
  }

  notifyParent(): void {
    this.notify.emit();
  }

  selectItem(item: { id: number; name: string }): void {
    this.itemSelected.emit(item);
  }
}

// parent.component.ts
@Component({
  selector: 'app-parent',
  template: `
    <h2>Parent Component</h2>
    <p>Received: {{ receivedData }}</p>
    <p>Selected: {{ selectedItem?.name }}</p>

    <app-child
      (dataSubmit)="onDataReceived($event)"
      (notify)="onNotification()"
      (itemSelected)="onItemSelected($event)">
    </app-child>
  `
})
export class ParentComponent {
  receivedData = '';
  selectedItem: { id: number; name: string } | null = null;

  onDataReceived(data: string): void {
    this.receivedData = data;
    console.log('Data received from child:', data);
  }

  onNotification(): void {
    console.log('Notification received from child');
  }

  onItemSelected(item: { id: number; name: string }): void {
    this.selectedItem = item;
    console.log('Item selected:', item);
  }
}
```

---

16. What are input and output decorators?

Answer:

- **@Input()**: Allows parent to pass data **into** child component
- **@Output()**: Allows child to send data **out** to parent component
- Together they enable **component communication**
- @Input uses **property binding** `[property]="value"`
- @Output uses **event binding** `(event)="handler($event)"`

Theoretical Keywords:

Property binding, **Event binding**, **Data flow**,
Component communication, **Decorators**, **EventEmitter**

Example:

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

// Reusable Counter Component
@Component({
  selector: 'app-counter',
  template: `
    <div class="counter">
      <button (click)="decrement()">-</button>
      <span>{{ count }}</span>
```

```
      <button (click)="increment()">+</button>
    </div>
  `
})
export class CounterComponent {
  // INPUT: Receive initial value from parent
  @Input() count: number = 0;
  @Input() step: number = 1;
  @Input() min: number = 0;
  @Input() max: number = 100;

  // OUTPUT: Notify parent of changes
  @Output() countChange = new EventEmitter<number>();
  @Output() limitReached = new EventEmitter<'min' | 'max'>();

  increment(): void {
    if (this.count < this.max) {
      this.count += this.step;
      this.countChange.emit(this.count);
    } else {
      this.limitReached.emit('max');
    }
  }

  decrement(): void {
    if (this.count > this.min) {
      this.count -= this.step;
      this.countChange.emit(this.count);
    } else {
      this.limitReached.emit('min');
    }
  }
}

// Parent using the counter
@Component({
  selector: 'app-product',
  template: `
    <h3>{{ product.name }}</h3>
    <p>Quantity:</p>
    <app-counter
      [count]="quantity"
      [step]="1"
      [min]="1"
      [max]="product.stock"
      (countChange)="onQuantityChange($event)"
      (limitReached)="onLimitReached($event)">
    </app-counter>
    <p>Total: {{ product.price * quantity | currency }}</p>
  `
})
export class ProductComponent {
  product = { name: 'Laptop', price: 999, stock: 10 };
  quantity = 1;

  onQuantityChange(newQuantity: number): void {
    this.quantity = newQuantity;
  }

  onLimitReached(limit: 'min' | 'max'): void {
    if (limit === 'max') {
      alert('Maximum stock reached!');
    }
  }
}
```

---

17. What is `@ViewChild` and `@ContentChild`?

Answer:

- **@ViewChild**: Access child component/element defined in **component's own template**
- **@ContentChild**: Access content **projected** into component via `<ng-content>`
- Both provide **reference** to child component, directive, or DOM element
- Available after `ngAfterViewInit` and `ngAfterContentInit` respectively
- Use `{ static: true }` if needed in `ngOnInit`

Theoretical Keywords:

Template query, **Content projection**, **Component reference**,
DOM access, **AfterViewInit**, **AfterContentInit**, **ng-content**

Example:

```
// ======================================
// @ViewChild - Access elements in own template
// ======================================

@Component({
  selector: 'app-parent',
  template: `
```

```
    <!-- Access child component -->
    <app-child #childRef></app-child>

    <!-- Access DOM element -->
    <input #inputRef type="text">

    <button (click)="focusInput()">Focus Input</button>
    <button (click)="callChildMethod()">Call Child Method</button>
  `
})
export class ParentComponent implements AfterViewInit {
  // Access child component
  @ViewChild('childRef') childComponent!: ChildComponent;

  // Access DOM element
  @ViewChild('inputRef') inputElement!: ElementRef<HTMLInputElement>;

  // Access by component type
  @ViewChild(ChildComponent) child!: ChildComponent;

  ngAfterViewInit(): void {
    // ViewChild is available here
    console.log('Child component:', this.childComponent);
  }

  focusInput(): void {
    this.inputElement.nativeElement.focus();
  }

  callChildMethod(): void {
    this.childComponent.doSomething();
  }
}

// ========================================
// @ContentChild - Access projected content
// ========================================

@Component({
  selector: 'app-card',
  template: `
    <div class="card">
      <div class="card-header">
        <ng-content select="[card-header]"></ng-content>
      </div>
      <div class="card-body">
        <ng-content></ng-content>
      </div>
    </div>
  `
})
export class CardComponent implements AfterContentInit {
  // Access projected content
  @ContentChild('headerContent') headerContent!: ElementRef;
  @ContentChild(HeaderComponent) headerComponent!: HeaderComponent;

  ngAfterContentInit(): void {
    // ContentChild is available here
    console.log('Header content:', this.headerContent);
  }
}

// Usage of CardComponent with content projection
@Component({
  selector: 'app-page',
  template: `
    <app-card>
      <!-- This is projected content -->
      <h2 card-header #headerContent>Card Title</h2>
      <p>Card body content goes here...</p>
      <app-header></app-header>
    </app-card>
  `
})
export class PageComponent { }
```

ViewChildren and ContentChildren:

```
@Component({
  selector: 'app-list',
  template: `
    <div *ngFor="let item of items">
      <app-list-item #itemRef [data]="item"></app-list-item>
    </div>
  `
})
export class ListComponent implements AfterViewInit {
  items = ['Item 1', 'Item 2', 'Item 3'];

  // Access multiple children
  @ViewChildren('itemRef') listItems!: QueryList<ListItemComponent>;

  ngAfterViewInit(): void {
    // Array of all child components
    this.listItems.forEach(item => console.log(item));
```

```
    // Listen for changes
    this.listItems.changes.subscribe(() => {
      console.log('List items changed');
    });
  }
}
```

PROJECT CONFIGURATION FILES ANSWERS

---

angular.json

---

18. What is `angular.json`?

Answer:

- **angular.json** is the **workspace configuration** file for Angular CLI
- Contains all **project settings**, build configurations, and CLI defaults
- Located at the **root** of Angular workspace
- Defines how Angular CLI commands work (build, serve, test, lint)
- Manages **multiple projects** in a single workspace

Theoretical Keywords:

Workspace configuration, **Angular CLI**, **Build settings**,
Project management, **CLI defaults**, **Architect targets**

Example:

```
{
  "$schema": "./node_modules/@angular/cli/lib/config/schema.json",
  "version": 1,
  "newProjectRoot": "projects",
  "projects": {
    "my-app": {
      "projectType": "application",
      "root": "",
      "sourceRoot": "src",
      "prefix": "app",
      "architect": {
        "build": { },
        "serve": { },
        "test": { }
      }
    }
  },
  "defaultProject": "my-app"
}
```

---

19. What is its role in an Angular project?

Answer:

The **angular.json** file serves several critical purposes:

- **Build configuration**: Output paths, optimization, bundling
- **Development server**: Port, proxy, SSL settings
- **Asset management**: Static files, styles, scripts
- **Environment configuration**: Development vs production
- **Testing configuration**: Karma, test files
- **Multiple project management**: Monorepo support

Theoretical Keywords:

Build configuration, **Serve settings**, **Asset management**,
Environment handling, **Test configuration**, **Monorepo**

---

20. How does `angular.json` manage build and serve configurations?

Answer:

- Uses **architect targets** for different operations
- **build**: Compiles the application
- **serve**: Runs development server
- Each target has **options** (defaults) and **configurations** (overrides)

Example:

```
{
  "projects": {
    "my-app": {
      "architect": {
        "build": {
          "builder": "@angular-devkit/build-angular:browser",
          "options": {
            "outputPath": "dist/my-app",
            "index": "src/index.html",
            "main": "src/main.ts",
            "polyfills": "src/polyfills.ts",
            "tsConfig": "tsconfig.app.json",
            "aot": true,
            "assets": ["src/favicon.ico", "src/assets"],
            "styles": ["src/styles.scss"],
            "scripts": []
          },
          "configurations": {
            "production": {
              "budgets": [
                { "type": "initial", "maximumWarning": "500kb", "maximumError": "1mb" }
              ],
              "fileReplacements": [
                { "replace": "src/environments/environment.ts", "with": "src/environments/environment.prod.ts" }
              ],
              "outputHashing": "all",
              "optimization": true,
              "sourceMap": false
            },
            "development": {
              "buildOptimizer": false,
              "optimization": false,
              "sourceMap": true
            }
          },
          "defaultConfiguration": "production"
        },
        "serve": {
          "builder": "@angular-devkit/build-angular:dev-server",
          "options": {
            "browserTarget": "my-app:build",
            "port": 4200,
            "open": true
          },
          "configurations": {
            "production": {
              "browserTarget": "my-app:build:production"
            },
            "development": {
              "browserTarget": "my-app:build:development"
            }
          }
        }
      }
    }
  }
}
```

---

21. How do you configure environments in `angular.json`?

Answer:

- Use **fileReplacements** in build configurations
- Create environment files for each configuration
- Angular CLI swaps files during build

Example:

```
// angular.json
{
  "configurations": {
    "production": {
      "fileReplacements": [
        {
          "replace": "src/environments/environment.ts",
          "with": "src/environments/environment.prod.ts"
        }
      ]
    },
    "staging": {
      "fileReplacements": [
        {
          "replace": "src/environments/environment.ts",
          "with": "src/environments/environment.staging.ts"
        }
      ]
    }
  }
}
```

```
// src/environments/environment.ts (development)
export const environment = {
  production: false,
  apiUrl: 'http://localhost:3000/api'
};

// src/environments/environment.prod.ts
export const environment = {
  production: true,
  apiUrl: 'https://api.myapp.com'
};

// src/environments/environment.staging.ts
export const environment = {
  production: false,
  apiUrl: 'https://staging-api.myapp.com'
};

// Usage in application
import { environment } from '../environments/environment';

@Injectable()
export class ApiService {
  private apiUrl = environment.apiUrl;
}

// Build commands:
// ng build --configuration=production
// ng build --configuration=staging
```

---

22. How do you add assets and styles globally using `angular.json`?

Answer:

- **assets**: Static files copied to output directory
- **styles**: Global CSS/SCSS files
- **scripts**: Global JavaScript files

Example:

```
{
  "architect": {
    "build": {
      "options": {
        "assets": [
          "src/favicon.ico",
          "src/assets",
          {
            "glob": "**/*",
            "input": "node_modules/some-library/assets",
            "output": "/library-assets/"
          },
          {
            "glob": "manifest.json",
            "input": "src",
            "output": "/"
          }
        ],
        "styles": [
          "src/styles.scss",
          "node_modules/bootstrap/dist/css/bootstrap.min.css",
          {
            "input": "node_modules/@fortawesome/fontawesome-free/css/all.min.css",
            "bundleName": "fontawesome"
          }
        ],
        "scripts": [
          "node_modules/jquery/dist/jquery.min.js",
          {
            "input": "node_modules/chart.js/dist/chart.min.js",
            "bundleName": "charts",
            "inject": false
          }
        ]
      }
    }
  }
}
```

---

package.json

---

23. What is `package.json`?

Answer:

- **package.json** is the **Node.js project manifest** file

- Defines project **metadata** and **dependencies**
- Contains **scripts** for running tasks
- Required for **npm/yarn** package management
- Lists both runtime and development dependencies

Theoretical Keywords:

Project manifest, **Dependencies**, **Scripts**, **npm/yarn**, **Metadata**, **Package management**

Example:

```
{
  "name": "my-angular-app",
  "version": "1.0.0",
  "description": "My Angular Application",
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "build": "ng build",
    "test": "ng test",
    "lint": "ng lint"
  },
  "dependencies": {
    "@angular/core": "^17.0.0",
    "rxjs": "^7.8.0"
  },
  "devDependencies": {
    "@angular/cli": "^17.0.0",
    "typescript": "~5.2.0"
  }
}
```

---

24. Why is `package.json` important?

Answer:

- **Dependency tracking**: Lists all required packages
- **Version management**: Controls package versions
- **Script automation**: Custom commands for development
- **Project sharing**: Others can install dependencies with `npm install`
- **CI/CD integration**: Build systems use it to install dependencies

Theoretical Keywords:

Dependency tracking, **Version control**, **Automation**, **Reproducibility**, **CI/CD**, **Team collaboration**

---

25. How does `package.json` manage dependencies?

Answer:

- Uses **semantic versioning** (semver)
- ^ prefix: Allow minor updates (^1.2.3 → 1.x.x)
- ~ prefix: Allow patch updates (~1.2.3 → 1.2.x)
- **Exact**: No prefix (1.2.3 → only 1.2.3)
- **package-lock.json**: Locks exact versions installed

Example:

```
{
  "dependencies": {
    "exact-version": "1.2.3",
    "minor-updates": "^1.2.3",
    "patch-updates": "~1.2.3",
    "any-version": "*",
    "range": ">=1.0.0 <2.0.0"
  }
}
```

---

26. What are `devDependencies` and `dependencies`?

Answer:

Type	Purpose	Example	Bundled in Production
dependencies	Required at runtime	@angular/core, rxjs	Yes
devDependencies	Development only	TypeScript, Karma, ESLint	No

Example:

```
{
  "dependencies": {
    "@angular/core": "^17.0.0",
    "@angular/common": "^17.0.0",
    "@angular/router": "^17.0.0",
    "rxjs": "^7.8.0",
    "zone.js": "^0.14.0"
  },
  "devDependencies": {
    "@angular/cli": "^17.0.0",
    "@angular/compiler-cli": "^17.0.0",
    "@types/node": "^20.0.0",
    "typescript": "~5.2.0",
    "karma": "^6.4.0",
    "jasmine-core": "^5.0.0",
    "eslint": "^8.0.0"
  }
}
```

---

27. What are scripts in `package.json` and how are they used?

Answer:

- **Scripts** are custom commands defined in package.json
- Run using `npm run <script-name>`
- Some scripts have shortcuts: `npm start`, `npm test`
- Can chain commands and pass arguments

Example:

```
{
  "scripts": {
    "ng": "ng",
    "start": "ng serve",
    "start:prod": "ng serve --configuration=production",
    "start:ssl": "ng serve --ssl --ssl-key ./ssl/server.key --ssl-cert ./ssl/server.crt",
    "build": "ng build",
    "build:prod": "ng build --configuration=production",
    "build:analyze": "ng build --stats-json && webpack-bundle-analyzer dist/my-app/stats.json",
    "test": "ng test",
    "test:ci": "ng test --no-watch --code-coverage",
    "lint": "ng lint",
    "lint:fix": "ng lint --fix",
    "e2e": "ng e2e",
    "format": "prettier --write \"src/**/*.{ts,html,scss}\"",
    "precommit": "npm run lint && npm run test:ci",
    "deploy": "npm run build:prod && firebase deploy"
  }
}
```

---

28. How to run custom npm scripts defined in `package.json`?

Answer:

```
# Standard scripts (shortcuts available)
npm start          # Runs "start" script
npm test           # Runs "test" script

# Custom scripts (require "run")
npm run build:prod
npm run lint:fix
npm run deploy

# Pass arguments
npm run start -- --port=4300
npm run build -- --configuration=staging

# Run multiple scripts
npm run lint && npm run test

# Using npx for one-off commands
npx ng generate component my-component
```

---

tsconfig.json

---

29. What is `tsconfig.json`?

Answer:

- **tsconfig.json** is the **TypeScript compiler configuration** file

- Defines how TypeScript code is **compiled to JavaScript**
- Contains **compiler options**, **file inclusion/exclusion**
- Angular has multiple tsconfig files for different purposes

Example:

```
{
  "compileOnSave": false,
  "compilerOptions": {
    "baseUrl": "./",
    "outDir": "./dist/out-tsc",
    "forceConsistentCasingInFileNames": true,
    "strict": true,
    "noImplicitOverride": true,
    "noPropertyAccessFromIndexSignature": true,
    "noImplicitReturns": true,
    "noFallthroughCasesInSwitch": true,
    "sourceMap": true,
    "declaration": false,
    "downlevelIteration": true,
    "experimentalDecorators": true,
    "moduleResolution": "node",
    "importHelpers": true,
    "target": "ES2022",
    "module": "ES2022",
    "lib": ["ES2022", "dom"]
  },
  "angularCompilerOptions": {
    "enableI18nLegacyMessageIdFormat": false,
    "strictInjectionParameters": true,
    "strictInputAccessModifiers": true,
    "strictTemplates": true
  }
}
```

---

30. What compiler options are most important?

Answer:

Option	Purpose
target	Output JavaScript version
module	Module code generation
strict	Enable all strict options
baseUrl	Base for module resolution
paths	Path mapping for imports
experimentalDecorators	Enable decorators
sourceMap	Generate source maps

Example:

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ES2022",
    "strict": true,
    "baseUrl": "src",
    "paths": {
      "@app/*": ["app/*"],
      "@core/*": ["app/core/*"],
      "@shared/*": ["app/shared/*"],
      "@env/*": ["environments/*"]
    },
    "experimentalDecorators": true,
    "emitDecoratorMetadata": true,
    "sourceMap": true
  }
}
```

---

31. What does strict mode do?

Answer:

strict: true enables all strict type-checking options:

- `strictNullChecks`: null/undefined handled explicitly
- `noImplicitAny`: Must specify types (no implicit any)
- `strictFunctionTypes`: Strict function type checking
- `strictBindCallApply`: Check bind, call, apply
- `strictPropertyInitialization`: Class properties must be initialized

Example:

```
// With strict mode enabled:

// strictNullChecks
let name: string;
// name.toUpperCase();  // Error: possibly undefined

// noImplicitAny
function greet(name) {  // Error: parameter has implicit any
  return `Hello, ${name}`;
}

// strictPropertyInitialization
class User {
  name: string;  // Error: not initialized

  constructor() {
    // Must initialize here or use !
  }
}

// Correct version
class UserCorrect {
  name: string;

  constructor(name: string) {
    this.name = name;
  }
}
```

---

32. Difference between `target` and `module`?

Answer:

Option	Purpose	Values
target	Output JS version	ES5, ES6, ES2020, ESNext
module	Module system	CommonJS, ES2020, ESNext

- **target**: Affects syntax (let, const, async/await)
- **module**: Affects import/export handling

```
{
  "compilerOptions": {
    "target": "ES2022",
    "module": "ES2022"
  }
}
```

---

33. How to enable `strictNullChecks` and `noImplicitAny`?

Answer:

```
{
  "compilerOptions": {
    "strict": true,
    // Or individually:
    "strictNullChecks": true,
    "noImplicitAny": true
  }
}
```

Webpack

---

34. What is Webpack?

Answer:

- **Webpack** is a **module bundler** for JavaScript applications
- Takes modules with dependencies and generates **static bundles**
- Handles: JavaScript, CSS, images, fonts, and more
- Uses **loaders** to transform files and **plugins** for optimization

Theoretical Keywords:

Module bundler, **Static assets**, **Loaders**, **Plugins**,
Code splitting, **Tree shaking**, **Hot Module Replacement**

---

35. How does Angular use Webpack internally?

Answer:

- Angular CLI uses Webpack **under the hood**
- Pre-configured through `@angular-devkit/build-angular`
- Handles: TypeScript compilation, bundling, optimization
- Features: Code splitting, lazy loading, tree shaking
- **Hidden by default** but can be customized

Theoretical Keywords:

Angular CLI, **Build system**, **Code splitting**,
Tree shaking, **AOT compilation**, **Dev server**

---

36. What is the role of loaders and plugins in Webpack?

Answer:

- **Loaders**: Transform files before bundling
- **Plugins**: Perform broader tasks (optimization, asset management)

Type	Purpose	Examples
Loaders	Transform source files	ts-loader, sass-loader, file-loader
Plugins	Bundle optimization	HtmlWebpackPlugin, MiniCssExtractPlugin

---

37. Difference between development and production build using Webpack?

Answer:

Feature	Development	Production
Mode	development	production
Source Maps	Full (inline)	Minimal/none
Optimization	Disabled	Enabled
Minification	No	Yes
Tree Shaking	No	Yes
Hot Reload	Enabled	Disabled

```
ng build                          # Production (default)
ng build --configuration=development  # Development
```

---

38. How can you customize Webpack in Angular?

Answer:

- Use **@angular-builders/custom-webpack**
- Create custom webpack configuration
- Merge with Angular's default config

Example:

```
npm install @angular-builders/custom-webpack --save-dev
```

```
// angular.json
{
  "architect": {
    "build": {
      "builder": "@angular-builders/custom-webpack:browser",
      "options": {
        "customWebpackConfig": {
          "path": "./webpack.config.js",
          "mergeRules": {
            "externals": "replace"
          }
        }
      }
    }
  }
}
```

```
// webpack.config.js
module.exports = {
  resolve: {
    alias: {
      '@components': path.resolve(__dirname, 'src/app/components')
    }
  },
  plugins: [
    new webpack.DefinePlugin({
      'APP_VERSION': JSON.stringify(require('./package.json').version)
```

```
    })
  ]
};
```

DIRECTIVES AND PIPES ANSWERS

---

39. What are directives in Angular?

Answer:

- **Directives** are classes that add behavior to elements in Angular templates
- Allow you to **manipulate the DOM** declaratively
- Three types: **Component**, **Structural**, **Attribute**

Theoretical Keywords:

DOM manipulation, **Template behavior**, **Component directive**,
Structural directive, **Attribute directive**, **Declarative**

Example:

```
// Built-in directives usage
@Component({
  template: `
    <!-- Structural Directives -->
    <div *ngIf="isVisible">Conditional content</div>
    <div *ngFor="let item of items">{{ item }}</div>
    <div [ngSwitch]="status">
      <span *ngSwitchCase="'active'">Active</span>
      <span *ngSwitchDefault>Unknown</span>
    </div>

    <!-- Attribute Directives -->
    <div [ngClass]="{'active': isActive, 'disabled': !isEnabled}">Styled</div>
    <div [ngStyle]="{'color': textColor, 'font-size': fontSize + 'px'}">Styled</div>
  `
})
export class AppComponent { }
```

---

40. What are the types of directives?

Answer:

Type	Purpose	Example
Component	Directive with template	`@Component()`
Structural	Change DOM layout	`*ngIf`, `*ngFor`, `*ngSwitch`
Attribute	Change appearance/behavior	`[ngClass]`, `[ngStyle]`, custom

Example:

```
// 1. Component Directive (with template)
@Component({
  selector: 'app-card',
  template: '<div class="card"><ng-content></ng-content></div>'
})
export class CardComponent { }

// 2. Structural Directive (changes DOM structure)
@Directive({
  selector: '[appUnless]',
  standalone: true
})
export class UnlessDirective {
  private hasView = false;

  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }

  @Input() set appUnless(condition: boolean) {
    if (!condition && !this.hasView) {
      this.viewContainer.createEmbeddedView(this.templateRef);
      this.hasView = true;
    } else if (condition && this.hasView) {
      this.viewContainer.clear();
      this.hasView = false;
    }
  }
}
```

```
// 3. Attribute Directive (changes appearance)
@Directive({
  selector: '[appHighlight]',
  standalone: true
})
export class HighlightDirective {
  @Input() appHighlight = 'yellow';

  constructor(private el: ElementRef) { }

  @HostListener('mouseenter') onMouseEnter() {
    this.el.nativeElement.style.backgroundColor = this.appHighlight;
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.el.nativeElement.style.backgroundColor = '';
  }
}
```

---

41. Difference between `*ngIf` and `[hidden]`?

Answer:

Feature	*ngIf	[hidden]
DOM	Removes element	Hides with CSS
Initialization	Creates on true	Always created
Performance	Better for heavy content	Better for toggles
CSS Override	Not affected	Can be overridden
Lifecycle Hooks	Triggers on each render	Only once

Example:

```
@Component({
  template: `
    <!-- *ngIf: Element is removed/added to DOM -->
    <heavy-component *ngIf="isVisible"></heavy-component>

    <!-- [hidden]: Element stays in DOM, just hidden -->
    <div [hidden]="!isVisible">Quick toggle content</div>

    <!-- *ngIf with else -->
    <div *ngIf="user; else noUser">
      Welcome, {{ user.name }}!
    </div>
    <ng-template #noUser>
      <div>Please log in</div>
    </ng-template>
  `
})
export class ComparisonComponent {
  isVisible = true;
  user: User | null = null;
}
```

---

42. What are custom directives and how do you create them?

Answer:

- Custom directives extend Angular's functionality
- Use `@Directive` decorator
- Can be **attribute** or **structural** directives

Example:

```
// Attribute Directive: Auto-focus
@Directive({
  selector: '[appAutoFocus]',
  standalone: true
})
export class AutoFocusDirective implements AfterViewInit {
  constructor(private el: ElementRef) { }

  ngAfterViewInit(): void {
    this.el.nativeElement.focus();
  }
}

// Attribute Directive: Tooltip
@Directive({
  selector: '[appTooltip]',
  standalone: true
})
export class TooltipDirective {
  @Input('appTooltip') tooltipText = '';
  private tooltipElement: HTMLElement | null = null;
```

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

  @HostListener('mouseenter') onMouseEnter() {
    this.showTooltip();
  }

  @HostListener('mouseleave') onMouseLeave() {
    this.hideTooltip();
  }

  private showTooltip(): void {
    this.tooltipElement = this.renderer.createElement('span');
    this.renderer.addClass(this.tooltipElement, 'tooltip');
    this.renderer.appendChild(
      this.tooltipElement,
      this.renderer.createText(this.tooltipText)
    );
    this.renderer.appendChild(this.el.nativeElement, this.tooltipElement);
  }

  private hideTooltip(): void {
    if (this.tooltipElement) {
      this.renderer.removeChild(this.el.nativeElement, this.tooltipElement);
      this.tooltipElement = null;
    }
  }
}

// Structural Directive: Repeat
@Directive({
  selector: '[appRepeat]',
  standalone: true
})
export class RepeatDirective {
  constructor(
    private templateRef: TemplateRef<any>,
    private viewContainer: ViewContainerRef
  ) { }

  @Input() set appRepeat(times: number) {
    this.viewContainer.clear();
    for (let i = 0; i < times; i++) {
      this.viewContainer.createEmbeddedView(this.templateRef, {
        $implicit: i,
        index: i
      });
    }
  }
}

// Usage
@Component({
  template: `
    <input appAutoFocus />
    <button [appTooltip]="'Click to submit'">Submit</button>
    <div *appRepeat="5; let i">Item {{ i + 1 }}</div>
  `
})
export class MyComponent { }
```

---

43. What are pipes?

Answer:

- **Pipes** transform data in templates
- Take input value and return transformed output
- Use the **pipe operator** `|`
- Can be **chained** and accept **parameters**

Theoretical Keywords:

Data transformation, **Template formatting**, **Pure pipes**,
Impure pipes, **Pipe operator**, **Chaining**

Example:

```
@Component({
  template: `
    <!-- Built-in Pipes -->
    <p>{{ name | uppercase }}</p>
    <p>{{ name | lowercase }}</p>
    <p>{{ title | titlecase }}</p>
    <p>{{ price | currency:'USD':'symbol':'1.2-2' }}</p>
    <p>{{ percentage | percent:'1.0-2' }}</p>
    <p>{{ today | date:'fullDate' }}</p>
    <p>{{ today | date:'short' }}</p>
    <p>{{ data | json }}</p>
    <p>{{ items | slice:0:5 }}</p>
    <p>{{ observable$ | async }}</p>
```

```
    <!-- Chaining Pipes -->
    <p>{{ name | uppercase | slice:0:10 }}</p>
  `
})
export class PipeExampleComponent {
  name = 'john doe';
  title = 'hello world';
  price = 42.50;
  percentage = 0.854;
  today = new Date();
  data = { key: 'value' };
  items = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
  observable$ = of('async value');
}
```

---

44. How to create custom pipes?

Answer:

- Use `@Pipe` decorator
- Implement `PipeTransform` interface
- Define `transform()` method

Example:

```
// Simple transform pipe
@Pipe({
  name: 'truncate',
  standalone: true
})
export class TruncatePipe implements PipeTransform {
  transform(value: string, limit: number = 50, trail: string = '...'): string {
    if (!value) return '';
    if (value.length <= limit) return value;
    return value.substring(0, limit) + trail;
  }
}

// Filter pipe
@Pipe({
  name: 'filter',
  standalone: true
})
export class FilterPipe implements PipeTransform {
  transform<T>(items: T[], field: keyof T, value: any): T[] {
    if (!items || !field || value === undefined) return items;
    return items.filter(item => item[field] === value);
  }
}

// Time ago pipe
@Pipe({
  name: 'timeAgo',
  standalone: true
})
export class TimeAgoPipe implements PipeTransform {
  transform(value: Date | string): string {
    const date = new Date(value);
    const now = new Date();
    const seconds = Math.floor((now.getTime() - date.getTime()) / 1000);

    if (seconds < 60) return 'just now';
    if (seconds < 3600) return `${Math.floor(seconds / 60)} minutes ago`;
    if (seconds < 86400) return `${Math.floor(seconds / 3600)} hours ago`;
    if (seconds < 2592000) return `${Math.floor(seconds / 86400)} days ago`;
    return `${Math.floor(seconds / 2592000)} months ago`;
  }
}

// Safe HTML pipe
@Pipe({
  name: 'safeHtml',
  standalone: true
})
export class SafeHtmlPipe implements PipeTransform {
  constructor(private sanitizer: DomSanitizer) { }

  transform(value: string): SafeHtml {
    return this.sanitizer.bypassSecurityTrustHtml(value);
  }
}

// Usage
@Component({
  imports: [TruncatePipe, FilterPipe, TimeAgoPipe, SafeHtmlPipe],
  template: `
    <p>{{ longText | truncate:100:'...' }}</p>
    <div *ngFor="let user of users | filter:'role':'admin'">
      {{ user.name }}
    </div>
    <p>{{ createdAt | timeAgo }}</p>
    <div [innerHTML]="htmlContent | safeHtml"></div>
```

```
  `
})
export class CustomPipeComponent {
  longText = 'Lorem ipsum dolor sit amet...';
  users = [
    { name: 'John', role: 'admin' },
    { name: 'Jane', role: 'user' }
  ];
  createdAt = new Date('2024-01-01');
  htmlContent = '<strong>Bold text</strong>';
}
```

Pure vs Impure Pipes

Answer:

Type	Execution	Use Case
Pure	Only when input changes (reference)	Most transformations
Impure	Every change detection cycle	Arrays, objects that mutate

Example:

```
// Pure Pipe (default)
@Pipe({
  name: 'purePipe',
  pure: true  // Default
})
export class PurePipe implements PipeTransform {
  transform(value: any): any {
    console.log('Pure pipe executed');
    return value;
  }
}

// Impure Pipe
@Pipe({
  name: 'impurePipe',
  pure: false  // Runs on every change detection
})
export class ImpurePipe implements PipeTransform {
  transform(items: any[], filterBy: string): any[] {
    console.log('Impure pipe executed');
    return items.filter(item => item.includes(filterBy));
  }
}

// Usage
@Component({
  template: `
    <!-- Pure: Won't update when array items change -->
    <div *ngFor="let item of items | purePipe">{{ item }}</div>

    <!-- Impure: Updates on any change -->
    <div *ngFor="let item of items | impurePipe:'a'">{{ item }}</div>
  `
})
export class PipeComponent {
  items = ['apple', 'banana', 'cherry'];

  addItem() {
    this.items.push('apricot');  // Pure pipe won't detect
    // To trigger pure pipe, create new array:
    // this.items = [...this.items, 'apricot'];
  }
}
```

SERVICES AND DEPENDENCY INJECTION ANSWERS

---

45. What is a Service in Angular?

Answer:

- A **Service** is a class that encapsulates **reusable business logic**
- Used for **data sharing**, **API calls**, **state management**
- Follows **Single Responsibility Principle**
- Provided via **Dependency Injection**
- Not tied to any component's lifecycle

Theoretical Keywords:

Reusable logic, **Separation of concerns**, **Singleton**,
API calls, **State management**, **Injectable**

Example:

```
// User service
@Injectable({
  providedIn: 'root'  // Singleton across the app
})
export class UserService {
  private apiUrl = 'https://api.example.com/users';

  constructor(private http: HttpClient) { }

  getUsers(): Observable<User[]> {
    return this.http.get<User[]>(this.apiUrl);
  }

  getUserById(id: number): Observable<User> {
    return this.http.get<User>(`${this.apiUrl}/${id}`);
  }

  createUser(user: User): Observable<User> {
    return this.http.post<User>(this.apiUrl, user);
  }

  updateUser(id: number, user: Partial<User>): Observable<User> {
    return this.http.put<User>(`${this.apiUrl}/${id}`, user);
  }

  deleteUser(id: number): Observable<void> {
    return this.http.delete<void>(`${this.apiUrl}/${id}`);
  }
}

// Usage in component
@Component({
  selector: 'app-user-list',
  template: `
    <div *ngFor="let user of users">{{ user.name }}</div>
  `
})
export class UserListComponent implements OnInit {
  users: User[] = [];

  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.userService.getUsers().subscribe(users => {
      this.users = users;
    });
  }
}
```

---

46. What is Dependency Injection (DI)?

Answer:

- **Dependency Injection** is a design pattern where dependencies are **provided externally**
- Angular's DI system **creates and manages** service instances
- Classes **declare** what they need, Angular **provides** it
- Promotes **loose coupling** and **testability**

Theoretical Keywords:

Design pattern, **Inversion of Control**, **Loose coupling**,
Testability, **Injector**, **Provider**

Example:

```
// Without DI (tight coupling)
class UserComponent {
  private userService: UserService;

  constructor() {
    this.userService = new UserService(new HttpClient());  // Hard to test
  }
}

// With DI (loose coupling)
@Component({
  selector: 'app-user'
})
export class UserComponent {
  constructor(private userService: UserService) { }  // Injected
}

// Benefits demonstrated in testing
describe('UserComponent', () => {
  let component: UserComponent;
  let mockUserService: jasmine.SpyObj<UserService>;

  beforeEach(() => {
```

```
    mockUserService = jasmine.createSpyObj('UserService', ['getUsers']);

    TestBed.configureTestingModule({
      providers: [
        { provide: UserService, useValue: mockUserService }
      ]
    });

    component = TestBed.createComponent(UserComponent).componentInstance;
  });
});
```

---

47. How does the Angular Injector work?

Answer:

- **Injector** is a service locator that stores providers
- Creates **hierarchical tree** of injectors
- **Root injector** for app-wide singletons
- **Module injectors** for lazy-loaded modules
- **Element injectors** for components

Theoretical Keywords:

Hierarchical DI, **Root injector**, **Module injector**,
Element injector, **Provider resolution**, **Token**

Example:

```
// Provider registration levels

// 1. Root level (app-wide singleton)
@Injectable({
  providedIn: 'root'
})
export class AppService { }

// 2. Module level
@NgModule({
  providers: [ModuleService]  // Available to module
})
export class FeatureModule { }

// 3. Component level (new instance per component)
@Component({
  providers: [ComponentService]
})
export class MyComponent { }

// Injector hierarchy example
@Injectable()
export class LoggerService {
  log(message: string): void {
    console.log(message);
  }
}

// Custom injector usage
const injector = Injector.create({
  providers: [
    { provide: LoggerService, useClass: LoggerService }
  ]
});

const logger = injector.get(LoggerService);
logger.log('Hello');
```

---

48. What is `providedIn: 'root'`?

Answer:

- **providedIn: 'root'** makes service a **singleton** at root level
- Service is **tree-shakable** if not used
- No need to add to module's providers array
- Recommended approach for most services

Theoretical Keywords:

Root singleton, **Tree-shakable**, **Automatic registration**,
Application-wide, **Lazy loading friendly**

Example:

```
// providedIn options
@Injectable({
  providedIn: 'root'  // App-wide singleton, tree-shakable
})
export class GlobalService { }

@Injectable({
  providedIn: 'any'  // New instance per lazy module
})
export class AnyService { }

@Injectable({
  providedIn: 'platform'  // Shared across apps (micro frontends)
})
export class PlatformService { }

@Injectable()  // Must be added to providers manually
export class ManualService { }

// Tree-shaking benefit
// If GlobalService is never injected anywhere,
// it won't be included in the final bundle
```

---

49. How to inject services into components?

Answer:

- Use **constructor injection**
- Angular's DI resolves and provides instance
- Use **access modifiers** for automatic property creation
- Can use `inject()` function as alternative

Example:

```
// Method 1: Constructor injection (traditional)
@Component({
  selector: 'app-user'
})
export class UserComponent {
  constructor(
    private userService: UserService,
    private router: Router,
    private http: HttpClient
  ) { }

  loadUsers(): void {
    this.userService.getUsers().subscribe(/*...*/);
  }
}

// Method 2: inject() function (modern, Angular 14+)
@Component({
  selector: 'app-user'
})
export class UserComponent {
  private userService = inject(UserService);
  private router = inject(Router);

  loadUsers(): void {
    this.userService.getUsers().subscribe(/*...*/);
  }
}

// Method 3: @Inject decorator (for tokens)
@Component({
  selector: 'app-config'
})
export class ConfigComponent {
  constructor(
    @Inject(API_URL) private apiUrl: string,
    @Inject(DOCUMENT) private document: Document
  ) { }
}

// Method 4: Optional injection
@Component({
  selector: 'app-optional'
})
export class OptionalComponent {
  constructor(
    @Optional() private optionalService?: OptionalService
  ) {
    if (this.optionalService) {
      // Service was provided
    }
  }
}
```

Provider Types

Answer:

Different ways to configure providers:

Provider Type	Use Case
useClass	Provide different implementation
useValue	Provide static value
useFactory	Dynamic creation logic
useExisting	Alias to another provider

Example:

```
// useClass: Different implementation
@Injectable()
export class MockUserService {
  getUsers(): Observable<User[]> {
    return of([{ id: 1, name: 'Mock User' }]);
  }
}

providers: [
  { provide: UserService, useClass: MockUserService }
]

// useValue: Static value or configuration
const API_CONFIG = {
  baseUrl: 'https://api.example.com',
  timeout: 5000
};

export const API_TOKEN = new InjectionToken<typeof API_CONFIG>('api.config');

providers: [
  { provide: API_TOKEN, useValue: API_CONFIG }
]

// useFactory: Dynamic creation
export function loggerFactory(http: HttpClient, config: Config) {
  return config.production
    ? new ProductionLogger(http)
    : new DevelopmentLogger();
}

providers: [
  {
    provide: LoggerService,
    useFactory: loggerFactory,
    deps: [HttpClient, Config]
  }
]

// useExisting: Alias
providers: [
  NewLogger,
  { provide: OldLogger, useExisting: NewLogger }
]
```

InjectionToken

Answer:

- **InjectionToken** creates unique tokens for non-class dependencies
- Used for **configuration objects**, **strings**, **functions**
- Prevents naming conflicts

Example:

```
// Creating injection tokens
export const API_URL = new InjectionToken<string>('api.url');
export const APP_CONFIG = new InjectionToken<AppConfig>('app.config');
export const FEATURE_FLAGS = new InjectionToken<FeatureFlags>('feature.flags');

// Providing values
@NgModule({
  providers: [
    { provide: API_URL, useValue: 'https://api.example.com' },
    {
      provide: APP_CONFIG,
      useValue: {
        appName: 'My App',
        version: '1.0.0'
      }
    },
    {
      provide: FEATURE_FLAGS,
      useFactory: () => ({
        darkMode: true,
        newDashboard: false
```

```
      })
    }
  ]
})
export class AppModule { }

// Injecting tokens
@Component({
  selector: 'app-root'
})
export class AppComponent {
  constructor(
    @Inject(API_URL) private apiUrl: string,
    @Inject(APP_CONFIG) private config: AppConfig,
    @Inject(FEATURE_FLAGS) private features: FeatureFlags
  ) {
    console.log(this.apiUrl);  // 'https://api.example.com'
  }

  // Or using inject() function
  private api = inject(API_URL);
}
```

ROUTING ANSWERS

---

50. What is Angular Routing?

Answer:

- **Angular Router** enables **navigation** between views/components
- Maps **URL paths** to **components**
- Supports **deep linking**, **guards**, **lazy loading**
- Single Page Application (SPA) navigation without page refresh

Theoretical Keywords:

Navigation, **URL mapping**, **SPA**, **Deep linking**,
Route configuration, **RouterModule**, **Guards**

Example:

```
// Route configuration
const routes: Routes = [
  { path: '', component: HomeComponent },
  { path: 'about', component: AboutComponent },
  { path: 'users', component: UserListComponent },
  { path: 'users/:id', component: UserDetailComponent },
  { path: '**', component: NotFoundComponent }
];

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

// Template navigation
@Component({
  template: `
    <nav>
      <a routerLink="/" routerLinkActive="active" [routerLinkActiveOptions]="{exact: true}">Home</a>
      <a routerLink="/about" routerLinkActive="active">About</a>
      <a routerLink="/users" routerLinkActive="active">Users</a>
    </nav>

    <router-outlet></router-outlet>
  `
})
export class AppComponent { }
```

---

51. How does lazy loading work?

Answer:

- **Lazy loading** loads modules **on demand**
- Reduces **initial bundle size**
- Uses **loadChildren** with dynamic import
- Creates **separate chunk** for each lazy module

Theoretical Keywords:

Code splitting, **On-demand loading**, **loadChildren**,
Dynamic import, **Chunk files**, **Performance**

Example:

```
// App routes with lazy loading
const routes: Routes = [
  { path: '', component: HomeComponent },
  {
    path: 'admin',
    loadChildren: () => import('./admin/admin.module')
      .then(m => m.AdminModule)
  },
  {
    path: 'products',
    loadChildren: () => import('./products/products.module')
      .then(m => m.ProductsModule)
  },
  // Standalone component lazy loading (Angular 14+)
  {
    path: 'settings',
    loadComponent: () => import('./settings/settings.component')
      .then(c => c.SettingsComponent)
  }
];

// Admin module with its own routes
@NgModule({
  imports: [
    RouterModule.forChild([
      { path: '', component: AdminDashboardComponent },
      { path: 'users', component: AdminUsersComponent },
      { path: 'settings', component: AdminSettingsComponent }
    ])
  ]
})
export class AdminModule { }

// Preloading strategies
@NgModule({
  imports: [
    RouterModule.forRoot(routes, {
      preloadingStrategy: PreloadAllModules  // Preload after initial load
    })
  ]
})
export class AppRoutingModule { }

// Custom preloading strategy
@Injectable({ providedIn: 'root' })
export class SelectivePreloadingStrategy implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    return route.data?.['preload'] ? load() : of(null);
  }
}

// Usage
const routes: Routes = [
  {
    path: 'important',
    loadChildren: () => import('./important/important.module')
      .then(m => m.ImportantModule),
    data: { preload: true }  // Will be preloaded
  }
];
```

---

52. What are Route Guards?

Answer:

- **Route Guards** control **access to routes**
- Execute **before navigation** or **before leaving**
- Can **allow**, **deny**, or **redirect**
- Types: CanActivate, CanDeactivate, CanLoad, Resolve, CanMatch

Theoretical Keywords:

Access control, **Navigation guard**, **Authentication**,
Authorization, **Route protection**, **Functional guards**

Example:

```
// Functional guards (Angular 15+)
export const authGuard: CanActivateFn = (route, state) => {
  const authService = inject(AuthService);
  const router = inject(Router);

  if (authService.isAuthenticated()) {
    return true;
```

```
  }

  return router.createUrlTree(['/login'], {
    queryParams: { returnUrl: state.url }
  });
};

export const roleGuard: CanActivateFn = (route) => {
  const authService = inject(AuthService);
  const requiredRole = route.data['role'];

  return authService.hasRole(requiredRole);
};

// Class-based guard
@Injectable({ providedIn: 'root' })
export class AuthGuard implements CanActivate {
  constructor(
    private authService: AuthService,
    private router: Router
  ) { }

  canActivate(
    route: ActivatedRouteSnapshot,
    state: RouterStateSnapshot
  ): boolean | UrlTree {
    if (this.authService.isAuthenticated()) {
      return true;
    }
    return this.router.createUrlTree(['/login']);
  }
}

// CanDeactivate guard (unsaved changes)
export const unsavedChangesGuard: CanDeactivateFn<ComponentWithUnsavedChanges> =
  (component) => {
    if (component.hasUnsavedChanges()) {
      return confirm('You have unsaved changes. Leave anyway?');
    }
    return true;
  };

// Route configuration
const routes: Routes = [
  {
    path: 'dashboard',
    component: DashboardComponent,
    canActivate: [authGuard]
  },
  {
    path: 'admin',
    component: AdminComponent,
    canActivate: [authGuard, roleGuard],
    data: { role: 'admin' }
  },
  {
    path: 'editor',
    component: EditorComponent,
    canDeactivate: [unsavedChangesGuard]
  }
];
```

---

53. What is the difference between `RouterModule.forRoot()` and `forChild()`?

Answer:

Method	Usage	Router Service	Where
forRoot()	Main app routes	Creates Router	AppModule only
forChild()	Feature routes	Reuses Router	Feature modules

Example:

```
// AppRoutingModule - use forRoot()
@NgModule({
  imports: [
    RouterModule.forRoot(routes, {
      enableTracing: false,  // Debug routing
      useHash: false,        // Use hash routing
      scrollPositionRestoration: 'enabled',
      paramsInheritanceStrategy: 'always'
    })
  ],
  exports: [RouterModule]
})
export class AppRoutingModule { }

// Feature module - use forChild()
@NgModule({
  imports: [RouterModule.forChild(featureRoutes)],
  exports: [RouterModule]
})
```

```
export class FeatureRoutingModule { }

// Why this matters:
// forRoot() creates the Router service instance
// forChild() uses the existing Router instance
// Using forRoot() in feature modules would create multiple router instances
```

---

54. How to pass route parameters?

Answer:

Multiple ways to pass data through routes:

- **Path parameters**: `/users/:id`
- **Query parameters**: `/users?page=1&sort=name`
- **Route data**: Static data in route config
- **State**: Navigation extras

Example:

```
// Route configuration
const routes: Routes = [
  { path: 'user/:id', component: UserComponent },
  {
    path: 'products',
    component: ProductsComponent,
    data: { title: 'Products Page', breadcrumb: 'Products' }
  }
];

// Passing parameters
@Component({
  template: `
    <!-- Path parameter -->
    <a [routerLink]="['/user', userId]">User Details</a>

    <!-- Query parameters -->
    <a [routerLink]="['/products']" [queryParams]="{page: 1, sort: 'name'}">
      Products
    </a>

    <!-- Preserve query params -->
    <a [routerLink]="['/next']" queryParamsHandling="preserve">Next</a>
  `
})
export class NavComponent {
  userId = 123;

  constructor(private router: Router) { }

  navigateProgrammatically(): void {
    // With path param
    this.router.navigate(['/user', this.userId]);

    // With query params
    this.router.navigate(['/products'], {
      queryParams: { page: 1, sort: 'name' }
    });

    // With state
    this.router.navigate(['/checkout'], {
      state: { items: this.cartItems }
    });
  }
}

// Reading parameters
@Component({
  selector: 'app-user'
})
export class UserComponent implements OnInit {
  constructor(
    private route: ActivatedRoute,
    private router: Router
  ) { }

  ngOnInit(): void {
    // Path parameter (snapshot - one time)
    const id = this.route.snapshot.paramMap.get('id');

    // Path parameter (observable - reactive)
    this.route.paramMap.subscribe(params => {
      const userId = params.get('id');
      this.loadUser(userId);
    });

    // Query parameters
    this.route.queryParamMap.subscribe(params => {
      const page = params.get('page');
      const sort = params.get('sort');
    });
```

```
    // Route data
    const title = this.route.snapshot.data['title'];

    // Navigation state
    const state = this.router.getCurrentNavigation()?.extras.state;
    // Or
    const historyState = history.state;
  }
}
```

Resolver

Answer:

- **Resolver** fetches data **before route activation**
- Ensures data is available when component loads
- Prevents empty state flicker

Example:

```
// Functional resolver (Angular 15+)
export const userResolver: ResolveFn<User> = (route) => {
  const userService = inject(UserService);
  const id = route.paramMap.get('id')!;
  return userService.getUserById(+id);
};

// Class-based resolver
@Injectable({ providedIn: 'root' })
export class UserResolver implements Resolve<User> {
  constructor(private userService: UserService) { }

  resolve(route: ActivatedRouteSnapshot): Observable<User> {
    const id = route.paramMap.get('id')!;
    return this.userService.getUserById(+id);
  }
}

// Route configuration
const routes: Routes = [
  {
    path: 'user/:id',
    component: UserComponent,
    resolve: {
      user: userResolver
    }
  }
];

// Using resolved data
@Component({
  selector: 'app-user'
})
export class UserComponent implements OnInit {
  user!: User;

  constructor(private route: ActivatedRoute) { }

  ngOnInit(): void {
    // Snapshot
    this.user = this.route.snapshot.data['user'];

    // Or reactive
    this.route.data.subscribe(data => {
      this.user = data['user'];
    });
  }
}
```

Child Routes

Answer:

- **Child routes** create nested navigation
- Rendered in parent's `<router-outlet>`
- Share parent's URL path

Example:

```
const routes: Routes = [
  {
    path: 'products',
    component: ProductsLayoutComponent,
    children: [
      { path: '', component: ProductListComponent },
      { path: ':id', component: ProductDetailComponent },
      { path: ':id/reviews', component: ProductReviewsComponent }
```

```
    ]
  },
  {
    path: 'admin',
    component: AdminLayoutComponent,
    children: [
      { path: '', redirectTo: 'dashboard', pathMatch: 'full' },
      { path: 'dashboard', component: AdminDashboardComponent },
      {
        path: 'users',
        component: UsersLayoutComponent,
        children: [
          { path: '', component: UserListComponent },
          { path: ':id', component: UserDetailComponent }
        ]
      }
    ]
  }
];

// Layout component with router-outlet
@Component({
  template: `
    <div class="admin-layout">
      <nav class="sidebar">
        <a routerLink="dashboard">Dashboard</a>
        <a routerLink="users">Users</a>
      </nav>
      <main>
        <router-outlet></router-outlet>
      </main>
    </div>
  `
})
export class AdminLayoutComponent { }
```

Named Router Outlets

Answer:

- Multiple `<router-outlet>` with names
- Display **auxiliary routes** simultaneously
- Independent navigation

Example:

```
const routes: Routes = [
  { path: 'home', component: HomeComponent },
  { path: 'chat', component: ChatComponent, outlet: 'sidebar' },
  { path: 'notifications', component: NotificationsComponent, outlet: 'sidebar' }
];

@Component({
  template: `
    <main>
      <router-outlet></router-outlet>
    </main>

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

    <!-- Navigation -->
    <a [routerLink]="[{ outlets: { primary: 'home', sidebar: 'chat' } }]">
      Home with Chat
    </a>
    <a [routerLink]="[{ outlets: { sidebar: null } }]">
      Close Sidebar
    </a>
  `
})
export class AppComponent { }

// URL: /home(sidebar:chat)
```

FORMS ANSWERS

---

55. What are Template-driven forms?

Answer:

- Forms built using **directives in templates**
- Use **ngModel** for two-way data binding
- **Simpler** for basic forms
- Validation through **HTML5 attributes** and directives

- Asynchronous by nature

Theoretical Keywords:

ngModel, **Two-way binding**, **FormsModule**, **Template directives**,
HTML5 validation, **Simple forms**, **Declarative**

Example:

```
// app.module.ts
import { FormsModule } from '@angular/forms';

@NgModule({
  imports: [FormsModule]
})
export class AppModule { }

// Component
@Component({
  selector: 'app-login',
  template: `
    <form #loginForm="ngForm" (ngSubmit)="onSubmit(loginForm)">
      <div class="form-group">
        <label for="email">Email</label>
        <input
          type="email"
          id="email"
          name="email"
          [(ngModel)]="user.email"
          required
          email
          #emailField="ngModel"
        >
        <div *ngIf="emailField.invalid && emailField.touched" class="error">
          <span *ngIf="emailField.errors?.['required']">Email is required</span>
          <span *ngIf="emailField.errors?.['email']">Invalid email format</span>
        </div>
      </div>

      <div class="form-group">
        <label for="password">Password</label>
        <input
          type="password"
          id="password"
          name="password"
          [(ngModel)]="user.password"
          required
          minlength="6"
          #passwordField="ngModel"
        >
        <div *ngIf="passwordField.invalid && passwordField.touched" class="error">
          <span *ngIf="passwordField.errors?.['required']">Password is required</span>
          <span *ngIf="passwordField.errors?.['minlength']">
            Min {{ passwordField.errors?.['minlength'].requiredLength }} characters
          </span>
        </div>
      </div>

      <button type="submit" [disabled]="loginForm.invalid">Login</button>

      <pre>Form Valid: {{ loginForm.valid }}</pre>
      <pre>Form Value: {{ loginForm.value | json }}</pre>
    </form>
  `
})
export class LoginComponent {
  user = {
    email: '',
    password: ''
  };

  onSubmit(form: NgForm): void {
    if (form.valid) {
      console.log('Form submitted:', this.user);
    }
  }
}
```

---

56. What are Reactive forms?

Answer:

- Forms built **programmatically** in component class
- Use **FormGroup**, **FormControl**, **FormArray**
- **More control** and flexibility
- **Synchronous** and easier to test
- Validation in component, not template

Theoretical Keywords:

FormGroup, **FormControl**, **ReactiveFormsModule**, **Programmatic**, **Synchronous**, **Testable**, **Validators**

Example:

```
// app.module.ts
import { ReactiveFormsModule } from '@angular/forms';

@NgModule({
  imports: [ReactiveFormsModule]
})
export class AppModule { }

// Component
@Component({
  selector: 'app-registration',
  template: `
    <form [formGroup]="registrationForm" (ngSubmit)="onSubmit()">
      <div class="form-group">
        <label>Email</label>
        <input type="email" formControlName="email">
        <div *ngIf="email.invalid && email.touched" class="error">
          <span *ngIf="email.errors?.['required']">Email is required</span>
          <span *ngIf="email.errors?.['email']">Invalid email</span>
        </div>
      </div>

      <div class="form-group">
        <label>Password</label>
        <input type="password" formControlName="password">
      </div>

      <div class="form-group">
        <label>Confirm Password</label>
        <input type="password" formControlName="confirmPassword">
        <div *ngIf="registrationForm.errors?.['passwordMismatch']" class="error">
          Passwords don't match
        </div>
      </div>

      <div formGroupName="address">
        <h4>Address</h4>
        <input formControlName="street" placeholder="Street">
        <input formControlName="city" placeholder="City">
        <input formControlName="zip" placeholder="ZIP">
      </div>

      <button type="submit" [disabled]="registrationForm.invalid">Register</button>
    </form>
  `
})
export class RegistrationComponent implements OnInit {
  registrationForm!: FormGroup;

  constructor(private fb: FormBuilder) { }

  ngOnInit(): void {
    this.registrationForm = this.fb.group({
      email: ['', [Validators.required, Validators.email]],
      password: ['', [Validators.required, Validators.minLength(6)]],
      confirmPassword: ['', Validators.required],
      address: this.fb.group({
        street: [''],
        city: ['', Validators.required],
        zip: ['', Validators.pattern(/^\d{5}$/)]
      })
    }, {
      validators: this.passwordMatchValidator
    });
  }

  get email() { return this.registrationForm.get('email')!; }

  passwordMatchValidator(form: FormGroup): ValidationErrors | null {
    const password = form.get('password')?.value;
    const confirmPassword = form.get('confirmPassword')?.value;
    return password === confirmPassword ? null : { passwordMismatch: true };
  }

  onSubmit(): void {
    if (this.registrationForm.valid) {
      console.log(this.registrationForm.value);
    }
  }
}
```

---

57. Difference between Template-driven and Reactive forms?

Answer:

Feature	Template-driven	Reactive

Feature	Template-driven	Reactive
Module	FormsModule	ReactiveFormsModule
Form model	Directives create	Explicit in class
Data binding	Two-way (ngModel)	Form control binding
Validation	Directives/HTML5	Validators in class
Testability	Harder (async)	Easier (sync)
Flexibility	Less	More
Best for	Simple forms	Complex forms
Dynamic	Harder	Easy (FormArray)

Example:

```
// Template-driven
@Component({
  template: `
    <input [(ngModel)]="name" required #nameField="ngModel">
    <span *ngIf="nameField.invalid">Required</span>
  `
})
export class TemplateComponent {
  name = '';
}

// Reactive
@Component({
  template: `
    <input [formControl]="nameControl">
    <span *ngIf="nameControl.invalid">Required</span>
  `
})
export class ReactiveComponent {
  nameControl = new FormControl('', Validators.required);
}
```

---

58. What is FormGroup and FormControl?

Answer:

- **FormControl**: Single input field
- **FormGroup**: Collection of controls
- **FormArray**: Dynamic array of controls

Example:

```
@Component({
  template: `
    <form [formGroup]="profileForm">
      <!-- FormControl -->
      <input formControlName="firstName">
      <input formControlName="lastName">

      <!-- Nested FormGroup -->
      <div formGroupName="address">
        <input formControlName="street">
        <input formControlName="city">
      </div>

      <!-- FormArray -->
      <div formArrayName="phones">
        <div *ngFor="let phone of phones.controls; let i = index">
          <input [formControlName]="i">
          <button (click)="removePhone(i)">Remove</button>
        </div>
        <button (click)="addPhone()">Add Phone</button>
      </div>
    </form>
  `
})
export class ProfileComponent {
  profileForm = new FormGroup({
    firstName: new FormControl('', Validators.required),
    lastName: new FormControl(''),
    address: new FormGroup({
      street: new FormControl(''),
      city: new FormControl('')
    }),
    phones: new FormArray([
      new FormControl('')
    ])
  });

  // Or using FormBuilder
  constructor(private fb: FormBuilder) {
    this.profileForm = this.fb.group({
      firstName: ['', Validators.required],
      lastName: [''],
      address: this.fb.group({
```

```
        street: [''],
        city: ['']
      }),
      phones: this.fb.array([''])
    });
  }

  get phones(): FormArray {
    return this.profileForm.get('phones') as FormArray;
  }

  addPhone(): void {
    this.phones.push(new FormControl(''));
  }

  removePhone(index: number): void {
    this.phones.removeAt(index);
  }
}
```

---

59. How to create custom validators?

Answer:

- **Sync validators**: Return errors or null immediately
- **Async validators**: Return Observable/Promise

Example:

```
// Sync validator function
function forbiddenNameValidator(forbiddenName: RegExp): ValidatorFn {
  return (control: AbstractControl): ValidationErrors | null => {
    const forbidden = forbiddenName.test(control.value);
    return forbidden ? { forbiddenName: { value: control.value } } : null;
  };
}

// Async validator function
function uniqueEmailValidator(userService: UserService): AsyncValidatorFn {
  return (control: AbstractControl): Observable<ValidationErrors | null> => {
    return userService.checkEmailExists(control.value).pipe(
      map(exists => exists ? { emailTaken: true } : null),
      catchError(() => of(null))
    );
  };
}

// Cross-field validator
function dateRangeValidator(control: AbstractControl): ValidationErrors | null {
  const startDate = control.get('startDate')?.value;
  const endDate = control.get('endDate')?.value;

  if (startDate && endDate && new Date(startDate) > new Date(endDate)) {
    return { dateRange: 'Start date must be before end date' };
  }
  return null;
}

// Usage
@Component({
  template: `
    <form [formGroup]="form">
      <input formControlName="username">
      <div *ngIf="username.errors?.['forbiddenName']">
        Name "{{ username.errors?.['forbiddenName'].value }}" is not allowed
      </div>

      <input formControlName="email">
      <div *ngIf="email.pending">Checking...</div>
      <div *ngIf="email.errors?.['emailTaken']">Email already taken</div>

      <input type="date" formControlName="startDate">
      <input type="date" formControlName="endDate">
      <div *ngIf="form.errors?.['dateRange']">{{ form.errors?.['dateRange'] }}</div>
    </form>
  `
})
export class ValidatorComponent {
  form: FormGroup;

  constructor(private fb: FormBuilder, private userService: UserService) {
    this.form = this.fb.group({
      username: ['', [
        Validators.required,
        forbiddenNameValidator(/admin/i)
      ]],
      email: ['',
        [Validators.required, Validators.email],
        [uniqueEmailValidator(userService)]
      ],
      startDate: [''],
      endDate: ['']
```

```
    }, {
      validators: dateRangeValidator
    });
  }

  get username() { return this.form.get('username')!; }
  get email() { return this.form.get('email')!; }
}

// Directive-based validator (for template-driven)
@Directive({
  selector: '[appForbiddenName]',
  providers: [{
    provide: NG_VALIDATORS,
    useExisting: ForbiddenNameDirective,
    multi: true
  }]
})
export class ForbiddenNameDirective implements Validator {
  @Input('appForbiddenName') forbiddenName = '';

  validate(control: AbstractControl): ValidationErrors | null {
    return this.forbiddenName
      ? forbiddenNameValidator(new RegExp(this.forbiddenName, 'i'))(control)
      : null;
  }
}
```

Form Control States

Answer:

FormControl tracks various states:

State	Description
valid/invalid	Validation status
pristine/dirty	User modified value
touched/untouched	User focused/blurred
pending	Async validation running
enabled/disabled	Can interact

Example:

```
@Component({
  template: `
    <input [formControl]="name">

    <div>
      <p>Value: {{ name.value }}</p>
      <p>Valid: {{ name.valid }}</p>
      <p>Invalid: {{ name.invalid }}</p>
      <p>Pristine: {{ name.pristine }}</p>
      <p>Dirty: {{ name.dirty }}</p>
      <p>Touched: {{ name.touched }}</p>
      <p>Untouched: {{ name.untouched }}</p>
      <p>Pending: {{ name.pending }}</p>
    </div>

    <!-- Show error only when touched and invalid -->
    <div *ngIf="name.invalid && name.touched" class="error">
      Name is required
    </div>

    <!-- CSS classes applied automatically -->
    <!-- .ng-valid, .ng-invalid -->
    <!-- .ng-pristine, .ng-dirty -->
    <!-- .ng-touched, .ng-untouched -->
  `,
  styles: [`
    input.ng-invalid.ng-touched {
      border-color: red;
    }
    input.ng-valid.ng-touched {
      border-color: green;
    }
  `]
})
export class FormStateComponent {
  name = new FormControl('', Validators.required);

  programmaticUpdates(): void {
    // Set value
    this.name.setValue('John');

    // Patch value (for FormGroup)
    // this.form.patchValue({ name: 'John' });

    // Mark states
    this.name.markAsTouched();
    this.name.markAsDirty();
```

```
    this.name.markAsPristine();
    this.name.markAsUntouched();

    // Reset
    this.name.reset();
    this.name.reset('default value');

    // Disable/Enable
    this.name.disable();
    this.name.enable();

    // Update validation
    this.name.setValidators([Validators.required, Validators.minLength(3)]);
    this.name.updateValueAndValidity();
  }
}
```

Dynamic Forms

Answer:

Use **FormArray** and **FormGroup** for dynamic form generation.

Example:

```
@Component({
  template: `
    <form [formGroup]="dynamicForm">
      <div formArrayName="questions">
        <div *ngFor="let question of questions.controls; let i = index" [formGroupName]="i">
          <input formControlName="label" placeholder="Question">
          <select formControlName="type">
            <option value="text">Text</option>
            <option value="number">Number</option>
            <option value="select">Select</option>
          </select>
          <input formControlName="required" type="checkbox"> Required
          <button (click)="removeQuestion(i)">Remove</button>
        </div>
      </div>
      <button (click)="addQuestion()">Add Question</button>
    </form>
  `
})
export class DynamicFormComponent {
  dynamicForm = new FormGroup({
    questions: new FormArray<FormGroup>([])
  });

  get questions(): FormArray<FormGroup> {
    return this.dynamicForm.get('questions') as FormArray<FormGroup>;
  }

  addQuestion(): void {
    const questionGroup = new FormGroup({
      label: new FormControl('', Validators.required),
      type: new FormControl('text'),
      required: new FormControl(false)
    });
    this.questions.push(questionGroup);
  }

  removeQuestion(index: number): void {
    this.questions.removeAt(index);
  }
}
```

STATE MANAGEMENT (NgRx) ANSWERS

---

60. What is NgRx?

Answer:

- **NgRx** is a **state management library** for Angular
- Based on **Redux pattern** and **RxJS**
- Provides **single source of truth** for application state
- Uses **unidirectional data flow**
- Includes: Store, Actions, Reducers, Effects, Selectors

Theoretical Keywords:

Redux pattern, **RxJS**, **Single source of truth**, **Unidirectional flow**,
Predictable state, **Store**, **Immutable**

Example:

```
// Installation
// npm install @ngrx/store @ngrx/effects @ngrx/store-devtools

// Basic flow
// Action → Reducer → Store → Selector → Component
```

---

61. Why use state management?

Answer:

- **Centralized state**: Single source of truth
- **Predictability**: State changes are explicit
- **Debugging**: Time-travel debugging with DevTools
- **Testability**: Pure functions are easy to test
- **Scalability**: Manages complex state interactions
- **Component decoupling**: Components don't need to know data source

Theoretical Keywords:

Centralized, **Predictable**, **Debuggable**, **Testable**,
Scalable, **Decoupled**, **Maintainable**

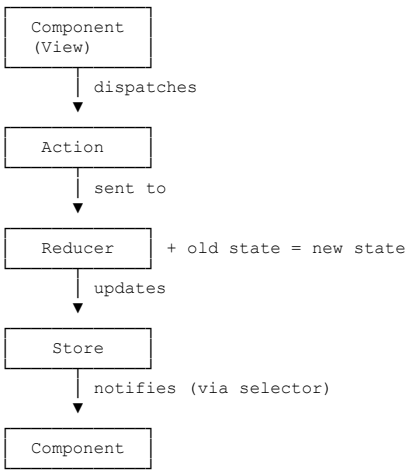
---

62. What is the Redux pattern?

Answer:

- **Store**: Single state container
- **Actions**: Describe what happened
- **Reducers**: Pure functions that update state
- **One-way data flow**: Action → Reducer → New State

Example:



---

63. What are Actions?

Answer:

- **Actions** are events that describe **what happened**
- Have a **type** (unique identifier) and optional **payload**
- Created using `createAction()`
- Dispatched to trigger state changes

Example:

```
// actions/user.actions.ts
import { createAction, props } from '@ngrx/store';

// Simple action
export const loadUsers = createAction('[User List] Load Users');

// Action with payload
export const loadUsersSuccess = createAction(
  '[User API] Load Users Success',
```

```
  props<{ users: User[] }>()
);

export const loadUsersFailure = createAction(
  '[User API] Load Users Failure',
  props<{ error: string }>()
);

export const addUser = createAction(
  '[User Form] Add User',
  props<{ user: User }>()
);

export const updateUser = createAction(
  '[User Form] Update User',
  props<{ id: number; changes: Partial<User> }>()
);

export const deleteUser = createAction(
  '[User List] Delete User',
  props<{ id: number }>()
);

// Dispatching actions in component
@Component({...})
export class UserListComponent {
  constructor(private store: Store) { }

  loadUsers(): void {
    this.store.dispatch(loadUsers());
  }

  addUser(user: User): void {
    this.store.dispatch(addUser({ user }));
  }

  deleteUser(id: number): void {
    this.store.dispatch(deleteUser({ id }));
  }
}
```

---

64. What are Reducers?

Answer:

- **Reducers** are **pure functions** that handle state transitions
- Take current state and action, return new state
- Must be **immutable** - never modify original state
- Created using `createReducer()`

Example:

```
// reducers/user.reducer.ts
import { createReducer, on } from '@ngrx/store';
import * as UserActions from '../actions/user.actions';

export interface UserState {
  users: User[];
  loading: boolean;
  error: string | null;
  selectedUserId: number | null;
}

export const initialState: UserState = {
  users: [],
  loading: false,
  error: null,
  selectedUserId: null
};

export const userReducer = createReducer(
  initialState,

  // Load users
  on(UserActions.loadUsers, (state) => ({
    ...state,
    loading: true,
    error: null
  })),

  on(UserActions.loadUsersSuccess, (state, { users }) => ({
    ...state,
    users,
    loading: false
  })),

  on(UserActions.loadUsersFailure, (state, { error }) => ({
    ...state,
    error,
    loading: false
  })),
```

```
  // Add user
  on(UserActions.addUser, (state, { user }) => ({
    ...state,
    users: [...state.users, user]
  })),

  // Update user
  on(UserActions.updateUser, (state, { id, changes }) => ({
    ...state,
    users: state.users.map(user =>
      user.id === id ? { ...user, ...changes } : user
    )
  })),

  // Delete user
  on(UserActions.deleteUser, (state, { id }) => ({
    ...state,
    users: state.users.filter(user => user.id !== id)
  })),

  // Select user
  on(UserActions.selectUser, (state, { id }) => ({
    ...state,
    selectedUserId: id
  }))
);
```

---

65. What are Selectors?

Answer:

- **Selectors** are **pure functions** to extract data from store
- Provide **memoization** for performance
- Compose smaller selectors into complex ones
- Decouple components from store structure

Example:

```
// selectors/user.selectors.ts
import { createFeatureSelector, createSelector } from '@ngrx/store';
import { UserState } from '../reducers/user.reducer';

// Feature selector
export const selectUserState = createFeatureSelector<UserState>('users');

// Basic selectors
export const selectAllUsers = createSelector(
  selectUserState,
  (state) => state.users
);

export const selectUsersLoading = createSelector(
  selectUserState,
  (state) => state.loading
);

export const selectUsersError = createSelector(
  selectUserState,
  (state) => state.error
);

export const selectSelectedUserId = createSelector(
  selectUserState,
  (state) => state.selectedUserId
);

// Composed selectors
export const selectSelectedUser = createSelector(
  selectAllUsers,
  selectSelectedUserId,
  (users, selectedId) => users.find(user => user.id === selectedId)
);

export const selectActiveUsers = createSelector(
  selectAllUsers,
  (users) => users.filter(user => user.isActive)
);

export const selectUserCount = createSelector(
  selectAllUsers,
  (users) => users.length
);

// Parameterized selector
export const selectUserById = (id: number) => createSelector(
  selectAllUsers,
  (users) => users.find(user => user.id === id)
);

// Using selectors in component
@Component({...})
export class UserListComponent {
```

```
  users$ = this.store.select(selectAllUsers);
  loading$ = this.store.select(selectUsersLoading);
  error$ = this.store.select(selectUsersError);
  activeUsers$ = this.store.select(selectActiveUsers);

  constructor(private store: Store) { }

  // With parameter
  getUserById(id: number): Observable<User | undefined> {
    return this.store.select(selectUserById(id));
  }
}
```

---

66. What are Effects?

Answer:

- **Effects** handle **side effects** (API calls, navigation, etc.)
- Listen for actions, perform async operations
- Dispatch new actions based on results
- Keep reducers pure and synchronous

Example:

```
// effects/user.effects.ts
import { Injectable } from '@angular/core';
import { Actions, createEffect, ofType } from '@ngrx/effects';
import { of } from 'rxjs';
import { map, exhaustMap, catchError, tap } from 'rxjs/operators';
import * as UserActions from '../actions/user.actions';
import { UserService } from '../services/user.service';

@Injectable()
export class UserEffects {
  constructor(
    private actions$: Actions,
    private userService: UserService,
    private router: Router
  ) { }

  // Load users effect
  loadUsers$ = createEffect(() =>
    this.actions$.pipe(
      ofType(UserActions.loadUsers),
      exhaustMap(() =>
        this.userService.getUsers().pipe(
          map(users => UserActions.loadUsersSuccess({ users })),
          catchError(error => of(UserActions.loadUsersFailure({
            error: error.message
          })))
        )
      )
    )
  );

  // Add user effect
  addUser$ = createEffect(() =>
    this.actions$.pipe(
      ofType(UserActions.addUser),
      exhaustMap(({ user }) =>
        this.userService.createUser(user).pipe(
          map(newUser => UserActions.addUserSuccess({ user: newUser })),
          catchError(error => of(UserActions.addUserFailure({
            error: error.message
          })))
        )
      )
    )
  );

  // Delete user effect
  deleteUser$ = createEffect(() =>
    this.actions$.pipe(
      ofType(UserActions.deleteUser),
      exhaustMap(({ id }) =>
        this.userService.deleteUser(id).pipe(
          map(() => UserActions.deleteUserSuccess({ id })),
          catchError(error => of(UserActions.deleteUserFailure({
            error: error.message
          })))
        )
      )
    )
  );

  // Navigation effect (no dispatch)
  addUserSuccess$ = createEffect(() =>
    this.actions$.pipe(
      ofType(UserActions.addUserSuccess),
      tap(() => this.router.navigate(['/users']))
    ),
    { dispatch: false }
```

```
  );

  // Show notification effect
  showError$ = createEffect(() =>
    this.actions$.pipe(
      ofType(
        UserActions.loadUsersFailure,
        UserActions.addUserFailure,
        UserActions.deleteUserFailure
      ),
      tap(({ error }) => {
        this.snackBar.open(error, 'Close', { duration: 5000 });
      })
    ),
    { dispatch: false }
  );
}
```

---

67. What is Store?

Answer:

- **Store** is the **single source of truth**
- Holds the entire application state
- State is **read-only**, changes via dispatching actions
- Components **select** slices of state

Example:

```
// app.module.ts - Store setup
import { StoreModule } from '@ngrx/store';
import { EffectsModule } from '@ngrx/effects';
import { StoreDevtoolsModule } from '@ngrx/store-devtools';

@NgModule({
  imports: [
    StoreModule.forRoot({
      users: userReducer,
      products: productReducer
    }),
    EffectsModule.forRoot([
      UserEffects,
      ProductEffects
    ]),
    StoreDevtoolsModule.instrument({
      maxAge: 25,
      logOnly: environment.production
    })
  ]
})
export class AppModule { }

// Feature module
@NgModule({
  imports: [
    StoreModule.forFeature('orders', orderReducer),
    EffectsModule.forFeature([OrderEffects])
  ]
})
export class OrderModule { }

// Using Store in component
@Component({
  selector: 'app-user-dashboard',
  template: `
    <div *ngIf="loading$ | async">Loading...</div>
    <div *ngIf="error$ | async as error" class="error">{{ error }}</div>

    <ul>
      <li *ngFor="let user of users$ | async">
        {{ user.name }}
      </li>
    </ul>

    <button (click)="loadUsers()">Refresh</button>
  `
})
export class UserDashboardComponent implements OnInit {
  users$ = this.store.select(selectAllUsers);
  loading$ = this.store.select(selectUsersLoading);
  error$ = this.store.select(selectUsersError);

  constructor(private store: Store) { }

  ngOnInit(): void {
    this.loadUsers();
  }

  loadUsers(): void {
    this.store.dispatch(loadUsers());
  }
}
```

---

68. When should you use NgRx?

Answer:

Use NgRx when:

- Large application with complex state
- Multiple components need same data
- State changes need to be tracked/debugged
- Offline support required
- Team needs consistent patterns

Don't use NgRx when:

- Simple application
- State is mostly local to components
- Team is unfamiliar with Redux
- Overhead not justified

Example Decision Matrix:

```
Application Size    | State Complexity | Use NgRx?
--------------------|------------------|----------
Small               | Simple           | No
Small               | Complex          | Maybe (signals)
Medium              | Simple           | No
Medium              | Complex          | Yes
Large               | Any              | Yes
```

---

69. Alternatives to NgRx?

Answer:

Library	Use Case
Angular Signals	Built-in, simple reactivity
Services + BehaviorSubject	Simple shared state
NGXS	Less boilerplate than NgRx
Akita	Entity-based state management
Elf	Lightweight, modern

Example:

```
// Simple service-based state management
@Injectable({ providedIn: 'root' })
export class UserStore {
  private usersSubject = new BehaviorSubject<User[]>([]);
  private loadingSubject = new BehaviorSubject<boolean>(false);

  users$ = this.usersSubject.asObservable();
  loading$ = this.loadingSubject.asObservable();

  constructor(private http: HttpClient) { }

  loadUsers(): void {
    this.loadingSubject.next(true);
    this.http.get<User[]>('/api/users').subscribe({
      next: (users) => {
        this.usersSubject.next(users);
        this.loadingSubject.next(false);
      },
      error: () => this.loadingSubject.next(false)
    });
  }

  addUser(user: User): void {
    const current = this.usersSubject.value;
    this.usersSubject.next([...current, user]);
  }
}

// Angular Signals (Angular 16+)
@Injectable({ providedIn: 'root' })
export class UserStore {
  users = signal<User[]>([]);
  loading = signal(false);

  // Computed
  activeUsers = computed(() =>
    this.users().filter(u => u.isActive)
  );

  loadUsers(): void {
    this.loading.set(true);
```

```
    // ...
    this.users.set(fetchedUsers);
    this.loading.set(false);
  }
}
```

LIFECYCLE HOOKS ANSWERS

---

70. What are lifecycle hooks?

Answer:

- **Lifecycle hooks** are methods that Angular calls at specific moments
- Allow you to **tap into** component/directive lifecycle
- From **creation** to **destruction**
- Execute code at precise moments

Theoretical Keywords:

Lifecycle events, **Component creation**, **Initialization**, **Change detection**, **Destruction**, **Hook methods**, **Order of execution**

Example:

```
@Component({
  selector: 'app-lifecycle-demo',
  template: '<p>{{ data }}</p>'
})
export class LifecycleDemoComponent implements
  OnChanges, OnInit, DoCheck,
  AfterContentInit, AfterContentChecked,
  AfterViewInit, AfterViewChecked, OnDestroy {

  @Input() data!: string;

  constructor() {
    console.log('1. Constructor');
    // Don't use for initialization logic
    // DI is available, but inputs are NOT set yet
  }

  ngOnChanges(changes: SimpleChanges): void {
    console.log('2. ngOnChanges', changes);
    // Called when @Input properties change
    // First call: before ngOnInit
  }

  ngOnInit(): void {
    console.log('3. ngOnInit');
    // Initialize component
    // @Input values are available
    // Good place for initial data fetch
  }

  ngDoCheck(): void {
    console.log('4. ngDoCheck');
    // Custom change detection
    // Called on EVERY change detection cycle
    // Expensive - use carefully
  }

  ngAfterContentInit(): void {
    console.log('5. ngAfterContentInit');
    // After <ng-content> is projected
    // @ContentChild/@ContentChildren available
  }

  ngAfterContentChecked(): void {
    console.log('6. ngAfterContentChecked');
    // After projected content is checked
  }

  ngAfterViewInit(): void {
    console.log('7. ngAfterViewInit');
    // After view (and child views) initialized
    // @ViewChild/@ViewChildren available
  }

  ngAfterViewChecked(): void {
    console.log('8. ngAfterViewChecked');
    // After view checked
  }

  ngOnDestroy(): void {
    console.log('9. ngOnDestroy');
    // Cleanup before component is destroyed
    // Unsubscribe, clear timers, etc.
```

```
  }
}
```

---

71. What is `ngOnInit` and why use it?

Answer:

- **ngOnInit** is called once after first `ngOnChanges`
- Used for **component initialization**
- **@Input** values are available (unlike constructor)
- Perfect for **API calls**, **data setup**

Theoretical Keywords:

Initialization, **Input available**, **Single execution**,
Data fetching, **Setup logic**, **After constructor**

Example:

```
@Component({
  selector: 'app-user-profile'
})
export class UserProfileComponent implements OnInit {
  @Input() userId!: number;
  user: User | null = null;

  constructor(
    private userService: UserService,
    private route: ActivatedRoute
  ) {
    // Constructor: Only for DI
    // this.userId is undefined here!
    console.log('Constructor - userId:', this.userId); // undefined
  }

  ngOnInit(): void {
    // ngOnInit: @Input is available
    console.log('ngOnInit - userId:', this.userId); // actual value

    // Fetch data
    this.loadUser();

    // Subscribe to route params
    this.route.params.subscribe(params => {
      this.loadUser(params['id']);
    });
  }

  private loadUser(id?: number): void {
    const userId = id || this.userId;
    this.userService.getUser(userId).subscribe(user => {
      this.user = user;
    });
  }
}
```

---

72. What is the difference between `ngOnInit` and `constructor`?

Answer:

Feature	Constructor	ngOnInit
Purpose	Dependency Injection	Initialization logic
@Input	Not available	Available
Called by	JavaScript engine	Angular
Timing	Object creation	After first ngOnChanges
Use for	DI only	Setup, API calls

Example:

```
@Component({
  selector: 'app-comparison'
})
export class ComparisonComponent implements OnInit {
  @Input() config!: Config;

  // WRONG: Don't do initialization in constructor
  constructor(private service: DataService) {
    // □ this.config is undefined
    // □ this.service.loadData(this.config); // Error!
  }

  // CORRECT: Do initialization in ngOnInit
  ngOnInit(): void {
```

```
    // □ this.config is available
    this.service.loadData(this.config);
  }
}

// Best practice pattern
@Component({
  selector: 'app-best-practice'
})
export class BestPracticeComponent implements OnInit, OnDestroy {
  @Input() userId!: number;

  private destroy$ = new Subject<void>();
  user$!: Observable<User>;

  constructor(
    private userService: UserService,  // Only DI
    private router: Router
  ) { }

  ngOnInit(): void {
    // All initialization here
    this.user$ = this.userService.getUser(this.userId);

    // Subscriptions with cleanup
    this.userService.notifications$
      .pipe(takeUntil(this.destroy$))
      .subscribe(/* ... */);
  }

  ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }
}
```

---

73. What is `ngOnDestroy` and why is it important?

Answer:

- **ngOnDestroy** is called just before component is destroyed
- Used for **cleanup** to prevent memory leaks
- Unsubscribe from observables, clear timers, detach event listeners

Theoretical Keywords:

Cleanup, **Memory leaks**, **Unsubscribe**, **Clear timers**,
Event listeners, **Resource release**, **Garbage collection**

Example:

```
@Component({
  selector: 'app-cleanup-demo'
})
export class CleanupDemoComponent implements OnInit, OnDestroy {
  private subscription!: Subscription;
  private intervalId!: any;
  private destroy$ = new Subject<void>();

  constructor(
    private dataService: DataService,
    private elementRef: ElementRef
  ) { }

  ngOnInit(): void {
    // Manual subscription (needs cleanup)
    this.subscription = this.dataService.getData().subscribe();

    // Interval (needs cleanup)
    this.intervalId = setInterval(() => {
      console.log('Tick');
    }, 1000);

    // Using takeUntil pattern (automatic cleanup)
    this.dataService.stream$
      .pipe(takeUntil(this.destroy$))
      .subscribe(data => console.log(data));

    // Event listener (needs cleanup)
    this.elementRef.nativeElement.addEventListener('click', this.handleClick);
  }

  handleClick = () => {
    console.log('Clicked');
  };

  ngOnDestroy(): void {
    // 1. Unsubscribe manual subscriptions
    if (this.subscription) {
      this.subscription.unsubscribe();
    }
```

```
    // 2. Clear intervals/timeouts
    if (this.intervalId) {
      clearInterval(this.intervalId);
    }

    // 3. Complete subject for takeUntil pattern
    this.destroy$.next();
    this.destroy$.complete();

    // 4. Remove event listeners
    this.elementRef.nativeElement.removeEventListener('click', this.handleClick);

    console.log('Component destroyed, resources cleaned up');
  }
}

// Using DestroyRef (Angular 16+)
@Component({
  selector: 'app-modern-cleanup'
})
export class ModernCleanupComponent {
  constructor(private destroyRef: DestroyRef) {
    // Automatic cleanup on destroy
    this.someObservable$
      .pipe(takeUntilDestroyed(this.destroyRef))
      .subscribe();
  }
}

// Or with inject
@Component({})
export class InjectCleanupComponent {
  private destroyRef = inject(DestroyRef);

  ngOnInit(): void {
    someObservable$
      .pipe(takeUntilDestroyed(this.destroyRef))
      .subscribe();
  }
}
```

Lifecycle Hooks Order

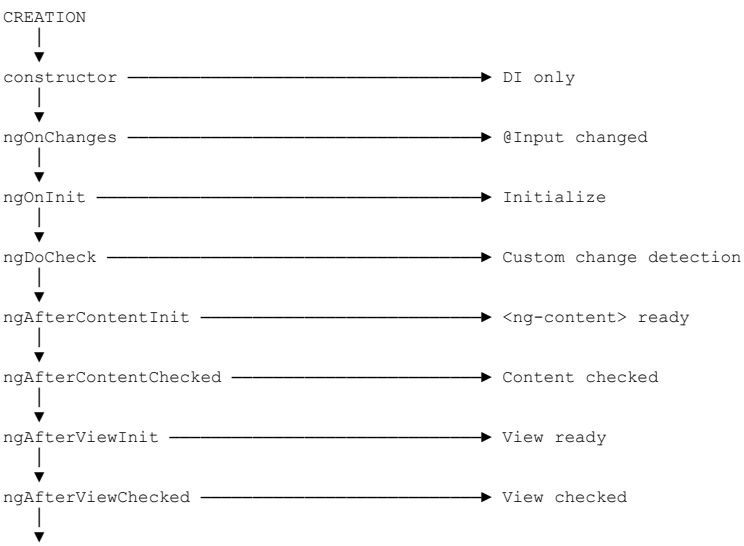
Answer:

```
1. constructor
2. ngOnChanges (if inputs exist)
3. ngOnInit
4. ngDoCheck
5. ngAfterContentInit
6. ngAfterContentChecked
7. ngAfterViewInit
8. ngAfterViewChecked

On subsequent changes:
- ngOnChanges (if inputs change)
- ngDoCheck
- ngAfterContentChecked
- ngAfterViewChecked

On destroy:
- ngOnDestroy
```

Visual Timeline:



```
(change detection cycles repeat above checks)
   |
   ▼
ngOnDestroy ──────────────────────────────────► Cleanup
```

---

ngOnChanges Deep Dive

Answer:

- Called whenever **@Input** property changes
- Receives **SimpleChanges** object with old/new values
- First call happens **before** ngOnInit

Example:

```
@Component({
  selector: 'app-user-card',
  template: `<div>{{ user?.name }}</div>`
})
export class UserCardComponent implements OnChanges, OnInit {
  @Input() user!: User;
  @Input() highlight = false;

  ngOnChanges(changes: SimpleChanges): void {
    console.log('Changes:', changes);

    // Check specific property
    if (changes['user']) {
      const userChange = changes['user'];
      console.log('Previous user:', userChange.previousValue);
      console.log('Current user:', userChange.currentValue);
      console.log('First change:', userChange.firstChange);

      if (!userChange.firstChange) {
        // React to user change (not initial)
        this.onUserChanged(userChange.currentValue);
      }
    }

    if (changes['highlight']) {
      this.updateHighlightStyle();
    }
  }

  ngOnInit(): void {
    // Called AFTER first ngOnChanges
    // user is already available here
  }

  private onUserChanged(newUser: User): void {
    // Handle user change
  }

  private updateHighlightStyle(): void {
    // Update styles
  }
}

// SimpleChanges structure
interface SimpleChanges {
  [propName: string]: SimpleChange;
}

interface SimpleChange {
  previousValue: any;
  currentValue: any;
  firstChange: boolean;
  isFirstChange(): boolean;
}
```

AfterViewInit vs AfterContentInit

Answer:

Hook	When	Access
AfterContentInit	After projected content	@ContentChild, @ContentChildren
AfterViewInit	After own view + children	@ViewChild, @ViewChildren

Example:

```
@Component({
  selector: 'app-tabs',
  template: `
    <!-- Projected content -->
    <ng-content></ng-content>
```

```
    <!-- Own view -->
    <div #tabContainer>
      <button *ngFor="let tab of tabs">{{ tab.label }}</button>
    </div>
  `
})
export class TabsComponent implements AfterContentInit, AfterViewInit {
  // From projected content (<ng-content>)
  @ContentChildren(TabComponent) projectedTabs!: QueryList<TabComponent>;

  // From own template
  @ViewChild('tabContainer') container!: ElementRef;

  tabs: TabComponent[] = [];

  ngAfterContentInit(): void {
    // Projected content is ready
    // projectedTabs QueryList is populated
    console.log('Projected tabs:', this.projectedTabs.length);
    this.tabs = this.projectedTabs.toArray();

    // Listen for changes
    this.projectedTabs.changes.subscribe(() => {
      this.tabs = this.projectedTabs.toArray();
    });
  }

  ngAfterViewInit(): void {
    // Own view is ready
    // ViewChild references are available
    console.log('Container:', this.container.nativeElement);

    // Safe to manipulate DOM here
    this.container.nativeElement.classList.add('initialized');
  }
}

// Usage
@Component({
  template: `
    <app-tabs>
      <app-tab label="Tab 1">Content 1</app-tab>
      <app-tab label="Tab 2">Content 2</app-tab>
    </app-tabs>
  `
})
export class AppComponent { }
```

CHANGE DETECTION ANSWERS

---

74. What is Change Detection?

Answer:

- **Change Detection** is Angular's mechanism to **sync model and view**
- Detects when data changes and **updates the DOM**
- Runs from **root component down** the component tree
- Triggered by: events, HTTP responses, timers, Promises

Theoretical Keywords:

Model-View sync, **DOM update**, **Component tree traversal**,
Zone.js, **Trigger events**, **Performance**, **Dirty checking**

Example:

```
@Component({
  selector: 'app-counter',
  template: `
    <p>Count: {{ count }}</p>
    <button (click)="increment()">+1</button>
  `
})
export class CounterComponent {
  count = 0;

  increment(): void {
    this.count++;  // Change detection runs after this
    // Angular detects 'count' changed and updates DOM
  }
}

// What triggers change detection:
// 1. User events (click, input, submit)
// 2. HTTP requests completing
// 3. setTimeout/setInterval
```

```
// 4. Promise resolution
// 5. Any async operation
```

---

75. What are the Change Detection strategies?

Answer:

Two strategies:

- **Default**: Check all components on every cycle
- **OnPush**: Check only when specific conditions met

Theoretical Keywords:

Default strategy, **OnPush strategy**, **Performance optimization**, **Immutable data**, **Observable inputs**, **Reference check**

Example:

```
// Default Strategy (checks every cycle)
@Component({
  selector: 'app-default',
  changeDetection: ChangeDetectionStrategy.Default,
  template: `{{ data.name }}`
})
export class DefaultComponent {
  @Input() data!: { name: string };
}

// OnPush Strategy (checks only when needed)
@Component({
  selector: 'app-onpush',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `{{ data.name }}`
})
export class OnPushComponent {
  @Input() data!: { name: string };
}

// Parent component
@Component({
  template: `
    <app-default [data]="user"></app-default>
    <app-onpush [data]="user"></app-onpush>
    <button (click)="mutate()">Mutate</button>
    <button (click)="replace()">Replace</button>
  `
})
export class ParentComponent {
  user = { name: 'John' };

  mutate(): void {
    // Mutation - Default updates, OnPush does NOT
    this.user.name = 'Jane';
  }

  replace(): void {
    // New reference - Both update
    this.user = { name: 'Jane' };
  }
}
```

---

76. What is OnPush Change Detection?

Answer:

OnPush only triggers change detection when:

1. **@Input reference** changes (not mutation)
2. **Event** originates from component or children
3. **Async pipe** receives new value
4. **Manually triggered** via ChangeDetectorRef

Example:

```
@Component({
  selector: 'app-user-card',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <div>{{ user.name }}</div>
    <div>{{ data$ | async }}</div>
    <button (click)="onClick()">Click</button>
  `
})
export class UserCardComponent {
```

```
  @Input() user!: User;

  data$ = this.dataService.getData();

  constructor(
    private dataService: DataService,
    private cdr: ChangeDetectorRef
  ) { }

  onClick(): void {
    // Event from this component - triggers change detection
  }

  // Manual trigger when needed
  forceUpdate(): void {
    this.cdr.markForCheck();  // Mark this and ancestors
    // or
    this.cdr.detectChanges();  // Run immediately on this component
  }
}

// Best practices with OnPush
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <!-- Use async pipe for observables -->
    <div *ngFor="let item of items$ | async">{{ item.name }}</div>

    <!-- Pass new object references, don't mutate -->
    <app-child [data]="immutableData"></app-child>
  `
})
export class OnPushBestPractices {
  items$: Observable<Item[]>;

  // GOOD: Create new array
  addItem(item: Item): void {
    this.items = [...this.items, item];
  }

  // BAD: Mutation won't trigger update
  badAddItem(item: Item): void {
    this.items.push(item);  // Won't work with OnPush!
  }
}
```

---

77. How to manually trigger Change Detection?

Answer:

Use **ChangeDetectorRef** methods:

- `markForCheck()`: Mark component and ancestors for check
- `detectChanges()`: Run change detection immediately
- `detach()`: Detach from change detection tree
- `reattach()`: Reattach to change detection tree

Example:

```
@Component({
  selector: 'app-manual-cd',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `<p>{{ value }}</p>`
})
export class ManualCDComponent implements OnInit {
  value = 'initial';

  constructor(
    private cdr: ChangeDetectorRef,
    private zone: NgZone
  ) { }

  ngOnInit(): void {
    // External update (WebSocket, etc.)
    externalLibrary.onUpdate((newValue) => {
      this.value = newValue;
      this.cdr.markForCheck();  // Tell Angular to check
    });
  }

  // markForCheck vs detectChanges
  usingMarkForCheck(): void {
    this.value = 'new value';
    this.cdr.markForCheck();
    // Marks for next change detection cycle
    // Angular will check on next tick
  }

  usingDetectChanges(): void {
    this.value = 'new value';
    this.cdr.detectChanges();
    // Runs change detection immediately
```

```
    // Only on this component and children
  }

  // Detach/Reattach for performance
  heavyProcessing(): void {
    this.cdr.detach();  // Stop automatic checks

    for (let i = 0; i < 10000; i++) {
      this.processItem(i);
    }

    this.cdr.reattach();  // Resume checks
    this.cdr.detectChanges();  // Update view
  }

  // Running outside Angular zone
  outsideAngular(): void {
    this.zone.runOutsideAngular(() => {
      // This won't trigger change detection
      setInterval(() => {
        this.someInternalUpdate();
      }, 100);
    });

    // When you need to update view
    this.zone.run(() => {
      this.cdr.detectChanges();
    });
  }
}

// Real-world example: Real-time chart
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class RealtimeChartComponent implements OnInit, OnDestroy {
  data: number[] = [];
  private subscription!: Subscription;

  constructor(
    private cdr: ChangeDetectorRef,
    private websocket: WebSocketService
  ) { }

  ngOnInit(): void {
    // Batch updates for performance
    this.subscription = this.websocket.dataStream$
      .pipe(
        bufferTime(100)  // Batch updates every 100ms
      )
      .subscribe(batch => {
        if (batch.length > 0) {
          this.data = [...this.data, ...batch].slice(-100);
          this.cdr.markForCheck();
        }
      });
  }

  ngOnDestroy(): void {
    this.subscription.unsubscribe();
  }
}
```

Zone.js and Change Detection

Answer:

- **Zone.js** patches async APIs to notify Angular
- Triggers change detection after async operations
- Can be bypassed for performance optimization

Example:

```
@Component({...})
export class ZoneExampleComponent {
  constructor(private zone: NgZone) { }

  // Inside Angular zone (triggers CD)
  insideZone(): void {
    setTimeout(() => {
      this.value = 'updated';  // CD runs automatically
    }, 1000);
  }

  // Outside Angular zone (no automatic CD)
  outsideZone(): void {
    this.zone.runOutsideAngular(() => {
      // Good for performance-critical operations
      setInterval(() => {
        this.internalCounter++;  // No CD triggered

        // Only update view when needed
```

```
        if (this.internalCounter % 100 === 0) {
          this.zone.run(() => {
            this.displayCounter = this.internalCounter;
          });
        }
      }, 10);
    });
  }

  // Check if inside Angular zone
  checkZone(): void {
    console.log('In Angular Zone:', NgZone.isInAngularZone());
  }
}

// Zoneless Angular (experimental in v18)
// bootstrapApplication(AppComponent, {
//   providers: [
//     provideExperimentalZonelessChangeDetection()
//   ]
// });
```

Change Detection Performance Tips

Answer:

1. Use **OnPush** strategy
2. Use **async pipe** with observables
3. **Immutable** data patterns
4. **trackBy** with ngFor
5. Avoid complex expressions in templates
6. Use **pure pipes** for transformations

Example:

```
// 1. OnPush + Immutable
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class OptimizedListComponent {
  @Input() items!: Item[];

  // 2. trackBy for ngFor
  trackById(index: number, item: Item): number {
    return item.id;
  }
}

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

// 3. Avoid this in templates
// BAD: Called every CD cycle
// <div>{{ getFullName() }}</div>

// GOOD: Use computed property or pipe
// <div>{{ fullName }}</div>
// <div>{{ user | fullName }}</div>

// 4. Pure pipe for transformations
@Pipe({ name: 'fullName', pure: true })
export class FullNamePipe implements PipeTransform {
  transform(user: User): string {
    return `${user.firstName} ${user.lastName}`;
  }
}
```

RxJS AND OBSERVABLES ANSWERS

---

78. What is RxJS?

Answer:

- **RxJS** (Reactive Extensions for JavaScript) is a library for **reactive programming**
- Uses **Observables** to work with async data streams
- Provides **operators** to transform, filter, combine streams
- Core part of Angular's architecture

Theoretical Keywords:

Reactive programming, **Observables**, **Data streams**,
Operators, **Async handling**, **Functional programming**

Example:

```
import { Observable, of, from, interval } from 'rxjs';
import { map, filter, take } from 'rxjs/operators';

// Creating observables
const simple$ = of(1, 2, 3, 4, 5);
const fromArray$ = from([1, 2, 3, 4, 5]);
const interval$ = interval(1000);  // Emits 0, 1, 2... every second

// Using operators
simple$.pipe(
  filter(n => n % 2 === 0),
  map(n => n * 10)
).subscribe(value => console.log(value));  // 20, 40
```

---

79. What is an Observable?

Answer:

- **Observable** is a stream of values over time
- Can emit **multiple values** (unlike Promise)
- **Lazy**: Doesn't execute until subscribed
- Can be **synchronous or asynchronous**
- Supports **cancellation** via unsubscribe

Theoretical Keywords:

Data stream, **Lazy evaluation**, **Multiple values**,
Subscribe, **Unsubscribe**, **Push-based**

Example:

```
// Creating Observable
const myObservable$ = new Observable<number>(subscriber => {
  subscriber.next(1);
  subscriber.next(2);
  subscriber.next(3);

  setTimeout(() => {
    subscriber.next(4);
    subscriber.complete();  // No more values
  }, 1000);

  // Cleanup function (called on unsubscribe)
  return () => {
    console.log('Cleanup');
  };
});

// Subscribing
const subscription = myObservable$.subscribe({
  next: (value) => console.log('Value:', value),
  error: (err) => console.error('Error:', err),
  complete: () => console.log('Complete')
});

// Output:
// Value: 1
// Value: 2
// Value: 3
// (after 1 second)
// Value: 4
// Complete
// Cleanup

// Unsubscribe to cancel
subscription.unsubscribe();
```

---

80. What is the difference between Observable and Promise?

Answer:

Feature	Observable	Promise
Values	Multiple	Single
Execution	Lazy (on subscribe)	Eager (immediate)
Cancellation	Yes (unsubscribe)	No
Operators	Many (map, filter, etc.)	Limited (then, catch)
Async	Sync or async	Always async

Example:

```
// Promise - Single value, eager
const promise = new Promise((resolve) => {
  console.log('Promise executing');  // Runs immediately
  resolve('done');
});

// Observable - Multiple values, lazy
const observable$ = new Observable(subscriber => {
  console.log('Observable executing');  // Only runs on subscribe
  subscriber.next('value 1');
  subscriber.next('value 2');
  subscriber.complete();
});

// Observable to Promise conversion
import { firstValueFrom, lastValueFrom } from 'rxjs';

const value = await firstValueFrom(observable$);
const last = await lastValueFrom(observable$);

// Promise to Observable conversion
import { from } from 'rxjs';
const obs$ = from(promise);
```

---

81. What are Subjects?

Answer:

- **Subject** is both **Observable** and **Observer**
- Can **multicast** to multiple subscribers
- Types: Subject, BehaviorSubject, ReplaySubject, AsyncSubject

Theoretical Keywords:

Multicast, **Hot observable**, **Observer pattern**,
Event emitter, **State management**, **Shared stream**

Example:

```
// Regular Subject
const subject = new Subject<number>();

subject.subscribe(v => console.log('A:', v));
subject.subscribe(v => console.log('B:', v));

subject.next(1);  // A: 1, B: 1
subject.next(2);  // A: 2, B: 2

// BehaviorSubject - Has current value
const behavior$ = new BehaviorSubject<number>(0);  // Initial value

behavior$.subscribe(v => console.log('Value:', v));  // Immediately: Value: 0
behavior$.next(1);  // Value: 1
console.log(behavior$.value);  // 1 (synchronous access)

// ReplaySubject - Replays N values to new subscribers
const replay$ = new ReplaySubject<number>(2);  // Buffer last 2

replay$.next(1);
replay$.next(2);
replay$.next(3);

replay$.subscribe(v => console.log('Replay:', v));
// Output: Replay: 2, Replay: 3 (last 2 values)

// AsyncSubject - Only emits last value on complete
const async$ = new AsyncSubject<number>();

async$.subscribe(v => console.log('Async:', v));
async$.next(1);
async$.next(2);
async$.next(3);
async$.complete();  // Now emits: Async: 3
```

---

82. What is BehaviorSubject?

Answer:

- **BehaviorSubject** is a Subject with a **current value**
- Requires an **initial value**
- New subscribers immediately receive **current value**
- Commonly used for **state management**

Example:

```
// State management with BehaviorSubject
@Injectable({ providedIn: 'root' })
export class AuthService {
  private currentUserSubject = new BehaviorSubject<User | null>(null);

  currentUser$ = this.currentUserSubject.asObservable();

  // Synchronous access to current value
  get currentUser(): User | null {
    return this.currentUserSubject.value;
  }

  login(credentials: Credentials): Observable<User> {
    return this.http.post<User>('/api/login', credentials).pipe(
      tap(user => this.currentUserSubject.next(user))
    );
  }

  logout(): void {
    this.currentUserSubject.next(null);
  }

  isLoggedIn(): boolean {
    return this.currentUserSubject.value !== null;
  }
}

// Component usage
@Component({
  template: `
    <div *ngIf="currentUser$ | async as user">
      Welcome, {{ user.name }}!
    </div>
  `
})
export class NavComponent {
  currentUser$ = this.authService.currentUser$;

  constructor(private authService: AuthService) { }
}
```

---

83. What are the most common RxJS operators?

Answer:

Categories of operators:

- **Creation**: of, from, interval, timer
- **Transformation**: map, switchMap, mergeMap
- **Filtering**: filter, take, debounceTime
- **Combination**: combineLatest, merge, forkJoin
- **Error handling**: catchError, retry

Example:

```
// Transformation operators
of(1, 2, 3).pipe(
  map(x => x * 10)  // 10, 20, 30
);

// Filtering operators
of(1, 2, 3, 4, 5).pipe(
  filter(x => x % 2 === 0),  // 2, 4
  take(1)  // Only first: 2
);

// Higher-order mapping
// switchMap - Cancel previous, use latest
this.searchInput.valueChanges.pipe(
  debounceTime(300),
  switchMap(term => this.searchService.search(term))
);

// mergeMap - Run all in parallel
from(userIds).pipe(
  mergeMap(id => this.userService.getUser(id))
);

// concatMap - Run sequentially
from(tasks).pipe(
  concatMap(task => this.taskService.process(task))
);

// exhaustMap - Ignore while processing
this.submitButton$.pipe(
  exhaustMap(() => this.formService.submit(this.form.value))
);

// Combination operators
```

```
combineLatest([user$, permissions$]).pipe(
  map(([user, permissions]) => ({ user, permissions }))
);

forkJoin({
  users: this.userService.getUsers(),
  products: this.productService.getProducts()
}).subscribe(({ users, products }) => {
  // Both completed
});

// Error handling
this.http.get('/api/data').pipe(
  retry(3),
  catchError(error => {
    console.error(error);
    return of([]);  // Return fallback value
  })
);
```

---

84. What is the difference between switchMap, mergeMap, and concatMap?

Answer:

Operator	Behavior	Use Case
switchMap	Cancel previous, only latest	Search, autocomplete
mergeMap	All run parallel	Parallel requests
concatMap	Run sequentially	Order matters
exhaustMap	Ignore until complete	Submit button

Example:

```
// switchMap - Typeahead search
// Cancels previous request when new search term arrives
this.searchControl.valueChanges.pipe(
  debounceTime(300),
  distinctUntilChanged(),
  switchMap(term => this.api.search(term))
).subscribe(results => {
  this.searchResults = results;
});

// mergeMap - Fetch all in parallel
// All requests run simultaneously
const userIds = [1, 2, 3, 4, 5];
from(userIds).pipe(
  mergeMap(id => this.userService.getUser(id), 3)  // Max 3 concurrent
).subscribe(user => {
  this.users.push(user);
});

// concatMap - Sequential operations
// Wait for each to complete before starting next
from(this.filesToUpload).pipe(
  concatMap(file => this.uploadService.upload(file))
).subscribe(result => {
  console.log('File uploaded:', result);
});

// exhaustMap - Prevent duplicate submissions
this.saveButton.pipe(
  exhaustMap(() => this.saveData())  // Ignore clicks while saving
).subscribe(result => {
  console.log('Saved');
});

// Visual comparison
// Input:    --1--2--3-->
// API call takes: =====>

// switchMap: --1==X2==X3====>  (cancels 1, 2)
// mergeMap:  --1=====>
//            ---2=====>
//            ----3=====>       (all parallel)
// concatMap: --1======2======3======>  (sequential)
// exhaustMap: --1=====>3=====>  (ignores 2)
```

---

85. How to handle errors in RxJS?

Answer:

- **catchError**: Handle error and return fallback
- **retry**: Retry failed operation
- **retryWhen**: Conditional retry with delay
- **finalize**: Execute on complete or error

Example:

```
// Basic error handling
this.http.get<User[]>('/api/users').pipe(
  catchError(error => {
    console.error('Error fetching users:', error);
    return of([]);  // Return empty array as fallback
  })
).subscribe(users => {
  this.users = users;
});

// Retry with delay
this.http.get('/api/data').pipe(
  retry({
    count: 3,
    delay: 1000  // Wait 1 second between retries
  }),
  catchError(error => {
    return throwError(() => new Error('Failed after 3 retries'));
  })
);

// Exponential backoff retry
this.http.get('/api/data').pipe(
  retryWhen(errors =>
    errors.pipe(
      scan((retryCount, error) => {
        if (retryCount >= 3) throw error;
        return retryCount + 1;
      }, 0),
      delay(attempt => Math.pow(2, attempt) * 1000)  // 1s, 2s, 4s
    )
  )
);

// Using finalize for cleanup
this.loading = true;
this.http.get('/api/data').pipe(
  finalize(() => {
    this.loading = false;  // Always runs
  })
).subscribe({
  next: (data) => this.data = data,
  error: (error) => this.error = error.message
});

// Global error handling with interceptor
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error.status === 401) {
          // Redirect to login
        }
        if (error.status === 500) {
          // Show error notification
        }
        return throwError(() => error);
      })
    );
  }
}
```

Common RxJS Patterns in Angular

Answer:

```
// 1. Unsubscribe pattern
export class MyComponent implements OnDestroy {
  private destroy$ = new Subject<void>();

  ngOnInit(): void {
    this.dataService.getData()
      .pipe(takeUntil(this.destroy$))
      .subscribe();
  }

  ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }
}

// 2. Async pipe (auto unsubscribe)
@Component({
  template: `<div *ngFor="let item of items$ | async">{{ item.name }}</div>`
})
export class ListComponent {
  items$ = this.service.getItems();
}
```

```
// 3. Share single HTTP request
@Injectable()
export class DataService {
  private data$?: Observable<Data>;

  getData(): Observable<Data> {
    if (!this.data$) {
      this.data$ = this.http.get<Data>('/api/data').pipe(
        shareReplay(1)
      );
    }
    return this.data$;
  }
}

// 4. Polling
interval(5000).pipe(
  startWith(0),
  switchMap(() => this.http.get('/api/status')),
  takeUntil(this.destroy$)
).subscribe(status => {
  this.status = status;
});

// 5. Combine multiple sources
combineLatest([
  this.route.params,
  this.route.queryParams,
  this.authService.currentUser$
]).pipe(
  switchMap(([params, query, user]) =>
    this.dataService.getData(params['id'], query, user)
  )
).subscribe(data => {
  this.data = data;
});
```

MODULES AND ARCHITECTURE ANSWERS

---

86. What are Angular Modules?

Answer:

- **NgModule** is a container for related code (components, services, etc.)
- Organizes application into **cohesive blocks**
- Controls **component visibility** and **dependencies**
- Enables **lazy loading**

Theoretical Keywords:

NgModule, **Organization**, **Encapsulation**, **Dependencies**,
Feature modules, **Shared modules**, **Lazy loading**

Example:

```
@NgModule({
  declarations: [    // Components, directives, pipes owned by this module
    UserListComponent,
    UserDetailComponent,
    UserFilterPipe
  ],
  imports: [         // Other modules needed
    CommonModule,
    FormsModule,
    SharedModule
  ],
  exports: [         // Make available to other modules
    UserListComponent
  ],
  providers: [       // Services (prefer providedIn: 'root')
    UserService
  ]
})
export class UserModule { }
```

---

87. Types of Angular modules?

Answer:

Type	Purpose	Example
Root (App)	Bootstrap application	AppModule
Feature	Encapsulate feature	UserModule, ProductModule

Type	Purpose	Example
Shared	Reusable components	SharedModule
Core	Singleton services	CoreModule
Routing	Route configuration	AppRoutingModule

Example:

```
// Root Module
@NgModule({
  declarations: [AppComponent],
  imports: [
    BrowserModule,
    AppRoutingModule,
    CoreModule,
    SharedModule
  ],
  bootstrap: [AppComponent]
})
export class AppModule { }

// Feature Module
@NgModule({
  declarations: [
    ProductListComponent,
    ProductDetailComponent
  ],
  imports: [
    CommonModule,
    ProductRoutingModule,
    SharedModule
  ]
})
export class ProductModule { }

// Shared Module
@NgModule({
  declarations: [
    ButtonComponent,
    CardComponent,
    LoadingSpinnerComponent,
    TruncatePipe
  ],
  imports: [CommonModule],
  exports: [
    // Re-export commonly used modules
    CommonModule,
    FormsModule,
    // Export shared components
    ButtonComponent,
    CardComponent,
    LoadingSpinnerComponent,
    TruncatePipe
  ]
})
export class SharedModule { }

// Core Module (singleton services)
@NgModule({
  providers: [
    AuthService,
    LoggingService,
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
  ]
})
export class CoreModule {
  // Prevent re-import
  constructor(@Optional() @SkipSelf() parentModule: CoreModule) {
    if (parentModule) {
      throw new Error('CoreModule already loaded. Import only in AppModule.');
    }
  }
}
```

---

88. What is the difference between declarations and imports?

Answer:

Property	Purpose	Contains
declarations	Define what belongs to module	Components, Directives, Pipes
imports	What this module needs	Other NgModules
exports	What to share with other modules	Components, Modules
providers	Services available	Services

Example:

```
@NgModule({
  // Components/Directives/Pipes that BELONG to this module
```

```
  declarations: [
    MyComponent,        // Component
    HighlightDirective, // Directive
    TruncatePipe        // Pipe
  ],

  // Modules this module NEEDS
  imports: [
    CommonModule,       // For *ngIf, *ngFor, etc.
    FormsModule,        // For ngModel
    HttpClientModule,   // For HttpClient
    FeatureModule       // Custom module
  ],

  // What to SHARE with modules that import this
  exports: [
    MyComponent,
    TruncatePipe,
    CommonModule  // Re-export
  ],

  // Services (prefer providedIn: 'root' instead)
  providers: [
    MyService
  ]
})
export class MyModule { }
```

---

89. What is a Shared Module?

Answer:

- Contains **commonly used** components, directives, pipes
- **Imported by multiple** feature modules
- **Exports** items for use elsewhere
- Does NOT contain services (usually)

Example:

```
// shared/shared.module.ts
@NgModule({
  declarations: [
    // UI Components
    ButtonComponent,
    CardComponent,
    ModalComponent,
    AlertComponent,
    LoadingComponent,
    PaginationComponent,

    // Directives
    ClickOutsideDirective,
    AutoFocusDirective,

    // Pipes
    DateFormatPipe,
    CurrencyFormatPipe,
    TruncatePipe,
    SafeHtmlPipe
  ],
  imports: [
    CommonModule,
    FormsModule,
    ReactiveFormsModule,
    RouterModule
  ],
  exports: [
    // Re-export Angular modules
    CommonModule,
    FormsModule,
    ReactiveFormsModule,
    RouterModule,

    // Export all shared components
    ButtonComponent,
    CardComponent,
    ModalComponent,
    AlertComponent,
    LoadingComponent,
    PaginationComponent,

    // Export directives
    ClickOutsideDirective,
    AutoFocusDirective,

    // Export pipes
    DateFormatPipe,
    CurrencyFormatPipe,
    TruncatePipe,
    SafeHtmlPipe
  ]
})
```

```
export class SharedModule { }

// Usage in feature module
@NgModule({
  imports: [SharedModule]  // Gets all exported items
})
export class FeatureModule { }
```

---

90. What is the Core Module?

Answer:

- Contains **singleton services** used across the app
- **Imported ONLY in AppModule**
- Contains **HTTP interceptors**, **guards**
- Prevents duplicate service instances

Example:

```
// core/core.module.ts
@NgModule({
  declarations: [],
  imports: [
    CommonModule,
    HttpClientModule
  ],
  providers: [
    // Singleton services
    AuthService,
    LoggingService,
    NotificationService,

    // HTTP Interceptors
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
    { provide: HTTP_INTERCEPTORS, useClass: ErrorInterceptor, multi: true },
    { provide: HTTP_INTERCEPTORS, useClass: LoadingInterceptor, multi: true },

    // Guards
    AuthGuard,
    AdminGuard
  ]
})
export class CoreModule {
  // Ensure CoreModule is only imported once
  constructor(@Optional() @SkipSelf() parentModule: CoreModule) {
    if (parentModule) {
      throw new Error(
        'CoreModule is already loaded. Import it only in AppModule.'
      );
    }
  }

  // Alternative: Static forRoot pattern
  static forRoot(): ModuleWithProviders<CoreModule> {
    return {
      ngModule: CoreModule,
      providers: [
        AuthService,
        LoggingService
      ]
    };
  }
}

// app.module.ts
@NgModule({
  imports: [
    BrowserModule,
    CoreModule,  // Import only here
    SharedModule,
    AppRoutingModule
  ]
})
export class AppModule { }
```

Standalone Components

Answer:

- **Angular 14+** feature
- Components without NgModule
- Declare dependencies directly
- Simpler, more tree-shakable

Example:

```
// Standalone component
@Component({
  selector: 'app-user-card',
  standalone: true,
  imports: [
    CommonModule,
    RouterModule,
    DatePipe
  ],
  template: `
    <div class="card">
      <h3>{{ user.name }}</h3>
      <p>{{ user.createdAt | date:'mediumDate' }}</p>
      <a [routerLink]="['/users', user.id]">View Details</a>
    </div>
  `
})
export class UserCardComponent {
  @Input() user!: User;
}

// Using in another standalone component
@Component({
  standalone: true,
  imports: [UserCardComponent],
  template: `
    <app-user-card [user]="user"></app-user-card>
  `
})
export class UserProfileComponent { }

// Bootstrapping with standalone
// main.ts
import { bootstrapApplication } from '@angular/platform-browser';
import { provideRouter } from '@angular/router';
import { provideHttpClient } from '@angular/common/http';

bootstrapApplication(AppComponent, {
  providers: [
    provideRouter(routes),
    provideHttpClient(),
    provideAnimations()
  ]
});

// Lazy loading standalone component
const routes: Routes = [
  {
    path: 'dashboard',
    loadComponent: () => import('./dashboard/dashboard.component')
      .then(c => c.DashboardComponent)
  }
];
```

Module Organization Best Practices

Answer:

```
src/
├── app/
│   ├── core/                    # Singleton services, guards
│   │   ├── services/
│   │   │   ├── auth.service.ts
│   │   │   └── api.service.ts
│   │   ├── guards/
│   │   ├── interceptors/
│   │   └── core.module.ts
│   │
│   ├── shared/                  # Reusable components
│   │   ├── components/
│   │   │   ├── button/
│   │   │   └── modal/
│   │   ├── directives/
│   │   ├── pipes/
│   │   └── shared.module.ts
│   │
│   ├── features/                # Feature modules
│   │   ├── users/
│   │   │   ├── components/
│   │   │   ├── services/
│   │   │   ├── users-routing.module.ts
│   │   │   └── users.module.ts
│   │   │
│   │   └── products/
│   │       ├── components/
│   │       ├── services/
│   │       └── products.module.ts
│   │
│   ├── layout/                  # Layout components
│   │   ├── header/
│   │   ├── footer/
│   │   └── sidebar/
│   │
│   ├── app-routing.module.ts
```

```
│   ├── app.component.ts
│   └── app.module.ts
```

FREQUENTLY ASKED AND SCENARIO QUESTIONS ANSWERS

---

91. What is AOT vs JIT compilation?

Answer:

Feature	AOT (Ahead-of-Time)	JIT (Just-in-Time)
When	Build time	Runtime (browser)
Bundle size	Smaller	Larger (includes compiler)
Startup	Faster	Slower
Error detection	Build time	Runtime
Default	Production	Development

Example:

```
# JIT compilation (development)
ng serve

# AOT compilation (production - default)
ng build --configuration=production

# AOT in development
ng serve --aot
```

```
// AOT catches template errors at build time
@Component({
  template: `{{ user.nmae }}`  // Typo: 'nmae' instead of 'name'
})
// AOT: Build error - Property 'nmae' does not exist
// JIT: Runtime error in browser
```

---

92. What are HTTP Interceptors?

Answer:

- **Interceptors** intercept HTTP requests/responses
- Used for: Auth tokens, logging, error handling, caching
- Implement `HttpInterceptor` interface

Example:

```
// Auth Interceptor
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) { }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = this.authService.getToken();

    if (token) {
      const authReq = req.clone({
        setHeaders: {
          Authorization: `Bearer ${token}`
        }
      });
      return next.handle(authReq);
    }

    return next.handle(req);
  }
}

// Error Interceptor
@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  constructor(private router: Router) { }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error.status === 401) {
          this.router.navigate(['/login']);
        }
        if (error.status === 403) {
          this.router.navigate(['/forbidden']);
        }
```

```
        return throwError(() => error);
      })
    );
  }
}

// Loading Interceptor
@Injectable()
export class LoadingInterceptor implements HttpInterceptor {
  constructor(private loadingService: LoadingService) { }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    this.loadingService.show();

    return next.handle(req).pipe(
      finalize(() => this.loadingService.hide())
    );
  }
}

// Register interceptors
@NgModule({
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true },
    { provide: HTTP_INTERCEPTORS, useClass: ErrorInterceptor, multi: true },
    { provide: HTTP_INTERCEPTORS, useClass: LoadingInterceptor, multi: true }
  ]
})
export class CoreModule { }

// Functional interceptor (Angular 15+)
export const authInterceptor: HttpInterceptorFn = (req, next) => {
  const token = inject(AuthService).getToken();

  if (token) {
    req = req.clone({
      setHeaders: { Authorization: `Bearer ${token}` }
    });
  }

  return next(req);
};

// Provide functional interceptor
bootstrapApplication(AppComponent, {
  providers: [
    provideHttpClient(withInterceptors([authInterceptor]))
  ]
});
```

---

93. What is View Encapsulation?

Answer:

- Controls how styles are **scoped** to component
- Three modes: **Emulated**, **None**, **ShadowDom**

Mode	Behavior
Emulated	Scoped via attributes (default)
None	Global styles
ShadowDom	Native Shadow DOM

Example:

```
// Emulated (default) - Styles scoped to component
@Component({
  selector: 'app-card',
  encapsulation: ViewEncapsulation.Emulated,
  styles: [`
    .title { color: blue; }  /* Only affects this component */
  `]
})
export class CardComponent { }

// None - Styles are global
@Component({
  selector: 'app-global',
  encapsulation: ViewEncapsulation.None,
  styles: [`
    .title { color: red; }  /* Affects entire application */
  `]
})
export class GlobalComponent { }

// ShadowDom - Native shadow DOM encapsulation
@Component({
  selector: 'app-shadow',
  encapsulation: ViewEncapsulation.ShadowDom,
  styles: [`
    .title { color: green; }  /* True isolation */
```

```
  `]
})
export class ShadowComponent { }

// Penetrating encapsulation with ::ng-deep
@Component({
  styles: [`
    :host ::ng-deep .child-style {
      color: purple;  /* Affects child components */
    }
  `]
})
export class ParentComponent { }
```

---

94. What is Content Projection?

Answer:

- Pass content from parent to child component
- Uses `<ng-content>` tag
- Supports **single slot**, **multi-slot**, **conditional**

Example:

```
// Single slot projection
@Component({
  selector: 'app-card',
  template: `
    <div class="card">
      <ng-content></ng-content>
    </div>
  `
})
export class CardComponent { }

// Usage
// <app-card>
//   <h1>Title</h1>
//   <p>Content goes here</p>
// </app-card>

// Multi-slot projection
@Component({
  selector: 'app-modal',
  template: `
    <div class="modal">
      <header>
        <ng-content select="[modal-header]"></ng-content>
      </header>
      <main>
        <ng-content select="[modal-body]"></ng-content>
      </main>
      <footer>
        <ng-content select="[modal-footer]"></ng-content>
      </footer>
    </div>
  `
})
export class ModalComponent { }

// Usage
// <app-modal>
//   <h2 modal-header>Title</h2>
//   <div modal-body>Content</div>
//   <button modal-footer>Close</button>
// </app-modal>

// Conditional projection with ngProjectAs
// <app-card>
//   <ng-container ngProjectAs="[modal-header]">
//     <h2 *ngIf="showHeader">Dynamic Header</h2>
//   </ng-container>
// </app-card>
```

---

95. What is ng-template?

Answer:

- **ng-template** defines template that isn't rendered directly
- Used with **structural directives**, **ngTemplateOutlet**
- For **conditional rendering**, **reusable templates**

Example:

```
@Component({
  template: `
    <!-- Basic ng-template -->
```

```
    <ng-template #loading>
      <div class="spinner">Loading...</div>
    </ng-template>

    <!-- Usage with *ngIf -->
    <div *ngIf="data; else loading">
      {{ data | json }}
    </div>

    <!-- ngTemplateOutlet for reuse -->
    <ng-container *ngTemplateOutlet="loading"></ng-container>

    <!-- Template with context -->
    <ng-template #userCard let-user let-index="index">
      <div class="card">
        {{ index + 1 }}. {{ user.name }}
      </div>
    </ng-template>

    <ng-container
      *ngTemplateOutlet="userCard; context: { $implicit: currentUser, index: 0 }">
    </ng-container>

    <!-- Passing template to child -->
    <app-list [itemTemplate]="userCard" [items]="users"></app-list>
  `
})
export class TemplateExampleComponent {
  data: any;
  currentUser = { name: 'John' };
  users = [{ name: 'John' }, { name: 'Jane' }];
}

// Child component receiving template
@Component({
  selector: 'app-list',
  template: `
    <div *ngFor="let item of items; let i = index">
      <ng-container *ngTemplateOutlet="itemTemplate; context: { $implicit: item, index: i }">
      </ng-container>
    </div>
  `
})
export class ListComponent {
  @Input() items: any[] = [];
  @Input() itemTemplate!: TemplateRef<any>;
}
```

---

96. How to communicate between components?

Answer:

Method	Use Case
@Input/@Output	Parent ↔ Child
Service	Any components
ViewChild	Parent → Child
ContentChild	Projected content
Router	Navigation data

Example:

```
// 1. @Input/@Output (Parent ↔ Child)
@Component({
  selector: 'app-child',
  template: `<button (click)="sendMessage()">Send</button>`
})
export class ChildComponent {
  @Input() data!: string;
  @Output() message = new EventEmitter<string>();

  sendMessage(): void {
    this.message.emit('Hello from child');
  }
}

// Parent
// <app-child [data]="parentData" (message)="onMessage($event)"></app-child>

// 2. Service (Any components)
@Injectable({ providedIn: 'root' })
export class CommunicationService {
  private messageSubject = new Subject<string>();
  message$ = this.messageSubject.asObservable();

  sendMessage(message: string): void {
    this.messageSubject.next(message);
  }
}

// Component A (sender)
```

```
this.communicationService.sendMessage('Hello');

// Component B (receiver)
this.communicationService.message$.subscribe(msg => console.log(msg));

// 3. @ViewChild (Parent → Child)
@Component({
  template: `<app-child #child></app-child>`
})
export class ParentComponent implements AfterViewInit {
  @ViewChild('child') childComponent!: ChildComponent;

  ngAfterViewInit(): void {
    this.childComponent.someMethod();
  }
}
```

---

97. What is the async pipe?

Answer:

- **Async pipe** subscribes to Observable/Promise
- **Automatically unsubscribes** on destroy
- Returns **latest value**
- Best practice for displaying async data

Example:

```
@Component({
  template: `
    <!-- Basic usage -->
    <div *ngIf="user$ | async as user">
      Welcome, {{ user.name }}!
    </div>

    <!-- With loading state -->
    <ng-container *ngIf="data$ | async as data; else loading">
      <div *ngFor="let item of data">{{ item.name }}</div>
    </ng-container>
    <ng-template #loading>Loading...</ng-template>

    <!-- Multiple subscriptions (avoid!) -->
    <div>{{ (user$ | async)?.name }}</div>
    <div>{{ (user$ | async)?.email }}</div>

    <!-- Better: Single subscription -->
    <ng-container *ngIf="user$ | async as user">
      <div>{{ user.name }}</div>
      <div>{{ user.email }}</div>
    </ng-container>

    <!-- With error handling -->
    <ng-container *ngIf="{
      data: data$ | async,
      error: error$ | async,
      loading: loading$ | async
    } as vm">
      <div *ngIf="vm.loading">Loading...</div>
      <div *ngIf="vm.error" class="error">{{ vm.error }}</div>
      <div *ngIf="vm.data">{{ vm.data | json }}</div>
    </ng-container>
  `
})
export class AsyncPipeComponent {
  user$ = this.userService.getCurrentUser();
  data$ = this.dataService.getData();
  error$ = this.errorService.error$;
  loading$ = this.loadingService.loading$;

  constructor(
    private userService: UserService,
    private dataService: DataService,
    private errorService: ErrorService,
    private loadingService: LoadingService
  ) { }
}
```

---

98. What is trackBy in *ngFor?

Answer:

- **trackBy** tells Angular how to **identify items**
- Prevents **unnecessary DOM re-renders**
- Improves **performance** for large lists

Example:

```
@Component({
  template: `
    <!-- Without trackBy - entire list re-renders -->
    <div *ngFor="let item of items">{{ item.name }}</div>

    <!-- With trackBy - only changed items re-render -->
    <div *ngFor="let item of items; trackBy: trackById">
      {{ item.name }}
    </div>
  `
})
export class ListComponent {
  items: Item[] = [];

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

  // Alternative: track by index (less ideal)
  trackByIndex(index: number): number {
    return index;
  }

  // When items change
  refreshData(): void {
    this.items = this.items.map(item => ({ ...item }));
    // Without trackBy: All DOM elements recreated
    // With trackBy: Only changed elements updated
  }
}
```

---

99. Scenario: Parent-child data flow

Question:

How would you pass data from parent to child and receive events back?

Answer:

```
// Parent Component
@Component({
  selector: 'app-parent',
  template: `
    <h1>Parent Component</h1>
    <p>Selected: {{ selectedUser?.name }}</p>

    <app-user-list
      [users]="users"
      [selectedId]="selectedUser?.id"
      (userSelected)="onUserSelected($event)"
      (userDeleted)="onUserDeleted($event)">
    </app-user-list>
  `
})
export class ParentComponent implements OnInit {
  users: User[] = [];
  selectedUser: User | null = null;

  constructor(private userService: UserService) { }

  ngOnInit(): void {
    this.userService.getUsers().subscribe(users => {
      this.users = users;
    });
  }

  onUserSelected(user: User): void {
    this.selectedUser = user;
  }

  onUserDeleted(userId: number): void {
    this.users = this.users.filter(u => u.id !== userId);
    if (this.selectedUser?.id === userId) {
      this.selectedUser = null;
    }
  }
}

// Child Component
@Component({
  selector: 'app-user-list',
  template: `
    <ul>
      <li *ngFor="let user of users; trackBy: trackById"
          [class.selected]="user.id === selectedId"
          (click)="selectUser(user)">
        {{ user.name }}
        <button (click)="deleteUser(user.id, $event)">Delete</button>
      </li>
    </ul>
  `
})
export class UserListComponent {
```

```
  @Input() users: User[] = [];
  @Input() selectedId: number | null = null;

  @Output() userSelected = new EventEmitter<User>();
  @Output() userDeleted = new EventEmitter<number>();

  trackById(index: number, user: User): number {
    return user.id;
  }

  selectUser(user: User): void {
    this.userSelected.emit(user);
  }

  deleteUser(userId: number, event: Event): void {
    event.stopPropagation();
    this.userDeleted.emit(userId);
  }
}
```

---

100. Scenario: Service-based communication

Question:

How would you share data between unrelated components?

Answer:

```
// Shared Service
@Injectable({ providedIn: 'root' })
export class NotificationService {
  private notificationSubject = new BehaviorSubject<Notification | null>(null);
  notification$ = this.notificationSubject.asObservable();

  show(message: string, type: 'success' | 'error' | 'info'): void {
    this.notificationSubject.next({ message, type, id: Date.now() });

    // Auto-hide after 5 seconds
    setTimeout(() => this.hide(), 5000);
  }

  hide(): void {
    this.notificationSubject.next(null);
  }
}

// Component A (sender - anywhere in app)
@Component({...})
export class FormComponent {
  constructor(private notificationService: NotificationService) { }

  onSubmit(): void {
    this.saveData().subscribe({
      next: () => this.notificationService.show('Saved!', 'success'),
      error: () => this.notificationService.show('Error saving', 'error')
    });
  }
}

// Component B (receiver - in app.component)
@Component({
  selector: 'app-notification',
  template: `
    <div *ngIf="notification$ | async as notification"
         [class]="notification.type">
      {{ notification.message }}
      <button (click)="close()">×</button>
    </div>
  `
})
export class NotificationComponent {
  notification$ = this.notificationService.notification$;

  constructor(private notificationService: NotificationService) { }

  close(): void {
    this.notificationService.hide();
  }
}
```

---

101. Scenario: Dynamic component loading

Question:

How would you load components dynamically at runtime?

Answer:

```
// Dynamic component loader
@Component({
  selector: 'app-dynamic-host',
  template: `<ng-container #container></ng-container>`
})
export class DynamicHostComponent implements OnDestroy {
  @ViewChild('container', { read: ViewContainerRef })
  container!: ViewContainerRef;

  private componentRef: ComponentRef<any> | null = null;

  loadComponent<T>(component: Type<T>, inputs?: Partial<T>): ComponentRef<T> {
    this.container.clear();
    this.componentRef = this.container.createComponent(component);

    if (inputs) {
      Object.assign(this.componentRef.instance, inputs);
    }

    return this.componentRef;
  }

  ngOnDestroy(): void {
    this.componentRef?.destroy();
  }
}

// Usage
@Component({
  template: `
    <app-dynamic-host></app-dynamic-host>
    <button (click)="loadChart()">Load Chart</button>
    <button (click)="loadTable()">Load Table</button>
  `
})
export class DashboardComponent {
  @ViewChild(DynamicHostComponent) host!: DynamicHostComponent;

  loadChart(): void {
    this.host.loadComponent(ChartComponent, {
      data: this.chartData
    });
  }

  loadTable(): void {
    this.host.loadComponent(TableComponent, {
      data: this.tableData,
      columns: ['name', 'age', 'email']
    });
  }
}
```

---

102. Scenario: Form with dynamic fields

Question:

How would you create a form with dynamic fields?

Answer:

```
@Component({
  selector: 'app-dynamic-form',
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <div formArrayName="fields">
        <div *ngFor="let field of fields.controls; let i = index" [formGroupName]="i">
          <input formControlName="label" placeholder="Field Label">
          <select formControlName="type">
            <option value="text">Text</option>
            <option value="number">Number</option>
            <option value="email">Email</option>
          </select>
          <input type="checkbox" formControlName="required"> Required
          <button type="button" (click)="removeField(i)">Remove</button>
        </div>
      </div>

      <button type="button" (click)="addField()">Add Field</button>
      <button type="submit" [disabled]="form.invalid">Submit</button>
    </form>

    <!-- Preview generated form -->
    <h3>Preview:</h3>
    <form [formGroup]="generatedForm">
      <div *ngFor="let field of fields.value">
        <label>{{ field.label }}</label>
        <input [type]="field.type" [formControlName]="field.label">
      </div>
    </form>
  `
})
export class DynamicFormComponent {
  form = new FormGroup({
```

```
    fields: new FormArray<FormGroup>([])
  });

  generatedForm = new FormGroup({});

  get fields(): FormArray<FormGroup> {
    return this.form.get('fields') as FormArray<FormGroup>;
  }

  addField(): void {
    const fieldGroup = new FormGroup({
      label: new FormControl('', Validators.required),
      type: new FormControl('text'),
      required: new FormControl(false)
    });
    this.fields.push(fieldGroup);
    this.updateGeneratedForm();
  }

  removeField(index: number): void {
    this.fields.removeAt(index);
    this.updateGeneratedForm();
  }

  updateGeneratedForm(): void {
    const newForm = new FormGroup({});
    this.fields.value.forEach(field => {
      const validators = field.required ? [Validators.required] : [];
      newForm.addControl(field.label, new FormControl('', validators));
    });
    this.generatedForm = newForm;
  }

  onSubmit(): void {
    console.log('Field definitions:', this.form.value);
    console.log('Generated form values:', this.generatedForm.value);
  }
}
```

MICRO FRONTENDS ANSWERS

---

103. What is Micro Frontend Architecture?

Answer:

- **Micro Frontends** extend microservices to the frontend
- Break large frontend into **smaller, independent applications**
- Each team owns end-to-end feature (including UI)
- Can use **different frameworks**
- Independent **deployment and development**

Theoretical Keywords:

Microservices for frontend, **Independent deployment**, **Team autonomy**, **Technology agnostic**, **Scalable architecture**, **Module Federation**

Example:

```
Traditional Monolith:
┌──────────────────────────────────────┐
│           Single Angular App         │
│  ┌─────┐ ┌─────┐ ┌─────┐ ┌─────┐     │
│  │Users│ │Cart │ │Prod │ │Auth │     │
│  └─────┘ └─────┘ └─────┘ └─────┘     │
└──────────────────────────────────────┘

Micro Frontend:
┌──────────────────────────────────────┐
│             Shell/Container          │
│  ┌──────────┐ ┌──────────┐ ┌──────────┐ │
│  │MFE: User │ │MFE: Cart │ │MFE: Prod │ │
│  │(Angular) │ │ (React)  │ │  (Vue)   │ │
│  │Team A    │ │ Team B   │ │ Team C   │ │
│  └──────────┘ └──────────┘ └──────────┘ │
└──────────────────────────────────────┘
```

---

104. What is Module Federation?

Answer:

- **Module Federation** is a Webpack 5 feature
- Allows **sharing code** between applications at runtime
- Load **remote modules** dynamically

- Share **dependencies** to avoid duplication

Example:

```
// webpack.config.js - Remote App (products)
const ModuleFederationPlugin = require('webpack/lib/container/ModuleFederationPlugin');

module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'products',
      filename: 'remoteEntry.js',
      exposes: {
        './ProductList': './src/app/product-list/product-list.component.ts',
        './ProductModule': './src/app/products/products.module.ts'
      },
      shared: {
        '@angular/core': { singleton: true, strictVersion: true },
        '@angular/common': { singleton: true, strictVersion: true },
        '@angular/router': { singleton: true, strictVersion: true },
        'rxjs': { singleton: true, strictVersion: true }
      }
    })
  ]
};

// webpack.config.js - Host App (shell)
module.exports = {
  plugins: [
    new ModuleFederationPlugin({
      name: 'shell',
      remotes: {
        products: 'products@http://localhost:4201/remoteEntry.js',
        cart: 'cart@http://localhost:4202/remoteEntry.js'
      },
      shared: {
        '@angular/core': { singleton: true, strictVersion: true },
        '@angular/common': { singleton: true, strictVersion: true },
        '@angular/router': { singleton: true, strictVersion: true }
      }
    })
  ]
};
```

---

105. How to implement Micro Frontends in Angular?

Answer:

Using **@angular-architects/module-federation**:

```
# Add Module Federation to Angular project
ng add @angular-architects/module-federation --project shell --port 4200 --type host
ng add @angular-architects/module-federation --project mfe1 --port 4201 --type remote

// Shell app - app-routing.module.ts
const routes: Routes = [
  {
    path: '',
    component: HomeComponent
  },
  {
    path: 'products',
    loadChildren: () => loadRemoteModule({
      type: 'module',
      remoteEntry: 'http://localhost:4201/remoteEntry.js',
      exposedModule: './ProductsModule'
    }).then(m => m.ProductsModule)
  },
  {
    path: 'cart',
    loadChildren: () => loadRemoteModule({
      type: 'module',
      remoteEntry: 'http://localhost:4202/remoteEntry.js',
      exposedModule: './CartModule'
    }).then(m => m.CartModule)
  }
];

// Remote app - webpack.config.js (auto-generated)
module.exports = {
  output: {
    uniqueName: "mfeProducts",
    publicPath: "auto"
  },
  plugins: [
    new ModuleFederationPlugin({
      name: "mfeProducts",
      filename: "remoteEntry.js",
      exposes: {
        './ProductsModule': './src/app/products/products.module.ts',
      },
      shared: share({
```

```
        "@angular/core": { singleton: true, strictVersion: true },
        "@angular/common": { singleton: true, strictVersion: true },
        "@angular/router": { singleton: true, strictVersion: true },
        "rxjs": { singleton: true, strictVersion: true }
      })
    })
  ]
};
```

---

106. What are the challenges of Micro Frontends?

Answer:

Challenge	Solution
Shared state	Event bus, shared service
Consistent UI	Design system, shared components
Routing	Shell manages routes
Authentication	Shared auth service
Versioning	Semantic versioning, compatible ranges
Performance	Lazy loading, shared dependencies
Debugging	Source maps, logging

Example - Shared State:

```
// Custom Events for cross-MFE communication
// In MFE1
window.dispatchEvent(new CustomEvent('user:selected', {
  detail: { userId: 123 }
}));

// In MFE2
window.addEventListener('user:selected', (event: CustomEvent) => {
  const userId = event.detail.userId;
  this.loadUserDetails(userId);
});

// Or using shared service via window
// shared-state.service.ts
@Injectable()
export class SharedStateService {
  private state = new BehaviorSubject<any>({});

  constructor() {
    // Expose to window for cross-MFE access
    (window as any).sharedState = this;
  }

  setState(key: string, value: any): void {
    const current = this.state.value;
    this.state.next({ ...current, [key]: value });
  }

  getState(key: string): Observable<any> {
    return this.state.pipe(map(state => state[key]));
  }
}
```

---

107. How to handle routing in Micro Frontends?

Answer:

```
// Shell App Routes
const routes: Routes = [
  { path: '', component: ShellHomeComponent },
  {
    path: 'products',
    loadChildren: () => loadRemoteModule({
      remoteEntry: 'http://localhost:4201/remoteEntry.js',
      exposedModule: './ProductsModule'
    }).then(m => m.ProductsModule),
    canActivate: [AuthGuard]
  }
];

// Remote MFE Routes (products)
const routes: Routes = [
  { path: '', component: ProductListComponent },
  { path: ':id', component: ProductDetailComponent }
];

// Full URLs would be:
// /products -> ProductListComponent (from MFE)
// /products/123 -> ProductDetailComponent (from MFE)
```

---

108. How to share dependencies between Micro Frontends?

Answer:

```
// webpack.config.js
shared: share({
  "@angular/core": {
    singleton: true,        // Only one instance
    strictVersion: true,    // Must match version
    requiredVersion: '^17.0.0'
  },
  "@angular/common": { singleton: true, strictVersion: true },
  "@angular/router": { singleton: true, strictVersion: true },
  "rxjs": {
    singleton: true,
    strictVersion: false    // Allow compatible versions
  },
  // Shared UI library
  "@my-org/ui-components": {
    singleton: true,
    import: '@my-org/ui-components',
    requiredVersion: '^2.0.0'
  }
})
```

---

109. What is Web Components approach?

Answer:

- Create **Custom Elements** from Angular components
- Framework-agnostic integration
- Uses **Angular Elements** package

Example:

```
// Create Angular Element
import { createCustomElement } from '@angular/elements';

@Component({
  selector: 'app-widget',
  template: `<div>{{ title }}</div>`
})
export class WidgetComponent {
  @Input() title = '';
}

@NgModule({
  declarations: [WidgetComponent],
  imports: [BrowserModule],
  entryComponents: [WidgetComponent]
})
export class AppModule {
  constructor(private injector: Injector) {
    const widgetElement = createCustomElement(WidgetComponent, { injector });
    customElements.define('my-widget', widgetElement);
  }

  ngDoBootstrap() { }
}

// Usage in any HTML (React, Vue, plain HTML)
// <my-widget title="Hello World"></my-widget>
```

---

110. What is Single-SPA?

Answer:

- **Single-SPA** is a framework for bringing together multiple JS frameworks
- Manages **lifecycle** of micro frontends
- Handles **mounting/unmounting** applications

Example:

```
// root-config.js (orchestrator)
import { registerApplication, start } from 'single-spa';

registerApplication({
  name: '@my-org/angular-app',
  app: () => System.import('@my-org/angular-app'),
  activeWhen: ['/angular']
});

registerApplication({
  name: '@my-org/react-app',
  app: () => System.import('@my-org/react-app'),
  activeWhen: ['/react']
```

```
});

start();

// Angular MFE setup
import { enableProdMode } from '@angular/core';
import { platformBrowserDynamic } from '@angular/platform-browser-dynamic';
import { singleSpaAngular } from 'single-spa-angular';

const lifecycles = singleSpaAngular({
  bootstrapFunction: () => platformBrowserDynamic().bootstrapModule(AppModule),
  template: '<app-root />',
  Router,
  NgZone
});

export const bootstrap = lifecycles.bootstrap;
export const mount = lifecycles.mount;
export const unmount = lifecycles.unmount;
```

---

111. How to handle authentication in Micro Frontends?

Answer:

```
// Shared Auth Service (in shell or shared library)
@Injectable({ providedIn: 'root' })
export class AuthService {
  private tokenKey = 'auth_token';

  getToken(): string | null {
    return localStorage.getItem(this.tokenKey);
  }

  setToken(token: string): void {
    localStorage.setItem(this.tokenKey, token);
    // Notify all MFEs
    window.dispatchEvent(new CustomEvent('auth:tokenChanged', {
      detail: { token }
    }));
  }

  isAuthenticated(): boolean {
    return !!this.getToken();
  }

  logout(): void {
    localStorage.removeItem(this.tokenKey);
    window.dispatchEvent(new CustomEvent('auth:logout'));
  }
}

// Each MFE listens for auth events
window.addEventListener('auth:logout', () => {
  // Clear local state, redirect to login
});

// Shared HTTP Interceptor
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = localStorage.getItem('auth_token');

    if (token) {
      req = req.clone({
        setHeaders: { Authorization: `Bearer ${token}` }
      });
    }

    return next.handle(req);
  }
}
```

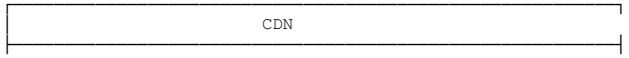
---

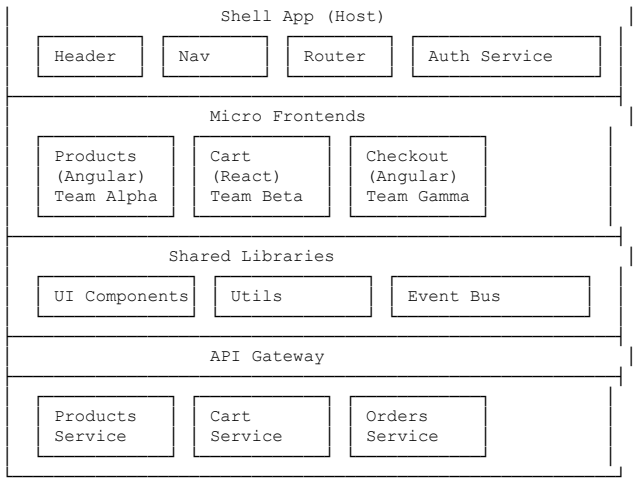
112. Micro Frontend Best Practices

Answer:

1. **Design System**: Shared UI components library
2. **API Gateway**: Single entry point for backends
3. **Event-driven communication**: Loose coupling
4. **Independent deployments**: CI/CD per MFE
5. **Monitoring**: Distributed tracing
6. **Versioning strategy**: Semantic versioning

Example Architecture:





ADVANCED TOPICS ANSWERS

PERFORMANCE & OPTIMIZATION

---

113. How to optimize Angular application performance?

Answer:

1. **OnPush Change Detection**
2. **Lazy Loading**
3. **trackBy in ngFor**
4. **Pure Pipes**
5. **Bundle optimization**
6. **Web Workers**

Example:

```
// 1. OnPush Change Detection
@Component({
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class OptimizedComponent { }

// 2. Lazy Loading
const routes: Routes = [
  {
    path: 'feature',
    loadChildren: () => import('./feature/feature.module')
      .then(m => m.FeatureModule)
  }
];

// 3. trackBy
<div *ngFor="let item of items; trackBy: trackById">
  {{ item.name }}
</div>

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

// 4. Pure Pipe instead of method in template
// BAD: <p>{{ getFullName() }}</p>
// GOOD: <p>{{ user | fullName }}</p>

// 5. Bundle Analysis
// ng build --stats-json
// npx webpack-bundle-analyzer dist/stats.json

// 6. Virtual Scrolling
import { ScrollingModule } from '@angular/cdk/scrolling';

@Component({
  template: `
    <cdk-virtual-scroll-viewport itemSize="50" class="viewport">
      <div *cdkVirtualFor="let item of items">{{ item.name }}</div>
    </cdk-virtual-scroll-viewport>
  `
})
export class VirtualScrollComponent { }
```

---

114. What is tree shaking?

Answer:

- **Tree shaking** removes **unused code** from bundle
- Works with **ES modules** (import/export)
- Requires **production build**
- Libraries must be tree-shakable

Example:

```
// Tree-shakable (ES modules)
import { map } from 'rxjs/operators';  // Only imports 'map'

// Not tree-shakable
import * as _ from 'lodash';  // Imports entire library

// Better: Import specific function
import { debounce } from 'lodash-es';

// Service tree-shaking
@Injectable({
  providedIn: 'root'  // Tree-shakable if not injected
})
export class MyService { }
```

---

115. How to measure Angular application performance?

Answer:

- **Lighthouse**: Overall performance audit
- **Chrome DevTools**: Performance tab, Memory
- **Angular DevTools**: Component profiler
- **Bundle Analyzer**: Bundle size analysis

Example:

```
// Enable Angular profiling
import { enableDebugTools } from '@angular/platform-browser';

platformBrowserDynamic().bootstrapModule(AppModule)
  .then(moduleRef => {
    const appRef = moduleRef.injector.get(ApplicationRef);
    const componentRef = appRef.components[0];
    enableDebugTools(componentRef);
  });

// In console: ng.profiler.timeChangeDetection()

// Custom performance logging
export class PerformanceService {
  mark(name: string): void {
    performance.mark(name);
  }

  measure(name: string, startMark: string, endMark: string): number {
    performance.measure(name, startMark, endMark);
    const measure = performance.getEntriesByName(name)[0];
    console.log(`${name}: ${measure.duration}ms`);
    return measure.duration;
  }
}
```

---

116. What is preloading strategy?

Answer:

- **Preloading** loads lazy modules **after initial load**
- Improves **perceived performance**
- Built-in strategies: NoPreloading, PreloadAllModules

Example:

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

```
// Custom preloading strategy
@Injectable({ providedIn: 'root' })
export class SelectivePreloadingStrategy implements PreloadingStrategy {
  preload(route: Route, load: () => Observable<any>): Observable<any> {
    // Only preload routes marked with data.preload
    if (route.data?.['preload']) {
      return load();
    }
    return of(null);
  }
}

// Route configuration
const routes: Routes = [
  {
    path: 'important',
    loadChildren: () => import('./important/important.module'),
    data: { preload: true }  // Will be preloaded
  },
  {
    path: 'rarely-used',
    loadChildren: () => import('./rarely/rarely.module')
    // Won't be preloaded
  }
];
```

---

117. How to use Web Workers in Angular?

Answer:

```
# Generate Web Worker
ng generate web-worker app

// app.worker.ts
addEventListener('message', ({ data }) => {
  const result = heavyComputation(data);
  postMessage(result);
});

function heavyComputation(data: number[]): number {
  return data.reduce((sum, n) => sum + n, 0);
}

// Component using Web Worker
@Component({...})
export class HeavyTaskComponent {
  result = 0;
  private worker: Worker;

  constructor() {
    if (typeof Worker !== 'undefined') {
      this.worker = new Worker(new URL('./app.worker', import.meta.url));
      this.worker.onmessage = ({ data }) => {
        this.result = data;
      };
    }
  }

  calculate(data: number[]): void {
    this.worker.postMessage(data);
  }

  ngOnDestroy(): void {
    this.worker?.terminate();
  }
}
```

ADVANCED RxJS

---

118. What is shareReplay?

Answer:

- **shareReplay** multicasts and caches emitted values
- New subscribers get **cached values**
- Useful for **HTTP caching**

Example:

```
@Injectable({ providedIn: 'root' })
export class DataService {
  private config$?: Observable<Config>;

  getConfig(): Observable<Config> {
    if (!this.config$) {
```

```
      this.config$ = this.http.get<Config>('/api/config').pipe(
        shareReplay(1)  // Cache 1 value
      );
    }
    return this.config$;
  }

  // Multiple components call this - only one HTTP request
}
```

---

119. What is the difference between share and shareReplay?

Answer:

Operator	Caching	Use Case
share	No caching	Live streams
shareReplay	Caches N values	HTTP, static data

Example:

```
// share: No caching
const stream$ = source$.pipe(share());
stream$.subscribe();  // Gets live values only
// Later...
stream$.subscribe();  // Misses earlier values

// shareReplay: Caches values
const cached$ = source$.pipe(shareReplay(1));
cached$.subscribe();  // Gets live values
// Later...
cached$.subscribe();  // Gets last cached value + new ones
```

---

120. What are custom RxJS operators?

Answer:

```
// Custom operator: Log values
function log<T>(tag: string): MonoTypeOperatorFunction<T> {
  return (source$) => source$.pipe(
    tap(value => console.log(`[${tag}]`, value))
  );
}

// Custom operator: Retry with exponential backoff
function retryWithBackoff<T>(
  maxRetries: number,
  delayMs: number
): MonoTypeOperatorFunction<T> {
  return (source$) => source$.pipe(
    retryWhen(errors$ =>
      errors$.pipe(
        scan((retryCount, error) => {
          if (retryCount >= maxRetries) throw error;
          return retryCount + 1;
        }, 0),
        delayWhen(retryCount =>
          timer(delayMs * Math.pow(2, retryCount))
        )
      )
    )
  );
}

// Usage
this.http.get('/api/data').pipe(
  log('API Response'),
  retryWithBackoff(3, 1000)
).subscribe();
```

---

121. What is the difference between combineLatest and forkJoin?

Answer:

Operator	Behavior	Use Case
combineLatest	Emits on any source emission	Live streams
forkJoin	Emits once when all complete	Parallel HTTP

Example:

```
// combineLatest: Emits every time any source emits
combineLatest([
  this.form.valueChanges,
```

```
  this.permissions$
]).subscribe(([formValue, permissions]) => {
  // Runs on every form change
});

// forkJoin: Waits for all to complete
forkJoin({
  users: this.http.get<User[]>('/api/users'),
  products: this.http.get<Product[]>('/api/products')
}).subscribe(({ users, products }) => {
  // Runs once when both complete
});
```

TESTING

---

122. How to unit test Angular components?

Answer:

```
describe('UserComponent', () => {
  let component: UserComponent;
  let fixture: ComponentFixture<UserComponent>;
  let userService: jasmine.SpyObj<UserService>;

  beforeEach(async () => {
    const spy = jasmine.createSpyObj('UserService', ['getUser']);

    await TestBed.configureTestingModule({
      declarations: [UserComponent],
      providers: [
        { provide: UserService, useValue: spy }
      ]
    }).compileComponents();

    userService = TestBed.inject(UserService) as jasmine.SpyObj<UserService>;
    fixture = TestBed.createComponent(UserComponent);
    component = fixture.componentInstance;
  });

  it('should create', () => {
    expect(component).toBeTruthy();
  });

  it('should display user name', () => {
    const user = { id: 1, name: 'John' };
    userService.getUser.and.returnValue(of(user));

    fixture.detectChanges();

    const element = fixture.nativeElement.querySelector('.user-name');
    expect(element.textContent).toContain('John');
  });

  it('should call service on init', () => {
    userService.getUser.and.returnValue(of({ id: 1, name: 'John' }));

    fixture.detectChanges();

    expect(userService.getUser).toHaveBeenCalled();
  });
});
```

---

123. How to test services?

Answer:

```
describe('UserService', () => {
  let service: UserService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [UserService]
    });

    service = TestBed.inject(UserService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  afterEach(() => {
    httpMock.verify();  // Verify no outstanding requests
  });

  it('should fetch users', () => {
    const mockUsers = [{ id: 1, name: 'John' }];
```

```
    service.getUsers().subscribe(users => {
      expect(users).toEqual(mockUsers);
    });

    const req = httpMock.expectOne('/api/users');
    expect(req.request.method).toBe('GET');
    req.flush(mockUsers);
  });

  it('should handle error', () => {
    service.getUsers().subscribe({
      error: (error) => {
        expect(error.status).toBe(500);
      }
    });

    const req = httpMock.expectOne('/api/users');
    req.flush('Error', { status: 500, statusText: 'Server Error' });
  });
});
```

---

124. How to test async code?

Answer:

```
describe('AsyncComponent', () => {
  // Using fakeAsync and tick
  it('should work with setTimeout', fakeAsync(() => {
    let value = false;

    setTimeout(() => { value = true; }, 1000);

    tick(1000);  // Advance time

    expect(value).toBe(true);
  }));

  // Using async/await with fixture
  it('should work with observables', async () => {
    const fixture = TestBed.createComponent(AsyncComponent);
    fixture.detectChanges();

    await fixture.whenStable();

    expect(fixture.nativeElement.textContent).toContain('loaded');
  });

  // Testing observables with marbles
  it('should debounce search', () => {
    const scheduler = new TestScheduler((actual, expected) => {
      expect(actual).toEqual(expected);
    });

    scheduler.run(({ cold, expectObservable }) => {
      const input = cold('a-b-c|');
      const expected = '---c|';

      const result = input.pipe(debounceTime(2, scheduler));

      expectObservable(result).toBe(expected);
    });
  });
});
```

---

125. What is TestBed?

Answer:

- **TestBed** configures and creates testing module
- Sets up **dependency injection** for tests
- Creates **component fixtures** for testing

Example:

```
beforeEach(async () => {
  await TestBed.configureTestingModule({
    imports: [
      HttpClientTestingModule,
      RouterTestingModule,
      SharedModule
    ],
    declarations: [
      MyComponent,
      MockChildComponent
    ],
    providers: [
      { provide: DataService, useClass: MockDataService },
      { provide: API_URL, useValue: 'http://test.com' }
    ],
```

```
    schemas: [NO_ERRORS_SCHEMA]  // Ignore unknown elements
  }).compileComponents();
});
```

SECURITY

---

126. What are common security vulnerabilities in Angular?

Answer:

1. **XSS** (Cross-Site Scripting)
2. **CSRF** (Cross-Site Request Forgery)
3. **Injection attacks**
4. **Insecure dependencies**

---

127. How does Angular prevent XSS?

Answer:

- Angular **sanitizes** values by default
- Automatically escapes dangerous HTML
- Use `DomSanitizer` for trusted content

Example:

```
// Automatic sanitization
@Component({
  template: `
    <!-- Safe: Angular escapes this -->
    <div>{{ userInput }}</div>

    <!-- Angular sanitizes innerHTML -->
    <div [innerHTML]="htmlContent"></div>
  `
})
export class SafeComponent {
  userInput = '<script>alert("XSS")</script>';  // Escaped
  htmlContent = '<b>Bold</b><script>alert("XSS")</script>';  // Script removed
}

// Bypass sanitization (use carefully!)
@Component({...})
export class TrustedComponent {
  constructor(private sanitizer: DomSanitizer) { }

  getTrustedHtml(html: string): SafeHtml {
    return this.sanitizer.bypassSecurityTrustHtml(html);
  }

  getTrustedUrl(url: string): SafeUrl {
    return this.sanitizer.bypassSecurityTrustUrl(url);
  }
}
```

---

128. How to implement CSRF protection?

Answer:

```
// HttpClient automatically reads XSRF token
// Backend sets cookie: XSRF-TOKEN
// Angular sends header: X-XSRF-TOKEN

// Custom configuration if needed
@NgModule({
  imports: [
    HttpClientModule,
    HttpClientXsrfModule.withOptions({
      cookieName: 'My-Xsrf-Cookie',
      headerName: 'My-Xsrf-Header'
    })
  ]
})
export class AppModule { }
```

---

129. Security Best Practices

Answer:

```
// 1. Use HttpOnly cookies for tokens
// Set by backend, not accessible via JavaScript

// 2. Validate and sanitize inputs
// 3. Use Content Security Policy headers
// 4. Keep dependencies updated
// npm audit fix

// 5. Avoid eval() and innerHTML when possible

// 6. Implement proper authentication
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    // Add token to requests
    // Handle 401 responses
  }
}

// 7. Use HTTPS only
// 8. Implement rate limiting (backend)
```

CLI & TOOLING

---

130. Useful Angular CLI commands

Answer:

```
# Generate
ng generate component user
ng generate service data
ng generate module feature --routing
ng generate guard auth
ng generate pipe transform
ng generate directive highlight

# Build
ng build --configuration=production
ng build --stats-json  # For bundle analysis

# Serve
ng serve --port 4300 --open
ng serve --proxy-config proxy.conf.json

# Test
ng test --code-coverage
ng e2e

# Update
ng update @angular/cli @angular/core

# Analyze
ng analytics
```

131-132. CLI Configuration & Schematics

Answer:

```
// angular.json customization
{
  "schematics": {
    "@schematics/angular:component": {
      "style": "scss",
      "changeDetection": "OnPush",
      "standalone": true
    }
  }
}
```

---

i18n (INTERNATIONALIZATION)

133-134. Angular Internationalization

Answer:

```
// 1. Mark text for translation
<h1 i18n="@@welcomeTitle">Welcome</h1>

// 2. Extract messages
// ng extract-i18n --output-path locale
```

```
// 3. Translate messages.xlf to messages.fr.xlf

// 4. Build with locale
// ng build --localize

// angular.json configuration
{
  "i18n": {
    "sourceLocale": "en-US",
    "locales": {
      "fr": "locale/messages.fr.xlf"
    }
  }
}
```

ACCESSIBILITY

135-137. Angular Accessibility

Answer:

```
@Component({
  template: `
    <!-- Proper ARIA labels -->
    <button aria-label="Close dialog" (click)="close()">×</button>

    <!-- Focus management -->
    <input #searchInput aria-describedby="search-hint">
    <span id="search-hint">Enter search term</span>

    <!-- Live regions -->
    <div aria-live="polite">{{ statusMessage }}</div>

    <!-- Keyboard navigation -->
    <div (keydown.enter)="select()" (keydown.space)="select()" tabindex="0">
      Selectable item
    </div>
  `
})
export class AccessibleComponent implements AfterViewInit {
  @ViewChild('searchInput') searchInput!: ElementRef;

  ngAfterViewInit(): void {
    this.searchInput.nativeElement.focus();
  }
}

// CDK A11y module
import { A11yModule, LiveAnnouncer } from '@angular/cdk/a11y';

constructor(private liveAnnouncer: LiveAnnouncer) { }

announce(message: string): void {
  this.liveAnnouncer.announce(message, 'polite');
}
```

SSR / ANGULAR UNIVERSAL

138-140. Server-Side Rendering

Answer:

```
# Add SSR to project
ng add @angular/ssr

// Server-side specific code
import { isPlatformBrowser, isPlatformServer } from '@angular/common';
import { PLATFORM_ID, inject } from '@angular/core';

@Component({...})
export class UniversalComponent {
  private platformId = inject(PLATFORM_ID);

  ngOnInit(): void {
    if (isPlatformBrowser(this.platformId)) {
      // Browser-only code (localStorage, window, etc.)
      localStorage.setItem('key', 'value');
    }

    if (isPlatformServer(this.platformId)) {
      // Server-only code
    }
  }
```

```
}

// Benefits of SSR:
// 1. Better SEO
// 2. Faster First Contentful Paint
// 3. Works without JavaScript enabled
```

STATE MANAGEMENT PATTERNS

141-144. State Management Alternatives

Answer:

```
// 1. BehaviorSubject Store Pattern
@Injectable({ providedIn: 'root' })
export class Store<T> {
  private state$: BehaviorSubject<T>;

  constructor(initialState: T) {
    this.state$ = new BehaviorSubject<T>(initialState);
  }

  get state(): T {
    return this.state$.value;
  }

  select<K>(selector: (state: T) => K): Observable<K> {
    return this.state$.pipe(
      map(selector),
      distinctUntilChanged()
    );
  }

  setState(newState: Partial<T>): void {
    this.state$.next({ ...this.state, ...newState });
  }
}

// 2. Angular Signals (Angular 16+)
@Injectable({ providedIn: 'root' })
export class UserStore {
  // State
  users = signal<User[]>([]);
  loading = signal(false);
  error = signal<string | null>(null);

  // Computed
  activeUsers = computed(() =>
    this.users().filter(u => u.isActive)
  );

  // Actions
  loadUsers(): void {
    this.loading.set(true);
    // ...
  }
}

// Component with signals
@Component({
  template: `
    <div *ngIf="store.loading()">Loading...</div>
    <div *ngFor="let user of store.users()">{{ user.name }}</div>
  `
})
export class UserListComponent {
  store = inject(UserStore);
}
```

SCENARIO-BASED QUESTIONS ANSWERS

Scenario 1: Component Communication in Complex Hierarchy

Question:

You have a deeply nested component structure (Parent → Child → GrandChild → GreatGrandChild). How would you communicate data from GreatGrandChild to Parent without prop drilling?

Answer:

```
// Solution 1: Shared Service with BehaviorSubject
@Injectable({ providedIn: 'root' })
```

```
export class CommunicationService {
  private dataSubject = new BehaviorSubject<any>(null);
  data$ = this.dataSubject.asObservable();

  sendData(data: any): void {
    this.dataSubject.next(data);
  }
}

// GreatGrandChild (sender)
@Component({
  selector: 'app-great-grandchild'
})
export class GreatGrandChildComponent {
  constructor(private commService: CommunicationService) { }

  sendToParent(data: any): void {
    this.commService.sendData(data);
  }
}

// Parent (receiver)
@Component({
  selector: 'app-parent',
  template: `<div>{{ receivedData | json }}</div>`
})
export class ParentComponent implements OnInit, OnDestroy {
  receivedData: any;
  private destroy$ = new Subject<void>();

  constructor(private commService: CommunicationService) { }

  ngOnInit(): void {
    this.commService.data$
      .pipe(takeUntil(this.destroy$))
      .subscribe(data => this.receivedData = data);
  }

  ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }
}

// Solution 2: Using Event Emitters chain (not recommended for deep nesting)
// Solution 3: NgRx for complex state management
```

Scenario 2: Large Data Table Performance

Question:

Your application has a table with 10,000 rows. The page is very slow. How would you optimize it?

Answer:

```
// Solution: Virtual Scrolling + OnPush + trackBy
import { ScrollingModule } from '@angular/cdk/scrolling';

@Component({
  selector: 'app-large-table',
  changeDetection: ChangeDetectionStrategy.OnPush,
  template: `
    <cdk-virtual-scroll-viewport itemSize="48" class="viewport">
      <table>
        <thead>
          <tr>
            <th *ngFor="let col of columns">{{ col.label }}</th>
          </tr>
        </thead>
        <tbody>
          <tr *cdkVirtualFor="let row of data; trackBy: trackById">
            <td *ngFor="let col of columns">{{ row[col.field] }}</td>
          </tr>
        </tbody>
      </table>
    </cdk-virtual-scroll-viewport>
  `,
  styles: [`
    .viewport {
      height: 500px;
      width: 100%;
    }
  `]
})
export class LargeTableComponent {
  @Input() data: any[] = [];
  @Input() columns: Column[] = [];

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

```
// Additional optimizations:
// 1. Pagination instead of loading all data
// 2. Server-side filtering/sorting
// 3. Memoize expensive computations with pure pipes
// 4. Use OnPush change detection
// 5. Avoid complex template expressions
```

Scenario 3: Handle Multiple HTTP Requests

Question:

You need to fetch user details from one API and their orders from another API. Both should complete before displaying the page. How would you handle this?

Answer:

```
// Solution 1: forkJoin for parallel requests
@Component({
  selector: 'app-user-dashboard'
})
export class UserDashboardComponent implements OnInit {
  user: User | null = null;
  orders: Order[] = [];
  loading = true;
  error: string | null = null;

  constructor(
    private userService: UserService,
    private orderService: OrderService,
    private route: ActivatedRoute
  ) { }

  ngOnInit(): void {
    const userId = this.route.snapshot.params['id'];

    forkJoin({
      user: this.userService.getUser(userId),
      orders: this.orderService.getOrders(userId)
    }).pipe(
      finalize(() => this.loading = false)
    ).subscribe({
      next: ({ user, orders }) => {
        this.user = user;
        this.orders = orders;
      },
      error: (err) => {
        this.error = 'Failed to load data';
        console.error(err);
      }
    });
  }
}

// Solution 2: Using Resolver
export const userDashboardResolver: ResolveFn<{ user: User; orders: Order[] }> =
  (route) => {
    const userService = inject(UserService);
    const orderService = inject(OrderService);
    const userId = route.params['id'];

    return forkJoin({
      user: userService.getUser(userId),
      orders: orderService.getOrders(userId)
    });
  };

// Route config
const routes: Routes = [
  {
    path: 'dashboard/:id',
    component: UserDashboardComponent,
    resolve: {
      data: userDashboardResolver
    }
  }
];

// Component using resolver data
export class UserDashboardComponent implements OnInit {
  data$ = this.route.data.pipe(
    map(data => data['data'])
  );

  constructor(private route: ActivatedRoute) { }
}
```

Scenario 4: Search with Debounce

Question:

Implement a search input that calls the API only after the user stops typing for 300ms and doesn't call for the same query twice.

Answer:

```
@Component({
  selector: 'app-search',
  template: `
    <input
      type="text"
      [formControl]="searchControl"
      placeholder="Search..."
    >

    <div *ngIf="loading">Searching...</div>

    <ul>
      <li *ngFor="let result of results$ | async">
        {{ result.name }}
      </li>
    </ul>
  `
})
export class SearchComponent implements OnInit {
  searchControl = new FormControl('');
  results$!: Observable<SearchResult[]>;
  loading = false;

  constructor(private searchService: SearchService) { }

  ngOnInit(): void {
    this.results$ = this.searchControl.valueChanges.pipe(
      debounceTime(300),              // Wait 300ms after typing stops
      distinctUntilChanged(),          // Don't search if same query
      filter(term => term.length >= 2), // Minimum 2 characters
      tap(() => this.loading = true),
      switchMap(term =>
        this.searchService.search(term).pipe(
          catchError(() => of([]))     // Return empty on error
        )
      ),
      tap(() => this.loading = false)
    );
  }
}

// Service
@Injectable({ providedIn: 'root' })
export class SearchService {
  private cache = new Map<string, SearchResult[]>();

  constructor(private http: HttpClient) { }

  search(term: string): Observable<SearchResult[]> {
    // Check cache first
    if (this.cache.has(term)) {
      return of(this.cache.get(term)!);
    }

    return this.http.get<SearchResult[]>(`/api/search?q=${term}`).pipe(
      tap(results => this.cache.set(term, results))
    );
  }
}
```

Scenario 5: Prevent Unsaved Changes

Question:

How would you warn users when they try to leave a page with unsaved form changes?

Answer:

```
// Interface for components with unsaved changes
export interface CanComponentDeactivate {
  canDeactivate(): boolean | Observable<boolean>;
}

// Guard
export const unsavedChangesGuard: CanDeactivateFn<CanComponentDeactivate> =
  (component) => {
    if (component.canDeactivate()) {
      return true;
    }
    return confirm('You have unsaved changes. Do you really want to leave?');
  };

// Route configuration
const routes: Routes = [
  {
    path: 'edit/:id',
    component: EditComponent,
```

```
    canDeactivate: [unsavedChangesGuard]
  }
];

// Component implementation
@Component({
  selector: 'app-edit'
})
export class EditComponent implements CanComponentDeactivate {
  form: FormGroup;
  private initialValue: any;

  constructor(private fb: FormBuilder) {
    this.form = this.fb.group({
      name: ['', Validators.required],
      email: ['', [Validators.required, Validators.email]]
    });

    // Store initial value
    this.initialValue = this.form.value;
  }

  canDeactivate(): boolean {
    // Compare current value with initial
    return JSON.stringify(this.form.value) === JSON.stringify(this.initialValue);
  }

  onSave(): void {
    if (this.form.valid) {
      // Save logic...
      this.initialValue = this.form.value;  // Update initial value
    }
  }
}

// Also handle browser back/refresh
@HostListener('window:beforeunload', ['$event'])
onBeforeUnload(event: BeforeUnloadEvent): void {
  if (!this.canDeactivate()) {
    event.returnValue = 'You have unsaved changes';
  }
}
```

Scenario 6: Authentication Flow

Question:

Implement a complete authentication flow with protected routes, auto-logout on token expiry, and token refresh.

Answer:

```
// Auth Service
@Injectable({ providedIn: 'root' })
export class AuthService {
  private currentUserSubject = new BehaviorSubject<User | null>(null);
  currentUser$ = this.currentUserSubject.asObservable();

  private tokenExpirationTimer: any;

  constructor(
    private http: HttpClient,
    private router: Router
  ) {
    this.autoLogin();
  }

  login(credentials: { email: string; password: string }): Observable<User> {
    return this.http.post<AuthResponse>('/api/login', credentials).pipe(
      map(response => {
        this.handleAuthentication(response);
        return response.user;
      })
    );
  }

  private handleAuthentication(response: AuthResponse): void {
    const expirationDate = new Date(new Date().getTime() + response.expiresIn * 1000);

    localStorage.setItem('authData', JSON.stringify({
      user: response.user,
      token: response.token,
      expirationDate: expirationDate.toISOString()
    }));

    this.currentUserSubject.next(response.user);
    this.setAutoLogout(response.expiresIn * 1000);
  }

  autoLogin(): void {
    const authDataString = localStorage.getItem('authData');
    if (!authDataString) return;

    const authData = JSON.parse(authDataString);
```

```
    const expirationDate = new Date(authData.expirationDate);

    if (expirationDate <= new Date()) {
      this.logout();
      return;
    }

    this.currentUserSubject.next(authData.user);
    const expiresIn = expirationDate.getTime() - new Date().getTime();
    this.setAutoLogout(expiresIn);
  }

  private setAutoLogout(duration: number): void {
    this.tokenExpirationTimer = setTimeout(() => {
      this.logout();
    }, duration);
  }

  logout(): void {
    this.currentUserSubject.next(null);
    localStorage.removeItem('authData');

    if (this.tokenExpirationTimer) {
      clearTimeout(this.tokenExpirationTimer);
    }

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

  getToken(): string | null {
    const authData = localStorage.getItem('authData');
    return authData ? JSON.parse(authData).token : null;
  }
}

// Auth Guard
export const authGuard: CanActivateFn = () => {
  const authService = inject(AuthService);
  const router = inject(Router);

  return authService.currentUser$.pipe(
    take(1),
    map(user => {
      if (user) return true;
      return router.createUrlTree(['/login']);
    })
  );
};

// Auth Interceptor
@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) { }

  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    const token = this.authService.getToken();

    if (token) {
      req = req.clone({
        setHeaders: { Authorization: `Bearer ${token}` }
      });
    }

    return next.handle(req).pipe(
      catchError((error: HttpErrorResponse) => {
        if (error.status === 401) {
          this.authService.logout();
        }
        return throwError(() => error);
      })
    );
  }
}
```

Scenario 7: Dynamic Form Generation

Question:

Build a form that generates fields dynamically based on JSON configuration from an API.

Answer:

```
// Field configuration interface
interface FormFieldConfig {
  type: 'text' | 'number' | 'email' | 'select' | 'checkbox' | 'textarea';
  name: string;
  label: string;
  required?: boolean;
  options?: { value: any; label: string }[];
  validators?: {
    min?: number;
    max?: number;
    minLength?: number;
```

```
    maxLength?: number;
    pattern?: string;
  };
}

@Component({
  selector: 'app-dynamic-form',
  template: `
    <form [formGroup]="form" (ngSubmit)="onSubmit()">
      <div *ngFor="let field of fields" class="form-field">
        <label [for]="field.name">{{ field.label }}</label>

        <ng-container [ngSwitch]="field.type">
          <input
            *ngSwitchCase="'text'"
            type="text"
            [id]="field.name"
            [formControlName]="field.name"
          >

          <input
            *ngSwitchCase="'number'"
            type="number"
            [id]="field.name"
            [formControlName]="field.name"
          >

          <input
            *ngSwitchCase="'email'"
            type="email"
            [id]="field.name"
            [formControlName]="field.name"
          >

          <select
            *ngSwitchCase="'select'"
            [id]="field.name"
            [formControlName]="field.name"
          >
            <option *ngFor="let opt of field.options" [value]="opt.value">
              {{ opt.label }}
            </option>
          </select>

          <textarea
            *ngSwitchCase="'textarea'"
            [id]="field.name"
            [formControlName]="field.name"
          ></textarea>

          <input
            *ngSwitchCase="'checkbox'"
            type="checkbox"
            [id]="field.name"
            [formControlName]="field.name"
          >
        </ng-container>

        <div *ngIf="form.get(field.name)?.invalid && form.get(field.name)?.touched"
             class="error">
          {{ getErrorMessage(field.name) }}
        </div>
      </div>

      <button type="submit" [disabled]="form.invalid">Submit</button>
    </form>
  `
})
export class DynamicFormComponent implements OnInit {
  @Input() fields: FormFieldConfig[] = [];
  @Output() formSubmit = new EventEmitter<any>();

  form!: FormGroup;

  ngOnInit(): void {
    this.buildForm();
  }

  private buildForm(): void {
    const group: { [key: string]: FormControl } = {};

    this.fields.forEach(field => {
      const validators = this.getValidators(field);
      group[field.name] = new FormControl('', validators);
    });

    this.form = new FormGroup(group);
  }

  private getValidators(field: FormFieldConfig): ValidatorFn[] {
    const validators: ValidatorFn[] = [];

    if (field.required) validators.push(Validators.required);
    if (field.type === 'email') validators.push(Validators.email);

    if (field.validators) {
      if (field.validators.min !== undefined)
```

```
        validators.push(Validators.min(field.validators.min));
      if (field.validators.max !== undefined)
        validators.push(Validators.max(field.validators.max));
      if (field.validators.minLength !== undefined)
        validators.push(Validators.minLength(field.validators.minLength));
      if (field.validators.maxLength !== undefined)
        validators.push(Validators.maxLength(field.validators.maxLength));
      if (field.validators.pattern)
        validators.push(Validators.pattern(field.validators.pattern));
    }

    return validators;
  }

  getErrorMessage(fieldName: string): string {
    const control = this.form.get(fieldName);
    if (!control?.errors) return '';

    if (control.errors['required']) return 'This field is required';
    if (control.errors['email']) return 'Invalid email format';
    if (control.errors['min']) return `Minimum value is ${control.errors['min'].min}`;
    if (control.errors['max']) return `Maximum value is ${control.errors['max'].max}`;

    return 'Invalid value';
  }

  onSubmit(): void {
    if (this.form.valid) {
      this.formSubmit.emit(this.form.value);
    }
  }
}
```

Scenario 8: Real-time Updates with WebSocket

Question:

Implement real-time notifications using WebSocket that displays toast messages.

Answer:

```
// WebSocket Service
@Injectable({ providedIn: 'root' })
export class WebSocketService {
  private socket$?: WebSocketSubject<any>;
  private messages$ = new Subject<any>();

  connect(url: string): Observable<any> {
    if (!this.socket$) {
      this.socket$ = webSocket({
        url,
        openObserver: {
          next: () => console.log('WebSocket connected')
        },
        closeObserver: {
          next: () => {
            console.log('WebSocket disconnected');
            this.socket$ = undefined;
          }
        }
      });

      this.socket$.pipe(
        retryWhen(errors =>
          errors.pipe(
            tap(err => console.log('WebSocket error:', err)),
            delay(5000)  // Retry after 5 seconds
          )
        )
      ).subscribe(message => {
        this.messages$.next(message);
      });
    }

    return this.messages$.asObservable();
  }

  send(message: any): void {
    this.socket$?.next(message);
  }

  disconnect(): void {
    this.socket$?.complete();
    this.socket$ = undefined;
  }
}

// Notification Service
@Injectable({ providedIn: 'root' })
export class NotificationService {
  private notifications$ = new Subject<Notification>();

  constructor(private wsService: WebSocketService) {
```

```
    this.wsService.connect('wss://api.example.com/notifications')
      .subscribe(message => {
        this.notifications$.next({
          id: Date.now(),
          type: message.type,
          message: message.text,
          read: false
        });
      });
  }

  getNotifications(): Observable<Notification> {
    return this.notifications$.asObservable();
  }
}

// Toast Component
@Component({
  selector: 'app-toast-container',
  template: `
    <div class="toast-container">
      <div
        *ngFor="let toast of toasts; trackBy: trackById"
        class="toast"
        [class]="toast.type"
        [@fadeInOut]
      >
        {{ toast.message }}
        <button (click)="remove(toast.id)">×</button>
      </div>
    </div>
  `,
  animations: [
    trigger('fadeInOut', [
      transition(':enter', [
        style({ opacity: 0, transform: 'translateY(-20px)' }),
        animate('300ms ease-out', style({ opacity: 1, transform: 'translateY(0)' }))
      ]),
      transition(':leave', [
        animate('300ms ease-in', style({ opacity: 0, transform: 'translateX(100%)' }))
      ])
    ])
  ]
})
export class ToastContainerComponent implements OnInit, OnDestroy {
  toasts: Toast[] = [];
  private destroy$ = new Subject<void>();

  constructor(private notificationService: NotificationService) { }

  ngOnInit(): void {
    this.notificationService.getNotifications()
      .pipe(takeUntil(this.destroy$))
      .subscribe(notification => {
        this.addToast(notification);
      });
  }

  addToast(notification: Notification): void {
    const toast: Toast = {
      id: notification.id,
      message: notification.message,
      type: notification.type
    };

    this.toasts.push(toast);

    // Auto-remove after 5 seconds
    setTimeout(() => this.remove(toast.id), 5000);
  }

  remove(id: number): void {
    this.toasts = this.toasts.filter(t => t.id !== id);
  }

  trackById(index: number, toast: Toast): number {
    return toast.id;
  }

  ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }
}
```

Scenario 9: Error Handling Strategy

Question:

Implement a global error handling strategy that catches HTTP errors, shows user-friendly messages, and logs errors for debugging.

Answer:

```
// Global Error Handler
@Injectable()
export class GlobalErrorHandler implements ErrorHandler {
  constructor(
    private injector: Injector,
    private zone: NgZone
  ) { }

  handleError(error: any): void {
    const notificationService = this.injector.get(NotificationService);
    const loggingService = this.injector.get(LoggingService);

    let message = 'An unexpected error occurred';
    let stack = '';

    if (error instanceof HttpErrorResponse) {
      message = this.getHttpErrorMessage(error);
    } else if (error instanceof Error) {
      message = error.message;
      stack = error.stack || '';
    }

    // Log error
    loggingService.logError({
      message,
      stack,
      timestamp: new Date(),
      url: window.location.href
    });

    // Show notification in Angular zone
    this.zone.run(() => {
      notificationService.showError(message);
    });

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

  private getHttpErrorMessage(error: HttpErrorResponse): string {
    switch (error.status) {
      case 400: return 'Bad request. Please check your input.';
      case 401: return 'Please log in to continue.';
      case 403: return 'You don\'t have permission to access this resource.';
      case 404: return 'Resource not found.';
      case 500: return 'Server error. Please try again later.';
      default: return 'An error occurred. Please try again.';
    }
  }
}

// Provide in AppModule
@NgModule({
  providers: [
    { provide: ErrorHandler, useClass: GlobalErrorHandler }
  ]
})
export class AppModule { }

// HTTP Error Interceptor
@Injectable()
export class HttpErrorInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler): Observable<HttpEvent<any>> {
    return next.handle(req).pipe(
      retry({
        count: 2,
        delay: (error, retryCount) => {
          if (error.status === 0 || error.status >= 500) {
            return timer(1000 * retryCount);  // Retry server errors
          }
          throw error;  // Don't retry client errors
        }
      }),
      catchError((error: HttpErrorResponse) => {
        // Transform error for better handling
        return throwError(() => ({
          status: error.status,
          message: error.message,
          originalError: error
        }));
      })
    );
  }
}
```

Scenario 10: Implementing Infinite Scroll

Question:

Implement infinite scroll for a list that loads more data when the user scrolls near the bottom.

Answer:

```
@Component({
  selector: 'app-infinite-list',
  template: `
    <div class="list-container" #scrollContainer>
      <div *ngFor="let item of items; trackBy: trackById" class="item">
        {{ item.name }}
      </div>

      <div *ngIf="loading" class="loading">
        Loading more...
      </div>

      <div *ngIf="!hasMore && items.length > 0" class="end">
        No more items
      </div>
    </div>
  `
})
export class InfiniteListComponent implements OnInit, AfterViewInit, OnDestroy {
  @ViewChild('scrollContainer') scrollContainer!: ElementRef;

  items: Item[] = [];
  loading = false;
  hasMore = true;
  private page = 1;
  private pageSize = 20;
  private destroy$ = new Subject<void>();

  constructor(private itemService: ItemService) { }

  ngOnInit(): void {
    this.loadItems();
  }

  ngAfterViewInit(): void {
    // Create scroll observable
    fromEvent(this.scrollContainer.nativeElement, 'scroll')
      .pipe(
        takeUntil(this.destroy$),
        debounceTime(200),
        filter(() => !this.loading && this.hasMore),
        filter(() => this.isNearBottom())
      )
      .subscribe(() => {
        this.loadItems();
      });
  }

  private isNearBottom(): boolean {
    const container = this.scrollContainer.nativeElement;
    const threshold = 100;  // pixels from bottom
    return container.scrollHeight - container.scrollTop - container.clientHeight < threshold;
  }

  private loadItems(): void {
    this.loading = true;

    this.itemService.getItems(this.page, this.pageSize)
      .pipe(
        finalize(() => this.loading = false)
      )
      .subscribe({
        next: (response) => {
          this.items = [...this.items, ...response.items];
          this.hasMore = response.hasMore;
          this.page++;
        },
        error: (err) => {
          console.error('Failed to load items:', err);
        }
      });
  }

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

  ngOnDestroy(): void {
    this.destroy$.next();
    this.destroy$.complete();
  }
}

// Alternative using Intersection Observer (better performance)
@Directive({
  selector: '[appInfiniteScroll]'
})
export class InfiniteScrollDirective implements OnInit, OnDestroy {
  @Output() scrolled = new EventEmitter<void>();

  private observer!: IntersectionObserver;

  constructor(private el: ElementRef) { }

  ngOnInit(): void {
    this.observer = new IntersectionObserver(
      (entries) => {
```

```
        if (entries[0].isIntersecting) {
          this.scrolled.emit();
        }
      },
      { threshold: 0.1 }
    );

    this.observer.observe(this.el.nativeElement);
  }

  ngOnDestroy(): void {
    this.observer.disconnect();
  }
}

// Usage
// <div class="sentinel" appInfiniteScroll (scrolled)="loadMore()"></div>
```

Angular Questions – Complete Prep

Beginner Level (0–2 Years)

Angular Basics

1. What is Angular?
2. Difference between AngularJS and Angular
3. What are components?
4. What is a module?
5. What is a service?
6. What is a directive?
7. What is data binding?
8. Types of data binding in Angular
9. What is interpolation?
10. What are templates?
11. What is the purpose of `@NgModule`?

Components and Templates

12. What is a component in Angular?
13. Difference between component and directive
14. How to pass data from parent to child component?
15. How to pass data from child to parent component?
16. What are input and output decorators?
17. What is `@ViewChild` and `@ContentChild`?

Project Configuration Files

angular.json

18. What is `angular.json`?
19. What is its role in an Angular project?
20. How does `angular.json` manage build and serve configurations?
21. How do you configure environments in `angular.json`?
22. How do you add assets and styles globally using `angular.json`?

package.json

23. What is `package.json`?
24. Why is `package.json` important?
25. How does `package.json` manage dependencies?
26. What are `devDependencies` and `dependencies`?
27. What are scripts in `package.json` and how are they used?
28. How to run custom npm scripts defined in `package.json`?

tsconfig.json

29. What is `tsconfig.json`?
30. What compiler options are most important?
31. What does strict mode do?
32. Difference between `target` and `module`?
33. How to enable `strictNullChecks` and `noImplicitAny`?

Webpack

34. What is Webpack?
35. How does Angular use Webpack internally?
36. What is the role of loaders and plugins in Webpack?
37. Difference between development and production build using Webpack?
38. How can you customize Webpack in Angular?

Intermediate Level (2–4 Years)

Directives and Pipes

39. What are structural directives?
40. What are attribute directives?
41. Common built-in directives (`ngIf`, `ngFor`, `ngSwitch`)
42. What is a pipe?
43. Difference between built-in and custom pipe
44. How to create a custom pipe

Services and Dependency Injection

45. What is a service in Angular?
46. How dependency injection works in Angular?
47. What is `providedIn: 'root'`?
48. Difference between service provided in root and module
49. Singleton services vs multiple instances

Routing

50. What is Angular Router?
51. Difference between `routerLink` and `href`
52. How to pass parameters in routes?
53. What are route guards?
54. What is lazy loading?

Forms

55. Difference between template-driven and reactive forms
56. How to create a reactive form?
57. Form validation (built-in and custom)
58. `FormControl`, `FormGroup`, `FormBuilder`
59. How to dynamically add form controls?

State Management

60. What is state management in Angular?
61. Difference between local component state and global application state
62. What is RxJS and how is it used in Angular?
63. What is NgRx?
64. Explain the concept of store, actions, reducers, and selectors
65. Difference between `BehaviorSubject`, `Subject`, and `ReplaySubject`
66. How to manage async state using NgRx Effects
67. How to connect components to store using `select` and `dispatch`
68. Difference between NgRx, Akita, and other state management libraries
69. How to debug state using NgRx DevTools

Advanced Level (4+ Years)

Lifecycle Hooks

70. What are lifecycle hooks?
71. Difference between `ngOnInit` and constructor
72. `ngOnChanges`, `ngDoCheck`, `ngAfterViewInit`, `ngAfterContentInit`
73. When to use each lifecycle hook

Change Detection

74. How change detection works in Angular?
75. What is `zone.js`?
76. Difference between default and `OnPush` change detection strategy
77. How to optimize change detection

RxJS and Observables

78. What is RxJS?
79. What is an Observable?
80. Difference between Promise and Observable
81. Common RxJS operators (`map`, `filter`, `switchMap`, `mergeMap`, `concatMap`, `exhaustMap`)
82. How to unsubscribe from Observables?
83. What is `Subject`, `BehaviorSubject`, `ReplaySubject`, `AsyncSubject`?
84. Difference between cold and hot Observables
85. How to handle error and retry logic in Observables

Angular Modules and Architecture

86. Difference between feature module and root module
87. Shared module vs Core module

88. What is Angular CLI and its usage?
89. What is Ahead-of-Time (AOT) compilation?
90. Difference between JIT and AOT compilation

Frequently Asked / Scenario Questions

91. Explain Angular architecture in your project
92. How did you implement routing in your project?
93. How did you handle state management?
94. Explain your usage of services and dependency injection
95. Explain lifecycle hooks you used in your project
96. How did you optimize change detection in your project?
97. How did you integrate RxJS Observables in your project?
98. Explain Angular forms implementation in your project
99. How do configuration files help in project management and build optimization?
100. Explain difference between development and production builds
101. Explain your usage of NgRx store, actions, reducers, and selectors in your project
102. How do you debug state issues using NgRx DevTools?

Micro Frontend / Micro-Component Questions

103. What is a Micro Frontend (MFE)? When would you use it?
104. Explain advantages and challenges of implementing MFEs in Angular.
105. How would you share common services (e.g., authentication, API service) across multiple MFEs?
106. How would you implement communication between two MFEs (e.g., cart and profile)?
107. How do you handle routing when multiple Angular MFEs are hosted on the same page?
108. How can you lazy load micro-components inside a larger Angular shell?
109. How to allow independent deployment of MFEs without affecting main app?
110. How do you handle versioning of shared libraries across multiple MFEs?
111. Using Web Components / Angular Elements for micro-components – how does it work?
112. How to package and distribute a micro-component for reuse?

Advanced / Missed Questions

Performance & Optimization

113. How does `trackBy` improve performance in `*ngFor`? Give an example.
114. How would you lazy load images or components to improve performance?
115. Explain `OnPush` change detection and when you would use it.
116. How can you use memoization with pipes to optimize performance?
117. How do you avoid unnecessary API calls or component re-rendering?

Advanced RxJS

118. Explain higher-order Observables (`switchMap`, `mergeMap`, `exhaustMap`) with examples.
119. How do you handle errors in Observables using `catchError`, `retry`, or `retryWhen`?
120. How do you combine multiple Observables using `forkJoin`, `combineLatest`, or `zip`?
121. How would you implement debounce or throttle for user input or search functionality?

Testing

122. How do you unit test components, services, pipes, and directives using Jasmine/Karma?
123. How do you test Observables and async operations?
124. How do you use `HttpClientTestingModule` to test API calls?
125. How do you perform end-to-end testing using Protractor or Cypress?

Security

126. How do you protect routes using Guards (`CanActivate`, `CanLoad`)?
127. How do you prevent XSS in Angular templates?
128. How do you secure APIs and tokens on the front-end?
129. How do you use Angular `DomSanitizer` to prevent security risks?

Angular CLI & Tooling

130. What are custom schematics and how do you use them?
131. How do you handle `ng update` and migration strategies?
132. How do you optimize builds using budgets, AOT, minification, and tree-shaking?

Internationalization (i18n)

133. How do you translate Angular apps using Angular i18n or `ngx-translate`?
134. How do you lazy load translations for large Angular applications?

Accessibility

135. How do you use ARIA attributes in Angular components?

136. How do you implement keyboard navigation for Angular components?
137. How do you ensure screen reader compatibility?

Server-side Rendering (Angular Universal)

138. When and why would you use server-side rendering (SSR) in Angular?
139. Difference between pre-rendering and live server rendering in Angular Universal.
140. How do you handle API calls when using Angular Universal?

State Management Patterns

141. How do you use NgRx selectors and memoization for performance?
142. How do you handle side effects for async operations in NgRx (Effects)?
143. Difference between local component state and global state in Angular.
144. How do you share state across multiple MFEs or micro-components?

```
## **Angular Scenario-Based Questions**

### **Component Communication & Performance**

1. You have a parent component and multiple child components. How would you optimize communication between them when the ap
2. Explain a situation where you would use `@ViewChild` vs `@ContentChild`. Give a real-world use case.
3. You notice your Angular app has performance issues on a large page with many bindings. How would you optimize change det
4. How would you prevent unnecessary API calls when multiple components subscribe to the same `Observable`?
5. Scenario: You have a form that is reused across multiple modules. How would you structure it for maintainability and reu
6. You have to lazy load multiple feature modules, but one module depends on another. How do you handle module dependencies
7. Explain a scenario where you would prefer reactive forms over template-driven forms and why.
8. You need to dynamically render components based on user roles. How would you implement this?
9. Your Angular app interacts with a third-party library that doesn't have TypeScript typings. How would you integrate it s
10. Scenario: A component needs to display real-time updates from a backend WebSocket connection. How would you handle subs

### **Micro Frontend (MFE) & Micro-Components**

11. What is a Micro Frontend (MFE)? When would you use it?
12. Explain the advantages and challenges of implementing MFEs in a large enterprise Angular application.
13. How would you share common services (for example authentication or API services) across multiple MFEs?
14. Scenario: Two MFEs need to communicate (for example a shopping cart MFE and a user profile MFE). How would you implemen
15. How do you handle routing in a Micro Frontend architecture where multiple Angular apps are hosted on the same page?
16. Explain how you can lazy load micro-components inside a larger Angular shell.
17. Scenario: A feature team wants to deploy their MFE independently without affecting the main app. How would you achieve
18. How do you handle versioning of shared libraries (like Angular Material or RxJS) across multiple MFEs?
19. Explain using Web Components or custom elements as a strategy for micro-components in Angular.
20. Scenario: A micro-component needs to be reused across multiple Angular applications. How would you package and distribu

### **Advanced Scenarios**

21. You have multiple child components with heavy computations. How would you prevent unnecessary change detection cycles f
22. Scenario: You need to implement a global notification system that works across multiple MFEs. How would you design it?
23. Explain how you would debug memory leaks in an Angular application with complex state management.
24. Scenario: Your Angular app uses NgRx. How would you handle side effects for multiple MFEs without duplicating code?
25. You need to integrate a legacy AngularJS application with a modern Angular shell using micro frontends. How would you a
ANGULAR SCENARIO-BASED QUESTIONS

1. You have a parent component and multiple child components. How would you optimize communication between them when the ap
2. Explain a situation where you would use @ViewChild vs @ContentChild. Give a real-world use case.
3. You notice your Angular app has performance issues on a large page with many bindings. How would you optimize change det
4. How would you prevent unnecessary API calls when multiple components subscribe to the same Observable?
5. Scenario: You have a form that is reused across multiple modules. How would you structure it for maintainability and cod
6. You have to lazy load multiple feature modules, but one module depends on another. How do you handle module dependencies
7. Explain a scenario where you would prefer reactive forms over template-driven forms and why.
8. You need to dynamically render components based on user roles. How would you implement this?
9. Your Angular app interacts with a third-party library that doesn't have TypeScript typings. How would you integrate it s
10. Scenario: A component needs to display real-time updates from a backend WebSocket connection. How would you handle subs

MICRO FRONTEND (MFE) AND MICRO-COMPONENT QUESTIONS

11. What is a Micro Frontend (MFE)? When would you use it?
12. Explain the advantages and challenges of implementing MFEs in a large enterprise Angular application.
13. How would you share common services (e.g., authentication, API service) across multiple MFEs?
14. Scenario: Two MFEs need to communicate (e.g., a shopping cart MFE and a user profile MFE). How would you implement cros
15. How do you handle routing in a Micro Frontend architecture where multiple Angular apps are hosted on the same page?
16. Explain how you can lazy load micro-components inside a larger Angular shell.
17. Scenario: A new feature team wants to deploy their MFE independently without affecting the main app. How would you achi
18. How do you handle versioning of shared libraries (like Angular Material or RxJS) across multiple MFEs?
19. Explain using Web Components or custom elements as a strategy for micro-components in Angular.
20. Scenario: A micro-component needs to be reused across multiple Angular applications. How would you package and distribu

ADVANCED SCENARIOS

21. You have multiple child components with heavy computations. How would you prevent unnecessary change detection cycles f
22. Scenario: You need to implement a global notification system that works across multiple MFEs. How would you design it?
23. Explain how you would debug memory leaks in an Angular application with complex state management.
24. Scenario: Your Angular app uses NgRx. How would you handle side effects for multiple MFEs without duplicating code?
25. You need to integrate a legacy AngularJS application with a modern Angular shell using micro frontends. How would you a
```

```
<style type="text/css">.markdown-body .octicon {
  display: inline-block;
  fill: currentColor;
  vertical-align: text-bottom;
```