Enhanced E-commerce Example with Advanced Input Properties

Introduction

This example builds on the logic found in Anatomy of a Component and demonstrates more in-depth use of Angular Component Inputs, including: - Multiple input properties - Input property aliases - Using get/set on input properties - Conditional input handling - Complex data passing

We expand the original e-commerce application by introducing a product “badge” and a configurable “highlight” toggle for product items.

Code

app.component.ts

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

@Component({
  selector: 'app-root',
  template: `
    <h1>Enhanced E-commerce Store</h1>
    <app-product-list (addToCart)="handleAddToCart($event)"></app-product-list>
    <app-cart [cartItems]="cartItems"></app-cart>
  `,
  styleUrls: ['./app.component.css']
})
export class AppComponent {
  cartItems: any[] = [];

  handleAddToCart(product: any): void {
    this.cartItems.push(product);
  }
}
```

product-list.component.ts

```
import { Component, OnInit, Output, EventEmitter } from '@angular/core';
import { ProductService } from '../product.service';

@Component({
  selector: 'app-product-list',
  template: `
    <h2>Enhanced Products</h2>
    <app-product-item
      *ngFor="let product of products"
      [product]="product"
      [highlight]="isPremiumProduct(product)"
      (add)="onAddToCart($event)"
    >
    </app-product-item>
  `,
  styleUrls: ['./product-list.component.css']
})
```

```
export class ProductListComponent implements OnInit {
  products: any[] = [];
  @Output() addToCart = new EventEmitter<any>();

  constructor(private productService: ProductService) {}

  ngOnInit(): void {
    this.products = this.productService.getProducts();
  }

  onAddToCart(product: any): void {
    this.addToCart.emit(product);
  }

  /**
   * Marks products over a certain price as "premium", to highlight them.
   */
  isPremiumProduct(product: any): boolean {
    return product.price > 1000;
  }
}
```

product-item.component.ts

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

@Component({
  selector: 'app-product-item',
  template: `
    <div class="product" [class.highlighted]="highlight">
      <app-product-badge
        [badgeLabel]="badgeLabel"
        [badgeType]="badgeType"
      ></app-product-badge>

      <h3>{{ product.name }}</h3>
      <p>{{ product.description }}</p>

      <!-- Content projection example (unused but available) -->
      <ng-content></ng-content>

      <button (click)="addToCart()">Add to Cart</button>
    </div>
  `,
  styleUrls: ['./product-item.component.css'],
  changeDetection: ChangeDetectionStrategy.OnPush
})
export class ProductItemComponent implements OnChanges {
  @Input() product: any;
  @Input() highlight = false; // Simple boolean input for conditionally styling

  /**
   * A dynamic label to be displayed in the product badge.
```

```
    * This input is optional. If not provided in the template, it defaults inside the
        badge component.
    */
  @Input() badgeLabel?: string;

  /**
   * An example demonstrating an input property alias.
   * The parent (ProductList) or template can bind to `[aliasBadgeType]` if preferred.
   */
  @Input('aliasBadgeType') badgeType?: string;

  @Output() add = new EventEmitter<any>();

  ngOnChanges(changes: SimpleChanges): void {
    if (changes['product']) {
      console.log('Product changed:', changes['product'].currentValue);
    }
    if (changes['highlight']) {
      console.log('Highlight changed:', changes['highlight'].currentValue);
    }
  }

  addToCart(): void {
    this.add.emit(this.product);
  }
}
```

product-badge.component.ts

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

@Component({
  selector: 'app-product-badge',
  template: `
    <span class="badge" [ngClass]="computedBadgeType">
      {{ computedBadgeLabel }}
    </span>
  `,
  styleUrls: ['./product-badge.component.css']
})
export class ProductBadgeComponent {
  private _badgeLabel?: string = 'Default';
  private _badgeType?: string = 'neutral';

  /**
   * Demonstrates the use of setter/getter on an input property.
   * If "badgeLabel" is empty or null, we fall back to "Default".
   */
  @Input() set badgeLabel(value: string | undefined) {
    this._badgeLabel = value && value.trim() ? value : 'Default';
  }

  get badgeLabel(): string {
    return this._badgeLabel ?? 'Default';
  }

  /**
   * A second input property—demonstrates a typical scenario with multiple inputs.
   * This is just a string that maps to a CSS class (like "premium", "sale", etc.).
   */
  @Input() set badgeType(value: string | undefined) {
    this._badgeType = value && value.trim() ? value : 'neutral';
```

```
  }

  get badgeType(): string {
    return this._badgeType ?? 'neutral';
  }

  /**
   * Dynamically determines the actual label text to display in the badge.
   */
  get computedBadgeLabel(): string {
    return this._badgeLabel ?? 'Default';
  }

  /**
   * Dynamically determines the CSS class to apply based on badgeType.
   */
  get computedBadgeType(): string {
    return 'badge-' + (this._badgeType ?? 'neutral');
  }
}
```

cart.component.ts

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

@Component({
  selector: 'app-cart',
  template: `
    <h2>Shopping Cart</h2>
    <div *ngFor="let item of cartItems">
      {{ item.name }} - {{ item.price | currency }}
    </div>
  `,
  styleUrls: ['./cart.component.css']
})
export class CartComponent {
  @Input() cartItems: any[] = [];
}
```

product.service.ts

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

@Injectable({
  providedIn: 'root'
})
export class ProductService {
  private products = [
    {
      id: 1,
      name: 'Laptop',
      description: 'A high-performance laptop',
      price: 1299.99
    },
    {
      id: 2,
      name: 'Smartphone',
      description: 'A powerful smartphone',
      price: 799.99
    },
```

```
    {
      id: 3,
      name: 'Headphones',
      description: 'Noise-cancelling headphones',
      price: 199.99
    },
    {
      id: 4,
      name: 'Camera',
      description: 'DSLR camera for professionals',
      price: 1499.99
    }
  ];

  getProducts() {
    return this.products;
  }
}
```

app.module.ts

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';

import { AppComponent } from './app.component';
import { ProductListComponent } from './product-list/product-list.component';
import { ProductItemComponent } from './product-item/product-item.component';
import { CartComponent } from './cart/cart.component';
import { ProductService } from './product.service';
import { ProductBadgeComponent } from './product-badge/product-badge.component';

@NgModule({
  declarations: [
    AppComponent,
    ProductListComponent,
    ProductItemComponent,
    CartComponent,
    ProductBadgeComponent
  ],
  imports: [BrowserModule],
  providers: [ProductService],
  bootstrap: [AppComponent]
})
export class AppModule {}
```

Minimal CSS (Optional)

```
.product {
  border: 1px solid #ccc;
  padding: 1rem;
  margin-bottom: 1rem;
}

.highlighted {
  border-color: gold;
}

.badge {
  display: inline-block;
  padding: 0.25rem 0.5rem;
```

```
  margin-right: 0.5rem;
  border-radius: 4px;
  font-size: 0.85rem;
  color: white;
}

.badge-neutral {
  background-color: gray;
}

.badge-promo {
  background-color: purple;
}

.badge-premium {
  background-color: goldenrod;
}

div {
  margin-bottom: 0.5rem;
}

h1 {
  text-align: center;
}
```

Explanation of the Code

Overview

This enhanced example demonstrates advanced component input features within an Angular e-commerce demo. It builds upon the base logic shown in Anatomy of a Component, adding examples of:

- Multiple `Input()` properties
- Input aliasing
- Getter and setter usage
- Conditional input handling

AppComponent

- **Purpose**:
  - Serves as the root component of the entire application.
  - Contains the global state for items added to the cart.
- **Template Highlights**:
  - Displays the main title “Enhanced E-commerce Store.”
  - Uses and listens for an `(addToCart)` event, which is emitted when a product is added.
  - Uses to show items currently in the cart by passing `[cartItems]="cartItems"`.
- **Key Properties and Methods**:
  - `cartItems: any[]`: An array of products that the user has added to the cart.
  - `handleAddToCart(product: any)`: A method that receives a product object and pushes it into `cartItems`.
- **Why This Matters**:
  - `AppComponent` acts as a global container where child components can communicate upward (e.g., adding items).
  - The parent can then distribute relevant data to other child components.

ProductListComponent

- **Purpose**:
  - Fetches a list of products from `ProductService` and presents each product.
  - Acts as a parent to `ProductItemComponent`, passing down product data as inputs.
- **Template Highlights**:
  - Enhanced Products heading.
  - Iterates through products with `*ngFor`.
  - Binds `[product]="product"` so each receives product data.
  - Binds `[highlight]="isPremiumProduct(product)"` to conditionally set a `highlight` flag on premium items.
  - Captures `(add)` event from using `(add)="onAddToCart($event)"` and re-emits that event upward to `AppComponent` with `@Output() addToCart`.
- **Key Properties and Methods**:
  - `products: any[]`: Array of product objects, fetched from `ProductService` in `ngOnInit()`.
  - `@Output() addToCart`: An `EventEmitter` that notifies the parent component (`AppComponent`) whenever a user clicks “Add to Cart.”
  - `onAddToCart(product: any)`: Method that re-emits the product object via `addToCart.emit(product)`.
  - `isPremiumProduct(product: any)`: Returns true if a product’s price is above a certain threshold (e.g., 1000). Used to highlight premium or expensive products.
- **Why This Matters**:
  - Demonstrates parent-child data flow, where `ProductListComponent` passes data down to `ProductItemComponent`.
  - Also shows upward data flow, turning child `add` events into a parent `addToCart` event.

ProductItemComponent

- **Purpose**:
  - Renders an individual product, along with an “Add to Cart” button.
  - Integrates a to display additional labels or designations (e.g., “Premium,” “Promo”).
- **Template Highlights**:
  - Uses `[class.highlighted]="highlight"` to activate a special CSS border if highlight is set to true.
  - Presents while binding `[badgeLabel]` and `[badgeType]`.
  - Provides an slot that allows future expansions or custom content injection.
- **Inputs**:
  - `@Input() product`: Data object containing name, description, price, etc.
  - `@Input() highlight`: Boolean that determines whether the product should have a special CSS border or some emphasis.
  - `@Input() badgeLabel?: string`: Optional text label displayed by `ProductBadgeComponent`.
  - `@Input('aliasBadgeType') badgeType?: string`: Demonstrates aliasing an input property. The template can reference this property as `[aliasBadgeType]="..."` if desired.
- **Output**:
  - `@Output() add`: `EventEmitter` that emits the associated product when the user clicks the “Add to Cart” button.
- **Lifecycle Hook**:
  - `ngOnChanges(changes: SimpleChanges)`: Monitors changes to both product and highlight. Logs them to the console for demonstration.
- **Why This Matters**:
  - Showcases advanced input usage (multiple inputs, optional inputs, property aliasing).
  - Illustrates the standard parent→child data flow (product, highlight) and child→parent event flow (add).
  - Provides the structure to expand upon for any additional meta-data (badges,

styles, etc.).

ProductBadgeComponent

- **Purpose**:
  - Renders a small “badge” on a product, showing a label and applying a style class based on the type.
- **Template Highlights**:
  - `<span class="badge" [ngClass]="computedBadgeType">{{ computedBadgeLabel }}`: The display includes text and a dynamic CSS class.
- **Private Fields**:
  - `_badgeLabel?: string = 'Default'`: The internally stored label.
  - `_badgeType?: string = 'neutral'`: The internally stored type.
- **Inputs (with getters and setters)**:
  - `badgeLabel`: The input setter checks if the provided string is non-empty, defaulting to “Default” if not.
  - `badgeType`: Similarly, the input setter assigns either the provided value or “neutral” if none is supplied.
- **Computed Properties**:
  - `computedBadgeLabel`: Returns the final label to display, either the user-provided one or “Default.”
  - `computedBadgeType`: Constructs a CSS class name such as “badge-premium” or “badge-promo.” If none is provided, we fall back to “badge-neutral.”
- **Why This Matters**:
  - Demonstrates how to use setter/getter logic on an `@Input()` to enforce custom validation or fallback logic.
  - Encourages consistent styling via computed property-based class names (the consumer of the component only passes in the type, not the actual CSS class name).

CartComponent

- **Purpose**:
  - Displays items that the user has added to the cart.
- **Template Highlights**:
  - Iterates over an array of cartItems with `*ngFor`.
  - Uses Angular’s currency pipe (`{{ item.price | currency }}`) to format the product’s price.
- **Input**:
  - `cartItems: any[] = []`: We receive this from the parent (`AppComponent`), storing each product the user chose to add.
- **Why This Matters**:
  - Demonstrates a simple read-only child that uses data from its parent to display relevant information.
  - Shows how built-in pipes (like `currency`) can be leveraged for quick formatting.

ProductService

- **Purpose**:
  - Provides the data for all products in the application.
  - Demonstrates how an Angular service can be used for data retrieval and application logic that’s separated from the components.
- **Methods**:
  - `getProducts()`: Returns an array of product objects, each containing id, name, description, and price.
- **Why This Matters**:
  - Separates data concerns from the UI.
  - Illustrates Angular’s dependency injection, as it’s set to `providedIn: 'root'`,

making it available across the entire app.

AppModule

- **Purpose**:
  - The root module that configures the entire Angular application by declaring components and importing necessary Angular features.
- **Declarations**:
  - `AppComponent`, `ProductListComponent`, `ProductItemComponent`, `CartComponent`, and `ProductBadgeComponent`. All components must be declared in a module before they can be used.
- **Imports**:
  - `BrowserModule`: Required for applications running in a browser.
- **Providers**:
  - `ProductService`: Registered here so that it can be dependency-injected into any component that needs it.
- **Bootstrap**:
  - `AppComponent`: The starting component of the application when it runs.

Additional Notes on Advanced Inputs

- **Use of Binders, Aliases, and Conditionals**:
  - We see standard inputs (`product`, `highlight`) used in `ProductItemComponent`.
  - We see an aliased input (`[aliasBadgeType]`) that can be used in a parent's template to set the property known internally as `badgeType`.
  - We see a condition `isPremiumProduct(product)` used in `ProductListComponent` to decide whether to pass highlight as true/false to `ProductItemComponent`.
- `Getters/Setters in ProductBadgeComponent`:
  - Illustrate how you can handle complex or user-supplied props gracefully. If no label is provided, the component uses "Default"; if no type is provided, it reverts to "neutral."
- **OnPush Change Detection**:
  - Applied to `ProductItemComponent`. This optimizes performance by telling Angular to check for changes only when bound input properties change or an event is fired. In large applications, this can reduce unnecessary checking.

Conclusion

This enhanced e-commerce example underscores several key Angular component input concepts: - Multiple `@Input()` properties per component. - Optional inputs with fallback logic (using getters and setters). - Input property aliasing (`badgeType as aliasBadgeType`). - Conditional input usage in the calling component (highlighting premium products). - Lifecycle hooks (`ngOnChanges`) to detect changes in input values.

By incorporating all of these techniques, we increase both the elegance and flexibility of our Angular application. Parents can pass data or flags selectively. Children can respond to changes, validate or normalize inputs, and communicate back up via events. Services (like `ProductService`) tie the data and UI together with a clean architecture. This design pattern fosters scalability, modularity, and maintainability as an application grows.