Week 11: Angular Services & RESTful Integration

FH University of Applied Sciences
TECHNIKUM WIEN

Week 11: Angular Services & RESTful Integration

In this session, we will:

- Learn about Angular Services & Dependency Injection.
- Use `HttpClient` to connect to a RESTful API.
- Fetch and send messages in the Messenger Frontend.
- Handle API responses asynchronously with Observables.

1. Why Use Angular Services?

The Role of Services

- **Encapsulation of Business Logic**: Keeps components focused on UI.
- **Reusability**: Services can be shared across multiple components.
- **Separation of Concerns**: API calls and state management are handled separately.

How Angular Services Work

- Services are **singleton instances** injected into components or other services.
- Registered using `providedIn: 'root'` in `@Injectable()`.
- Used with **Dependency Injection (DI)**.

2. Creating and Using Angular Services

Generate a Service

```
ng generate service message
```

MessageService Implementation

```
import { Injectable } from '@angular/core';
import { HttpClient } from '@angular/common/http';
import { Observable } from 'rxjs';

@Injectable({ providedIn: 'root' })
export class MessageService {
  private apiUrl = 'https://api.example.com/messages'; // Replace with actual backend URL

  constructor(private http: HttpClient) { }

  getMessages(): Observable<Message[]> {
    return this.http.get<Message[]>(this.apiUrl);
  }

  sendMessage(message: Message): Observable<any> {
    return this.http.post(this.apiUrl, message);
  }
}
```

Injecting the Service in a Component

```
constructor(private messageService: MessageService) { }
```

3. Making HTTP Requests with Angular

Setting Up HttpClientModule

Import `HttpClientModule` in `AppModule`:

```
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  imports: [HttpClientModule],
})
export class AppModule {}
```

Performing GET & POST Requests

- Fetching messages:

```
this.http.get<Message[]>(this.apiUrl)
```

- Sending a message:

```
this.http.post(this.apiUrl, message)
```

- Observables allow handling of asynchronous data.

4. Handling API Responses

Observables & Async Handling

- Use `subscribe()` to receive async data.
- Handle errors with `catchError()`.
- Use the `async` pipe for auto-subscribing in templates.

Example: Fetching Messages in a Component

```
messages: Message[] = [];
ngOnInit() {
  this.messageService.getMessages().subscribe(data => this.messages = data);
}
```

Displaying Messages in the Template

```
<div *ngFor="let msg of messages">
  <strong>{{ msg.sender }}</strong>: {{ msg.content }}
</div>
```

5. Hands-On Practice: Extending the Messenger App

Task 1: Set Up API Communication

- Import `HttpClientModule`.
- Use `HttpClient` to fetch messages (`GET /messages`).
- Display messages dynamically using `ngFor`.

Task 2: Implement a Send Message Feature

- Add an input field & button in `MessageListComponent`.
- Call `sendMessage()` on button click.
- Append message dynamically to the list.

6. Sample Solution Overview

message-list.component.ts

```
import { Component, OnInit } from '@angular/core';
import { MessageService } from '../message.service';

@Component({
  selector: 'app-message-list',
  templateUrl: './message-list.component.html',
  styleUrls: ['./message-list.component.css']
})
export class MessageListComponent implements OnInit {
  messages = [];
  newMessage = '';
  isLoading = true;
  errorMessage = '';

  constructor(private messageService: MessageService) {}

  ngOnInit() {
    this.messageService.getMessages().subscribe(
      data => { this.messages = data; this.isLoading = false; },
      error => { this.errorMessage = 'Failed to load messages'; this.isLoading = false; }
    );
  }

  sendMessage() {
    if (!this.newMessage.trim()) return;

    const message = { sender: 'User', content: this.newMessage };
    this.messageService.sendMessage(message).subscribe(
      () => this.messages.push(message),
      error => this.errorMessage = 'Failed to send message'
    );

    this.newMessage = '';
  }
}
```

7. Key Takeaways

- **Angular Services** help separate business logic from components.
- **HttpClientModule** enables API interaction.
- **Observables & Async Data Handling** allow efficient updates.
- **Error Handling & UX Enhancements** improve the application experience.

Next Steps

- **Week 12:** Add **Routing** (Login & Chat views) and **Real-Time Messaging**.
- **Week 13:** Finalize UI/UX, Best Practices, and Deployment.

Happy coding! 🎉