Independent Skill Development Mission

Independent Skill Development Mission NXT
EMPOWERING FUTURES, SHAPING TOMORROW

ISDM (INDEPENDENT SKILL DEVELOPMENT MISSION

REAL-TIME APPLICATIONS & UNIT TESTING (WEEKS 9-10)

USING WEBSOCKETS IN ANGULAR

CHAPTER 1: INTRODUCTION TO WEBSOCKETS

1.1 What are WebSockets?

WebSockets are a **real-time, full-duplex communication protocol** that allows a client (browser) and a server to exchange data **without requiring repeated HTTP requests**. Unlike traditional **request-response models**, WebSockets maintain a **persistent connection**, making them ideal for real-time applications.

✓ **Full-duplex communication** – Both client and server can send/receive data simultaneously.
✓ **Low latency** – Reduces the need for repeated polling, improving performance.
✓ **Efficient use of resources** – Reduces unnecessary network requests.

1.2 Why Use WebSockets in Angular?

WebSockets are widely used in **real-time applications,** such as:

✓ **Chat applications** – Sending and receiving messages instantly.
✓ **Live stock market updates** – Displaying financial data dynamically.
✓ **Collaborative tools** – Real-time document editing (e.g., Google Docs).
✓ **Gaming applications** – Synchronizing game state for multiple players.

CHAPTER 2: SETTING UP WEBSOCKETS IN ANGULAR

2.1 Installing Dependencies

Angular does not provide built-in WebSocket support, but we can use **RxJS Observables** and **WebSocket APIs** to implement it.

To simplify WebSocket handling, install ngx-socket-io:

```
npm install ngx-socket-io
```

✓ This package simplifies **WebSocket connections** using Angular services.

2.2 Setting Up a WebSocket Server (Node.js)

We need a WebSocket server to communicate with our Angular app. Let's create a **basic WebSocket server** using **Socket.io**.

Step 1: Install Required Packages

```
npm install express socket.io cors
```

Step 2: Create a WebSocket Server (server.js)

```
const express = require('express');
const http = require('http');
const socketIo = require('socket.io');
const cors = require('cors');

const app = express();
const server = http.createServer(app);
const io = socketIo(server, { cors: { origin: "*" } });

io.on('connection', (socket) => {
  console.log('New client connected:', socket.id);

  // Send a welcome message
  socket.emit('message', 'Welcome to WebSocket Server!');

  // Listen for messages from clients
  socket.on('chatMessage', (msg) => {
    console.log('Message received:', msg);
    io.emit('chatMessage', msg); // Broadcast message to all clients
  });
```

```
  // Handle client disconnect
  socket.on('disconnect', () => {
    console.log('Client disconnected');
  });
});

server.listen(3000, () => {
  console.log('WebSocket server running on port 3000');
});
```

✓ **Creates an Express server with WebSocket support**.
✓ **Handles real-time messaging between clients**.

CHAPTER 3: CREATING A WEBSOCKET SERVICE IN ANGULAR

3.1 Generate an Angular Service for WebSockets

Run the following command to generate a WebSocket service:

```
ng generate service websocket
```

3.2 Implement the WebSocket Service (websocket.service.ts)

Modify websocket.service.ts to connect to the WebSocket server:

```
import { Injectable } from '@angular/core';
import { Socket } from 'ngx-socket-io';
```

```
import { Observable } from 'rxjs';


@Injectable({
 providedIn: 'root'
})
export class WebsocketService {
 constructor(private socket: Socket) {}


 // Send a message to the WebSocket server
 sendMessage(msg: string) {
  this.socket.emit('chatMessage', msg);
 }


 // Listen for incoming messages
 getMessages(): Observable<string> {
  return this.socket.fromEvent<string>('chatMessage');
 }
}
```

✓ **sendMessage()** – Sends data to the WebSocket server.
✓ **getMessages()** – Listens for messages from the server.

3.3 Provide the WebSocket Configuration (app.module.ts)

Modify app.module.ts to include the WebSocket configuration:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { AppComponent } from './app.component';
import { SocketIoModule, SocketIoConfig } from 'ngx-socket-io';


const config: SocketIoConfig = { url: 'http://localhost:3000', options: {} };


@NgModule({
 declarations: [AppComponent],
 imports: [BrowserModule, SocketIoModule.forRoot(config)],
 bootstrap: [AppComponent]
})
export class AppModule { }
```

✓ Connects Angular to the WebSocket **server running on localhost:3000**.

CHAPTER 4: IMPLEMENTING WEBSOCKETS IN COMPONENTS

4.1 Modify the Component to Use WebSockets (app.component.ts)

Modify app.component.ts to send and receive WebSocket messages:

```
import { Component } from '@angular/core';
import { WebsocketService } from './websocket.service';

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent {
 message: string = '';
 messages: string[] = [];

 constructor(private websocketService: WebsocketService) {
  this.websocketService.getMessages().subscribe(msg => {
   this.messages.push(msg);
  });
 }

 sendMessage() {
```

```
  if (this.message.trim()) {
    this.websocketService.sendMessage(this.message);
    this.message = ''; // Clear input after sending
  }
 }
}
```

✓ Subscribes to **incoming WebSocket messages**.
✓ Sends messages **to the WebSocket server** when the user clicks send.

4.2 Create the Chat UI (app.component.html)

Modify app.component.html to create a simple chat interface:

```
<div class="container">
 <h2>Angular WebSocket Chat</h2>

 <ul class="list-group">
  <li *ngFor="let msg of messages" class="list-group-item">{{ msg }}</li>
 </ul>

 <input [(ngModel)]="message" class="form-control mt-3" placeholder="Type a message..." />
```

```
<button (click)="sendMessage()" class="btn btn-primary mt-2">Send</button>

</div>
```

✓ Displays **real-time chat messages**.
✓ Uses **two-way binding ([(ngModel)])** for message input.

CHAPTER 5: HANDLING WEBSOCKET ERRORS AND DISCONNECTS

Modify websocket.service.ts to **handle disconnects and errors**:

```
this.socket.on('disconnect', () => {
  console.warn('WebSocket disconnected. Attempting to reconnect...');
});

this.socket.on('connect_error', (error) => {
  console.error('WebSocket connection error:', error);
});
```

✓ **Detects when WebSocket disconnects** and attempts reconnection.

Case Study: How a Stock Trading Platform Used WebSockets for Real-Time Updates

Background

A stock trading platform needed **real-time price updates** for its dashboard.

Challenges

✓ **High server load** from frequent HTTP polling.
✓ **Delayed price updates,** reducing trading accuracy.
✓ **Poor user experience** due to slow refresh rates.

Solution: Implementing WebSockets

✓ Used **WebSockets to stream stock prices instantly**.
✓ **Reduced polling requests,** decreasing server load by **60%**.
✓ Implemented **real-time data visualization** with Angular components.

Results

🚀 **Instant stock price updates,** improving trading decisions.
⚡ **Faster UI updates,** reducing latency.
🔍 **Better server efficiency,** handling more users with fewer resources.

By using **WebSockets in Angular,** the trading platform **improved accuracy, performance, and user engagement**.

Exercise

1. Modify the **WebSocket service** to allow **private chat rooms**.
2. Implement a **"User is typing..." feature** using WebSockets.
3. Display **server connection status** (Online/Offline).
4. Create a **real-time notification system** using WebSockets.

Conclusion

✓ WebSockets enable real-time data exchange in Angular applications.
✓ Using RxJS Observables improves WebSocket event handling.
✓ A WebSocket server (Node.js) can manage multiple client connections.

IMPLEMENTING REAL-TIME CHAT FEATURES IN ANGULAR

CHAPTER 1: INTRODUCTION TO REAL-TIME CHAT IN ANGULAR

1.1 What is Real-Time Chat?

Real-time chat applications allow users to send and receive messages **instantly** without manually refreshing the page.

✓ **Enables instant messaging and notifications**.
✓ **Uses WebSockets or Firebase for live updates**.
✓ **Improves user engagement in applications**.

1.2 Technologies Used for Real-Time Chat

Real-time chat in Angular is implemented using:

✓ **WebSockets** – Full-duplex communication between client and server.
✓ **Firebase Realtime Database** – Google's cloud database for live updates.
✓ **RxJS Observables** – Handles real-time event streams.

CHAPTER 2: SETTING UP A WEBSOCKET-BASED CHAT APPLICATION

2.1 What is WebSocket?

WebSockets provide **bi-directional, event-driven communication** between the server and clients.

✓ **Faster than HTTP polling** – Reduces latency.
✓ **Persistent connection** – Messages are sent and received instantly.
✓ **Efficient resource usage** – No need for frequent API calls.

2.2 Setting Up a WebSocket Server

Step 1: Install ws WebSocket Package in Node.js

Run the following command to install WebSocket support:

```
npm install ws
```

Step 2: Create a WebSocket Server in Node.js

```
const WebSocket = require('ws');

const server = new WebSocket.Server({ port: 8080 });

server.on('connection', socket => {
  console.log('Client connected');

  socket.on('message', message => {
    console.log(`Received: ${message}`);

    // Broadcast message to all connected clients
    server.clients.forEach(client => {
```

```
        if (client.readyState === WebSocket.OPEN) {
            client.send(message);
        }
    });
});

socket.on('close', () => {
    console.log('Client disconnected');
});
});

console.log('WebSocket server running on ws://localhost:8080');
```

✓ Starts a **WebSocket server** on port **8080**.
✓ Handles **message broadcasting** between connected clients.

CHAPTER 3: IMPLEMENTING WEBSOCKET IN AN ANGULAR CHAT APPLICATION

3.1 Creating a WebSocket Service in Angular

Step 1: Install WebSocket Support in Angular

ng new angular-chat

cd angular-chat

npm install rxjs

Step 2: Create a WebSocket Service

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

@Injectable({
 providedIn: 'root'
})
export class ChatService {
 private socket$ = new WebSocketSubject('ws://localhost:8080');

 sendMessage(message: string) {
  this.socket$.next(message);
 }

 getMessages() {
  return this.socket$;
 }
}
```

✓ Connects Angular to the **WebSocket server**.
✓ **sendMessage()** sends messages.
✓ **getMessages()** listens for incoming messages.

3.2 Creating the Chat Component

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

import { ChatService } from '../services/chat.service';


@Component({

 selector: 'app-chat',

 template: `

  <div *ngFor="let msg of messages">{{ msg }}</div>

  <input [(ngModel)]="message" (keyup.enter)="sendMessage()"
placeholder="Type a message">

  <button (click)="sendMessage()">Send</button>

 `,

 styles: [`div { padding: 10px; border-bottom: 1px solid #ccc; }`]

})

export class ChatComponent {

 message = '';

 messages: string[] = [];


 constructor(private chatService: ChatService) {

  this.chatService.getMessages().subscribe(msg =>
this.messages.push(msg));
```

```
 }


 sendMessage() {
  if (this.message.trim()) {
   this.chatService.sendMessage(this.message);
   this.message = '';
  }
 }
}
```

✓ Listens for **real-time messages** using getMessages().
✓ Updates UI **instantly** when a message arrives.

CHAPTER 4: USING FIREBASE FOR REAL-TIME CHAT

4.1 Why Use Firebase?

✓ **No backend required** – Cloud-based database.
✓ **Built-in WebSockets** – Handles live updates automatically.
✓ **Secure authentication** – Supports Google, Facebook, and email login.

4.2 Setting Up Firebase in Angular

Step 1: Install Firebase SDK

npm install firebase @angular/fire

Step 2: Configure Firebase in environment.ts

Get your Firebase configuration from **Firebase Console** and add it to your environment.ts:

```
export const environment = {
 firebaseConfig: {
  apiKey: "YOUR_API_KEY",
  authDomain: "your-app.firebaseapp.com",
  projectId: "your-app",
  storageBucket: "your-app.appspot.com",
  messagingSenderId: "YOUR_SENDER_ID",
  appId: "YOUR_APP_ID"
 }
};
```

4.3 Creating a Chat Service Using Firebase

```
import { Injectable } from '@angular/core';
import { AngularFireDatabase } from
'@angular/fire/compat/database';


@Injectable({
 providedIn: 'root'
})
```

```
export class FirebaseChatService {

 constructor(private db: AngularFireDatabase) {}


 sendMessage(username: string, message: string) {

  this.db.list('messages').push({ username, message, timestamp:
Date.now() });

 }


 getMessages() {

  return this.db.list('messages', ref =>
ref.orderByChild('timestamp')).valueChanges();

 }

}
```

✓ Stores chat messages in Firebase's **Realtime Database**.
✓ Fetches messages **in real-time**.

4.4 Creating the Firebase Chat Component

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

import { FirebaseChatService } from '../services/firebase-
chat.service';


@Component({
```

```
 selector: 'app-firebase-chat',
 template: `
  <div *ngFor="let msg of messages">
   <strong>{{ msg.username }}:</strong> {{ msg.message }}
  </div>
  <input [(ngModel)]="message" (keyup.enter)="sendMessage()" placeholder="Type a message">
  <button (click)="sendMessage()">Send</button>
 `,
 styles: [`div { padding: 10px; border-bottom: 1px solid #ccc; }`]
})
export class FirebaseChatComponent {
 message = '';
 messages: any[] = [];
 username = 'User' + Math.floor(Math.random() * 1000);

 constructor(private chatService: FirebaseChatService) {
  this.chatService.getMessages().subscribe(messages => this.messages = messages);
 }

 sendMessage() {
```

```
  if (this.message.trim()) {
    this.chatService.sendMessage(this.username, this.message);
    this.message = '';
  }
 }
}
```

✓ Retrieves messages **in real-time** from Firebase.
✓ Displays the **username and message** dynamically.

CHAPTER 5: BEST PRACTICES FOR REAL-TIME CHAT IN ANGULAR

✓ **Use WebSockets for fast communication**.
✓ **Use Firebase if backend setup is not needed**.
✓ **Handle message formatting for better readability**.
✓ **Use RxJS debounceTime() to prevent unnecessary messages**.
✓ **Implement authentication for user security**.

Case Study: How a Customer Support Platform Used Real-Time Chat in Angular

Background

A **customer support platform** needed:

✓ **Real-time customer-agent messaging**.
✓ **Scalability for thousands of concurrent users**.
✓ **Cross-platform support (web and mobile)**.

Challenges

- **Slow message delivery** affected customer experience.
- **High server load due to HTTP polling**.
- **No user authentication for chat rooms**.

Solution: Implementing WebSockets and Firebase

✓ Used **WebSockets** for instant messaging.
✓ Used **Firebase for cloud-based real-time chat**.
✓ Integrated **user authentication for secure messaging**.

Results

🚀 **75% faster message delivery**.
🔍 **Lower server load using event-driven communication**.
⚡ **Seamless real-time chat with zero downtime**.

By implementing **WebSockets and Firebase,** the company provided a **smooth, real-time chat experience**.

Exercise

1. Create a **WebSocket-based chat service** in Angular.
2. Build a **Firebase chat system** with message storage.
3. Implement **real-time message streaming using RxJS Observables**.
4. Add **user authentication** to prevent unauthorized messages.

Conclusion

In this section, we explored:

✓ **How WebSockets and Firebase enable real-time chat.**

✓ **How to create a WebSocket server and connect it to Angular.**

✓ **How to use Firebase for chat applications.**

HANDLING ASYNCHRONOUS DATA STREAMS WITH RXJS IN ANGULAR

CHAPTER 1: INTRODUCTION TO RXJS AND ASYNCHRONOUS DATA STREAMS

1.1 What is RxJS?

RxJS (Reactive Extensions for JavaScript) is a powerful library for handling **asynchronous data streams** in Angular applications.

✓ **Manages asynchronous operations efficiently**.
✓ **Works seamlessly with HTTP requests, WebSockets, and user events**.
✓ **Uses Observables to handle and manipulate data streams**.

- **Key Concepts in RxJS**:
  - **Observable** – Represents a data stream that emits values over time.
  - **Observer** – Listens to an Observable and reacts to emitted values.
  - **Subscription** – Connects an Observer to an Observable.
  - **Operators** – Transform and manipulate data streams.

1.2 Why Use RxJS in Angular?

✓ **Handles multiple asynchronous operations** (e.g., API requests, user interactions).

✓ **Prevents callback hell** by chaining operators.
✓ **Enables real-time data updates** with WebSockets.
✓ **Optimizes performance by reducing unnecessary computations**.

CHAPTER 2: WORKING WITH OBSERVABLES IN ANGULAR

2.1 Creating and Using Observables

An **Observable** represents a **data stream** that emits values over time.

Example: Creating a Simple Observable

```
import { Observable } from 'rxjs';


const myObservable = new Observable(observer => {
  observer.next('First Value');
  observer.next('Second Value');
  observer.complete(); // Marks the end of the stream
});


myObservable.subscribe(value => console.log(value));
```

✓ **observer.next()** emits values.
✓ **observer.complete()** signals completion.
✓ **.subscribe()** listens for values and executes a callback function.

2.2 Subscribing to Observables in Angular Components

Observables are commonly used in **services to fetch API data**.

Example: Fetching API Data with RxJS in a Service

Modify api.service.ts:

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

@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/users';

  constructor(private http: HttpClient) {}

  getUsers(): Observable<any> {
    return this.http.get(this.apiUrl);
  }
}
```

✓ Returns an **Observable** from the API request.

2.3 Subscribing to the Observable in a Component

Modify app.component.ts to consume the data stream:

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

@Component({
 selector: 'app-root',
 templateUrl: './app.component.html',
 styleUrls: ['./app.component.css']
})
export class AppComponent implements OnInit {
 users: any = [];

 constructor(private apiService: ApiService) {}

 ngOnInit() {
  this.apiService.getUsers().subscribe(data => {
   this.users = data;
  });
 }
```

}

✓ **Subscribes** to the getUsers() Observable.

✓ **Updates the UI dynamically** when new data arrives.

2.4 Displaying Observable Data in the Template

Modify app.component.html to display the API data:

```
<h2>User List</h2>

<ul>

 <li *ngFor="let user of users">

  {{ user.name }} - {{ user.email }}

 </li>

</ul>
```

✓ Uses *ngFor to dynamically display data from the **Observable**.

CHAPTER 3: USING RXJS OPERATORS FOR DATA STREAMS

3.1 What Are RxJS Operators?

Operators modify and control data streams efficiently.

- **Common RxJS Operators**:

Operator	Description
map()	Transforms emitted values.
filter()	Emits only values that match a condition.

mergeMap()	Flattens nested Observables.
debounceTime()	Delays execution to prevent multiple rapid events.
catchError()	Handles errors in streams.

3.2 Transforming Data with map()

map() transforms the emitted data before passing it to the subscriber.

Example: Extracting Only User Names

```
import { map } from 'rxjs/operators';


getUsers(): Observable<string[]> {
 return this.http.get<any[]>(this.apiUrl).pipe(
  map(users => users.map(user => user.name))
 );
}
```

✓ Extracts **only names** from API data.
✓ **Reduces data size** sent to components.

3.3 Filtering Data with filter()

filter() removes unwanted values from the stream.

Example: Filtering Users by Age

```
import { filter } from 'rxjs/operators';


getUsers(): Observable<any[]> {
 return this.http.get<any[]>(this.apiUrl).pipe(
  map(users => users.filter(user => user.age >= 18))
 );
}
```

✓ Removes users **under 18** from the API response.

3.4 Handling API Errors with catchError()

catchError() prevents application crashes due to API failures.

Example: Handling API Request Failures

```
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';


getUsers(): Observable<any> {
 return this.http.get(this.apiUrl).pipe(
  catchError(error => {
   console.error('Error fetching users', error);
   return throwError(() => new Error('Failed to load data'));
  })
```

```
  );
}
```

✓ **Logs API errors** without breaking the application.
✓ Returns a **fallback error message** to handle failures gracefully.

CHAPTER 4: USING SUBJECT AND BEHAVIORSUBJECT FOR REAL-TIME DATA

4.1 Understanding Subject and BehaviorSubject

- **Subject** – Broadcasts data to multiple subscribers.
- **BehaviorSubject** – Stores the **last emitted value** and provides it immediately to new subscribers.

4.2 Using Subject for Cross-Component Communication

Modify event-bus.service.ts to create an event bus:

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

@Injectable({
  providedIn: 'root'
})
export class EventBusService {
  private eventSubject = new Subject<string>();
```

```
  event$ = this.eventSubject.asObservable();

  emitEvent(data: string) {
    this.eventSubject.next(data);
  }
}
```

✓ Allows **multiple components** to share data dynamically.

4.3 Subscribing to the Subject in Components

Modify receiver.component.ts to listen for messages:

```
import { Component, OnInit } from '@angular/core';
import { EventBusService } from '../event-bus.service';

@Component({
  selector: 'app-receiver',
  templateUrl: './receiver.component.html'
})
export class ReceiverComponent implements OnInit {
  receivedMessage = '';

  constructor(private eventBus: EventBusService) {}
```

```
 ngOnInit() {
   this.eventBus.event$.subscribe(message => {
     this.receivedMessage = message;
   });
 }
}
```

✓ Updates UI dynamically when new data arrives.

Case Study: How a Stock Market App Used RxJS for Real-Time Data Streaming

Background

A stock market platform needed **real-time stock price updates**.

Challenges

✓ **Slow updates** due to polling APIs every few seconds.
✓ **High memory usage** from multiple API calls.
✓ **Delayed price changes affecting traders.**

Solution: Implementing RxJS Observables

✓ Used WebSocketSubject for real-time stock data.
✓ Applied map() and filter() to optimize stock updates.
✓ Used BehaviorSubject to **store the latest price** for users joining late.

Results

🚀 **Instant stock price updates,** improving trading efficiency.
📉 **Reduced server requests by 60%,** optimizing API usage.
⚡ **Enhanced user experience,** ensuring real-time accuracy.

Exercise

1. **Create an Observable** that emits three values and logs them.
2. **Fetch data from an API using HttpClient and Observables**.
3. **Use map() to transform the response and display only names**.
4. **Use Subject to share data between two components**.
5. **Use catchError() to handle API failures gracefully**.

Conclusion

In this section, we explored:
✓ **How to create and subscribe to Observables in Angular**.
✓ **How to use RxJS operators (map(), filter(), catchError())**.
✓ **How Subject and BehaviorSubject enable real-time updates**.

INTRODUCTION TO TESTING IN ANGULAR (JASMINE & KARMA)

CHAPTER 1: INTRODUCTION TO TESTING IN ANGULAR

1.1 Why is Testing Important in Angular?

Testing in Angular ensures that applications are:

✓ **Reliable** – Detects and prevents bugs early.
✓ **Maintainable** – Ensures new updates do not break existing features.
✓ **Scalable** – Allows the application to grow without introducing errors.
✓ **Efficient** – Reduces manual testing efforts.

Angular provides a **built-in testing environment** with:

- **Jasmine** – A testing framework for writing test cases.
- **Karma** – A test runner that executes tests in different browsers.

1.2 Types of Testing in Angular

Testing Type	Description	Example Use Case
Unit Testing	Tests individual components, services, or functions	Checking if a method returns the correct value

Integration Testing	Tests how different parts of the app work together	Ensuring a form interacts correctly with a service
End-to-End (E2E) Testing	Tests the entire application from a user's perspective	Validating login functionality in a real browser

✓ **Unit tests** focus on **small, isolated** parts of the application.
✓ **Integration tests** check if different parts **work well together**.
✓ **E2E tests** simulate **real user interactions**.

CHAPTER 2: SETTING UP JASMINE & KARMA FOR ANGULAR TESTING

2.1 Installing Jasmine and Karma

Jasmine and Karma **come pre-installed** with Angular. To verify the setup, run:

ng version

To reinstall if needed:

npm install --save-dev jasmine-core karma karma-jasmine

2.2 Running Tests in Angular

To execute all test cases, use:

ng test

✓ **Launches the Karma test runner**.
✓ **Runs all test cases** in a browser window.
✓ **Auto-refreshes on code changes**.

CHAPTER 3: WRITING UNIT TESTS WITH JASMINE

3.1 Basic Structure of a Jasmine Test

A test file is created automatically when generating a component or service (.spec.ts).

Example: Sample Jasmine Test

```
describe('Basic Math Test', () => {
  it('should add two numbers correctly', () => {
    const sum = 2 + 3;
    expect(sum).toBe(5);
  });
});
```

✓ **describe()** – Defines a test suite.
✓ **it()** – Defines an individual test case.
✓ **expect()** – Checks expected results.

3.2 Testing an Angular Service

Step 1: Create a Service Using Angular CLI

ng generate service services/calculator

Step 2: Implement the Service Logic

Modify calculator.service.ts:

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


@Injectable({
 providedIn: 'root'
})
export class CalculatorService {
 add(a: number, b: number): number {
  return a + b;
 }
}
```

Step 3: Write Unit Tests for the Service

Modify calculator.service.spec.ts:

```
import { CalculatorService } from './calculator.service';


describe('CalculatorService', () => {
 let service: CalculatorService;


 beforeEach(() => {
  service = new CalculatorService();
```

```
});

it('should return the sum of two numbers', () => {
  expect(service.add(2, 3)).toBe(5);
});

it('should return a negative number when adding negative values',
() => {
  expect(service.add(-2, -3)).toBe(-5);
});
});
```

✓ Uses beforeEach() to **set up test instances**.
✓ Ensures the add() method works as expected.

CHAPTER 4: TESTING ANGULAR COMPONENTS

4.1 Writing Tests for an Angular Component

Step 1: Generate a New Component

ng generate component components/greeting

Step 2: Implement Component Logic

Modify greeting.component.ts:

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

```
@Component({
 selector: 'app-greeting',
 template: `<h1>{{ message }}</h1>`
})
export class GreetingComponent {
 message = "Hello, Angular!";
}
```

Step 3: Write Tests for the Component

Modify greeting.component.spec.ts:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { GreetingComponent } from './greeting.component';

describe('GreetingComponent', () => {
 let component: GreetingComponent;
 let fixture: ComponentFixture<GreetingComponent>;

 beforeEach(() => {
  TestBed.configureTestingModule({
   declarations: [GreetingComponent]
  });
```

```
  fixture = TestBed.createComponent(GreetingComponent);
  component = fixture.componentInstance;
 });

 it('should display the correct message', () => {
  fixture.detectChanges(); // Detects changes in the component
  const compiled = fixture.nativeElement;

expect(compiled.querySelector('h1').textContent).toContain('Hello, Angular!');
 });
});
```

✓ **TestBed.configureTestingModule()** sets up the testing environment.
✓ **fixture.detectChanges()** ensures the component is rendered before testing.

CHAPTER 5: MOCKING DEPENDENCIES IN UNIT TESTS

5.1 Using Spies to Mock Services

Mocking is used when a component **depends on a service** but does not require actual API calls.

Example: Mocking an API Call

Modify user.service.ts:

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


@Injectable({
 providedIn: 'root'
})
export class UserService {
 getUser() {
  return { name: "John Doe", email: "john@example.com" };
 }
}
```

Modify user.service.spec.ts:

```
import { UserService } from './user.service';


describe('UserService', () => {
 let service: UserService;

 beforeEach(() => {
  service = new UserService();
 });

 it('should return a user object', () => {
```

```
  spyOn(service, 'getUser').and.returnValue({ name: 'Test User', email: 'test@example.com' });

  expect(service.getUser().name).toBe('Test User');

 });

});
```

✓ **spyOn()** replaces the actual function with a mock response.
✓ **Ensures unit tests are isolated** from real API calls.

Case Study: How a Banking App Used Unit Tests to Improve Stability

Background

A **banking application** needed to:

✓ Ensure **fund transfer logic was error-free**.
✓ Prevent **unexpected application crashes**.
✓ Automate **testing for key financial transactions**.

Challenges

- **Bugs in transaction calculations** led to incorrect balances.
- **Manually testing components** took too much time.
- **Lack of automated tests** made regression testing difficult.

Solution: Implementing Angular Unit Tests

The team implemented:

✓ **Unit tests for banking transactions** using Jasmine.
✓ **Mocking APIs** to prevent dependency failures.
✓ **Automated testing with Karma,** reducing manual checks.

Results

🚀 **Detected and fixed 80% of transaction-related bugs before deployment**.
⚡ **Reduced manual testing time by 60%**.
🔄 **Increased reliability,** ensuring user trust in financial transactions.

By integrating **Angular unit testing,** the banking app **achieved stability and faster development cycles**.

Exercise

1. Write a unit test for a **simple addition function** using Jasmine.
2. Create a **mock API service** and test its response using spyOn().
3. Test a **component with an @Input() property** and verify its behavior.
4. Run ng test and analyze test results in Karma.

Conclusion

In this section, we explored:
✓ **How Jasmine & Karma automate testing in Angular**.
✓ **How to test components, services, and APIs using mocks**.
✓ **How unit tests improve stability and maintainability**.

WRITING UNIT TESTS FOR COMPONENTS & SERVICES IN ANGULAR

CHAPTER 1: INTRODUCTION TO UNIT TESTING IN ANGULAR

1.1 What is Unit Testing?

Unit testing is a **software testing technique** where individual components, services, or functions are tested in **isolation** to ensure they function correctly.

✓ Ensures **code reliability and stability**.
✓ Helps **detect bugs early** in the development cycle.
✓ Provides **confidence for refactoring code**.

1.2 Why Unit Test in Angular?

✓ **Prevents regressions** when making changes.
✓ **Improves maintainability** by catching issues early.
✓ **Automates testing**, saving time on manual checks.

Angular uses:

- **Jasmine** – A testing framework for writing unit tests.
- **Karma** – A test runner that executes tests in a browser.

✅ **Run all tests** using:

ng test

✅ **Run tests with coverage report**:

ng test --code-coverage

CHAPTER 2: UNIT TESTING COMPONENTS IN ANGULAR

2.1 Generating a Testable Component

Generate a new component:

ng generate component test-example

This creates:

- test-example.component.ts (Component logic)
- test-example.component.spec.ts (Test file)

2.2 Understanding the Default Test File (.spec.ts)

Angular CLI automatically generates a test file:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
import { TestExampleComponent } from './test-example.component';

describe('TestExampleComponent', () => {
  let component: TestExampleComponent;
  let fixture: ComponentFixture<TestExampleComponent>;

  beforeEach(async () => {
```

```
  await TestBed.configureTestingModule({
    declarations: [ TestExampleComponent ]
  })
  .compileComponents();
});

beforeEach(() => {
  fixture = TestBed.createComponent(TestExampleComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
});

it('should create the component', () => {
  expect(component).toBeTruthy();
});
});
```

✓ **describe()** – Defines the test suite for the component.
✓ **beforeEach()** – Runs before each test to set up the component.
✓ **expect(component).toBeTruthy()** – Checks if the component is created successfully.

2.3 Writing a Simple Unit Test for a Component

Modify test-example.component.ts:

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


@Component({
 selector: 'app-test-example',
 template: `<h2>{{ title }}</h2>`
})
export class TestExampleComponent {
 title = 'Hello Angular!';
}
```

Modify test-example.component.spec.ts to test the title:

```
it('should have a title "Hello Angular!"', () => {
 expect(component.title).toEqual('Hello Angular!');
});
```

✓ Verifies that the component's title is correctly set.

2.4 Testing Component HTML Rendering

Modify test-example.component.html:

```
<h2>{{ title }}</h2>
```

Add a test to check if the title appears in the template:

```
import { ComponentFixture, TestBed } from '@angular/core/testing';
```

```
import { TestExampleComponent } from './test-
example.component';

import { By } from '@angular/platform-browser';


describe('TestExampleComponent', () => {
 let component: TestExampleComponent;
 let fixture: ComponentFixture<TestExampleComponent>;


 beforeEach(async () => {
  await TestBed.configureTestingModule({
   declarations: [ TestExampleComponent ]
  })
  .compileComponents();
 });


 beforeEach(() => {
  fixture = TestBed.createComponent(TestExampleComponent);
  component = fixture.componentInstance;
  fixture.detectChanges();
 });
```

```
 it('should render title in an h2 tag', () => {
   const compiled = fixture.nativeElement;

expect(compiled.querySelector('h2').textContent).toContain('Hello Angular!');
 });
});
```

✓ This test ensures the **title is correctly rendered in the HTML**.

CHAPTER 3: UNIT TESTING SERVICES IN ANGULAR

3.1 Generating a Testable Service

Generate a new service:

```
ng generate service test-service
```

This creates:

- test-service.service.ts (Service logic)
- test-service.service.spec.ts (Test file)

3.2 Writing a Simple Service and Testing It

Modify test-service.service.ts:

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

@Injectable({
```

```
  providedIn: 'root'
})
export class TestService {
  getMessage(): string {
    return 'Hello from Service!';
  }
}
```

Modify test-service.service.spec.ts to test the service:

```
import { TestBed } from '@angular/core/testing';
import { TestService } from './test-service.service';

describe('TestService', () => {
  let service: TestService;

  beforeEach(() => {
    TestBed.configureTestingModule({});
    service = TestBed.inject(TestService);
  });

  it('should return a message "Hello from Service!"', () => {
    expect(service.getMessage()).toEqual('Hello from Service!');
```

```
  });
});
```

✓ This test **ensures that the service method returns the correct message**.

3.3 Testing an HTTP Service

Modify test-service.service.ts to fetch data from an API:

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

@Injectable({
  providedIn: 'root'
})
export class TestService {
  private apiUrl = 'https://jsonplaceholder.typicode.com/posts';

  constructor(private http: HttpClient) {}

  getPosts(): Observable<any> {
    return this.http.get(this.apiUrl);
```

```
  }
}
```

Modify test-service.service.spec.ts to mock HTTP requests:

```
import { TestBed } from '@angular/core/testing';
import { HttpClientTestingModule, HttpTestingController } from '@angular/common/http/testing';
import { TestService } from './test-service.service';

describe('TestService', () => {
  let service: TestService;
  let httpMock: HttpTestingController;

  beforeEach(() => {
    TestBed.configureTestingModule({
      imports: [HttpClientTestingModule],
      providers: [TestService]
    });

    service = TestBed.inject(TestService);
    httpMock = TestBed.inject(HttpTestingController);
  });
```

```
 it('should fetch posts from API', () => {
   const dummyPosts = [{ id: 1, title: 'Post 1' }, { id: 2, title: 'Post 2' }];

   service.getPosts().subscribe(posts => {
    expect(posts.length).toBe(2);
    expect(posts).toEqual(dummyPosts);
   });

   const req =
httpMock.expectOne('https://jsonplaceholder.typicode.com/posts');
   expect(req.request.method).toBe('GET');
   req.flush(dummyPosts);
 });

 afterEach(() => {
   httpMock.verify();
 });
});
```

✓ **Mocks an HTTP request** and verifies that the correct data is fetched.

Case Study: How a Banking App Improved Stability with Unit Tests

Background

A banking application needed **reliable account transactions and user authentication services**.

Challenges

✓ **Frequent API failures** causing app crashes.
✓ **Unstable login service** affecting user experience.
✓ **Poor test coverage,** making debugging difficult.

Solution: Implementing Unit Tests for Components & Services

✓ **Tested login service to handle API failures gracefully**.
✓ **Verified transaction calculations** using unit tests.
✓ **Mocked backend responses** to simulate real-world scenarios.

Results

🚀 **Reduced production bugs by 80%**.
🔍 **Faster debugging,** improving development speed.
⚡ **Improved system reliability,** increasing user trust.

By integrating **unit testing in Angular,** the banking app **improved performance, stability, and security**.

Exercise

1. Create a new Angular **service** that returns user data.
2. Write a **unit test** for the service using **Jasmine & Karma**.
3. Create an Angular **component** that displays a list of users.

4️⃣ Write a **unit test** to verify that the component correctly displays user names.

Conclusion

✓ Unit testing ensures Angular applications are stable and error-free.
✓ Jasmine & Karma provide a powerful testing framework for Angular.
✓ Mocking HTTP requests improves API testing reliability.

DEBUGGING AND ERROR HANDLING IN ANGULAR

CHAPTER 1: INTRODUCTION TO DEBUGGING AND ERROR HANDLING IN ANGULAR

1.1 Why is Debugging and Error Handling Important?

Debugging and error handling are essential for **building robust and scalable** Angular applications.

✓ **Identifies and fixes issues quickly**.
✓ **Prevents crashes and improves performance**.
✓ **Enhances user experience by handling errors gracefully**.

Angular provides **built-in tools** for debugging and mechanisms for **handling errors efficiently** at various levels.

CHAPTER 2: DEBUGGING ANGULAR APPLICATIONS

2.1 Using Browser Developer Tools

Most debugging in Angular is done using **Chrome DevTools or Firefox DevTools**.

Common Debugging Features

✓ **Console Tab** – Displays logs, errors, and warnings.
✓ **Elements Tab** – Inspects and edits the DOM.
✓ **Network Tab** – Monitors API calls and responses.
✓ **Sources Tab** – Adds breakpoints for debugging TypeScript code.

Example: Logging Debug Information

```
console.log("Debugging Angular Component");

console.error("An error occurred!");

console.warn("This is a warning message");

console.table([{ id: 1, name: "John" }, { id: 2, name: "Alice" }]);
```

✓ Logs useful information for debugging **variables, API responses, and errors**.

2.2 Debugging Using Augury (Angular DevTools Extension)

Augury is a Chrome extension that helps debug Angular applications.

Augury Features

✓ **View component structure** – See all components in the app.
✓ **Inspect component properties** – Modify @Input() values in real-time.
✓ **Analyze dependency injection (DI)** – Identify provided services.

Installing Augury

1. Install **Angular DevTools** from the Chrome Web Store.
2. Open Developer Tools (F12) → **Angular** tab.
3. Inspect components and **modify their state live**.

2.3 Setting Breakpoints in TypeScript

Breakpoints pause execution at specific points in the code, helping identify issues.

Example: Setting a Breakpoint in Chrome DevTools

1. Open the **Sources** tab in Chrome DevTools.
2. Find the TypeScript file (.ts).
3. Click on the **line number** to set a breakpoint.

Example: Using debugger in TypeScript

```
function processData(data: any) {
  debugger; // Pauses execution here
  console.log("Processing Data:", data);
}
```

✓ The **execution stops**, allowing inspection of variables.

CHAPTER 3: ERROR HANDLING IN ANGULAR

3.1 Types of Errors in Angular

Error Type	Description
Runtime Errors	Occur during execution (e.g., null value access).
Compile-Time Errors	Happen when the TypeScript code has syntax errors.
HTTP Errors	Failed API calls (e.g., 404 Not Found, 500 Server Error).

Unhandled Promise Rejections	Occur when Promise or Observable errors are not caught.

CHAPTER 4: HANDLING ERRORS IN ANGULAR COMPONENTS

4.1 Using try-catch for Handling Runtime Errors

```
try {
  let data = JSON.parse('{invalid json}');
} catch (error) {
  console.error("Error parsing JSON:", error);
}
```

✓ Prevents **application crashes** when invalid data is encountered.

4.2 Using ngOnInit() for Error Handling in Components

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

@Component({
  selector: 'app-error-demo',
  template: `<p>{{ message }}</p>`
})
export class ErrorDemoComponent implements OnInit {
  message = "";
```

```
 ngOnInit() {
  try {
   throw new Error("Something went wrong!");
  } catch (error) {
   this.message = "An error occurred!";
   console.error(error);
  }
 }
}
```

✓ **Displays user-friendly messages** instead of crashing the app.

CHAPTER 5: HANDLING HTTP ERRORS WITH HTTPINTERCEPTOR

5.1 What is an HTTP Interceptor?

An **HttpInterceptor** allows intercepting HTTP requests and responses **to handle errors globally**.

5.2 Implementing an HTTP Error Handler

Step 1: Create an Interceptor Service

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

import { HttpInterceptor, HttpRequest, HttpHandler, HttpEvent, HttpErrorResponse } from '@angular/common/http';
```

```
import { Observable, throwError } from 'rxjs';
import { catchError } from 'rxjs/operators';


@Injectable()
export class HttpErrorInterceptor implements HttpInterceptor {
 intercept(req: HttpRequest<any>, next: HttpHandler):
Observable<HttpEvent<any>> {
   return next.handle(req).pipe(
    catchError((error: HttpErrorResponse) => {
     let errorMsg = "Unknown error occurred";


     if (error.status === 404) {
      errorMsg = "Resource not found";
     } else if (error.status === 500) {
      errorMsg = "Internal server error";
     }


     console.error("HTTP Error:", errorMsg);
     return throwError(errorMsg);
    })
   );
```

```
  }
}
```

✓ **Handles API errors globally** instead of writing catch in every request.

Step 2: Register the Interceptor in app.module.ts

```
import { HttpClientModule, HTTP_INTERCEPTORS } from '@angular/common/http';
import { HttpErrorInterceptor } from './http-error.interceptor';

@NgModule({
  imports: [HttpClientModule],
  providers: [
    { provide: HTTP_INTERCEPTORS, useClass: HttpErrorInterceptor, multi: true }
  ]
})
export class AppModule {}
```

✓ **Automatically handles all HTTP errors** in the application.

CHAPTER 6: GLOBAL ERROR HANDLING IN ANGULAR

6.1 Using the ErrorHandler Class

Angular allows **global error handling** using the ErrorHandler service.

Step 1: Create a Global Error Handler

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


@Injectable()
export class GlobalErrorHandler implements ErrorHandler {
  handleError(error: any) {
    console.error("Global Error:", error);
    alert("Something went wrong. Please try again.");
  }
}
```

✓ Captures **all uncaught errors** across the application.

Step 2: Register the Global Error Handler in app.module.ts

```
@NgModule({
  providers: [{ provide: ErrorHandler, useClass: GlobalErrorHandler }]
})
export class AppModule {}
```

✓ Ensures **application-wide error handling**.

Case Study: How an Online Banking Platform Used Error Handling to Improve User Experience

Background

A **banking application** needed a **robust error handling mechanism** to:

✓ Prevent **transaction failures from crashing the app**.
✓ Display **clear user-friendly error messages**.
✓ Log **API errors for debugging**.

Challenges

- Users received **blank screens** on errors.
- HTTP failures **were not logged,** making debugging difficult.
- **Slow debugging process** due to lack of error tracking.

Solution: Implementing Comprehensive Error Handling

✓ **Used HttpInterceptor** to catch API errors globally.
✓ **Implemented ErrorHandler** for uncaught errors.
✓ **Integrated logging system** to store errors in a database.

Results

🚀 **Error detection time reduced by 70%**.
🔍 **Clear error messages improved user trust**.
⚡ **Faster debugging, reducing development costs**.

By implementing **a structured error handling approach,** the platform **improved reliability and user experience**.

Exercise

1. Use try-catch to **handle runtime errors** in a component.
2. Implement an HttpInterceptor to **catch API errors** globally.
3. Create a **global error handler** using ErrorHandler.
4. Use console.table() to **log API response data for debugging**.

Conclusion

In this section, we explored:

✓ **How to debug Angular applications using DevTools and Augury**.

✓ **How to use try-catch and console.log() for error handling**.

✓ **How to implement HttpInterceptor for global API error management**.

✓ **How to use ErrorHandler for application-wide error handling**.

ASSIGNMENT:

BUILD A REAL-TIME ANGULAR APPLICATION WITH WEBSOCKET COMMUNICATION AND UNIT TESTS

ASSIGNMENT SOLUTION: BUILDING A REAL-TIME ANGULAR APPLICATION WITH WEBSOCKET COMMUNICATION AND UNIT TESTS

- **Features of the Application**

✓ Establish **WebSocket communication** using WebSocketSubject.
✓ Display **real-time data updates** in a component.
✓ Implement **unit tests** for WebSocket communication using Jasmine and Karma.

Step 1: Setting Up the Angular Project

First, create a new Angular project and navigate into it:

```
ng new real-time-app
cd real-time-app
```

✓ This sets up the **basic project structure**.

Next, install RxJS WebSocket support:

```
npm install rxjs
```

✓ This package enables **real-time WebSocket communication**.

Step 2: Creating a WebSocket Service

We need a **service** to handle WebSocket connections and manage incoming messages.

- **Generate WebSocket Service**

ng generate service webSocket

Modify web-socket.service.ts to create a WebSocket connection:

```
import { Injectable } from '@angular/core';
import { WebSocketSubject } from 'rxjs/webSocket';
import { Observable } from 'rxjs';

@Injectable({
 providedIn: 'root'
})
export class WebSocketService {
 private socket$: WebSocketSubject<any>;

 constructor() {
  this.socket$ = new
WebSocketSubject('wss://echo.websocket.org'); // Public test
WebSocket server
 }

 // Send message to WebSocket server
 sendMessage(message: string): void {
  this.socket$.next(message);
 }
```

```
 // Receive real-time messages
 getMessages(): Observable<any> {
  return this.socket$.asObservable();
 }

 // Close WebSocket connection
 closeConnection(): void {
  this.socket$.complete();
 }
}
```

✓ Establishes a WebSocket connection to **wss://echo.websocket.org** (a public test WebSocket).
✓ sendMessage() sends data to the server.
✓ getMessages() listens for incoming messages.
✓ closeConnection() properly closes the WebSocket.

Step 3: Creating a Real-Time Component

We will create a component to **send and receive messages in real time**.

- **Generate a WebSocket Component**

ng generate component real-time

Modify real-time.component.ts:

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

```
import { WebSocketService } from '../web-socket.service';

@Component({
 selector: 'app-real-time',
 templateUrl: './real-time.component.html',
 styleUrls: ['./real-time.component.css']
})
export class RealTimeComponent implements OnInit {
 messages: string[] = [];
 newMessage: string = '';

 constructor(private webSocketService: WebSocketService) {}

 ngOnInit() {
  this.webSocketService.getMessages().subscribe(
   (message) => this.messages.push(message),
   (error) => console.error('WebSocket Error:', error)
  );
 }

 sendMessage() {
  if (this.newMessage.trim()) {
```

```
    this.webSocketService.sendMessage(this.newMessage);
    this.newMessage = ''; // Clear input after sending
  }
 }

 closeConnection() {
  this.webSocketService.closeConnection();
 }
}
```

✓ Listens for **incoming messages** and updates the UI.
✓ Sends messages via the **WebSocket service**.
✓ Closes the connection when needed.

Step 4: Building the UI for Real-Time Messages

Modify real-time.component.html:

```
<h2>Real-Time Chat</h2>

<input type="text" [(ngModel)]="newMessage" placeholder="Type a message..." />
<button (click)="sendMessage()">Send</button>
<button (click)="closeConnection()">Close Connection</button>

<ul>
```

```
  <li *ngFor="let msg of messages">{{ msg }}</li>
</ul>
```

✓ **ngModel** binds the input field to newMessage.
✓ ***ngFor** displays received messages dynamically.
✓ Provides buttons to **send messages and close the connection**.

Step 5: Adding WebSocket Component to the App

Modify app.component.html to include our real-time component:

```
<app-real-time></app-real-time>
```

✓ The WebSocket-based chat feature **is now part of the main app**.

Step 6: Writing Unit Tests for the WebSocket Service

To ensure **WebSocketService works correctly**, we write **unit tests** using Jasmine and Karma.

Modify web-socket.service.spec.ts:

```
import { TestBed } from '@angular/core/testing';
import { WebSocketService } from './web-socket.service';
import { WebSocketSubject } from 'rxjs/webSocket';
import { cold } from 'jasmine-marbles';

describe('WebSocketService', () => {
  let service: WebSocketService;
  let mockSocket$: WebSocketSubject<any>;
```

```
 beforeEach(() => {
  TestBed.configureTestingModule({});
  service = TestBed.inject(WebSocketService);
  mockSocket$ = new
WebSocketSubject('wss://echo.websocket.org');
 });

 it('should create WebSocket connection', () => {
  expect(service).toBeTruthy();
 });

 it('should send a message via WebSocket', () => {
  spyOn(mockSocket$, 'next');
  service.sendMessage('Hello WebSocket');
  expect(mockSocket$.next).toHaveBeenCalledWith('Hello
WebSocket');
 });

 it('should receive messages from WebSocket', () => {
  const message$ = cold('-a|', { a: 'Test Message' });
  service.getMessages().subscribe((msg) => {
   expect(msg).toBe('Test Message');
```

```
  });
 });

 it('should close WebSocket connection', () => {
  spyOn(mockSocket$, 'complete');
  service.closeConnection();
  expect(mockSocket$.complete).toHaveBeenCalled();
 });
});
```

✓ **Creates a fake WebSocketSubject** for testing.
✓ **Tests message sending and receiving behavior**.
✓ **Verifies that WebSocket closes correctly**.

Step 7: Running Unit Tests

Run the tests using:

```
ng test
```

✓ Ensures WebSocket functionality **works as expected**.

Case Study: How a Finance Company Used WebSockets in Angular for Real-Time Stock Prices

Background

A **finance company** wanted to provide **live stock price updates** in its trading platform.

Challenges

✓ **Slow data updates due to polling**.
✓ **High memory usage from multiple API calls**.
✓ **User experience was affected by delayed updates**.

Solution: Using WebSockets with Angular

✓ Implemented WebSocketService using RxJS.
✓ Created a StockPriceComponent to **display live updates**.
✓ Used Subject to **broadcast stock updates to all subscribers**.

Results

🚀 **Instant stock price updates,** improving user experience.
📉 **Reduced API requests by 60%,** optimizing server usage.
⚡ **Better trading experience,** enabling real-time decision-making.

By **switching from polling to WebSockets,** the company improved **performance and scalability**.

Exercise

1. **Modify the WebSocket component** to display the sender's name along with the message.
2. **Write a unit test** to check if closeConnection() correctly closes the WebSocket.
3. **Enhance the UI** by adding timestamps to messages.
4. **Implement WebSockets in a stock market tracker** that updates real-time prices.

Conclusion

In this assignment, we built:

✓ A **real-time WebSocket-based Angular application**.

✓ A **WebSocket service** for handling communication.

✓ **Unit tests using Jasmine and Karma**.