Recruitment Platform - API & Architecture Documentation

1) Project Overview & Architecture

I built a simple recruitment platform prototype with a React frontend and an Express backend. The goal was to implement user registration, login, and profile display using JWT-based authentication.

The project structure is small but organised:

```
server/
 controllers/authentication.controller.js # Handles registration, login
 routes/authentication.route.js
                                           # Auth API routes
 models/user.model.js
                                          # User schema with validation
 utils/authentication.js
                                          # JWT helpers
 utils/database.js
                                           # MongoDB connection
 server.js
                                           # Server setup
client/
 src/pages/Register.jsx
                                          # Registration page
 src/pages/Login.jsx
                                          # Login page
                                           # Profile page
 src/pages/Dashboard.jsx
                                           # Routes
 src/App.jsx
 src/main.jsx
                                           # App bootstrap
```

Why I structured it this way:

- Keep controllers separate from routes to make the code clean and reusable.
- Keep models and utils in separate folders for clarity and scalability.
- Frontend has separate pages for each feature, so it's easy to navigate.

2) Authentication Flow & Security

The authentication is simple but secure:

- 1. Passwords are hashed with bcrypt before saving in the database.
- 2. When a user logs in, we compare the password using bcrypt.compare.
- 3. We generate a JWT with only the user ID, so sensitive info is never in the token.
- 4. The JWT is stored as an httpOnly cookie so it cannot be accessed by JavaScript.
- 5. The token is valid for 1 hour.
- 6. There's a /authenticate-me endpoint that checks the token and fetches user info when dashboard page is accessed.

Sequence:

- 1. User registers \rightarrow inputs validated \rightarrow password hashed \rightarrow user saved.
- 2. User logs in → credentials checked → JWT generated → httpOnly cookie set.



- 3. User visits dashboard \rightarrow token verified \rightarrow user data returned.
- 4. Logout → cookie cleared.

3) API Endpoints

Base URL: \${VITE_BACKEND_URL} (dev: http://localhost:3000)

POST /api/auth/register

• Request:

```
{
  "name": "Arman Ul Haq",
  "email": "arman@gmail.com",
  "password": "#1Strong@123"
}
```

• Responses:

- o 201: User registered successfully
- 400: Validation error or duplicate email
- o 500: Server error

POST /api/auth/login

• Request:

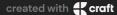
```
{
  "email": "arman@gmail.com",
  "password": "#1Strong@123"
}
```

• Responses:

- o 200: Login successful + sets token httpOnly cookie
- 401: Invalid credentials
- o 500: Server error

GET /api/auth/authenticate-me

- Requires cookie token.
- Responses:
 - 200: Returns user info { name, email }
 - 401: Unauthorised if token is missing/invalid
 - o 500: Server error



POST /api/auth/logout

- Clears the cookie.
- Response: 200: Logout successful

4) Database Schema

I used Mongoose as ODM for MongoDB. It provides a simple, schema-based solution to model my application data.

```
const userSchema = new mongoose.Schema({
    name: {
        type: String,
        trim: true,
        required: [true, "Name is required"],
    },
    email: {
        type: String,
        lowercase: true,
        trim: true,
        required: [true, "Email is required"],
        unique: true,
    },
    password: {
        type: String,
        trim: true,
        required: [true, "Password is required"],
    },
},{timestamps: true,});
```

- Email is unique and normalised.
- Timestamps automatically track when users registers.

5) Error Handling

- Registration: checks for name length, email format, password strength.
- Login: invalid credentials → sends 401.
- Token validation: invalid or missing token → 401.
- Server errors → 500, with logs on backend.

Client UX:

• Forms show instant validation errors.

6) Scaling & Improvements

If I were to extend this prototype:



• Security:

• Role-based access control for recruiters/candidates.

• Features:

- Add profiles with bio, location, skills.
- Add jobs, applications, and saved jobs collections.

• Frontend:

- Global auth context for protected routes.
- Better UX for server errors (toasts, retry).