Angular: Sharing Data Between Components with Services

Objective

*The goal of this hands-on lab is to build a simple Angular dashboard page composed of multiple components: a **Header**, a **Sidebar**, and a main content area called **Dashboard**. Through this exercise, you will learn how to share data and state across these independent components by utilizing a common Angular service.*

In Angular, services play a vital role by acting as centralized data stores and logic providers that can be injected into components via Angular's powerful dependency injection (DI) system. This DI mechanism allows components to request the same service instance, enabling a shared source of truth for data that multiple components can access and update.

By managing shared state through services—often using reactive patterns such as RxJS BehaviorSubjects—you establish a unidirectional data flow. Components update state in the service, and other subscribed components reactively respond to changes. This architecture streamlines communication between components without tightly coupling them, which improves the maintainability and scalability of your application.

This lab will give you hands-on experience with:

- *Creating Angular components representing different parts of a dashboard UI*
- *Building and injecting a shared service to manage cross-component data*
- *Leveraging RxJS observables to broadcast and react to data changes in real time*
- *Understanding how dependency injection supports modular, testable Angular apps*

By the end of this tutorial, you will have a working dashboard app illustrating best practices for state sharing and component communication in Angular projects, boosting your confidence in building more sophisticated interfaces.

Prerequisites

Before starting this lab, ensure you have the following set up and ready:

- **Node.js and npm:** Install the latest LTS version of Node.js, which includes npm, from nodejs.org. Angular CLI requires a compatible Node.js version (preferably 14.x or higher) to work smoothly.
- **Angular CLI:** Install globally by running `npm install -g @angular/cli`. This tool helps generate projects, components, and services efficiently.

- **Angular Project:** Either have an existing Angular project or create a new one using `ng new dashboard-app`. Use defaults or customize features such as routing and stylesheet formats during setup.
- **Basic Angular Knowledge:** Familiarity with Angular components, modules, templates, and the Angular project structure is essential.
- **Code Editor and Terminal:** Use a modern code editor like Visual Studio Code with integrated terminal access to run Angular CLI commands and serve the application.

Project Setup

Open the Integrated Terminal

1. Launch VS Code

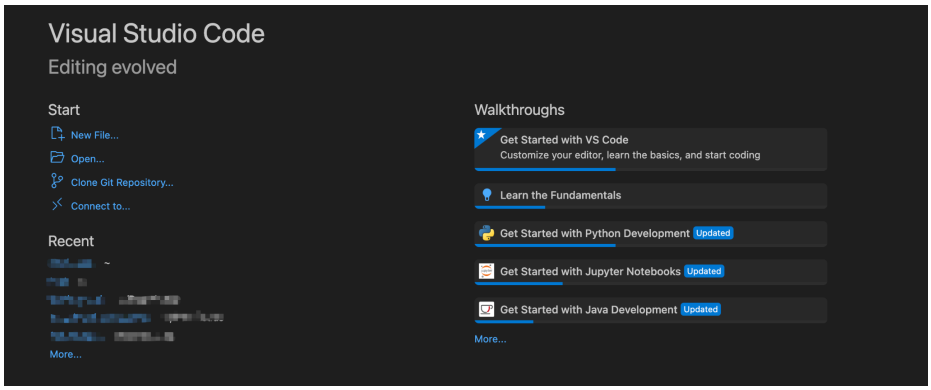

***Figure 1**: VS Code Default Window*

2. Open your project folder or any

    a. Click on “Open”.

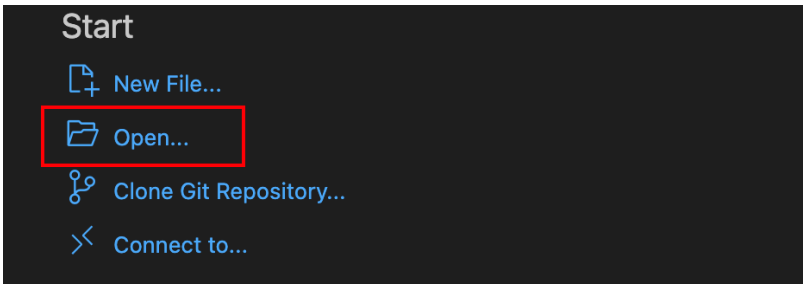

***Figure 2**: Opening the project directory in VS Code*

    b. Select the folder and click on “Open”.

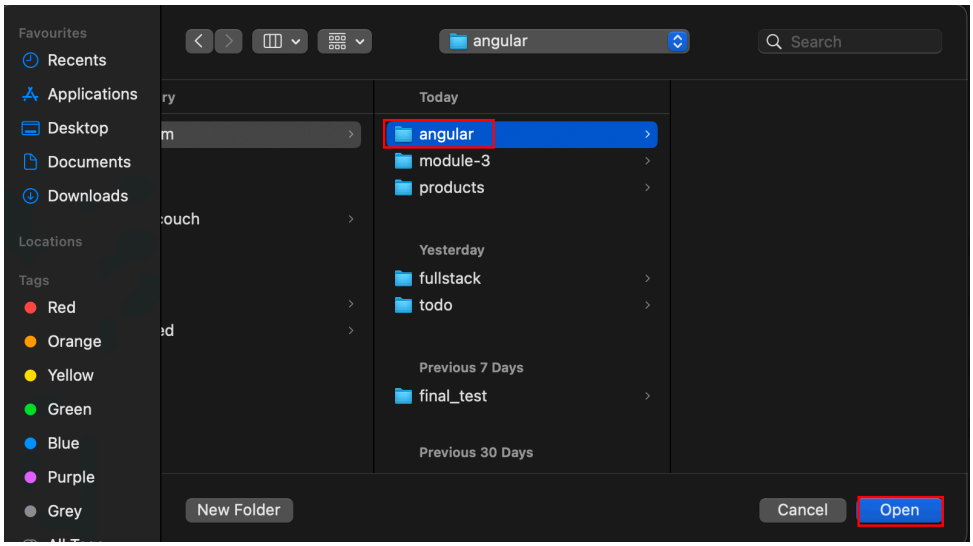

***Figure 3**: Selecting the project directory*

3. Open the terminal with Ctrl+` (backtick)

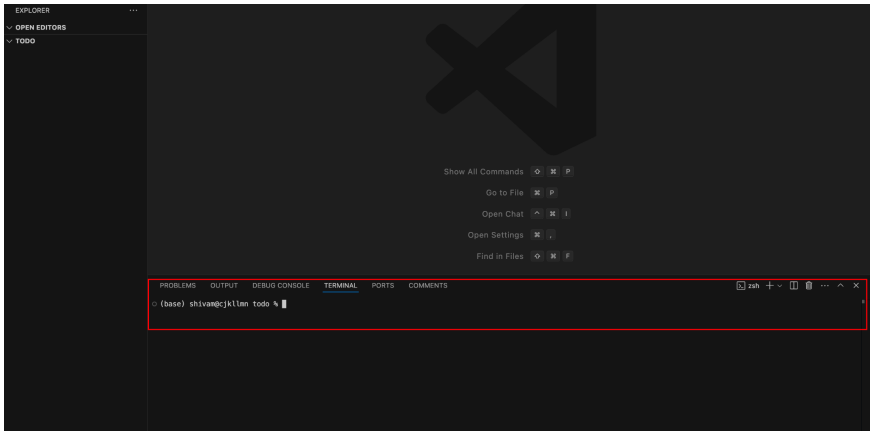

***Figure 4**: VS Code's Terminal*

In this section, you will set up the Angular project environment for this lab. If you haven't created an Angular project yet, start by running the following command in your VS Code's terminal to generate a new Angular workspace and app with default settings:

```
npm install -g @angular/cli
```

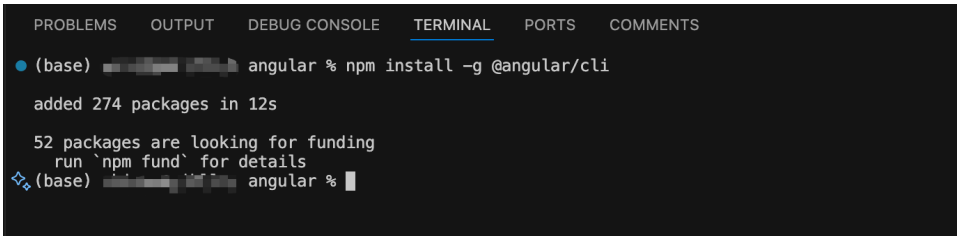

***Figure 5**: Angular Installed*

```
ng new dashboard-app
```

Keep on clicking “**Enter**” to continue with the default settings till the installation starts.

This command will:

- Create a new folder `dashboard-app` with all necessary Angular project files.
- Configure routing (if you choose) and set up a default stylesheet format.
- Install all dependencies automatically via npm.

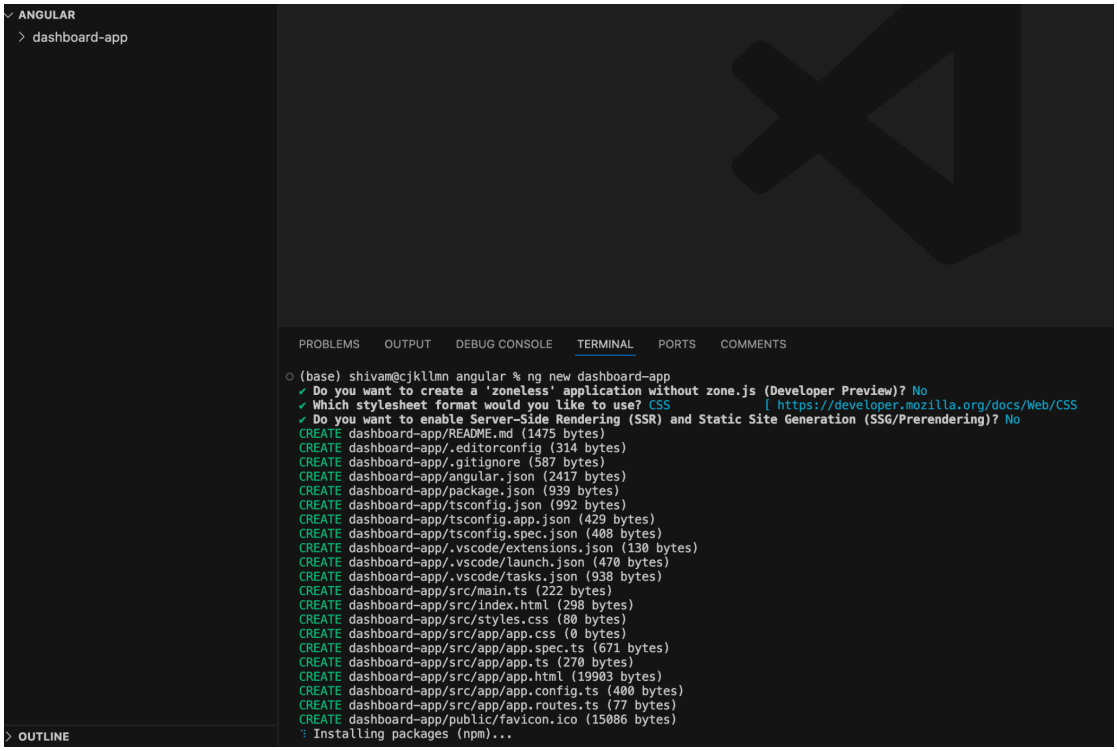

***Figure 6**: Project Basic setup done*

Once your project is created, navigate into the project folder:

```
cd dashboard-app
```

Next, generate the three components required for the dashboard layout: **HeaderComponent**, **SidebarComponent**, and **DashboardComponent**. Use the Angular CLI commands below to create these components inside the `src/app/` folder:

```
ng generate component header
ng generate component sidebar
ng generate component dashboard
```

***Figure 7**: Directories Generated Successfully*

Each command generates a directory along with the following files:

- `header.ts`, `header.html`, `header.component.css`, and `header.component.spec.ts` under `src/app/header/`
- `sidebar.ts`, `sidebar.html`, etc., under `src/app/sidebar/`
- `dashboard.ts`, `dashboard.html`, etc., under `src/app/dashboard/`

Following the components, create a shared service that will manage and share state among these components. Run:

```
ng generate service shared-data
```

The CLI will create two files: `shared-data.ts` and `shared-data.service.spec.ts`, located in `src/app/`. This service will be the centralized point to hold and broadcast shared data.

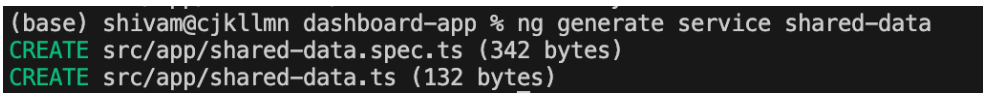

***Figure 8**: Files Generated Successfully*

After generating these files, verify your project structure looks like this (partial view):

- `src/app/header/header.ts`

- `src/app/sidebar/sidebar.ts`
- `src/app/dashboard/dashboard.ts`
- `src/app/shared-data.ts`

Successful generation of components and services will be confirmed by CLI messages like:

```
CREATE src/app/header/header.ts (xxx bytes)
CREATE src/app/header/header.html (xxx bytes)
...
CREATE src/app/shared-data.ts (xxx bytes)
```

With this setup complete, you are ready to begin building the dashboard layout and implementing shared data flow.

Build the Layout

Now that the components have been generated, the next step is to create a simple responsive layout that arranges the **Header**, **Sidebar**, and **Dashboard** components inside the main app view. We will modify the primary layout file `src/app/app.html` to embed the three components using their selectors, and add CSS styles in `src/app/app.css` to organize them visually.

AppComponent Template Setup

Open the `src/app/app.html` file and **replace** its contents with the following code. This lays out the page with the header spanning full width across the top, a vertical sidebar on the left, and the dashboard occupying the remaining main area:

```
<app-header></app-header>
<div class="container">
  <app-sidebar class="sidebar"></app-sidebar>
  <app-dashboard class="dashboard"></app-dashboard>
</div>
```

Here, `<app-header>` displays the header, followed by a `div.container` that serves as a flex container holding the sidebar and dashboard components side by side.

CSS Styling for Layout

Next, open `src/app/app.css` and add the following styles to arrange the layout using CSS Flexbox. These styles create a vertical stacking for the header and horizontal side-by-side layout for the sidebar and dashboard:

```css
/* Make the entire app take full viewport height */
:host {
  display: flex;
  flex-direction: column;
  height: 100vh;
  margin: 0;
}

/* Header occupies fixed height at the top, full width */
app-header {
  flex: 0 0 60px;
  background-color: #3f51b5;
  color: white;
  display: flex;
  align-items: center;
  padding: 0 1rem;
  font-size: 1.25rem;
  font-weight: 500;
  box-shadow: 0 2px 4px rgba(0, 0, 0, 0.1);
}

/* Container below header holds sidebar and dashboard as flex row */
.container {
  flex: 1;
  display: flex;
  height: calc(100vh - 60px);
  background: #f5f5f5;
}

/* Sidebar styling: fixed width, full height */
.sidebar {
  flex: 0 0 250px;
  background-color: #e8eaf6;
  border-right: 1px solid #c5cae9;
  padding: 1rem;
  box-sizing: border-box;
  overflow-y: auto;
}

/* Dashboard styling: takes remaining width */
.dashboard {
  flex: 1;
  padding: 1rem;
```

```
  overflow-y: auto;
  background-color: white;
}
```

This CSS ensures:

- The app fills the entire browser viewport height.
- The header is fixed in height at the top and spans full width.
- The main content area below the header is a flex container with sidebar and dashboard side-by-side.
- The sidebar has a fixed width of 250px with a subtle background and border.
- The dashboard area fills the remaining horizontal space.
- Both sidebar and dashboard are scrollable if their content exceeds available height.

After adding these files, save your changes and run `ng serve` if your app is not already running. You should now see the header on top, the sidebar on the left, and a blank dashboard area on the right arranged clearly in the browser window.

This clean layout sets the stage for injecting shared data and dynamic behavior into each of these components in upcoming steps.

Create SharedDataService

To enable sharing state and communication between components in your Angular dashboard, we'll create a service named `SharedDataService`. This service acts as a centralized data store that multiple components can both read from and update.

The core of this service will leverage RxJS's `BehaviorSubject`. Unlike a regular `Subject`, a `BehaviorSubject` stores the current value and immediately emits this latest value to any new subscribers. This ensures that whenever a component subscribes to the shared data, it instantly receives the most recent state without waiting for a new emission.

Follow these steps to build the service in the file `src/app/shared-data.ts`:

Service Implementation

```
import { Injectable } from '@angular/core';
import { BehaviorSubject, Observable } from 'rxjs';

@Injectable({
 providedIn: 'root' // Service is provided application-wide
```

```
})
export class SharedDataService {
 // Private BehaviorSubject holding the current message state
 private currentMessage = new BehaviorSubject<string>('Welcome to the
Dashboard');

 constructor() { }

 /**
  * Returns the current message as an Observable.
  * Components can subscribe to this Observable to react to changes.
  */
 getMessage(): Observable<string> {
   return this.currentMessage.asObservable();
 }

 /**
  * Updates the message state.
  * This triggers all subscribers to receive the new message immediately.
  * @param newMessage The new message string to broadcast
  */
 updateMessage(newMessage: string): void {
   this.currentMessage.next(newMessage);
 }
}
```

Explanation of key parts:

- `private currentMessage = new BehaviorSubject<string>('Welcome to the Dashboard');`
  Initializes the BehaviorSubject with a default message. This is the shared piece of state visible to all components subscribing to it.
- `getMessage(): Observable<string>`
  Exposes the BehaviorSubject as an `Observable` for components to subscribe to without directly accessing the subject's internal state. This encapsulation forces all updates to go through defined methods, following good design principles.
- `updateMessage(newMessage: string): void`
  Updates the current message by pushing the new value to the `BehaviorSubject` using `next()`. This causes all subscribed components to receive the updated data in real-time.

By using this method, any component injecting `SharedDataService` can easily subscribe to `getMessage()` to get live updates and call `updateMessage()` to broadcast changes. The service maintains a single source of truth, enabling unidirectional data flow.

Inject and Use the Service in Components

In this section, you will learn how to inject the `SharedDataService` into your **HeaderComponent** and **SidebarComponent** and use it to share data between them. The *SidebarComponent* will update the shared message based on user input, while the *HeaderComponent* will subscribe to changes and display the current message live.

Injecting and Using SharedDataService in SidebarComponent

The `SidebarComponent` will provide a simple text input and a button to update the shared message via the service. Follow these steps to implement this in `src/app/sidebar/sidebar.ts` and `src/app/sidebar/sidebar.html`.

sidebar.ts

```
import { Component } from '@angular/core';
import { FormsModule } from '@angular/forms';
import { SharedDataService } from  '../shared-data';

@Component({
 selector: 'app-sidebar',
 standalone: true,
 imports: [FormsModule],
 templateUrl: './sidebar.html',
 styleUrls: ['./sidebar.css']
})
export class SidebarComponent {
 newMessage: string = '';

 constructor(private sharedDataService: SharedDataService) {}

 updateSharedMessage(): void {
   if (this.newMessage.trim()) {
     this.sharedDataService.updateMessage(this.newMessage.trim());
     this.newMessage = '';
   }
 }
}
```

sidebar.html

```
<div>
  <h3>Sidebar</h3>
  <label for="messageInput">Enter new dashboard message:</label><br>
  <input
    id="messageInput"
    type="text"
    [(ngModel)]="newMessage"
    placeholder="Type a message"
    aria-label="New message input"
  />
  <button (click)="updateSharedMessage()">Update Message</button>
</div>
```

Notes:

- The `updateSharedMessage()` method calls the service's `updateMessage()` method to broadcast the new message.

Subscribing to Shared Data in HeaderComponent

The `HeaderComponent` will listen to changes from the `SharedDataService` and display the current message live. This involves subscribing to the message observable in `ngOnInit` and updating the template accordingly.

header.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';
import { SharedDataService } from '../shared-data';

@Component({
 selector: 'app-header',
 templateUrl: './header.html',
 styleUrls: ['./header.css']
})
export class HeaderComponent implements OnInit, OnDestroy {
 // Holds the current message from the shared service
 currentMessage: string = '';

 // Store the subscription so we can unsubscribe to prevent memory leaks
 private messageSubscription!: Subscription;
```

```
 // Inject the shared data service to access the observable
 constructor(private sharedDataService: SharedDataService) { }

 /**
  * Subscribe to the shared message observable when the component
initializes.
  * Every emission updates the currentMessage displayed in the header.
  */
 ngOnInit(): void {
   this.messageSubscription =
this.sharedDataService.getMessage().subscribe(message => {
     this.currentMessage = message;
   });
 }

 /**
  * Unsubscribe from the observable when the component is destroyed
  * to avoid memory leaks, following Angular best practices.
  */
 ngOnDestroy(): void {
   if (this.messageSubscription) {
     this.messageSubscription.unsubscribe();
   }
 }
}
```

header.html

```
<header>
  <h1>Dashboard Header</h1>
  <p>Current Message: <strong>{{ currentMessage }}</strong></p>
</header>
```

Important Concepts Illustrated:

- **Dependency Injection:** The `SharedDataService` is injected into each component's constructor, giving access to the shared state and methods.
- **Subscription Management:** Subscribing to the `getMessage()` observable inside `ngOnInit` enables the component to react to live data changes.

- **Unsubscribe on Destroy:** To prevent memory leaks, you unsubscribe inside `ngOnDestroy`, which is a recommended best practice when subscribing to observables directly.
- **Reactive and Unidirectional Data Flow:** The `SidebarComponent` pushes data to the service; the `HeaderComponent` subscribes and reflects updates in real-time.

Enhance the DashboardComponent

Now that the `HeaderComponent` and `SidebarComponent` are interacting through the shared `SharedDataService`, let's extend this data sharing functionality to the `DashboardComponent`. This will illustrate how all three components stay synchronized dynamically through the service.

The `DashboardComponent` will subscribe to the shared message observable just like the header, allowing it to display the current message and react to changes in real time. We'll also demonstrate simple conditional styling that updates based on the message content, providing a visual cue that this component responds to the shared state.

Injecting and Subscribing to SharedDataService

Open the file `src/app/dashboard/dashboard.ts` and modify it as follows:

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { Subscription } from 'rxjs';
import { SharedDataService } from '../shared-data'
import { CommonModule } from '@angular/common';

@Component({
 selector: 'app-dashboard',
 standalone: true,
 imports: [CommonModule],
 templateUrl: './dashboard.html',
 styleUrls: ['./dashboard.css']
})
export class DashboardComponent implements OnInit, OnDestroy {
 currentMessage: string = '';
 private messageSubscription!: Subscription;

 constructor(private sharedDataService: SharedDataService) {}

 ngOnInit(): void {
   this.messageSubscription =
this.sharedDataService.getMessage().subscribe(message => {
```

```
      this.currentMessage = message;
    });
  }

  ngOnDestroy(): void {
    if (this.messageSubscription) {
      this.messageSubscription.unsubscribe();
    }
  }
}
```

Dynamic Template to Display Message

Next, open the `src/app/dashboard/dashboard.html` file and update it to bind and display the shared message dynamically. We'll also add conditional CSS classes to change the background color based on specific keywords in the message as a simple demonstration of reactive UI behavior.

```
<div class="dashboard-panel" [ngClass]="{
  'alert-warning': currentMessage.toLowerCase().includes('alert'),
  'welcome-message': currentMessage.toLowerCase().includes('welcome')
}">
  <h2>Dashboard</h2>
  <p>Latest Message:</p>
  <blockquote>{{ currentMessage }}</blockquote>

  <!-- Placeholder for future dynamic panels or charts -->
  <div class="data-panel">
    <p>This panel can update dynamically based on the shared message.</p>
  </div>
</div>
```

Supporting CSS for Visual Feedback

Optionally, define CSS styles in `src/app/dashboard/dashboard.css` to visually differentiate the dashboard area depending on the message state:

```
.dashboard-panel {
```

```
  border: 1px solid #ccc;
  padding: 1rem;
  border-radius: 5px;
  background-color: #fafafa;
  transition: background-color 0.3s ease;
}

.alert-warning {
  background-color: #fff3cd; /* light yellow */
  border-color: #ffeeba;
}

.welcome-message {
  background-color: #d1e7dd; /* light green */
  border-color: #badbcc;
}

.data-panel {
  margin-top: 1rem;
  padding: 1rem;
  background: #e9ecef;
  border-radius: 4px;
  font-style: italic;
  color: #495057;
}
```

Update `app.ts`

This step is often missed and **critical**. Open `src/app/app.ts` and update as follows:

```
import { Component } from '@angular/core';
import { HeaderComponent } from './header/header';
import { SidebarComponent } from './sidebar/sidebar';
import { DashboardComponent } from './dashboard/dashboard';

@Component({
 selector: 'app-root',
 standalone: true,
 imports: [HeaderComponent, SidebarComponent, DashboardComponent],
 templateUrl: './app.html',
 styleUrls: ['./app.css']
})
```

```
export class AppComponent {
 title = 'dashboard-app';
}
```

Update `main.ts`

```
import { bootstrapApplication } from '@angular/platform-browser';
import { appConfig } from './app/app.config';
import { AppComponent } from './app/app';

bootstrapApplication(AppComponent, appConfig)
 .catch((err) => console.error(err));
```

These styles visually signal message types such as alerts or welcomes, enhancing UX by tying the UI appearance to shared data state.

This example shows how the `DashboardComponent` reacts seamlessly to changes made in other components (like the SidebarComponent updating the message), by subscribing to the centralized `SharedDataService`. Such synchronized state sharing across multiple components enables building rich, reactive dashboards without tightly coupling component logic.

In future enhancements, you can extend this pattern with more complex data models, or integrate data visualization libraries like *ngx-charts* or *Chart.js* to represent dashboard data dynamically as shared state updates.

Testing the Data Flow

After implementing the shared service and subscribing to its data in your components, it is essential to verify that the data flow and component communication work as expected in your Angular dashboard.

Running the Angular Application

From your project root directory (where `angular.json` is located), run the following command in your terminal or integrated editor terminal to start a development server and serve the application:

```
ng serve
```

This command compiles the application and launches a local server, typically accessible at http://localhost:4200 in your web browser. It also watches your files for changes and reloads automatically.

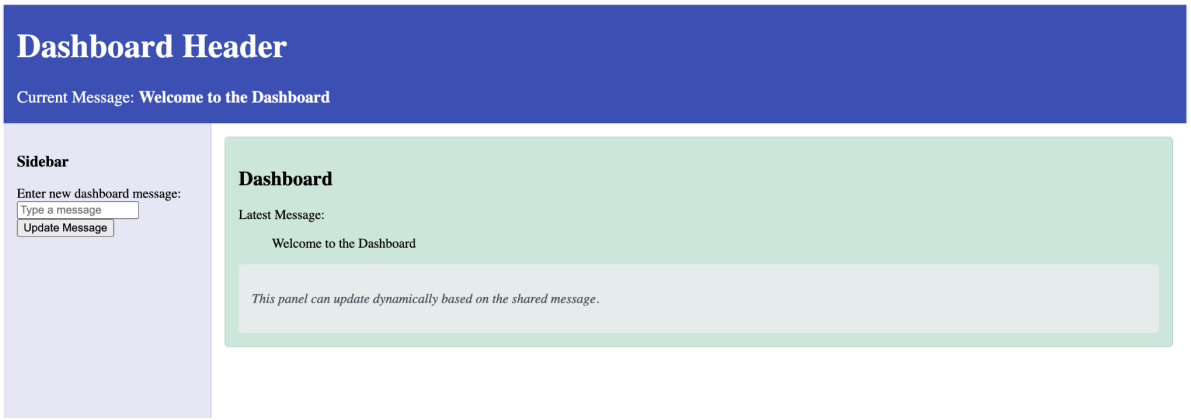

***Figure 9**: Final Dashboard UI*

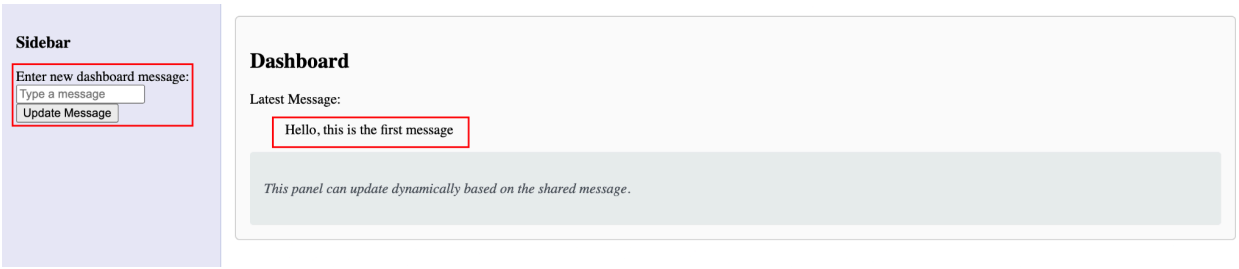

***Figure 10**: Latest message visibility through Update Message Button*

Observing Component Interaction

Once the app is running in the browser, you should see the dashboard layout with the **Header** across the top, the **Sidebar** on the left with an input box and button, and the **Dashboard** area displaying the shared message.

To test the data flow:

- Type a new message into the input field inside the *Sidebar* component.
- Click the `Update Message` button.
- Immediately, observe that the *Header* updates its displayed message to match the input.
- The *Dashboard* component should also reactively update its displayed message and styling if implemented.

This real-time update demonstrates successful unidirectional data flow through the `SharedDataService` and Angular's dependency injection system, confirming that components are synchronized via the shared BehaviorSubject.

Using Developer Tools for Debugging

Open your browser's developer console (usually `F12` or `Ctrl+Shift+I`) to:

- Check for any errors or warnings that might indicate issues with service injection or subscriptions.
- Use Angular devtools or console logs to inspect component states or verify that subscriptions are active.

This inspection helps confirm that components are correctly subscribing to the service's observable and that no unexpected lifecycle or binding issues occur.

Troubleshooting Tips

- If updates in the Sidebar do not reflect in Header or Dashboard, verify that `SharedDataService` is properly injected in all components.
- Ensure subscriptions to `getMessage()` are set up in `ngOnInit`, and subscriptions are not prematurely unsubscribed.
- Try stopping and restarting `ng serve` to rebuild the application in case of any build or caching issues.
- Verify your selectors in the `app.html` are correct (e.g., <app-header>, <app-sidebar>, <app-dashboard>).

Conclusion

In this hands-on lab, you built a modular Angular dashboard with Header, Sidebar, and Dashboard components that communicated through a shared Angular service. The core achievement was implementing `SharedDataService` using RxJS `BehaviorSubject` to maintain a centralized, reactive state.

This approach enabled unidirectional data flow, where components updated the service and subscribed to react to changes—promoting clean, scalable architecture beyond traditional Input/Output bindings. Using Angular's dependency injection, you decoupled components while ensuring consistent state management.

To build on this foundation, you can:

- Introducing NgRx for structured, scalable global state management.
- Extend services to manage more complex data structures.
- Implement lazy-loaded modules for better performance.

Mastering service-based communication prepares you to build maintainable, reactive Angular applications with loosely coupled components.