Week 12: Angular Routing, Component Interaction & State Management

FH University of Applied Sciences
TECHNIKUM WIEN

Week 12: Angular Routing, Component Interaction & State Management

In this session, we will:

- Learn how to set up **Routing in Angular**.
- Implement **Component Communication**.
- Introduce **State Management** for user authentication.
- Extend the **Messenger Frontend** to manage user sessions.

1. Angular Routing & Navigation

Why Routing?

- Enables switching between multiple views without full page reloads.
- Organizes components logically (e.g., Login vs. Chat views).
- Improves user experience by allowing deep linking.

Setting Up Routing in Angular

1. Define Routes in `app-routing.module.ts`

```
import { NgModule } from '@angular/core';
import { RouterModule, Routes } from '@angular/router';
import { LoginComponent } from './login/login.component';
import { ChatComponent } from './chat/chat.component';

const routes: Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'chat', component: ChatComponent },
  { path: '', redirectTo: '/login', pathMatch: 'full' }
];

@NgModule({
  imports: [RouterModule.forRoot(routes)],
  exports: [RouterModule]
})
export class AppRoutingModule {}
```

2. Navigation Between Routes

- Using `routerLink` in a template:

```
<a routerLink="/chat">Go to Chat</a>
```

- Programmatically navigating using Angular Router:

```
constructor(private router: Router) {}
this.router.navigate(['/chat']);
```

2. Component Communication

Why Component Interaction is Needed?

- Allows passing data from **parent to child** (`@Input()`).
- Enables event handling from **child to parent** (`@Output()`).
- Helps manage state across components using **Services**.

1. Parent-to-Child Communication (`@Input()`)

```
@Component({ selector: 'app-child', template: `<p>{{message}}</p>` })
export class ChildComponent {
  @Input() message!: string;
}
```

```
<app-child [message]="parentMessage"></app-child>
```

2. Child-to-Parent Communication (`@Output()`)

```
@Component({ selector: 'app-child', template: `<button (click)="sendMessage()">Send</button>` })
export class ChildComponent {
  @Output() messageEvent = new EventEmitter<string>();
  sendMessage() { this.messageEvent.emit('Hello Parent!'); }
}
```

```
<app-child (messageEvent)="receiveMessage($event)"></app-child>
```

3. State Management Basics

Why State Management?

- Stores global user authentication state.
- Prevents data loss when switching routes.
- Avoids prop-drilling (passing data multiple levels deep).

1. Using a Service to Manage User State

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

@Injectable({ providedIn: 'root' })
export class AuthService {
  private userSubject = new BehaviorSubject<string | null>(localStorage.getItem('username') || null);
  user$ = this.userSubject.asObservable();

  setUser(username: string) {
    localStorage.setItem('username', username);
    this.userSubject.next(username);
  }
}
```

2. Using the Service in a Component

```
constructor(private authService: AuthService) {}
ngOnInit() {
  this.authService.user$.subscribe(user => this.username = user);
}
```

4. Hands-On Practice: Implementing Routing & State

Task 1: Implement Routing in Messenger App

- **Generate Components** for Login & Chat:

```
ng generate component login
ng generate component chat
```

- **Modify `AppRoutingModule` to include both views.**
- **Ensure LoginComponent has a form to accept a username and store it globally.**

Task 2: Implement Component Communication

- **Pass username from `LoginComponent` to `ChatComponent`.**
- **Use `AuthService` with BehaviorSubject to store the username globally.**
- **Modify Chat UI to show the logged-in user.**

Task 3: Improve State Management

- **Store username in localStorage for persistence.**
- **Modify AuthService to check stored username on reload.**

5. Sample Solution Overview

1. `app-routing.module.ts` (Routing Setup)

```
const routes: Routes = [
  { path: 'login', component: LoginComponent },
  { path: 'chat', component: ChatComponent },
  { path: '', redirectTo: '/login', pathMatch: 'full' }
];
```

2. `auth.service.ts` (State Management)

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

@Injectable({ providedIn: 'root' })
export class AuthService {
  private userSubject = new BehaviorSubject<string | null>(localStorage.getItem('username') || null);
  user$ = this.userSubject.asObservable();

  setUser(username: string) {
    localStorage.setItem('username', username);
    this.userSubject.next(username);
  }
}
```

3. `login.component.ts` (User Login)

```ts
import { Component } from '@angular/core';
import { AuthService } from '../auth.service';
import { Router } from '@angular/router';

@Component({ selector: 'app-login', templateUrl: './login.component.html' })
export class LoginComponent {
  username = '';

  constructor(private authService: AuthService, private router: Router) {}

  login() {
    if (this.username.trim()) {
      this.authService.setUser(this.username);
      this.router.navigate(['/chat']);
    }
  }
}
```

4. `chat.component.ts` (Chat View)

```ts
import { Component } from '@angular/core';
import { AuthService } from '../auth.service';

@Component({ selector: 'app-chat', templateUrl: './chat.component.html' })
export class ChatComponent {
  username: string | null = '';

  constructor(private authService: AuthService) {
    this.authService.user$.subscribe(user => this.username = user);
  }
}
```

6. Summary & Next Steps

- **Week 12:** Angular Routing, Component Interaction & State Management.
- **Week 13:** Final UI/UX, Implement Real-Time Features (WebSockets), and Deployment.

Happy coding! 🎉