## **Q1: Why do we need a directive?**

Ans: In Angular, directives are like special instructions you give to HTML elements. They act as a bridge between your component's code and the HTML template, allowing you to add new behaviors or modify existing ones for those elements. Imagine them as tiny tools that enhance your HTML's capabilities.

Here's why directives are essential for Angular development:

1. **Adding New Behaviors:** Directives let you extend the functionality of HTML elements. You can't normally add features like showing/hiding content based on conditions or highlighting errors directly in HTML. Directives enable you to do just that!

**Example:** Angular provides a built-in directive called \*ngIf. Let's say you want to display a welcome message only if a user is logged in. You can't achieve this with plain HTML. But, with \*ngIf, you can write:

HTML

<p \*ngIf="isLoggedIn">Welcome, {{ username }}!</p>

Here, \*ngIf acts as a directive. It checks the isLoggedIn property in your component's code. If it's true, the welcome message is displayed; otherwise, it's hidden.

1. **Modifying Element Appearance:** Directives can also change how elements look or behave. For instance, you might want to make a button turn red when clicked. Directives come in handy for such styling tweaks.

**Example:** Another built-in directive, [ngClass], lets you dynamically add or remove CSS classes based on conditions. Imagine you have a button with the class active for styling when clicked. You can use [ngClass] like this:

HTML

<button [ngClass]="{'active': isActive}">Click me</button>

Here, [ngClass] checks the isActive property. If it's true, the active class is added to the button, applying the desired styling.

1. **Customizing Components:** Directives can even be used to create custom components that you can reuse throughout your application. These components can encapsulate specific functionalities and styles, making your code more modular.

In essence, directives empower you to go beyond basic HTML functionalities and create a more interactive and dynamic user experience in your Angular applications. They provide a clean way to separate your component logic from the view (HTML template), promoting better code organization and maintainability.

## Q2: **What is "?" i.e. "Safe Navigation Operator"?**

Ans: The "?," symbol, also known as the safe navigation operator (or optional chaining operator), is a handy tool in Angular (and many other programming languages) to prevent errors when working with potentially null objects.

Imagine you're building a form in Angular to display user information. You have a user object in your component that might contain properties like name, email, and address. But what if the user object is initially null because the data hasn't been fetched yet?

Here's where the safe navigation operator comes in:

**Without Safe Navigation Operator:**

HTML

<p>Name: {{ user.name }}</p> <p>Email: {{ user.email }}</p> ```

If `user` is null, trying to access `user.name` or `user.email` would throw an error, crashing your application.

\*\*Using Safe Navigation Operator:\*\*

```html

<p>Name: {{ user?.name }}</p>

<p>Email: {{ user?.email }}</p>

Now, with the safe navigation operator (?), if user is null, it simply returns null for user.name and user.email, preventing the error. You can then handle this null value in your template or component logic.

**Example with Handling Null Values:**

HTML

<p>Name: {{ user?.name || 'No name available' }}</p>

<p>Email: {{ user?.email || 'No email available' }}</p>

Here, the || operator (logical OR) is used as a fallback. If user.name or user.email is null, the message "No name available" or "No email available" will be displayed instead.

**Benefits of Safe Navigation Operator:**

* **Prevents Errors:** It safeguards your code from crashing due to null references.
* **Improves Readability:** Makes your code clearer by explicitly indicating that you're checking for null before accessing properties.
* **Cleaner Templates:** Avoids long conditional statements within templates to handle null checks.

## Q3: **How can we share data among components?**

Ans: In Angular, sharing data between components is essential for building interactive and dynamic applications. Here are the common methods you can use, explained in a beginner-friendly way with code examples:

**1. Parent-to-Child Data Sharing (using @Input):**

* Imagine you have a parent component (parent.component.ts) displaying a product list and a child component (product.component.ts) showing product details.
* You want to pass the selected product information from the parent to the child component.

**Parent Component (parent.component.ts):**

TypeScript

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

@Component({

selector: 'app-parent',

template: `

<h2>Product List</h2>

<ul>

<li \*ngFor="let product of products" (click)="onProductSelected(product)">

{{ product.name }}

</li>

</ul>

<app-product [selectedProduct]="selectedProduct"></app-product>

`

})

export class ParentComponent {

products = [

{ id: 1, name: 'Product 1' },

{ id: 2, name: 'Product 2' }

];

selectedProduct: any;

onProductSelected(product: any) {

this.selectedProduct = product;

}

}

* The @Input() decorator is used on the child component's property (selectedProduct) in the parent's template. This allows the parent to pass data down to the child.
* When a product is clicked in the list, the onProductSelected method in the parent component sets the selectedProduct property with the clicked product details.
* The child component receives this data through the [selectedProduct] binding in the parent's template.

**2. Child-to-Parent Data Sharing (using @Output and EventEmitter):**

* Now, imagine the child component (product.component.ts) has a button to add a new product. You want to notify the parent component when this button is clicked.

**Child Component (product.component.ts):**

TypeScript

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

@Component({

selector: 'app-product',

template: `

<h2>Product Details</h2>

<p \*ngIf="selectedProduct">Name: {{ selectedProduct.name }}</p>

<button (click)="onAddProduct()">Add New Product</button>

`

})

export class ProductComponent {

@Input() selectedProduct: any;

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

onAddProduct() {

this.addProduct.emit(); // Emit an event when the button is clicked

}

}

* The @Output() decorator is used on the child component's event (addProduct). This allows the child to emit an event that the parent can listen to.
* The EventEmitter class creates an event object that can be used to emit data.
* When the "Add New Product" button is clicked, the onAddProduct method emits an event using this.addProduct.emit().
* The parent component can then listen to this event using the (addProduct) binding in the template.

**3. Sharing Data Across Unrelated Components (using Services):**

* If two components that aren't in a parent-child relationship need to share data (e.g., a shopping cart component and a checkout component), you can use a service.

**Data Service (data.service.ts):**

TypeScript

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

import { BehaviorSubject } from 'rxjs'; // Import from RxJS

@Injectable({

providedIn: 'root' // Register as a root service (app-wide)

})

export class DataService {

private cartItems = new BehaviorSubject<any[]>([]);

currentCartItems = this.cartItems.asObservable();

addToCart(item: any) {

const currentItems = this.cartItems.getValue();

currentItems.push(item);

this.cartItems.next(currentItems);

}

}

* This service holds a BehaviorSubject that stores the cart items as an observable.
* The addToCart method adds items to the cart and updates the BehaviorSubject.
* Both the shopping cart and checkout components can inject this service and use its methods and observables to share and access cart data.

These are the primary methods for data sharing in Angular. The best approach depends on the relationship. You can definitely use session storage and local storage for data sharing in Angular applications

Q4: What is routing in Angular, and why is it essential in single-page applications (SPAs)?

* Routing in Angular refers to the process of navigating between different views or components based on the URL.
* In SPAs, where the entire application runs within a single HTML page, routing allows users to navigate between different sections of the application without the need for full-page reloads.
* Routing enables a more seamless and interactive user experience by providing a way to organize and navigate through different parts of the application.

Q5: How do you configure routing in an Angular application? Explain the role of RouterModule and routes array.

* Routing in Angular is configured using the RouterModule.forRoot() method in the AppModule and RouterModule.forChild() method in feature modules.
* The routes array defines the routes of the application, specifying the URL paths and corresponding components to be displayed when those paths are matched.
* Each route object in the routes array consists of properties such as path (URL path), component (component to be displayed), pathMatch (matching strategy), redirectTo (redirect route), and canActivate (route guards).