**Reusability in Angular**

Reusability is a cornerstone of efficient and maintainable software development.1 In Angular, it's achieved primarily through the use of **Components**.2

**Key Concepts for Reusability in Angular:**

* **Components as Building Blocks:**
  + Angular's core concept is built around the idea of creating small, self-contained components.3
  + Each component encapsulates its own logic, template, and styles, making it independent and reusable across different parts of your application.4
* **Data Input and Output:**
  + **@Input() Decorator:** Allows you to pass data into a component from its parent component.5 This makes the component flexible and adaptable to different data scenarios.6
  + **@Output() Decorator:** Enables the component to emit events to its parent, allowing it to communicate changes or actions back to the parent component.7
* **Services:**
  + Services are injectable classes that provide data and functionality to components.8
  + By creating reusable services, you can share data and logic across multiple components without tightly coupling them.
* **Directives:**
  + Directives modify the behavior or appearance of DOM elements.9
  + Reusable directives can be created to encapsulate common UI interactions or styling.
* **Pipes:**
  + Pipes transform data within the template.10
  + Reusable pipes can be created to format data, filter data, or perform other transformations.11

**Benefits of Reusability:**

* **Reduced Development Time:** Avoids redundant code and speeds up development.12
* **Improved Maintainability:** Changes to a reusable component only need to be made in one place, reducing the risk of errors and making updates easier.13
* **Increased Code Quality:** Encourages the creation of well-defined, modular, and testable components.
* **Better Code Organization:** Leads to cleaner, more organized, and easier-to-understand codebases.

**Example:**

Imagine you have a reusable component that displays a user's profile picture.

* **UserAvatarComponent:** This component would accept the user's image URL as an @Input() and display it.
* You can then use this UserAvatarComponent in various places throughout your application (e.g., user profiles, chat lists, etc.) without having to write the image display logic repeatedly.

By embracing these reusability principles, you can build more efficient, maintainable, and scalable Angular applications.14

Example:

TypeScript

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

@Component({

selector: 'app-counter',

template: `

<button (click)="increment()">+</button>

<span>{{ count }}</span>

<button (click)="decrement()">-</button>

`,

})

export class CounterComponent {

@Input() initialValue: number = 0;

@Input() step: number = 1;

count: number = 0;

ngOnInit() {

this.count = this.initialValue;

}

increment() {

this.count += this.step;

}

decrement() {

this.count -= this.step;

}

}

**Explanation:**

1. **Inputs:**
   * initialValue: Defines the initial count value for the counter. Defaults to 0.
   * step: Defines the increment/decrement step for the counter. Defaults to 1.
2. **ngOnInit():**
   * In the ngOnInit lifecycle hook, we initialize the count property with the provided initialValue.
3. **Increment/Decrement Methods:**
   * The increment() method increases the count by the step value.
   * The decrement() method decreases the count by the step value.

**Usage in Parent Component:**

HTML

<app-counter [initialValue]="5"></app-counter>

<app-counter [initialValue]="10" [step]="2"></app-counter>

* The first instance of app-counter will start with an initial value of 5 and increment/decrement by 1.
* The second instance will start with an initial value of 10 and increment/decrement by 2.

**Key Improvements:**

* **Flexibility:** This component is highly reusable as it can be easily customized with different initial values and step sizes.
* **Maintainability:** The counter logic is encapsulated within a single component, making it easier to maintain and update.
* **Readability:** The component's template is clean and concise, improving code readability.

This reusable CounterComponent demonstrates how to leverage Angular's component architecture and input properties to create flexible and reusable UI elements.

Example2:

TypeScript

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

@Component({

selector: 'app-todo-list',

template: `

<ul>

<li \*ngFor="let todo of todos">

{{ todo }}

<button (click)="removeTodo(todo)">Remove</button>

</li>

</ul>

<input type="text" [(ngModel)]="newTodo" placeholder="Add Todo">

<button (click)="addTodo()">Add</button>

`,

})

export class TodoListComponent {

@Input() todos: string[] = [];

@Output() todoAdded = new EventEmitter<string>();

newTodo: string = '';

addTodo() {

if (this.newTodo.trim() !== '') {

this.todoAdded.emit(this.newTodo);

this.newTodo = '';

}

}

removeTodo(todo: string) {

this.todos = this.todos.filter(t => t !== todo);

}

}

**Explanation:**

1. **Inputs:**
   * todos: An array of strings representing the list of todo items. This input allows the parent component to provide the initial list of todos.
2. **Outputs:**
   * todoAdded: An EventEmitter that emits a new todo item when the user adds it. This allows the parent component to update its own todo list.
3. **Template:**
   * The template uses \*ngFor to iterate over the todos array and display each todo item with a "Remove" button.
   * An input field allows the user to enter a new todo.
   * Buttons are provided for adding and removing todo items.
4. **Methods:**
   * addTodo(): Adds the new todo to the list and emits the new todo through the todoAdded event.
   * removeTodo(): Removes the specified todo item from the list.

**Usage in Parent Component:**

HTML

<app-todo-list [todos]="myTodos" (todoAdded)="addTodo($event)"></app-todo-list>

* [todos]="myTodos": Passes the parent component's myTodos array to the todos input of the TodoListComponent.
* (todoAdded)="addTodo($event)": Binds the todoAdded event emitted by the child component to the addTodo() method in the parent component.

**Parent Component (Example):**

TypeScript

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

@Component({

selector: 'app-parent',

template: `

<app-todo-list [todos]="myTodos" (todoAdded)="addTodo($event)"></app-todo-list>

`,

})

export class ParentComponent {

myTodos: string[] = ['Task 1', 'Task 2'];

addTodo(newTodo: string) {

this.myTodos.push(newTodo);

}

}

This reusable TodoListComponent demonstrates how to create a flexible and maintainable component with input and output properties, making it easy to integrate into various parts of your Angular application.