Integrating the Angular frontend with the Express.js API

1. Set Up the Express.js Backend

Make sure your **Express.js API** is running and serving data correctly.

Example: Basic Express.js API (`server.js`)

```
const express = require("express");
const cors = require("cors");

const app = express();
app.use(express.json()); // To parse JSON request bodies
app.use(cors()); // Enable CORS for frontend requests

// Sample API route
app.get("/api/data", (req, res) => {
  res.json({ message: "Hello from Express API" });
});

const PORT = 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

2. Run the Express Server

node server.js

This will start your backend on `http://localhost:5000`.

3. Configure CORS in Express.js (if needed)

CORS is required to allow requests from your Angular app running on a different port.

If using `cors` package

```
const cors = require("cors");
app.use(cors({ origin: "http://localhost:4200" })); // Allow Angular requests
```

4. Set Up Angular HTTP Client

Ensure **HttpClientModule** is imported in your `app.module.ts`:

```
import { NgModule } from '@angular/core';
import { BrowserModule } from '@angular/platform-browser';
import { HttpClientModule } from '@angular/common/http';

@NgModule({
  declarations: [],
  imports: [
    BrowserModule,
    HttpClientModule  // Import HttpClientModule for API calls
  ],
  providers: [],
  bootstrap: []
})
export class AppModule { }
```

5. Create Angular Service for API Calls

Create a service (`api.service.ts`) to interact with Express API.

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

@Injectable({
  providedIn: 'root'
})
export class ApiService {
  private apiUrl = 'http://localhost:5000/api/data'; // Backend API URL

  constructor(private http: HttpClient) {}

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

6. Use the API Service in Angular Components

In a component (`app.component.ts`), call the API service.

```
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 {
  data: any;

  constructor(private apiService: ApiService) {}

  ngOnInit() {
    this.apiService.getData().subscribe(response => {
      this.data = response;
    });
  }
}
```

7. Display Data in HTML (`app.component.html`)

```
<h1>Angular-Express Integration</h1>
<p>API Response: {{ data?.message }}</p>
```

8. Run the Angular App

ng serve

Ensure your **Express.js backend** is running on port `5000` and **Angular frontend** is running on port `4200`.

9. Deploying the MEAN Stack App

- Use **NGINX or Apache** to serve Angular and proxy requests to Express.
- Use **PM2** to manage the Express server.
- Use **MongoDB Atlas** or a local MongoDB instance for data persistence.

Authentication and user management integration

Integrating **authentication and user management** in a **MEAN stack application** involves the following key components:

1. **User Registration** (Sign Up)
2. **User Login** (JWT Authentication)
3. **Protecting Routes** (Middleware)
4. **User Roles & Permissions** (Optional)
5. **Token Management** (Angular Interceptor)

Step 1: Install Required Packages

In the Express.js backend, install the required authentication packages:

npm install express jsonwebtoken bcryptjs mongoose cors dotenv

- `jsonwebtoken` → Generates & verifies JWT tokens.
- `bcryptjs` → Hashes and compares passwords.
- `mongoose` → Manages MongoDB interactions.

Step 2: Set Up MongoDB with Mongoose

Ensure MongoDB is running, then create a `models/User.js` file.

```
javascript
const mongoose = require("mongoose");

const UserSchema = new mongoose.Schema({
  name: { type: String, required: true },
  email: { type: String, required: true, unique: true },
  password: { type: String, required: true },
});

module.exports = mongoose.model("User", UserSchema);
```

Step 3: Create Express.js Authentication Routes

Create an `auth.js` file inside a `routes` folder.

```
const express = require("express");
const bcrypt = require("bcryptjs");
const jwt = require("jsonwebtoken");
const User = require("../models/User");

const router = express.Router();
const SECRET_KEY = "your_secret_key"; // Replace with a secure secret

// Register User
```

```
router.post("/register", async (req, res) => {
  const { name, email, password } = req.body;

  try {
    let user = await User.findOne({ email });
    if (user) return res.status(400).json({ message: "User already exists" });

    const hashedPassword = await bcrypt.hash(password, 10);
    user = new User({ name, email, password: hashedPassword });

    await user.save();
    res.status(201).json({ message: "User registered successfully" });
  } catch (err) {
    res.status(500).json({ message: "Server error" });
  }
});

// Login User
router.post("/login", async (req, res) => {
  const { email, password } = req.body;

  try {
    const user = await User.findOne({ email });
    if (!user) return res.status(400).json({ message: "User not found" });
```

```
    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) return res.status(400).json({ message: "Invalid credentials" });

    const token = jwt.sign({ userId: user._id }, SECRET_KEY, { expiresIn: "1h" });
    res.json({ token });
  } catch (err) {
    res.status(500).json({ message: "Server error" });
  }
});

module.exports = router;
```

Step 4: Middleware to Protect Routes

Create `middleware/auth.js`:

```
const jwt = require("jsonwebtoken");
const SECRET_KEY = "your_secret_key";

module.exports = (req, res, next) => {
  const token = req.header("Authorization");
  if (!token) return res.status(401).json({ message: "Access denied" });
```

```
  try {
    const decoded = jwt.verify(token.split(" ")[1], SECRET_KEY);
    req.user = decoded;
    next();
  } catch (err) {
    res.status(401).json({ message: "Invalid token" });
  }
};
```

Step 5: Apply Authentication Middleware

In `server.js`:

```
const express = require("express");
const mongoose = require("mongoose");
const cors = require("cors");
const dotenv = require("dotenv");

dotenv.config();
const app = express();

app.use(express.json());
app.use(cors());
```

```
mongoose.connect("mongodb://localhost:27017/mean_auth", { useNewUrlParser: true, useUnifiedTopology: true });


app.use("/auth", require("./routes/auth")); // Auth routes


const authMiddleware = require("./middleware/auth");


// Protected route example
app.get("/api/protected", authMiddleware, (req, res) => {
  res.json({ message: "This is a protected route!" });
});



app.listen(5000, () => console.log("Server running on port 5000"));
```

Step 6: Create Angular Authentication Service

Create `auth.service.ts` in Angular:

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

```
@Injectable({
  providedIn: 'root'
})
export class AuthService {
  private apiUrl = 'http://localhost:5000/auth';

  constructor(private http: HttpClient) {}

  register(user: any): Observable<any> {
    return this.http.post(`${this.apiUrl}/register`, user);
  }

  login(credentials: any): Observable<any> {
    return this.http.post(`${this.apiUrl}/login`, credentials);
  }

  saveToken(token: string) {
    localStorage.setItem('token', token);
  }

  getToken(): string | null {
```

```
    return localStorage.getItem('token');
  }

  isAuthenticated(): boolean {
    return !!this.getToken();
  }

  logout() {
    localStorage.removeItem('token');
  }
}
```

Step 7: Use Authentication in Angular Components

Login Component

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

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

```
  email = '';
  password = '';

  constructor(private authService: AuthService) {}

  login() {
    this.authService.login({ email: this.email, password: this.password }).subscribe(
      (res: any) => {
        this.authService.saveToken(res.token);
        alert('Login successful');
      },
      (error) => alert(error.error.message)
    );
  }
}
```

Login HTML

```
<form (ngSubmit)="login()">
  <input type="email" [(ngModel)]="email" name="email" placeholder="Email" required>
  <input type="password" [(ngModel)]="password" name="password" placeholder="Password" required>
  <button type="submit">Login</button>
```

```
</form>
```

Step 8: Protect Angular Routes

Modify `app-routing.module.ts`:

```
import { Injectable } from '@angular/core';
import { CanActivate, Router } from '@angular/router';
import { AuthService } from './auth.service';

@Injectable({
  providedIn: 'root'
})
export class AuthGuard implements CanActivate {
  constructor(private authService: AuthService, private router: Router) {}

  canActivate(): boolean {
    if (!this.authService.isAuthenticated()) {
      this.router.navigate(['/login']);
      return false;
    }
    return true;
  }
```

```
}
```

Apply guard to protected routes:

```
const routes: Routes = [
  { path: 'protected', component: ProtectedComponent, canActivate: [AuthGuard] }
];
```

Step 9: Add Token to HTTP Requests (Interceptor)

Create `auth.interceptor.ts`:

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler } from
'@angular/common/http';
import { AuthService } from './auth.service';

@Injectable()
export class AuthInterceptor implements HttpInterceptor {
  constructor(private authService: AuthService) {}

  intercept(req: HttpRequest<any>, next: HttpHandler) {
    const token = this.authService.getToken();
    if (token) {
      const cloned = req.clone({
        setHeaders: { Authorization: `Bearer ${token}` }
```

```
      });
      return next.handle(cloned);
    }
    return next.handle(req);
  }
}
```

Register the interceptor in `app.module.ts`:

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


providers: [
  { provide: HTTP_INTERCEPTORS, useClass: AuthInterceptor, multi: true }
]
```

Handling real-time data with WebSockets

Handling **real-time data** in a **MEAN stack** application requires **WebSockets**, which allow bi-directional communication between the client (Angular) and the server (Express.js). In this guide, we'll use **Socket.IO**, a popular WebSocket library.

Step 1: Install Socket.IO

In the Express.js backend, install **Socket.IO**:

```
npm install socket.io
```

For Angular, install the client-side **Socket.IO package**:

```
npm install socket.io-client
```

Step 2: Set Up WebSockets in Express.js

Modify your `server.js` to integrate **Socket.IO**.

```
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); // Create HTTP server
const io = socketIo(server, {
  cors: {
    origin: "http://localhost:4200", // Allow Angular frontend
    methods: ["GET", "POST"]
  }
});

app.use(express.json());
app.use(cors());

// Listen for client connections
io.on("connection", (socket) => {
  console.log("New client connected:", socket.id);

  // Listen for messages from the client
  socket.on("message", (data) => {
    console.log("Message received:", data);
    io.emit("message", data); // Broadcast to all clients
  });

  // Handle client disconnection
  socket.on("disconnect", () => {
    console.log("Client disconnected:", socket.id);
  });
});

const PORT = 5000;
server.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

Step 3: Create Angular WebSocket Service

Create a **service** to handle WebSocket connections.

`socket.service.ts`

```
import { Injectable } from '@angular/core';
import { io, Socket } from 'socket.io-client';
import { Observable } from 'rxjs';

@Injectable({
  providedIn: 'root'
})
export class SocketService {
  private socket: Socket;
  private SERVER_URL = 'http://localhost:5000';

  constructor() {
    this.socket = io(this.SERVER_URL);
  }

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

  // Receive messages from the server
  onMessage(): Observable<string> {
    return new Observable(observer => {
      this.socket.on('message', (data: string) => {
        observer.next(data);
      });
    });
  }

  // Disconnect socket when component is destroyed
  disconnect() {
    this.socket.disconnect();
  }
```

```
}
```

Step 4: Implement WebSocket in an Angular Component

Modify a component to use real-time data.

chat.component.ts

```
import { Component, OnInit, OnDestroy } from '@angular/core';
import { SocketService } from '../socket.service';

@Component({
  selector: 'app-chat',
  templateUrl: './chat.component.html',
  styleUrls: ['./chat.component.css']
})
export class ChatComponent implements OnInit, OnDestroy {
  messages: string[] = [];
  messageInput = '';

  constructor(private socketService: SocketService) {}

  ngOnInit() {
    this.socketService.onMessage().subscribe((message: string) => {
      this.messages.push(message);
```

```
    });
  }

  sendMessage() {
    if (this.messageInput.trim()) {
      this.socketService.sendMessage(this.messageInput);
      this.messageInput = '';
    }
  }

  ngOnDestroy() {
    this.socketService.disconnect();
  }
}
```

Step 5: Create Chat UI

Modify `chat.component.html` to display real-time messages.

```
<h2>Chat Room</h2>

<div class="chat-box">
  <div *ngFor="let msg of messages">
    <p>{{ msg }}</p>
```

```
  </div>
</div>


<input type="text" [(ngModel)]="messageInput" placeholder="Type a message..."
/>
<button (click)="sendMessage()">Send</button>
```

Add styles (`chat.component.css`):

```
.chat-box {
  width: 100%;
  max-height: 300px;
  overflow-y: auto;
  border: 1px solid #ccc;
  padding: 10px;
  margin-bottom: 10px;
}

input {
  width: 80%;
  padding: 5px;
}

button {
```

```
  padding: 5px 10px;
  cursor: pointer;
}
```

Step 6: Run the Application

Start Express.js Server

```
node server.js
```

Run Angular Frontend

```
ng serve
```

Test the Chat

1. Open **two browser tabs** with `http://localhost:4200/chat`.
2. Send messages from one tab and see them appear in the other in real-time.

Error handling and testing

1. Error Handling in Express.js

1.1 Global Error Handling Middleware

Create a global error-handling middleware in **Express.js** (`middleware/errorHandler.js`):

```
const errorHandler = (err, req, res, next) => {
  console.error(err.stack);

  const statusCode = err.statusCode || 500;
  res.status(statusCode).json({
```

```
    success: false,
    message: err.message || "Internal Server Error"
  });
};

module.exports = errorHandler;
```

1.2 Implement Middleware in `server.js`

Import and use the middleware:

```
const express = require("express");
const cors = require("cors");
const errorHandler = require("./middleware/errorHandler");

const app = express();
app.use(express.json());
app.use(cors());

// Routes
app.use("/auth", require("./routes/auth"));

// Global error handler (should be the last middleware)
app.use(errorHandler);

const PORT = 5000;
app.listen(PORT, () => console.log(`Server running on port ${PORT}`));
```

1.3 Use `try...catch` for API Error Handling

Modify the `auth.js` file:

```
router.post("/login", async (req, res, next) => {
  try {
    const { email, password } = req.body;
    const user = await User.findOne({ email });
```

```
    if (!user) {
      const error = new Error("User not found");
      error.statusCode = 404;
      throw error;
    }

    const isMatch = await bcrypt.compare(password, user.password);
    if (!isMatch) {
      const error = new Error("Invalid credentials");
      error.statusCode = 401;
      throw error;
    }

    const token = jwt.sign({ userId: user._id }, "your_secret_key", { expiresIn:
"1h" });
    res.json({ token });
  } catch (error) {
    next(error);
  }
});
```

2. Error Handling in Angular

2.1 Create an Error Interceptor

Create `error.interceptor.ts` to catch API errors in Angular.

```
import { Injectable } from '@angular/core';
import { HttpInterceptor, HttpRequest, HttpHandler, HttpErrorResponse }
from '@angular/common/http';
import { catchError } from 'rxjs/operators';
import { throwError } from 'rxjs';

@Injectable()
export class ErrorInterceptor implements HttpInterceptor {
  intercept(req: HttpRequest<any>, next: HttpHandler) {
    return next.handle(req).pipe(
```

```
      catchError((error: HttpErrorResponse) => {
        let errorMessage = 'An unknown error occurred!';

        if (error.error.message) {
          errorMessage = error.error.message;
        } else if (error.status === 0) {
          errorMessage = 'Network error! Please check your connection.';
        }

        alert(errorMessage); // Show error alert
        return throwError(() => new Error(errorMessage));
      })
    );
  }
}
```

2.2 Register Interceptor in `app.module.ts`

Modify `app.module.ts`:

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

providers: [
  { provide: HTTP_INTERCEPTORS, useClass:
ErrorInterceptor, multi: true }
]
```

3. Testing in MEAN Stack

3.1 Testing Express.js APIs

Install **Jest & Supertest** for backend testing:

```
npm install --save-dev jest supertest mongoose-memory-
server
```

3.1.1 Setup Jest in `package.json`

Modify `package.json` to include:

```
"scripts": {
  "test": "jest"
},
"jest": {
  "testEnvironment": "node"
}
```

3.1.2 Write API Tests

Create `tests/auth.test.js`:

```
const request = require("supertest");
const app = require("../server");

describe("Authentication API", () => {
  it("should return 404 for non-existing user", async ()
=> {
    const res = await
request(app).post("/auth/login").send({
      email: "test@example.com",
      password: "password123"
    });

    expect(res.statusCode).toEqual(404);
    expect(res.body.message).toEqual("User not found");
  });
});
```

3.1.3 Run Tests

```
npm test
```

3.2 Testing Angular Components

3.2.1 Setup Angular Testing

Angular uses **Jasmine & Karma** by default. Run:

```
ng test
```

3.2.2 Writing Unit Tests for Services

Modify `auth.service.spec.ts`:

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

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

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

    service = TestBed.inject(AuthService);
    httpMock = TestBed.inject(HttpTestingController);
  });

  it('should call login API and return token', () => {
    const mockResponse = { token: '12345' };

    service.login({ email: 'test@example.com', password:
'password' }).subscribe(res => {
      expect(res.token).toEqual('12345');
    });

    const req =
httpMock.expectOne('http://localhost:5000/auth/login');
    expect(req.request.method).toBe('POST');
    req.flush(mockResponse);
  });
```

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

4. Running Angular Tests

```
ng test
```

- Runs all **unit tests** and **integration tests**.
- Shows **code coverage**.