Task Statement: Express + MySQL Auth API (4 endpoints)

Objective

Build a small Node.js/Express REST API with **four endpoints**:

1. **GET /** – public homepage that returns static JSON.
2. **POST /auth/register** – user registration (MySQL).
3. **POST /auth/login** – user login (email/password) → returns a **JWT**.
4. **GET /profile** – protected route using an **auth middleware** that validates the JWT and returns the user's profile.

References: Express official docs; MDN Express intro.

Tech requirements

- **Runtime:** Node.js (LTS).
- **Framework:** Express.
- **DB Driver:** mysql2 (promise API).
- **Validation:** joi for request body schemas.
- **Password Hashing:** bcrypt (hash & compare).
- **Auth tokens:** jsonwebtoken for issuing/verifying JWTs.

Data model (MySQL)

Create a users table with these columns:

- id (PK, auto-increment INT)
- name (VARCHAR 100, required)
- email (VARCHAR 255, required, **UNIQUE**)
- password_hash (VARCHAR 255, required) – store **bcrypt hash**, never the raw password
- age (INT, required, e.g., 13–120)
- created_at (TIMESTAMP default CURRENT_TIMESTAMP)

Driver: use mysql2 with promises for queries and prepared statements.

Validation rules (with Joi)

For **registration** (POST /auth/register):

- name: string, 2–100 chars, required
- email: valid email, required
- password: string, min 8 chars, required (you'll hash it before saving)
- age: integer, min 13, max 120, required

For **login** (POST /auth/login):

- email: valid email, required
- password: string, required

Implement with joi schemas and return **400 Bad Request** on validation errors.

Endpoint specs

1) GET

/

- **Access:** public
- **Response (200):** { "message": "Welcome to the API" } (or similar static content)

2) POST

/auth/register

- **Access:** public
- **Body:** { name, email, password, age } (validated with Joi)
- **Process:**
    - Validate input (Joi).
    - Check if email already exists (return 409 if it does).
    - Hash password using **bcrypt** with a sensible salt rounds value (e.g., 10–12).
    - Insert user with password_hash.
- **Responses:**
    - 201 Created with a minimal user object (exclude password fields).
    - Errors: 400 (validation), 409 (email in use), 500 (server).

      References: bcrypt hashing & compare.

3) POST

/auth/login

- **Access:** public
- **Body:** { email, password } (validated with Joi)

- **Process:**
    - Validate input (Joi).
    - Find user by email; if not found, return 401 Unauthorized.
    - Compare provided password with stored password_hash using **bcrypt.compare**.
    - If match, sign a **JWT** (e.g., payload { id, email }) using jsonwebtoken, with expiry (e.g., 1h).
- **Responses:**
    - 200 OK: { token }
    - Errors: 400, 401 (bad creds), 500.

      References: jsonwebtoken usage & JWT libraries.

4) GET

/profile

- **Access: protected**
- **Auth:** Bearer token in Authorization: Bearer <token>
- **Middleware flow:**
    1. Read Authorization header.
    2. Verify JWT with jsonwebtoken.verify and your secret.
    3. On success, attach req.user (e.g., { id, email }) and continue.
    4. On failure/missing token, return 401 Unauthorized.
- **Response (200):** return the user profile (at least: id, name, email, age, created_at).

  References: jsonwebtoken verify patterns.

Middleware:

auth

Create auth middleware that:

- Extracts token from Authorization header (format: Bearer <token>).
- Verifies token using jsonwebtoken; on success sets req.user; on failure sends 401.