1. Create a Book Inventory Service

File: `src/app/book-inventory/book-inventory.service.ts`

Steps:

- **a. Generate the Service:**
  You can generate a service using Angular CLI (if you wish). This creates a service file for you.:

  ```
  ng generate service book-inventory
  ```

- **b. Define the Service:**
  In the service file, create a property to hold the inventory (or fetch it from an API in a real-world scenario) and methods for retrieving the inventory and deleting a book.

Example Code:

```
// src/app/book-inventory.service.ts
import { Injectable } from '@angular/core';
import { Book } from '../book';

@Injectable({
  providedIn: 'root'
})
export class BookInventoryService {
  // Hard-coded inventory (in a real app, you might fetch
this data from an API)
  private inventory: Book[] = [
    {
      ISBN: "978-1492056205",
      title: "Angular Up & Running",
      author: "Shyam Seshadri",
      year: 2018,
      price: 39,
      featured: true,
      coverImages: ["/assets/angular-up-and-running.png"]
    },
    {
      ISBN: "978-1593279509",
      title: "Eloquent JavaScript, 3rd Edition",
```

```
      author: "Marijn Haverbeke",
      year: 2018,
      price: 29.99,
      featured: false,
      coverImages: ["/assets/eloquent-javascript.jpg"]
    },
    {
      ISBN: "978-1491904244",
      title: "You Don't Know JS Yet: Get Started",
      author: "Kyle Simpson",
      year: 2020,
      price: 34.99,
      featured: false,
      coverImages: ["/assets/ydkjs-cover.jpg"]
    },
    {
      ISBN: "978-1449331818",
      title: "Learning JavaScript Design Patterns",
      author: "Addy Osmani",
      year: 2012,
      price: 25.99,
      featured: true,
      coverImages: ["/assets/js-design-patterns.png"]
    }
  ];

  // Return the current inventory
  getInventory(): Book[] {
    return [...this.inventory]; // return a copy
  }

  // Delete a book by ISBN
  deleteBook(book: Book): void {
    this.inventory = this.inventory.filter(b => b.ISBN !==
book.ISBN);
  }
}
```

Explanation:

- We mark the service as `@Injectable({ providedIn: 'root' })` so it's a singleton provided at the root level.

- The service holds a private inventory array.
- The `getInventory()` method returns a copy of the inventory.
- The `deleteBook(book: Book)` method updates the inventory by filtering out the specified book.

2. Update the Book Inventory Component to Use the Service

File: `src/app/book-inventory/book-inventory.component.ts`

Steps:

- **a. Import and Inject the Service:**
  Import your service and add it to the component's constructor.
- **b. Retrieve the Inventory:**
  In `ngOnInit()`, call the service's `getInventory()` method to set your local `inventory` property.
- **c. Update the Delete Method:**
  Instead of modifying the inventory directly, call the service's `deleteBook()` method and then refresh your local inventory (if necessary).

Example Code:

```
// src/app/book-inventory/book-inventory.component.ts
import { Component, OnInit } from '@angular/core';
import { Book } from '../book';
import { CommonModule } from '@angular/common';
import { FormsModule } from '@angular/forms';
import { HoverHighlightDirective } from './hover-
highlight.directive';
import { BookFilterPipe } from '../book-filter.pipe';
import { BookInventoryService } from './book-
inventory.service';

@Component({
  selector: 'app-book-inventory',
  standalone: true,
  imports: [CommonModule, HoverHighlightDirective,
FormsModule, BookFilterPipe],
  templateUrl: './book-inventory.component.html',
  styleUrls: ['./book-inventory.component.css']
})
export class BookInventoryComponent implements OnInit {
```

```
  currentDate: Date = new Date();
  searchTerm: string = '';
  inventory: Book[] = [];

  constructor(private bookService: BookInventoryService) {}

  ngOnInit(): void {
    // Get the inventory from the service
    this.inventory = this.bookService.getInventory();
  }

  trackByISBN(index: number, book: Book): string {
    return book.ISBN;
  }

  deleteBook(book: Book): void {
    // Use the service to delete the book
    this.bookService.deleteBook(book);
    // Update local inventory after deletion
    this.inventory = this.bookService.getInventory();
  }
}
```

Explanation:

- We inject `BookInventoryService` in the constructor.
- On initialization, we retrieve the inventory from the service.
- When deleting a book, we delegate that responsibility to the service and then refresh the local inventory.

3. Update the Book Inventory Template

No changes are needed in the template for the service integration if the data-binding remains the same. Your template can still use `inventory | bookFilter:searchTerm` in the `*ngFor` loop:

```
<!-- src/app/book-inventory/book-inventory.component.html
-->
<div class="container my-4">
  <h1 class="mb-4 text-center">Current Inventory</h1>
```

```
  <!-- Search input for filtering books -->
  <div class="mb-3 text-center">
    <input type="text" [(ngModel)]="searchTerm"
placeholder="Search books" class="form-control w-50 mx-
auto" />
  </div>

  <p *ngIf="inventory.length === 0" class="alert alert-info
text-center">
    There are no books in inventory.
  </p>

  <div class="row">
    <div class="col-12 col-sm-6 col-md-4 col-lg-3 mb-4"
         *ngFor="let book of (inventory |
bookFilter:searchTerm); trackBy: trackByISBN">
      <div class="card h-100 shadow-sm"
appHoverHighlight="#e1e1e1">
        <div class="ratio ratio-4x3">
          <img *ngIf="book.coverImages.length > 0"
               [src]="book.coverImages[0]"
               class="card-img-top"
               alt="{{ book.title }} cover"
               style="object-fit: cover;">
        </div>
        <div class="card-body d-flex flex-column">
          <h5 class="card-title">{{ book.title }}</h5>
          <p class="card-text flex-grow-1">
            <strong>Author:</strong> {{ book.author }}<br>
            <strong>Year:</strong> {{ book.year }}<br>
            <strong>ISBN:</strong> {{ book.ISBN }}<br>
            <strong>Price:</strong> {{ book.price |
currency:'USD':'symbol' }}
          </p>
          <p *ngIf="book.featured" class="badge bg-primary
mb-2">Featured Book</p>
          <div [ngSwitch]="book.price > 30 ? 'expensive' :
'affordable'" class="mb-2">
            <p *ngSwitchCase="'expensive'" class="text-
danger mb-0">Premium selection!</p>
```

```
            <p *ngSwitchCase="'affordable'" class="text-
success mb-0">Budget-friendly!</p>
            <p *ngSwitchDefault class="text-muted
mb-0">Great value!</p>
          </div>
          <button (click)="deleteBook(book)" class="btn
btn-danger mt-auto">Delete</button>
        </div>
      </div>
    </div>
  </div>
</div>
```

Summary of Steps

1. **Create a Service:**
    - Create `BookInventoryService` with methods to get and delete books.
    - Mark the service as injectable at the root level.
2. **Update Component (TS):**
    - Inject the service into `BookInventoryComponent`.
    - On initialization, retrieve the inventory from the service.
    - Delegate deletion logic to the service and refresh the local inventory.
3. **Template:**
    - No changes needed in the template; it continues to use the `inventory` property and custom filter pipe as before.

By offloading data management to a service, you improve separation of concerns, making your component easier to maintain and test.

Dependency Injection (DI) is a design pattern used in Angular (and many other frameworks) to supply components or services with their required dependencies rather than having them create those dependencies themselves. Here's a detailed explanation:

1. What is Dependency Injection?

- **Concept:**
  DI is a pattern where a class receives its dependencies (services, objects, or values) from an external source rather than creating them itself. This helps to decouple components from their dependencies.

- **Angular Implementation:**
  Angular has a built-in dependency injection system that automatically provides instances of services to components, directives, pipes, or even other services. When you declare a dependency in a constructor, Angular's DI system "injects" the required instance.

```
// Example:
constructor(private bookService: BookInventoryService) { }

//In this example, Angular will automatically provide an instance of BookInventoryService when it creates the component.
```

2. Benefits of Dependency Injection

- **Decoupling:**
  Components don't need to know how to create or configure their dependencies; they just declare what they need.

- **Testability:**
  It's easier to mock dependencies in unit tests since you can replace real services with mocks or stubs.

- **Reusability:**
  Services and components can be reused in different parts of an application without worrying about how dependencies are constructed.

- **Maintainability:**
  By managing dependencies centrally (e.g., via providers), you have a single place to configure or update them.

3. How Angular's DI Works

- **Providers:**
  You register classes (services) with the Angular injector by declaring them as providers. For example, using `@Injectable({ providedIn: 'root' })` makes a service available application-wide as a singleton.

- **Injection Tokens:**
  Angular uses tokens to identify dependencies. Tokens can be class types, strings, or custom objects. When you declare a dependency, Angular matches the type or token to a provider.

- **Constructor Injection:**
  Dependencies are typically injected via constructor parameters. Angular inspects the types of constructor parameters and supplies the appropriate instances.

4. Alternatives to Dependency Injection

While DI is the recommended pattern in Angular, here are some alternatives and why they might be less favorable:

- **Manual Instantiation:**

  - **Description:**
    A component could create its own dependencies using the `new` keyword.

  Example:

```
export class SomeComponent {
  private bookService = new BookInventoryService();
}
```

  - **Drawbacks:**
    - Tight coupling: The component is directly responsible for constructing the service.
    - Difficult to test: It's harder to replace the dependency with a mock.
    - No centralized configuration: Changing how the service is created requires updating each component.

- **Service Locator Pattern:**

  - **Description:**
    Instead of injecting dependencies, a component could request a service from a global service locator.
  - **Example:**

```
const bookService =
ServiceLocator.get(BookInventoryService);
```

  - **Drawbacks:**
    - Hidden dependencies: It's not clear from the component's constructor what services it depends on.

        - Harder to test: It requires configuring the service locator before tests.
        - Reduced transparency: DI makes dependencies explicit; a service locator obscures that relationship.

- **Factory Functions (like FormBuilder):**
    - **Description:**
      Instead of direct instantiation, you could use factory functions to create instances. Angular supports factories as providers.
    - **Usage:**
      This approach is still part of DI, but you can control instantiation more precisely.
    - **Drawbacks:**
        - More configuration: It's more verbose than simple DI.

5. Conclusion

Dependency Injection in Angular is a powerful and preferred pattern because it:

- **Decouples** components from their dependencies.
- **Simplifies testing** by making it easy to inject mocks.
- **Centralizes configuration** for services.
- **Improves maintainability** and readability by making dependencies explicit.