Web Application Architectures

Lab 3 - Create an API offering authentication and authorization

# **Step 1 - Create the project**

**Create a new project folder:** auth-api-ts

Open the folder in VS Code.

**Open a terminal window and run:**

npm init -y

npm install express cors jsonwebtoken

npm install --save-dev typescript ts-node @types/node @types/express @types/cors @types/jsonwebtoken

**Initialize TypeScript:**

|  |  |
| --- | --- |
| In Terminal run:  npx tsc –init  This will create tsconfig.json  In tsconfig.json, enable:  {  "compilerOptions": {  "target": "ES2020",  "module": "CommonJS",  "strict": true,  "esModuleInterop": true,  "outDir": "dist"  }  } |  |

# **Step 2 - Basic Express server**

|  |  |
| --- | --- |
| In the project folder create a new folder: src |  |

**In src directory create a file named server.ts**

**Add the following to server.ts:**

|  |  |
| --- | --- |
| import express, { Request, Response } from "express";  const app = express();  app.get("/", (req: Request, res: Response) => {  res.send("Hello World with TypeScript!");  });  app.listen(8000, () => {  console.log("Server running at http://localhost:8000");  }); |  |

This program sets up a very simple web server using Express and TypeScript. It first brings in the Express library along with type definitions for requests and responses. Then, it initializes an Express application.

A single route is defined that listens for GET requests on the root path (/). When someone visits this path in the browser, the server responds with the text “Hello World with TypeScript!”.

Finally, the application starts listening on port 8000, and once it’s running, a message is printed to the console confirming that the server is available at http://localhost:8000.

**Run:** npx ts-node src/server.ts

In a web browser open: localhost:8000

A screen shot of a computer

AI-generated content may be incorrect.

# **Step 3. Middleware (CORS & JSON parsing)**

As this API will be used by different applications/domains we will need to enable CORS.

|  |  |
| --- | --- |
| **Add the following to server.ts:**  import cors from "cors";  **After const app = express(); add the following:**  app.use(cors());  app.use(express.json()); |  |

This step adds middleware to the server. The CORS middleware allows the API to accept requests from different origins (useful when the frontend and backend run on different domains or ports). The JSON parser middleware enables the server to automatically read and process incoming request bodies formatted as JSON.

**Test the app:**

Stop the server if it is running (ctrl + c)

**Run:** npx ts-node src/server.ts

In a web browser open: localhost:8000

Make sure there are no error in the terminal window

A screen shot of a computer

AI-generated content may be incorrect.

# **Step 4. Define user type and mock database**

**Add the following to server.ts:**

|  |  |
| --- | --- |
| interface User {  id: number;  email: string;  username: string;  role: string;  password: string;  }  const users: User[] = [  {  id: 1,  email: "admin@mail.com",  username: "admin",  role: "admin",  password: "123123"  }  ]; |  |

This part defines a User interface to describe the structure of a user object, ensuring each user has an ID, email, username, role, and password. It then creates a small in-memory list of users, starting with a single sample user (an admin account).

**Test the app:**

Stop the server if it is running (ctrl + c)

**Run:** npx ts-node src/server.ts

In a web browser open: localhost:8000

Make sure there are no error in the terminal window

A screen shot of a computer

AI-generated content may be incorrect.

# **Step 5. Login route with JWT**

**Add the following to server.js:**

|  |  |
| --- | --- |
| import jwt from "jsonwebtoken";  const SECRET = "shhhhh"; // later: move to process.env  app.post("/api/login", (req: Request, res: Response) => {  const { email, password } = req.body as {  email: string;  password: string;  };  const foundUser = users.find((user) => user.email === email);  if (!foundUser || foundUser.password !== password) {  return res.status(400).json({ message: "Invalid credentials" });  }  const token = jwt.sign({ id: foundUser.id }, SECRET);  res.json({ token });  }); |  |

This section introduces JWT authentication. It imports the jsonwebtoken library and defines a secret key for signing tokens. A new login endpoint is added, where the server checks the submitted email and password against the stored users. If the credentials are invalid, it responds with an error. If valid, it generates a signed JWT containing the user’s ID and returns that token to the client, which can be used for future authenticated requests.

**Test the app:**

Stop the server if it is running (ctrl + c)

**Run:** npx ts-node src/server.ts

In a web browser open: localhost:8000

Make sure there are no error in the terminal window

# **Step 6. JWT verification middleware**

**Add the following to server.js:**

|  |  |
| --- | --- |
| import { NextFunction } from "express";  interface AuthRequest extends Request {  user?: any;  }  const verifyJWT = (req: AuthRequest, res: Response, next: NextFunction) => {  const authHeader = req.headers.authorization;  if (!authHeader?.startsWith("Bearer ")) {  return res.status(401).json({ message: "Unauthorized" });  }  const token = authHeader.split(" ")[1];  try {  const decoded = jwt.verify(token, SECRET);  req.user = decoded;  next();  } catch {  res.status(401).json({ message: "Unauthorized" });  }  }; |  |

This part defines a middleware function for verifying JSON Web Tokens. It extends the request object so it can carry a user property once a token is validated. The middleware checks whether the request includes an Authorization header starting with “Bearer”. If no valid token is found, it rejects the request with an Unauthorized response. If a token is present, it verifies it using the secret key; when valid, the decoded payload is attached to the request and the request continues to the next handler.

**Test the app:**

Stop the server if it is running (ctrl + c)

**Run:** npx ts-node src/server.ts

In a web browser open: localhost:8000

Make sure there are no error in the terminal window

# **Step 7. Protected route**

**Add the following to server.js:**

|  |  |
| --- | --- |
| app.get("/api/protected", verifyJWT, (req: AuthRequest, res: Response) => {  res.json({ message: "Welcome, authorized user!", user: req.user });  }); |  |

This defines a protected API route. It uses the JWT verification middleware so only requests with a valid token can access it. When the request is authorized, the server responds with a success message along with the decoded user information from the token.

**Test the app:**

Stop the server if it is running (ctrl + c)

**Run:** npx ts-node src/server.ts

In a web browser open: localhost:8000

Make sure there are no error in the terminal window

# **Step 8. “Users” route**

**Add the following route to server.ts**

|  |  |
| --- | --- |
| app.get("/api/users", verifyJWT, (req: AuthRequest, res: Response) => {  const user = users.find((user) => user.id === req.user.id);  if (!user) return res.status(404).json({ message: "User not found" });  const { password, ...userWithoutPassword } = user;  res.json(userWithoutPassword);  }); |  |

This route provides an endpoint to retrieve the currently logged-in user’s information. It is protected by the JWT middleware, so only requests with a valid token are allowed. Once authorized, the server looks up the user in the in-memory list using the ID from the token. If the user exists, their details are returned with the password removed; if not, a “User not found” error is sent back.

**Test the app:**

Stop the server if it is running (ctrl + c)

**Run:** npx ts-node src/server.ts

In a web browser open: localhost:8000

Make sure there are no error in the terminal window

A screen shot of a computer

AI-generated content may be incorrect.

# **Step 9. Compile and run**

**Build with:**

npx tsc

**Run compiled code:**

node dist/server.js

**Or run directly in dev mode:**

npx ts-node src/server.ts

What we now have:

A TypeScript Express API with:

Login route → returns JWT

Middleware for authentication

Protected route

/users route for user info

Type safety for req.body, users, and middleware.

# **Step 10. Test with Postman**

**Test /api/login**

Method: POST

URL: http://localhost:8000/api/login

Headers:

Content-Type: application/json

Body (raw JSON):

{

"email": "admin@mail.com",

"password": "password"

}

A screenshot of a computer

AI-generated content may be incorrect.

**Test /api/users**

Method: GET

URL: http://localhost:8000/api/users

Headers:

Authorization: Bearer <your\_token\_from\_login>

A screenshot of a computer

AI-generated content may be incorrect.

# **Observing Authorization**

Without login: Requests to /api/protected or /api/users fail.

After login: Requests succeed using the token.

Purpose: This demonstrates authorization — only authenticated users with a valid JWT can access protected resources.