**Angular Interview Questions**

**Why angular**

Angular is a powerful front-end framework developed by Google, widely used for building dynamic, scalable, and high-performance web applications. Here’s why it’s trending:

Why Use Angular?

* Component-Based Architecture: Breaks UI into reusable components, improving maintainability.
* TypeScript Support: Enhances code quality, debugging, and scalability.
* Two-Way Data Binding: Synchronizes data between the model and view automatically.
* Dependency Injection: Improves modularity and testability.
* Built-in Routing & State Management: Enables seamless navigation and efficient state handling.
* RxJS for Reactive Programming: Handles asynchronous data streams effectively.

Why Is Angular Trending?

* Backed by Google: Ensures long-term support and continuous improvements.
* Enterprise Adoption: Used by companies like Google, IBM, and Microsoft for large-scale applications.
* Scalability & Performance: Ideal for complex applications requiring structured development.
* Strong Developer Community: Extensive resources, libraries, and support.
* Modern Features: Regular updates with enhanced security, performance, and usability.

Angular remains a top choice for developers due to its robust ecosystem and ability to handle complex

**1)Angular Lifecycle hooks**

Angular lifecycle hooks are methods that allow you to tap into key moments in a component's lifecycle. Here's the typical execution sequence:

1. **constructor** - Runs when Angular instantiates the component.
2. **ngOnChanges** - Called when input properties change.
3. **ngOnInit** - Runs once after the component's inputs are initialized.
4. **ngDoCheck** - Invoked during every change detection cycle.
5. **ngAfterContentInit** - Runs once after projected content is initialized.
6. **ngAfterContentChecked** - Called after projected content is checked.
7. **ngAfterViewInit** - Runs once after the component's view is initialized.
8. **ngAfterViewChecked** - Called after the component's view is checked.
9. **ngOnDestroy** - Runs before the component is destroyed.

**2)View Encapsulation in Angular**

View Encapsulation in Angular controls how styles are applied to components, ensuring they don't unintentionally affect other parts of the application. Angular provides three encapsulation modes:

1. **ViewEncapsulation.Emulated** - Angular modifies CSS selectors so styles apply only to the component's view, mimicking Shadow DOM behavior.
2. **ViewEncapsulation**.**ShadowDom** - Uses the browser's native Shadow DOM API to encapsulate styles within the component.
3. **ViewEncapsulation**.**None** - No encapsulation; styles are globally applied and can affect any element in the application.

**3)How componenet share data or interact with each other**

Angular components can share data and interact with each other in several ways, depending on their relationship. Here are the common methods:

1. Parent to Child Communication
   * Using @**Input**() decorator: The parent component passes data to the child component via property binding.
   * Example: <child-component [data]="parentData"></child-component>
2. Child to Parent Communication
   * Using @**Output**() decorator with EventEmitter: The child component emits an event that the parent listens to.
   * Example: <child-component (event)="handleEvent($event)"></child-component>
3. Sibling Component Communication
   * Using a shared service: A service acts as a mediator, storing and sharing data between sibling components.
   * Example: A service with a **BehaviorSubject** or **Subject** can be used to notify components of changes.
4. Communication Without Direct Relationship
   * Using a shared service with RxJS observables: Components subscribe to a service that broadcasts data updates.
   * Using Angular's **ViewChild** or **ContentChild**: Allows a parent component to access a child component's properties and methods directly.

**4)Data Bindings in angular**

Data binding in Angular is a powerful mechanism that keeps the component and the view in sync. It allows data to flow between the template and the component efficiently. There are four types of data binding:

1. **Interpolation** ({{ }})
   * Used to bind component properties to the view.
   * Example: <h1>{{ title }}</h1>
2. **Property** **Binding** ([property] syntax)
   * Binds a property of an HTML element to a component property.
   * Example: <img [src]="imageUrl">
3. **Event** **Binding** ((event) syntax)
   * Allows the component to respond to user interactions.
   * Example: <button (click)="handleClick()">Click Me</button>
4. **Two**-**Way** **Binding** ([(ngModel)] syntax)
   * Synchronizes data between the component and the view.
   * Example: <input [(ngModel)]="userName">

5)**Event Binding in Angular**

Event binding in Angular allows components to listen for and respond to user interactions like clicks, key presses, and mouse movements. It uses a special syntax where the event name is enclosed in parentheses and linked to a method in the component.

**Example:**

<button (click)="handleClick()">Click Me</button>

In this example, when the button is clicked, the handleClick() method in the component is executed.

**Key Features:**

* **Binding to native events**: You can bind to standard events like click, **keyup**, **mouseover**, etc.
* **Passing event data**: Use $event to access event details.
* <input (keyup)="onKeyPress($event)">
* **Using event modifiers**: Angular allows filtering events using modifiers like .enter, .shift, etc.
* <input (keyup.enter)="submitForm()">

**6)Attribute Binding in angular**

Attribute binding in Angular allows you to set values for attributes dynamically in your templates. It is useful when dealing with attributes that do not have corresponding DOM properties, such as ARIA attributes or SVG attributes.

**Syntax:**

<p [attr.attribute-name]="expression"></p>

If the expression evaluates to null or undefined, Angular removes the attribute altogether.

**Common Use Cases:**

1. **Binding ARIA attributes** (for accessibility)
2. <button [attr.aria-label]="actionName">{{ actionName }}</button>
3. **Binding colspan in tables** (since colspan is an attribute, not a property)
4. <td [attr.colspan]="1 + 1">Merged Cells</td>
5. **Setting custom attributes dynamically**
6. <div [attr.data-custom]="customValue"></div>

7)**Class Binding in Angular**

Class binding in Angular allows you to dynamically add or remove CSS classes from an element based on component properties. It helps in styling elements conditionally.

**Syntax:**

<button [class.active]="isActive">Click Me</button>

In this example, the active class is applied only if isActive is true.

**Different Ways to Use Class Binding:**

1. **Binding a Single Class**
2. <div [class.highlight]="isHighlighted"></div>
   * Adds the highlight class when isHighlighted is true.
3. **Binding Multiple Classes Using Object Syntax**
4. <div [class]="{'bold': isBold, 'italic': isItalic}"></div>
   * Adds bold and italic classes based on the respective boolean values.
5. **Binding Multiple Classes Using String Syntax**
6. <div [class]="dynamicClasses"></div>
   * dynamicClasses can be a string like "class1 class2".

**7)Style binding in Angular**

Style binding in Angular allows you to dynamically set styles on elements based on component properties. It helps in applying styles conditionally and making UI elements more interactive.

Syntax:

<p [style.color]="textColor">Styled Text</p>

In this example, the color style is set based on the textColor property in the component.

Different Ways to Use Style Binding:

1. Binding a Single Style Property
2. <div [style.background-color]="bgColor"></div>
   * Sets the background color dynamically.
3. Binding Multiple Styles Using Object Syntax
4. <div [style]="{'width': widthValue, 'height': heightValue}"></div>
   * Allows multiple styles to be applied dynamically.
5. Adding Units to Style Values
6. <p [style.font-size.px]="fontSize">Dynamic Font Size</p>
   * .px, .em, or % can be used to specify units.

**2 way data binding in Angular**

Two-way data binding in Angular allows synchronization between the model and the view, meaning changes in the UI automatically update the model, and vice versa. This is achieved using the [(ngModel)] directive.

How It Works:

* Property Binding ([]): Updates the view when the model changes.
* Event Binding (()): Updates the model when the view changes.
* Two-Way Binding ([()]): Combines both property and event binding.

Example:

<input [(ngModel)]="username">

<p>Hello, {{ username }}!</p>

**Directives in angular**

Directives in Angular are used to extend the functionality of HTML elements by adding custom behavior. They allow developers to manipulate the DOM efficiently. Angular has three types of directives:

1. Component Directives

* These are the most common directives in Angular.
* They include a template and logic, making them reusable UI elements.
* Example: Any Angular component (@Component decorator).

2. Attribute Directives

* Modify the appearance or behavior of an element, component, or another directive.
* Examples:
  + NgClass: Dynamically adds or removes CSS classes.
  + NgStyle: Applies inline styles dynamically.
  + NgModel: Enables two-way data binding.

3. Structural Directives

* Modify the DOM layout by adding or removing elements.
* Examples:
  + NgIf: Conditionally adds or removes elements.
  + NgFor: Iterates over a collection and renders elements.
  + NgSwitch: Displays elements based on a condition.

**Angular Forms**

Angular provides two main types of forms: Template-Driven Forms and Reactive Forms, each with distinct characteristics and use cases.

1. Template-Driven Forms

* Use Case: Best for simple forms like login or contact forms.
* How It Works: Uses directives (ngModel) to manage form state.
* Example:
* <form #myForm="ngForm">
* <input type="text" name="username" [(ngModel)]="username">
* </form>

2. Reactive Forms

* Use Case: Ideal for complex forms requiring dynamic validation or multi-step processes.
* How It Works: Uses FormControl and FormGroup for structured form management.
* **Example:**
* this.myForm = new FormGroup({
* username: new FormControl('')
* });

**Key Differences**

| **Feature** | **Template-Driven Forms** | **Reactive Forms** |
| --- | --- | --- |
| **Setup** | **Implicit (via directives)** | **Explicit (via component class)** |
| **Data Model** | **Mutable** | **Immutable** |
| **Data Flow** | **Asynchronous** | **Synchronous** |
| **Validation** | **Directive-based** | **Function-based** |
| **Scalability** | **Less scalable** | **Highly scalable** |

**Design Patterns in Angular**

Design patterns in Angular help developers build scalable, maintainable, and efficient applications. Some commonly used design patterns in Angular include:

* **Dependency Injection (DI)**: A technique where a class receives its dependencies from an external source rather than creating them itself. This promotes loose coupling and flexibility.
* **Lazy Loading**: Loads modules only when needed, improving performance by reducing initial load time.
* **Singleton Pattern**: Ensures that only one instance of a service exists throughout the application.
* **Observer Pattern**: Used for event-driven programming, where components react to changes in data.
* **Model-View-Controller (MVC)**: Separates concerns by organizing code into models, views, and controllers.
* **Factory Pattern**: Creates objects without specifying the exact class, improving modularity.
* **State Management**: Helps manage application state efficiently, often implemented using libraries like NgRx.

**Observables vs Promises**

Observables and Promises are both used for handling asynchronous operations in JavaScript, but they have key differences:

Promises

* Handles a single value: A Promise resolves once and provides a single result.
* Not cancellable: Once a Promise starts, it will execute regardless of whether the result is needed.
* Simpler syntax: Uses .then() and .catch() for handling success and errors.
* Example:
* fetch('https://api.example.com/data')
* .then(response => response.json())
* .then(data => console.log(data))
* .catch(error => console.error(error));

Observables

* Handles multiple values over time: Can emit multiple values asynchronously.
* Cancellable: You can unsubscribe from an Observable to stop receiving data.
* Uses RxJS operators: Provides powerful operators like map(), filter(), and retry().
* Example:
* import { Observable } from 'rxjs';
* const observable = new Observable(subscriber => {
* subscriber.next('First value');
* setTimeout(() => subscriber.next('Second value'), 1000);
* setTimeout(() => subscriber.complete(), 2000);
* });
* observable.subscribe({
* next: value => console.log(value),
* complete: () => console.log('Completed')
* });

**Key Differences**

| **Feature** | **Promises** | **Observables** |
| --- | --- | --- |
| **Number of values** | **Single** | **Multiple** |
| **Cancellation** | **Not possible** | **Possible via unsubscribe()** |
| **Execution** | **Eager (executes immediately)** | **Lazy (executes only when subscribed)** |
| **Operators** | **Limited (then, catch)** | **Extensive (map, filter, retry)** |

**Observables are more flexible and powerful, especially for handling streams of data like WebSockets or user interactions.**

**Pipes in Angular**

Pipes in Angular are used to transform data in templates, making it easier to format and display values dynamically. They allow you to apply transformations directly within the HTML without modifying the component logic.

Types of Pipes

1. Built-in Pipes:
   * DatePipe: Formats dates.
   * CurrencyPipe: Formats numbers as currency.
   * UpperCasePipe / LowerCasePipe: Converts text to uppercase or lowercase.
   * PercentPipe: Formats numbers as percentages.
   * JsonPipe: Converts objects to JSON format.
2. Custom Pipes:
   * You can create your own pipes using the @Pipe decorator.
   * Example:
   * import { Pipe, PipeTransform } from '@angular/core';
   * @Pipe({ name: 'reverse' })
   * export class ReversePipe implements PipeTransform {
   * transform(value: string): string {
   * return value.split('').reverse().join('');
   * }
   * }

Usage in Templates

<p>{{ today | date:'short' }}</p>

<p>{{ price | currency:'USD' }}</p>

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

Ques-if pipe got response as null or empty what will happen

Ans-some error it wil throw

Que-What is async pipe

The **AsyncPipe** in Angular is used to automatically subscribe to an **Observable** or **Promise** and retrieve the latest emitted value. It simplifies handling asynchronous data in templates without manually subscribing and unsubscribing.

**Key Features of AsyncPipe**

* **Automatic Subscription & Unsubscription**: It subscribes to an Observable or Promise and unsubscribes when the component is destroyed, preventing memory leaks.
* **Efficient Change Detection**: When a new value is emitted, the AsyncPipe marks the component for change detection.
* **Simplifies Code**: Eliminates the need for explicit .subscribe() calls in the component.

**Example Usage**

<p>Data: {{ myObservable | async }}</p>

In this example, myObservable emits values asynchronously, and the AsyncPipe automatically updates the template when new data arrives.

**What are subjects in angular**

In Angular, Subjects are a special type of Observable from RxJS that allow values to be multicasted to multiple subscribers. Unlike regular Observables, which are unicast (each subscriber gets an independent execution), Subjects share the same execution among all subscribers.

Types of Subjects in Angular

1. Subject:
   * Acts as both an Observable and an Observer.
   * Can emit values using .next().
   * Example:
   * import { Subject } from 'rxjs';
   * const subject = new Subject<number>();
   * subject.subscribe(value => console.log(`Observer A: ${value}`));
   * subject.subscribe(value => console.log(`Observer B: ${value}`));
   * subject.next(1);
   * subject.next(2);
2. BehaviorSubject:
   * Stores the latest value and emits it to new subscribers.
   * Requires an initial value.
   * Example:
   * import { BehaviorSubject } from 'rxjs';
   * const behaviorSubject = new BehaviorSubject<number>(0);
   * behaviorSubject.subscribe(value => console.log(`Observer A: ${value}`));
   * behaviorSubject.next(1);
   * behaviorSubject.next(2);
3. ReplaySubject:
   * Stores a specified number of previous values and emits them to new subscribers.
   * Useful for caching values.
   * Example:
   * import { ReplaySubject } from 'rxjs';
   * const replaySubject = new ReplaySubject<number>(2);
   * replaySubject.next(1);
   * replaySubject.next(2);
   * replaySubject.next(3);
   * replaySubject.subscribe(value => console.log(`Observer A: ${value}`));
4. AsyncSubject:
   * Emits only the last value when .complete() is called.
   * Example:
   * import { AsyncSubject } from 'rxjs';
   * const asyncSubject = new AsyncSubject<number>();
   * asyncSubject.subscribe(value => console.log(`Observer A: ${value}`));
   * asyncSubject.next(1);
   * asyncSubject.next(2);
   * asyncSubject.complete()

**Queuing and parallel request handling in angular**

In Angular, handling multiple HTTP requests efficiently often involves using **mergeMap**, **switchMap**, and queuing techniques with RxJS.

**Key Operators for HTTP Requests**

1. **mergeMap**:
   * Executes multiple HTTP requests concurrently.
   * Useful when you need all requests to complete, regardless of order.
   * Example:
   * this.source$.pipe(
   * mergeMap(value => this.http.get(`https://api.example.com/data/${value}`))
   * ).subscribe(response => console.log(response));
2. **switchMap**:
   * Cancels the previous request if a new one starts.
   * Ideal for scenarios where only the latest request matters (e.g., search autocomplete).
   * Example:
   * this.searchTerm$.pipe(
   * switchMap(term => this.http.get(`https://api.example.com/search?q=${term}`))
   * ).subscribe(response => console.log(response));

**Queuing HTTP Requests**

* If you need to queue requests (execute them sequentially), you can use **concatMap**.
* Example:
* this.source$.pipe(
* concatMap(value => this.http.get(`https://api.example.com/data/${value}`))
* ).subscribe(response => console.log(response));

**HttpVerbs difference**

HTTP methods define the actions that can be performed on a resource in a web application. Here’s a breakdown of the key differences:

**GET vs POST in HTTP Requests**

Both GET and POST are HTTP methods used to communicate with a server, but they serve different purposes and have key differences.

**1. GET Method**

* **Purpose**: Retrieves data from the server.
* **Characteristics**:
  + Data is sent in the URL as query parameters.
  + Can be cached by browsers.
  + Can be bookmarked.
  + Should not be used for sensitive data (visible in the URL).
  + Limited data length due to URL restrictions.
* **Example**:
* this.http.get('https://api.example.com/users');

**2. POST Method**

* **Purpose**: Sends data to the server, often creating or modifying a resource.
* **Characteristics**:
  + Data is sent in the request body (not visible in the URL).
  + Cannot be cached or bookmarked.
  + No restrictions on data length.
  + More secure for sending sensitive data.
* **Example**:
* this.http.post('https://api.example.com/users', { name: 'John' });

**Key Differences**

| **Feature** | **GET** | **POST** |
| --- | --- | --- |
| Data Location | URL (query parameters) | Request body |
| Security | Less secure (visible in URL) | More secure (hidden in body) |
| Caching | Can be cached | Cannot be cached |
| Bookmarking | Can be bookmarked | Cannot be bookmarked |
| Data Length | Limited by URL | No restrictions |

For a deeper dive, check out [this guide](https://www.w3schools.com/tags/ref_httpmethods.asp) or [this discussion](https://stackoverflow.com/questions/2080863/what-is-the-difference-between-a-http-get-and-http-post-and-why-is-http-post-wea). Let me know if you need further clarification! 🚀

3**PUT vs PATCH in HTTP Requests**

Both PUT and PATCH are used to update resources on a server, but they have key differences in how they handle updates.

**1. PUT Method**

* **Purpose**: Replaces an entire resource with a new version.
* **Idempotent**: Multiple calls result in the same state.
* **Example**:
* this.http.put('https://api.example.com/users/1', { name: 'John', age: 30 });
  + If the resource originally had { name: 'Alice', age: 25, city: 'NY' }, after the PUT request, it will be **completely replaced** with { name: 'John', age: 30 }, potentially removing city.

**2. PATCH Method**

* **Purpose**: Partially updates a resource without replacing the entire object.
* **Not necessarily idempotent**: Multiple calls may modify different parts of the resource.
* **Example**:
* this.http.patch('https://api.example.com/users/1', { age: 30 });
  + If the resource originally had { name: 'Alice', age: 25, city: 'NY' }, after the PATCH request, only age will be updated, keeping name and city unchanged.

**Key Differences**

| **Feature** | **PUT** | **PATCH** |
| --- | --- | --- |
| Update Type | Full replacement | Partial update |
| Idempotency | Yes | Not always |
| Data Sent | Entire resource | Only changed fields |
| Use Case | Overwriting an object | Modifying specific fields |

**HttpDelete**

**HTTP DELETE Method**

**The HTTP DELETE method is used to remove a resource from the server. It is idempotent, meaning multiple identical DELETE requests should have the same effect as a single request.**

**Key Characteristics**

* **Purpose: Deletes a specified resource.**
* **Idempotent: Multiple calls result in the same state (resource deleted).**
* **Not Cacheable: DELETE requests are not stored by browsers.**
* **No Request Body: Typically, DELETE requests do not require a body.**

**Example Usage in Angular**

**this.http.delete('https://api.example.com/users/1')**

**.subscribe(response => console.log('User deleted:', response));**

**Common Response Codes**

| **Status Code** | **Meaning** |
| --- | --- |
| **200 OK** | **Resource deleted successfully, response body included.** |
| **204 No Content** | **Resource deleted successfully, no response body.** |
| **404 Not Found** | **Resource does not exist.** |
| **403 Forbidden** | **User lacks permission to delete the resource.** |