ECE
ÉCOLE D'INGÉNIEURS
ENGINEERING SCHOOL

Title:

Design and Implementation of a Secure REST API Using Node.js, MongoDB, and JWT

Course:

Distributed Systems Security

Institution: ECE Paris

Instructor:

Yaya Doumbia

Academic Year: 2025-2026

Group Member:

OFUNNEKA JENNIFER OKONKWOABUTU

DATE: 15th January 2026

TABLE OF CONTENTS

1.0 INTRODUCTION

1.1 Purpose of this Project

The purpose of this lab is to design, implement, and document a secure RESTful API capable of managing two related entities: Equipes and Players.

For this project:

- Users can create, read, update, and delete(CRUD) equipe and player records.
- Entity relationships are maintained: each player is linked to a team(equipe) using MongoDB ObjectID references.
- Routes allow fetching all players for a team and finding the team of a given player.
- Users must authenticate before modifying or deleting resources via JWT.
- Search functionality for players by name is prepared for future implementation.

This report is written as an operational document, such that it can be followed step-by-step to reproduce the system from scratch.

1.2 Objectives of the project

Functional Objectives:

1. Create a REST API using Node.js and Express.
2. Store data using MongoDB.
3. Implement full CRUD operations.
4. Manage relationships between two entities (teams and players).
5. Secure write operations using JWT authentication.

1.3 Source Code Reference

The complete source code for this project is available in a public Github repository and can be used to reproduce the system described in this report.

Repository Link:
https://github.com/JanieAbutu/ECE_Distributed-Systems-Security/tree/main/Projects/Lab%201_PlayerEntity_RestAPI_Project

2.0 DEVELOPMENT ENVIRONMENT CONFIGURATION

2.1 Tools and Technologies Used

Node.js:
This is used as the JavaScript runtime to execute server-side code. Node.js comes with the npm package manager which is used to manage dependencies and run scripts.

Express.js
This provides routing, middleware support, and HTTP request handling.

MongoDB
This is a NoSQL document-oriented database used for data persistence.

Mongoose
This is an Object Data Modeling (ODM) library that provides schemas, validation, and database abstraction.

JSON Web Token (JWT)
This is used for stateless authentication and authorization.

Postman
This is used to manually test and verify API endpoints.

Summary:
Each tool was selected to address a specific responsibility within the system.

2.2 Environment setup

This section describes how the development environment was configured before starting the project. Proper environment setup is critical to ensure that the REST API runs correctly, dependencies are managed consistently, and errors related to tooling are avoided.

Step 1: Install Required Software

1. **Node.js includes npm (Node Package Manager):**
- Download LTS version (recommended) from https://nodejs.org/en/ .

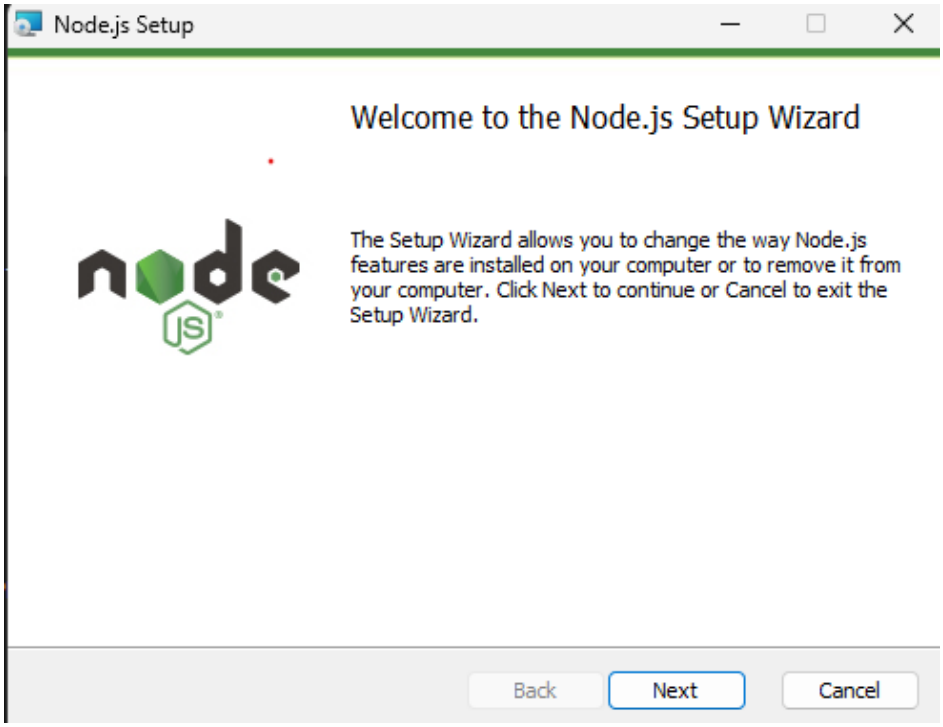

Fig 1: Node.js installation interface

- Verify the installation of Node.js by running the commands below on command prompt.

node -v

npm -v

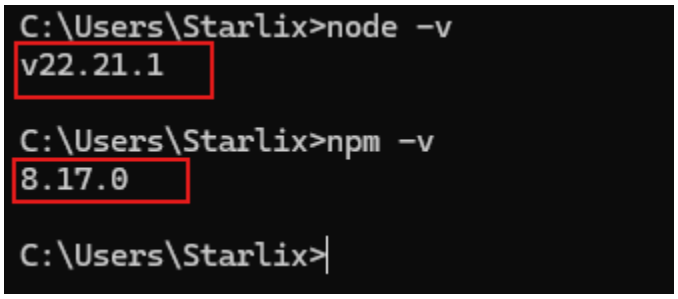

Fig 2: Output of command verifying version of node.js and npm installed

2. MongoDB Community Server

- Download MongoDB Community Server from https://www.mongodb.com/ and install.

-

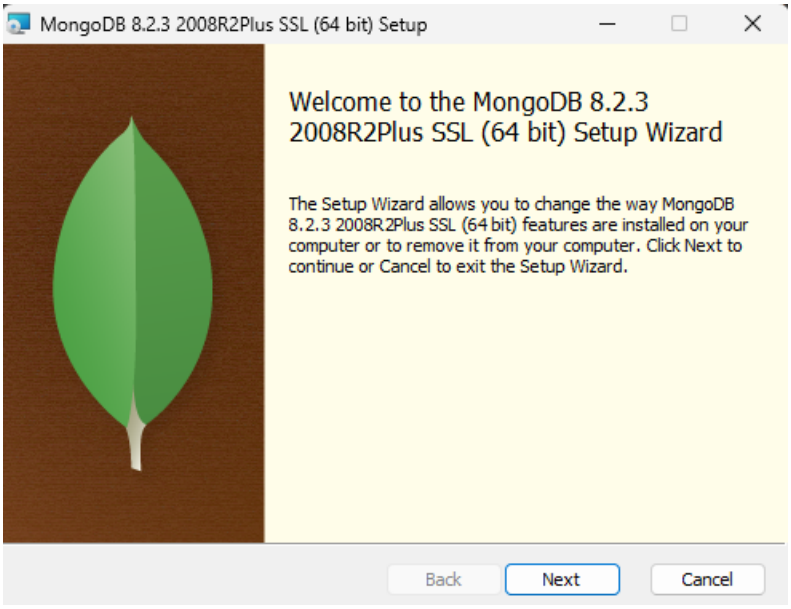

Fig 3:MongoDB Community Server installation interface

-

- The **MongoDB Compass** installs immediately after this from the same package. It provides the graphical interface to visualise and manage database data as seen below.

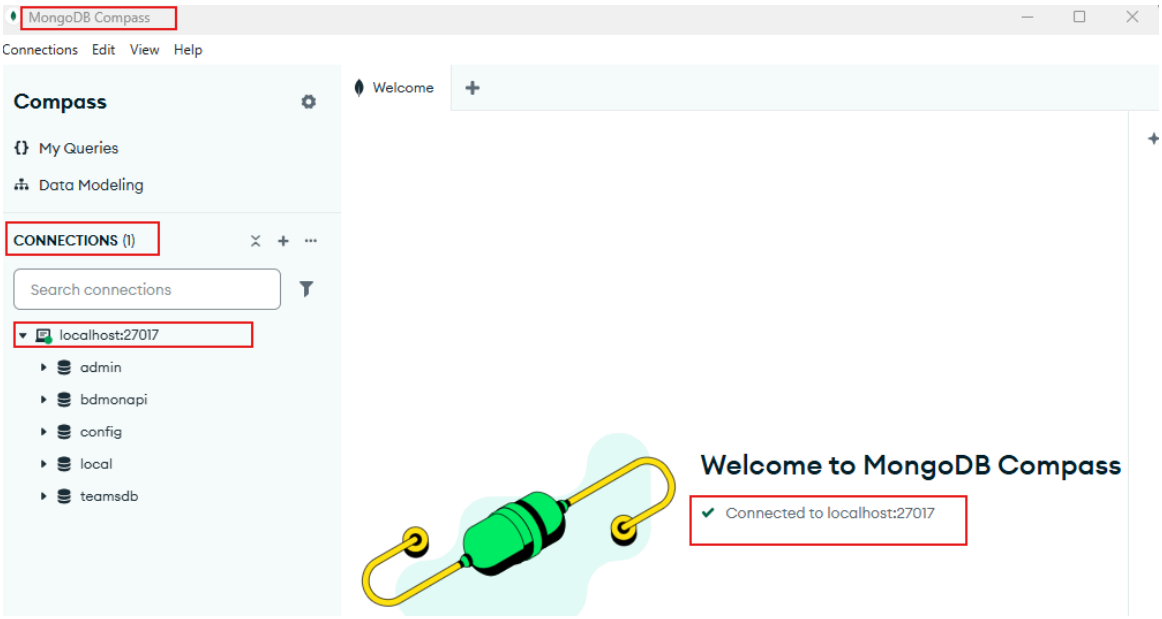

Fig 4:MongoDB Compass interface

- After installation, open MongoDB Compass, then click on 'Localhost:27017' to connect. Successful connection will be confirmed as seen in above image.

3. **Postman**

- Download the postman application from https://www.postman.com/ and install. Register, login, and create a workspace to get started.

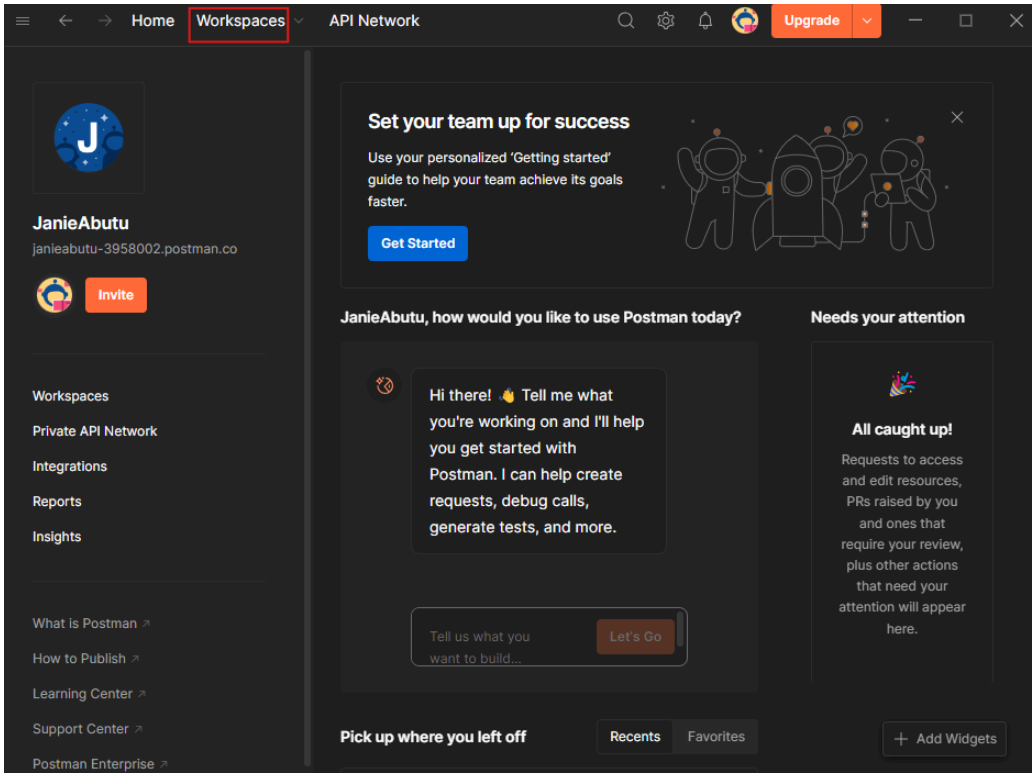

Fig 5:Postman interface

Step 2: Create the Project Workspace

- Create a dedicated folder to contain all application files, all source code, configuration files, and dependencies for one Node.js project. The name of the folder created for this project is ***PlayerEntity_RestAPI_Project***.

- Open the folder in a code editor (Visual Studio - VS Code recommended).

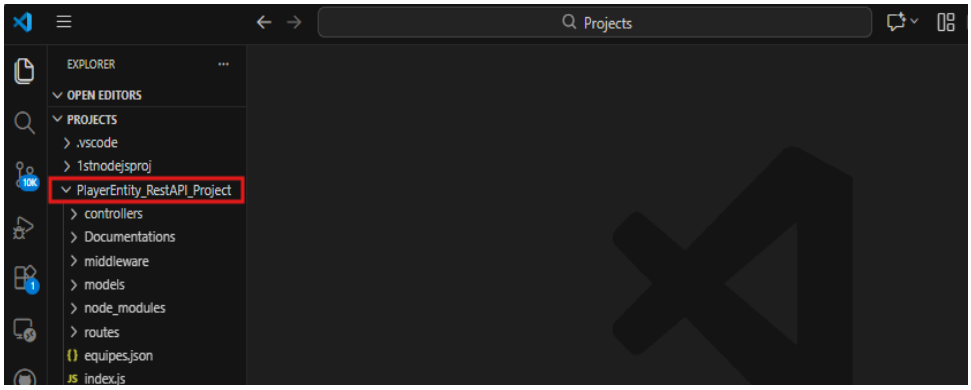

Fig 6:Folder ***PlayerEntity_RestAPI_Project*** opened on VS Code

- On the terminal on VS Code, navigate to the project folder as seen below. This is to ensure that all activities and dependencies to be installed will be done within the project.

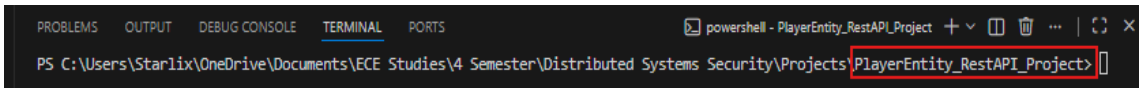

Fig 7: Navigated into the directory ***PlayerEntity_RestAPI_Project*** opened on VS Code terminal

Step 3: Initialize Node.js Project

- On the VS Code terminal navigated to the project directory, run the command **npm init.**
- This initializes the Node.js project and generates a package.json file which describes the project, lists the installed dependencies, and defines the application entry point as seen below

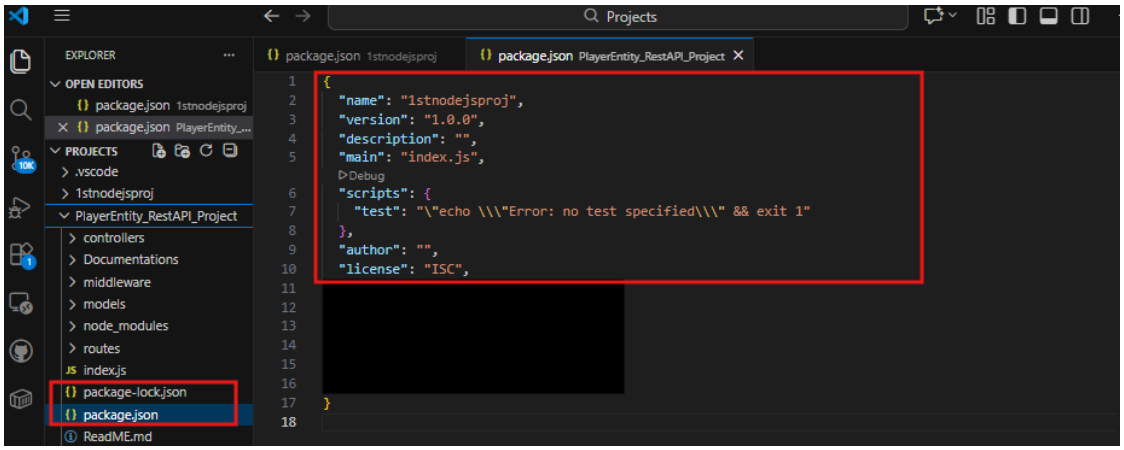

Fig 8: package.json file generated after initialization

Step 4: Install Project Dependencies

- The following dependencies are installed locally for the project.
  - express, a web framework
  - mongoose, mongoDB ODM
  - jsonwebtoken, jwt authentication
  - bcrypt, password hashing

- The dependencies are stored in node_modules/ and referenced in package.json as seen below:

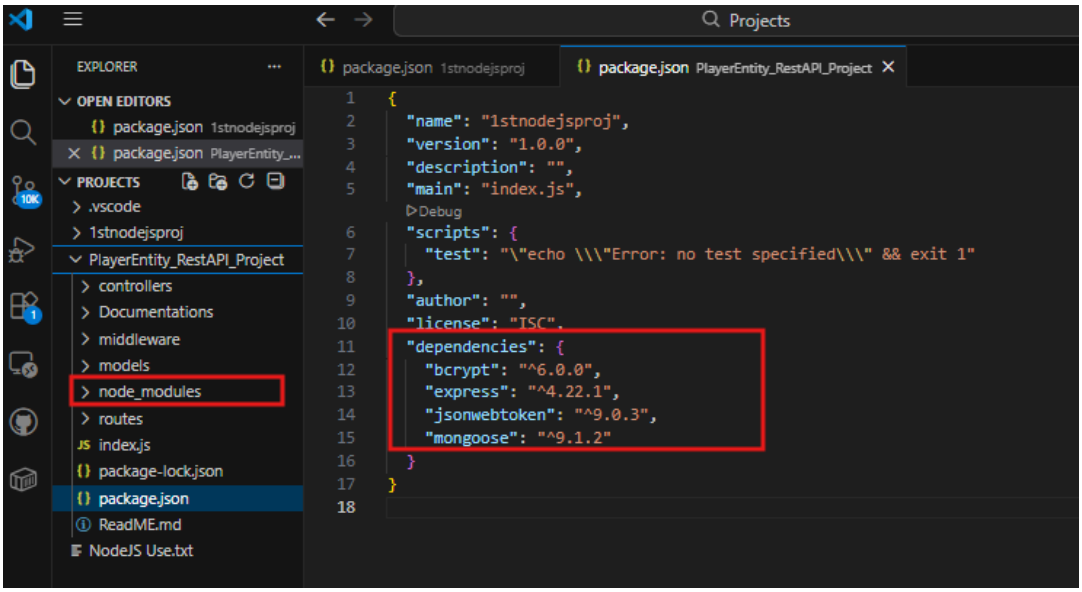

Fig 9: node_modules and Installed dependencies referenced in the package.json

Step 5: Verify Environment Configuration

- The command node index.js was run to test the configuration.
- Node.js server running without errors, mongoDB connecting as seen below confirms the environment configuration was successfully done.

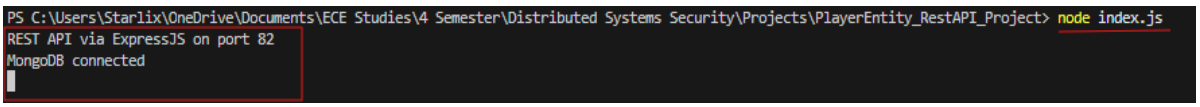

Fig 10: Successful server and database connection

2.3 Conclusion

This configuration is essential before API implementation as it ensures that errors are minimized during development, and that the application can be executed and tested reliably.

3.0 GLOBAL SYSTEM ARCHITECTURE

This section explains how responsibilities are divided inside the application for structure, clarity, and to enforce clean code organisation and predictable behaviour.

3.1 Architectural Layers

1. Client (Postman)
2. Routes (HTTP end points)
3. Middleware (Security checks)
4. Controllers (Business logic)
5. Models (Data structure and validation)
6. Database (Persistent storage)

3.2 Project Structure

Action: Manually create the following folders and files in the order below:

```
project/
├── models/                      # Database structure (schemas)
│   ├── Equipe.js                # Team schema
│   ├── Player.js                # Player schema
│   └── User.js                  # User model for authentication
│
├── controllers/                 # Business rules/logic
│   ├── equipe.js                # Team logic
│   ├── playerController.js      # Player logic
│   └── authController.js  # Login/signup logic
│
├── routes/                       # HTTP routing (API endpoints)
│   ├── equipe.js                # Team routes
│   ├── players.js               # Player routes
│   └── auth.js                  # Login/signup routes
│
├── middleware/                   # Security and request filtering
│   └── auth.js                  # JWT verification middleware
│
├── index.js                      # Application entry point
└── package.json                 # Dependencies
```

3.3 Conclusion

The layered architecture and structure ensures clarity, maintainability, request handling, and reduced complexity.

4.0 DATABASE DESIGN AND DATA MODELING

The database layer stores all application data persistently.

4.1 Entities

Equipe (Team)

- ➔ Id (Automatically assigned by MongoDB)
- ➔ Name
- ➔ country

Player

- ➔ Id (Automatically assigned by MongoDB)
- ➔ name
- ➔ number
- ➔ position
- ➔ teamId (reference to Equipe)

Relationships

- ➔ One Equipe → Many Players
- ➔ Player references Equipe using MongoDB ObjectId

4.2 Database Creation and Setup

Step 1: Database Creation

The database is created automatically when the application connects. MongoDB creates the database.

Step 2: Collections Creation

Collections are automatically created by Mongoose when documents are inserted:

- ➔ equipe
- ➔ player
- ➔ users

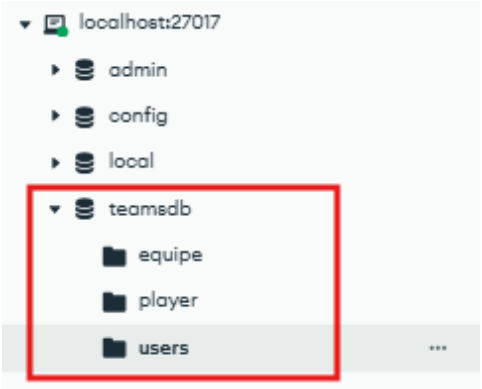

Fig 4.2: Created Database and Collections on MongoDB

4.3 Implement Database Models (Schemas)

Objective: Define how data is stored and validated in MongoDB.

Step 1: Team Model (Equipe.js)

Purpose - Represent a team in the database.

Actions Performed:

- Create models/Equipe.js
- Define fields: name and country

```
Equipe.js
1  const mongoose = require('mongoose');
2
3  const equipeSchema = new mongoose.Schema({
4    name: { type: String, required: true },
5    country: { type: String, required: true }
6  });
7
8  module.exports = mongoose.model('Equipe', equipeSchema, 'equipe');
9
```

Fig 4.3.1: Equipe.js Schema Code

Step 2: Player Model (Player.js)

Purpose - Represent a player and link them to a team using a reference.

Actions Performed:

- Create models/Player.js
- Define fields: name, number, position, teamId(MongoDB ObjectId reference)

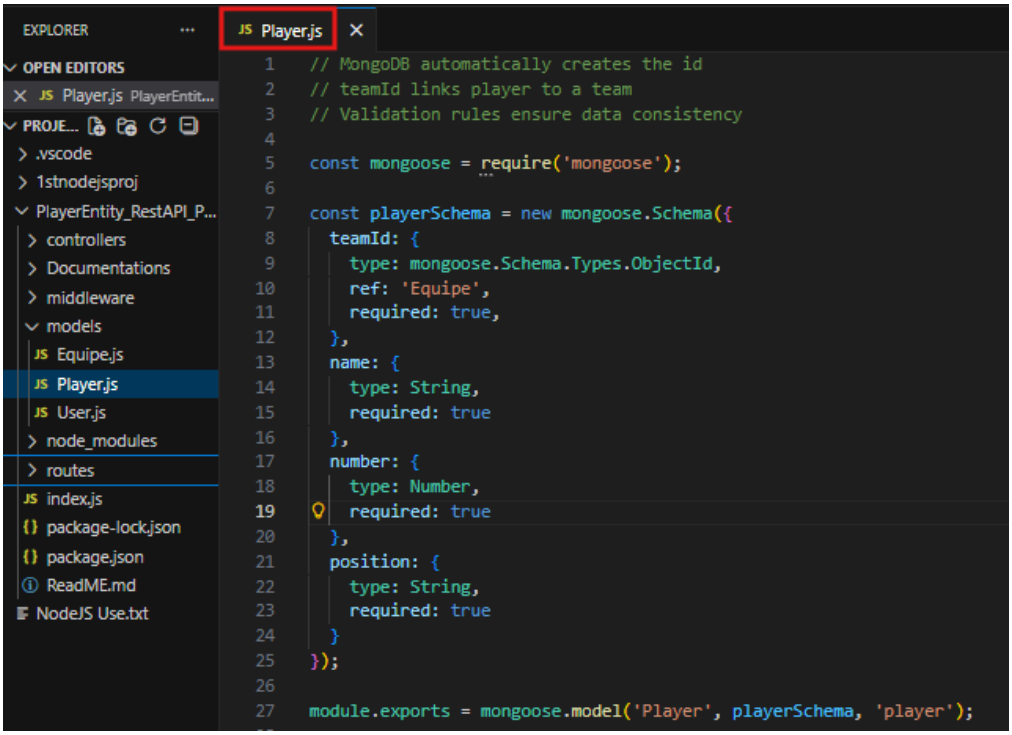

Fig 4.3.2: Player.jsSchema Code

Step 3: User Model (User.js)

Purpose - Stores users.

Actions Performed:

- Create models/User.js
- Define fields: email, password

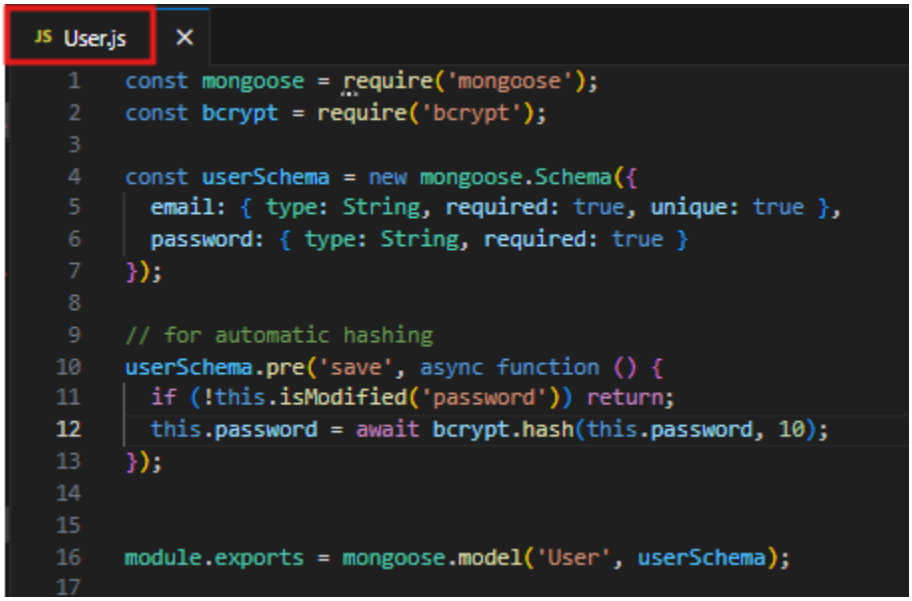

Fig 4.3.3: Equipe.js Schema Code

4.4 Conclusion:

MongoDB combined with Mongoose allows automatic database creation, schema validation, and scalable data storage without manual SQL setup. Models formally define the database structure and relationships. Using references ensures scalability and avoids data duplication.

5.0 APPLICATION IMPLEMENTATION AND REQUEST PROCESSING

5.1 Implement Controllers (Business Logic)

Objective: Centralize all application logic and data handling separately from routing to ensure scalability.

CRUD Operations handled by Controllers:

Create (POST) - Adds new resources to the database.

- ➔ Accepts JSON input
- ➔ Validates data
- ➔ Stores it in MongoDB

Read (GET) - Retrieves one or multiple resources.

- ➔ Does not modify data

Update (PUT) - Modifies an existing resource.

- ➔ Locates an existing resource
- ➔ Modifies existing data with new data
- ➔ Saves the modified data back to the database

Delete (DELETE)- Removes a resource permanently.

CRUD operations form the foundation of all data-driven systems and are implemented as below:

Step 1 - Team Controller (Equipe.js)

Purpose: Handles all team related operations

Actions Performed:

- Create *controller/Equipe.js*
- Implement CRUD logic for team:
  - ➔ Create team
  - ➔ Get all teams
  - ➔ Get team by ID
  - ➔ Update team
  - ➔ Delete team

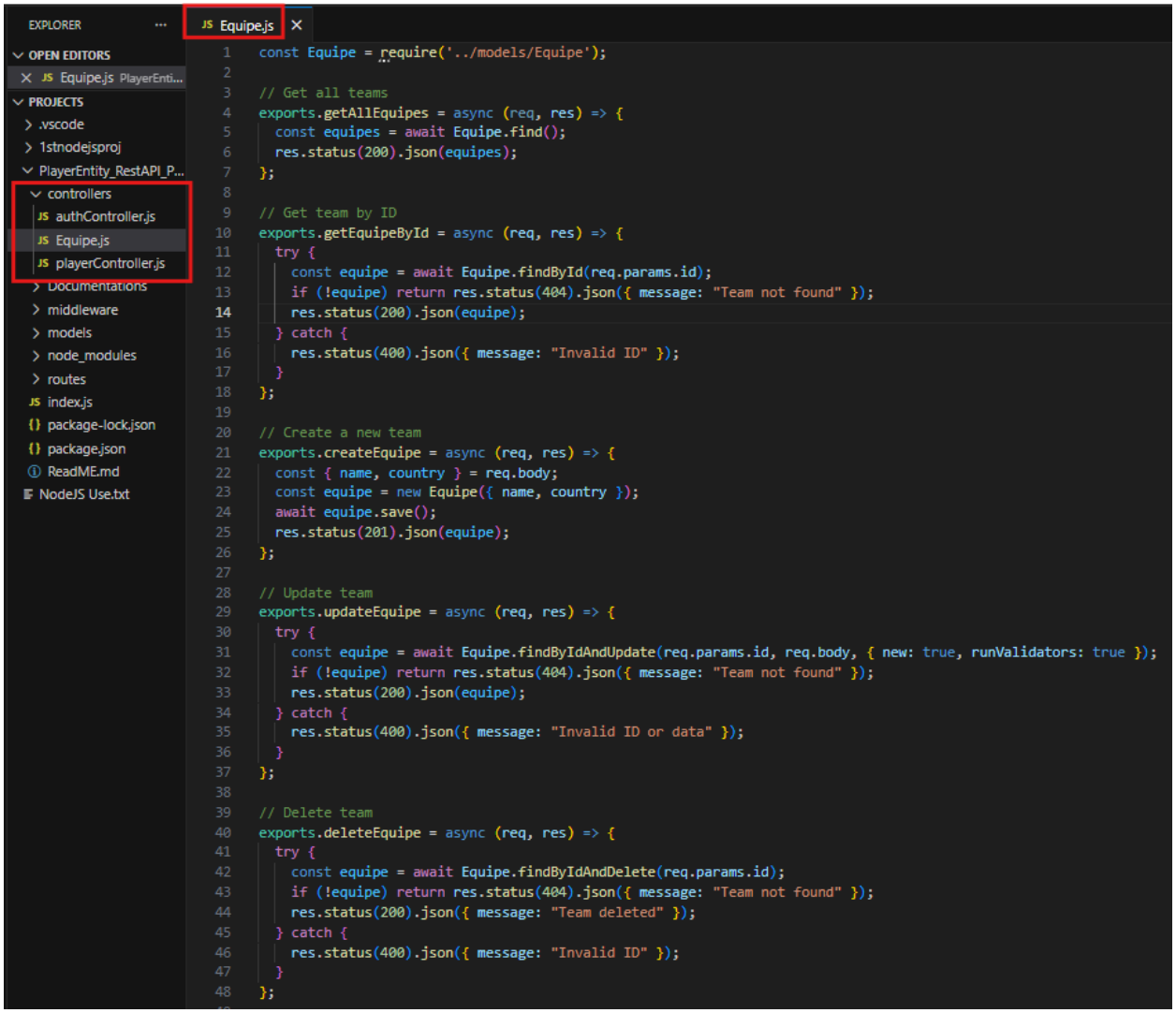

Fig 5.1.1: Equipe.js with CRUD functions

Step 2 - Player Controller (playerController.js)

Purpose: Handles players operations and entity relationships.

Actions Performed:

- Create *controller/playerController.js*
- Implement CRUD logic for players:
  - ➔ Create player
  - ➔ Get all players
  - ➔ Get player by playerID
  - ➔ Update player
  - ➔ Delete player
- Get players on a team by teamID
- Get team of a player by player ID
- Placeholder search by name

playerController.js

```
const Player = require('../models/Player');
const Equipe = require('../models/Equipe');

// Create a new player using team name instead of manually providing teamId
exports.createPlayer = async (req, res) => {
  try {
    const { teamName, name, number, position } = req.body;

    // Find the team dynamically by its name
    const team = await Equipe.findOne({ name: teamName });
    if (!team) return res.status(404).json({ message: "Team not found" });

    // Create player with the proper teamId
    const player = new Player({
      teamId: team._id,
      name,
      number,
      position
    });

    await player.save();
    res.status(201).json(player);
  } catch (err) {
    res.status(500).json({ message: "Error creating player", error: err.message });
  }
};

exports.createPlayersBulk = async (req, res) => {
  try {
    const { teamName, players } = req.body; // expect teamName + array of players

    const team = await Equipe.findOne({ name: teamName });
    if (!team) return res.status(404).json({ message: "Team not found" });

    // Add teamId to each player
    const playersWithTeam = players.map(p => ({
      ...p,
      teamId: team._id
    }));

    const result = await Player.insertMany(playersWithTeam);
    res.status(201).json(result);
  } catch (err) {
    res.status(400).json({ message: "Invalid data", error: err.message });
  }
};
```

```
// Get all players
exports.getAllPlayers = async (req, res) => {
  const players = await Player.find();
  res.status(200).json(players);
};

// Get player by ID
exports.getPlayerById = async (req, res) => {
  try {
    const player = await Player.findById(req.params.id);
    if (!player) return res.status(404).json({ message: "Player not found" });
    res.status(200).json(player);
  } catch {
    res.status(400).json({ message: "Invalid ID" });
  }
};

// Update player
exports.updatePlayer = async (req, res) => {
  try {
    const player = await Player.findByIdAndUpdate(req.params.id, req.body, { new: true, runValidators: true });
    if (!player) return res.status(404).json({ message: "Player not found" });
    res.status(200).json(player);
  } catch {
    res.status(400).json({ message: "Invalid ID or data" });
  }
};

// Delete player
exports.deletePlayer = async (req, res) => {
  try {
    const player = await Player.findByIdAndDelete(req.params.id);
    if (!player) return res.status(404).json({ message: "Player not found" });
    res.status(200).json({ message: "Player deleted" });
  } catch {
    res.status(400).json({ message: "Invalid ID" });
  }
};

// List all players on a team
exports.getPlayersByTeam = async (req, res) => {
  const players = await Player.find({ teamId: req.params.teamId });
  res.status(200).json(players);
};

// Get the team of a player
exports.getTeamByPlayer = async (req, res) => {
  try {
    const player = await Player.findById(req.params.id);
    if (!player) return res.status(404).json({ message: "Player not found" });

    const team = await Equipe.findById(player.teamId);
    if (!team) return res.status(404).json({ message: "Team not found" });

    res.status(200).json(team);
  } catch {
    res.status(400).json({ message: "Invalid ID" });
  }
};
```

```
117  // Placeholder search by name
118  exports.searchPlayerByName = async (req, res) => {
119    const { name } = req.params;
120    try {
121      const players = await Player.find({ name: { $regex: name, $options: "i" } });
122      res.status(200).json(players);
123    } catch (err) {
124      res.status(500).json({ message: "Server error", error: err });
125    }
126  };
```

Fig 5.1.2: playerController.js showing relationship queries

Step 3 - Authentication Controller (authController.js)

Authentication and Authorization was done using JWT.

Authentication Flow:

- ➔ User signs up
- ➔ Password is hashed
- ➔ User logs in
- ➔ JWT is issued
- ➔ Token is used in protected requests

Purpose: Manages authentication and token issuance.

Actions Performed:

- Create *controller/authController.js*
- Implement CRUD logic for players:
  - ➔ Signup (hash password using bcrypt)
  - ➔ Login (verify password)
  - ➔ JWT token generation

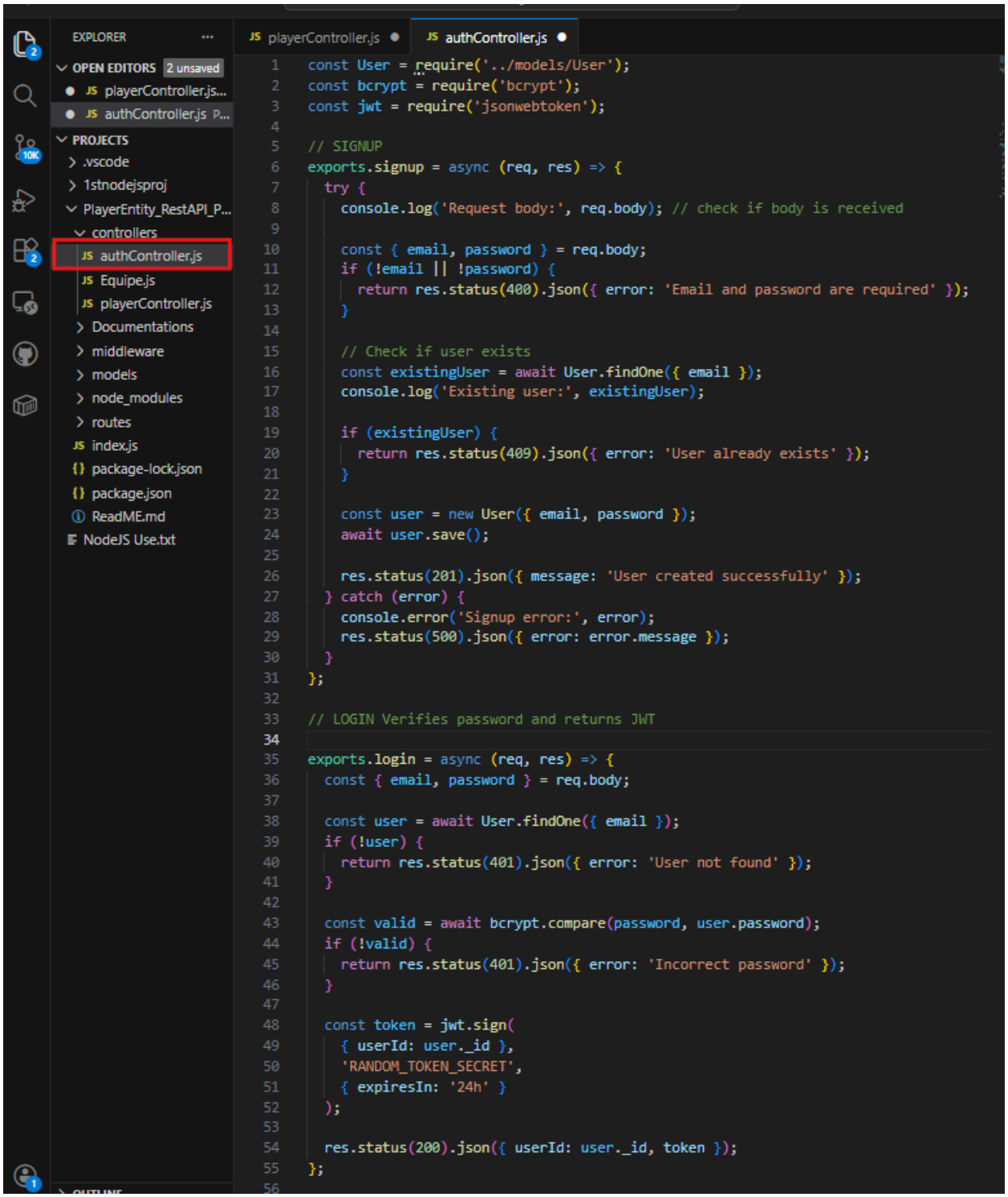

Fig 5.1.3: authController.js signup and login logic

Conclusion: Controllers cover all business rules and database interactions. Authentication controllers protect sensitive operations and identify users.Using JWT enables secure and scalable authentication.

5.2 Implement Routes (HTTP Endpoints)

Objective: Expose controllers through RESTful URLs.

Step 1 - Team Routes(Equipe.js)

Actions Performed:

- Create *routes/Equipe.js*
- Map HTTP methods to controller functions:
  - ➔ GET
  - ➔ POST
  - ➔ PUT
  - ➔ DELETE

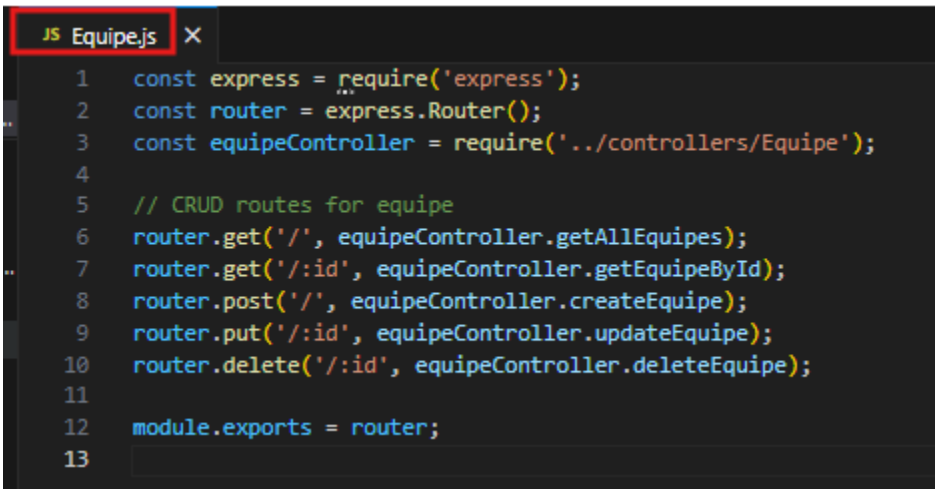

Fig 5.2.1: routes/Equipe.js

Step 2 - Player Routes(routes/players.js**)**

Actions Performed:

- Create *routes/players.js*
- Define routes for:
  - ➔ GET
  - ➔ POST
  - ➔ PUT
  - ➔ DELETE
  - ➔ Players by team
  - ➔ Team by player
  - ➔ Search placeholder

players.js

```
const express = require('express');
const router = express.Router();
const auth = require('../middleware/auth');
const playerController = require('../controllers/playerController');

// CRUD routes for players: private APIs must have the middleware-auth for protect:
router.get('/', auth, playerController.getAllPlayers);
router.get('/:id', playerController.getPlayerById);
router.post('/bulk', auth, playerController.createPlayersBulk);
router.post('/', auth, playerController.createPlayer);
router.put('/:id', auth, playerController.updatePlayer);
router.delete('/:id', auth, playerController.deletePlayer);

// Special routes
router.get('/team/:teamId', playerController.getPlayersByTeam);
router.get('/:id/team', playerController.getTeamByPlayer);
router.get('/search/:name', playerController.searchPlayerByName);

module.exports = router;
```

Fig 5.2.2:routes/players.js

Step 3 - Authentication Routes(routes/auth.js**)**

Actions Performed:

- Create *routes/auth.js*
- Define routes:
  - ➔ POST /signup
  - ➔ POST /login

auth.js

```
const express = require('express');
const router = express.Router();
const authController = require('../controllers/authController');

router.post('/signup', authController.signup);
router.post('/login', authController.login);

module.exports = router;
```

Fig 5.2.3: routes/auth.js

Conclusion:
Routes connect external HTTP requests to internal logic.

5.3 Implement Authentication Middleware

Objective: Protect sensitive routes using JWT verification.

Middleware and Security Enforcement
Purpose - Apply security checks before controllers are executed. It is responsible for token extraction, verification and request blocking or forwarding. Middleware centralizes security logic.

Actions Performed:

- Create *middleware/auth.js*
- Extract token from request header
- Verify token validity
- Allow or block request

Fig 5.2.3: middleware/auth.js

Conclusion:
This ensures that security is enforced consistently across protected routes preventing unauthorized access to them.

5.4 Implement Application Entry Point

Objective: This ties the entire application together.

Actions Performed:

- Initialize Express
- Connect to MongoDB
- Register middleware
- Register routes
- Start server on port 82

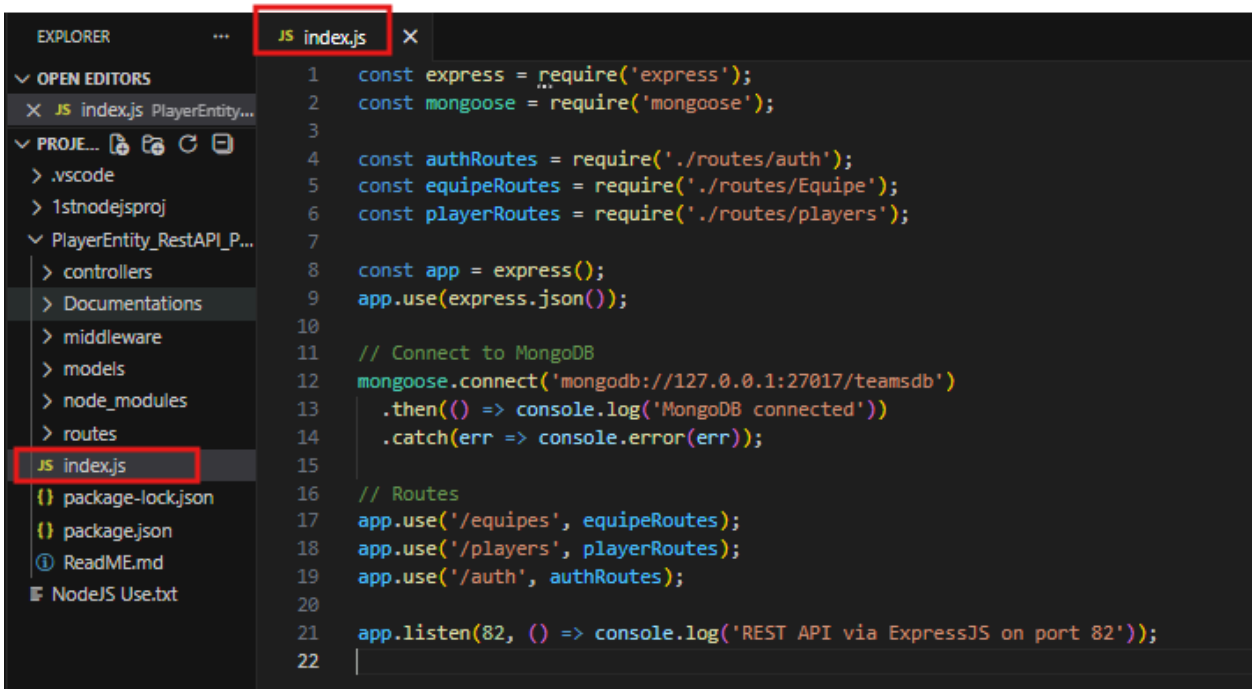

Fig 5.2.3: index.js showing app.use(), mongoose.connect(), app.listen()

Conclusion:
This ensures that the application becomes executable and accessible.

6.0 TESTING AND VERIFICATION USING POSTMAN

Purpose - Validate correctness, security, and stability of the REST API

6.1 Precondition

- Node.js server is running
- MongoDB service is running
- Collections exists

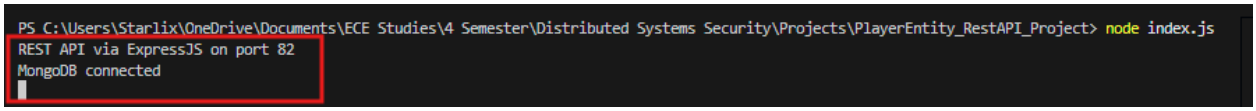

Fig 6.1: Running services

6.2 Step by Step Testing of All API Endpoints

Step 1: User Signup Test

➔ Send POST requests to /auth/signup

➔ Provide email and password

➔ **INPUT**:

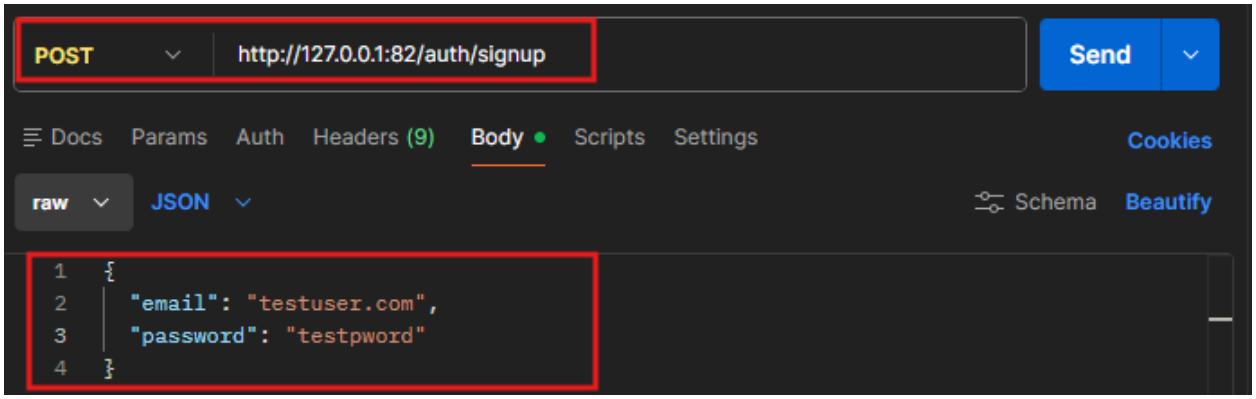

Fig 6.2.1a: Signup Request

➔ **OUTPUT:**

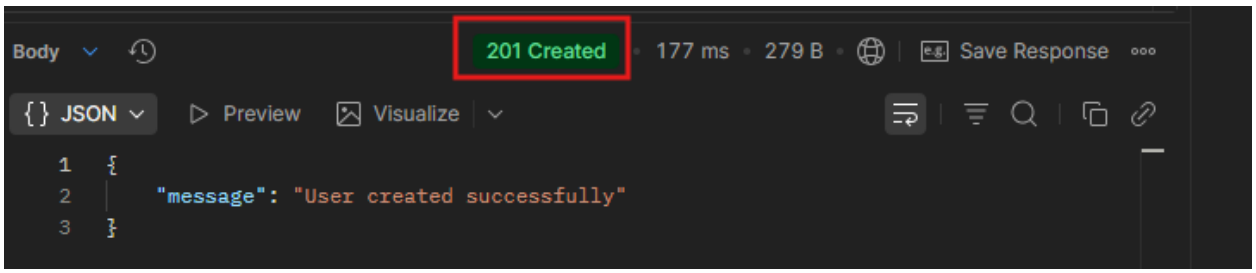

Fig 6.2.1b: Output of Signup Request (Successful user creation)

Fig 6.2.1c: Created user as saved in the database

Step 2: User Login Test

➔ Send POST requests to /auth/login

➔ Receive JWT token

➔ **INPUT**:

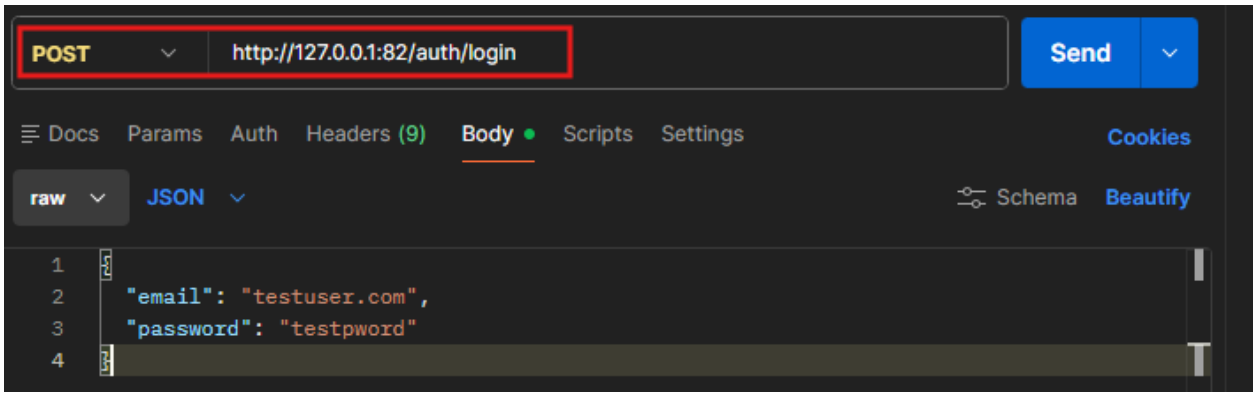

Fig 6.2.2a: Login request using created user credentials

➔ **OUTPUT:**

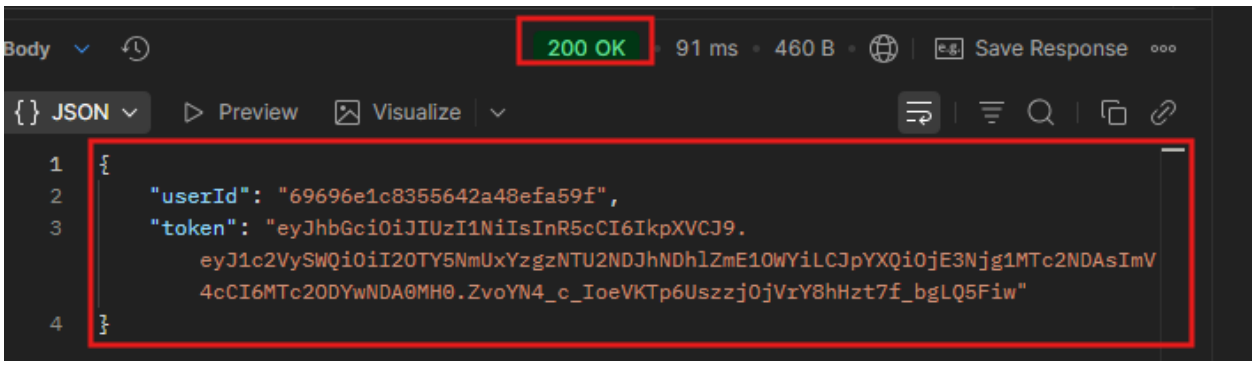

Fig 6.2.2b: Login response containing JWT

Step 3: Access Protected Route Without Token (Create Player)

➔ Call POST /players without Authorization header

➔ **INPUT:**
Attempted to create a player through a protected route without an authorization header.

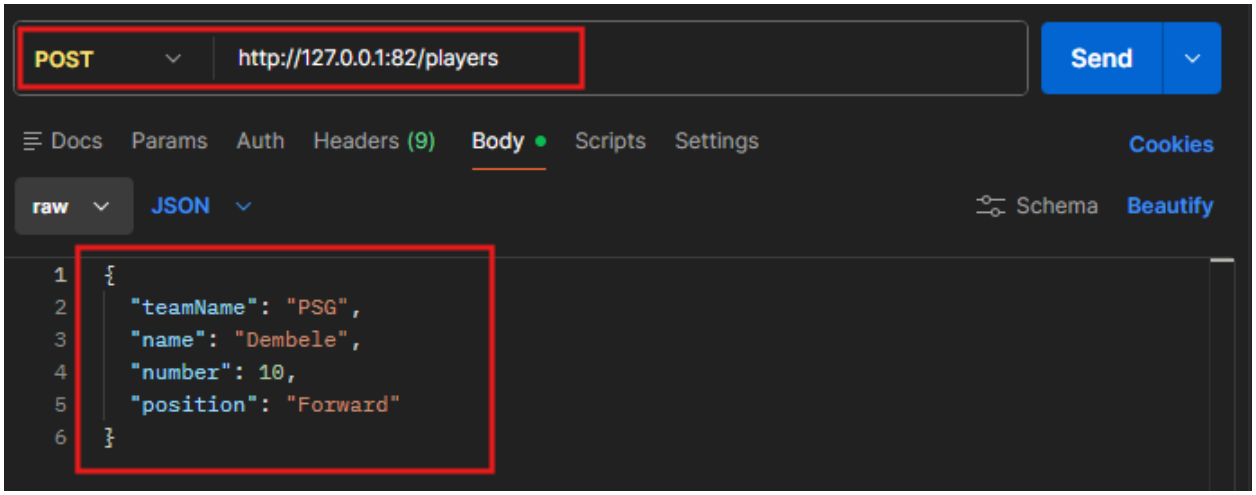

Fig 6.2.3a: Create Player without Authorization header

➔ **OUTPUT**:

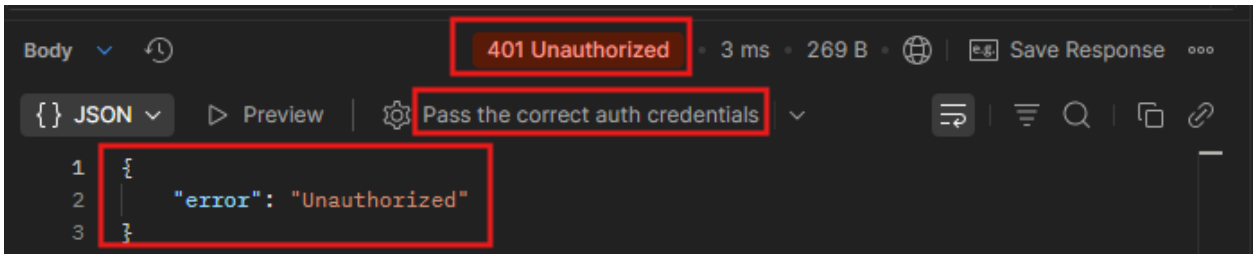

Fig 6.2.3b: 401 Unauthorized response

Step 4: Access Protected Route With Token (Create Player)

➔ Add header: Authorization: Bearer <token>

➔ Call POST /players with Authorization header

➔ **INPUT**:

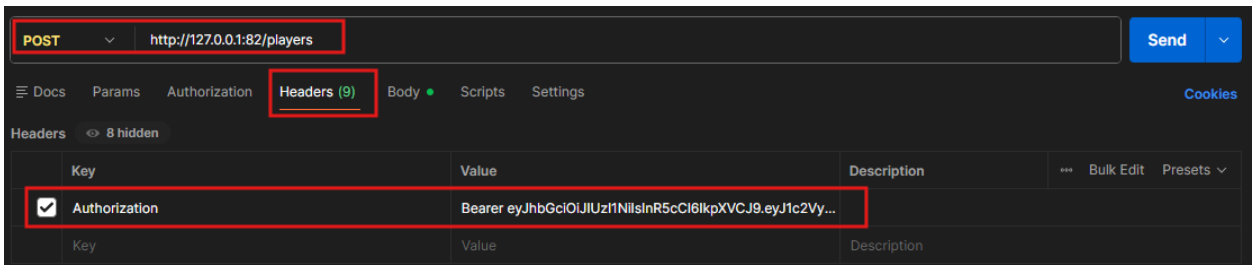

Fig 6.2.4a: Create Player with Authorization header

➔ **OUTPUT**:

Fig 6.2.4b: Successful protected request

➔ **Database Output:**

Fig 6.2.4c: Created Player on the database

Step 5: Other CRUD Validation

➔ Create, read, update, delete players

1. **GET ALL PLAYERS:**

➔ **INPUT**:

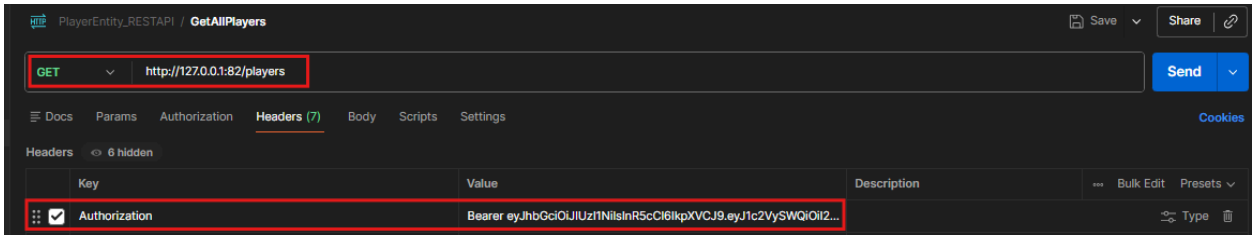

Fig 6.2.5a: Get all Players request

➔ **OUTPUT:**

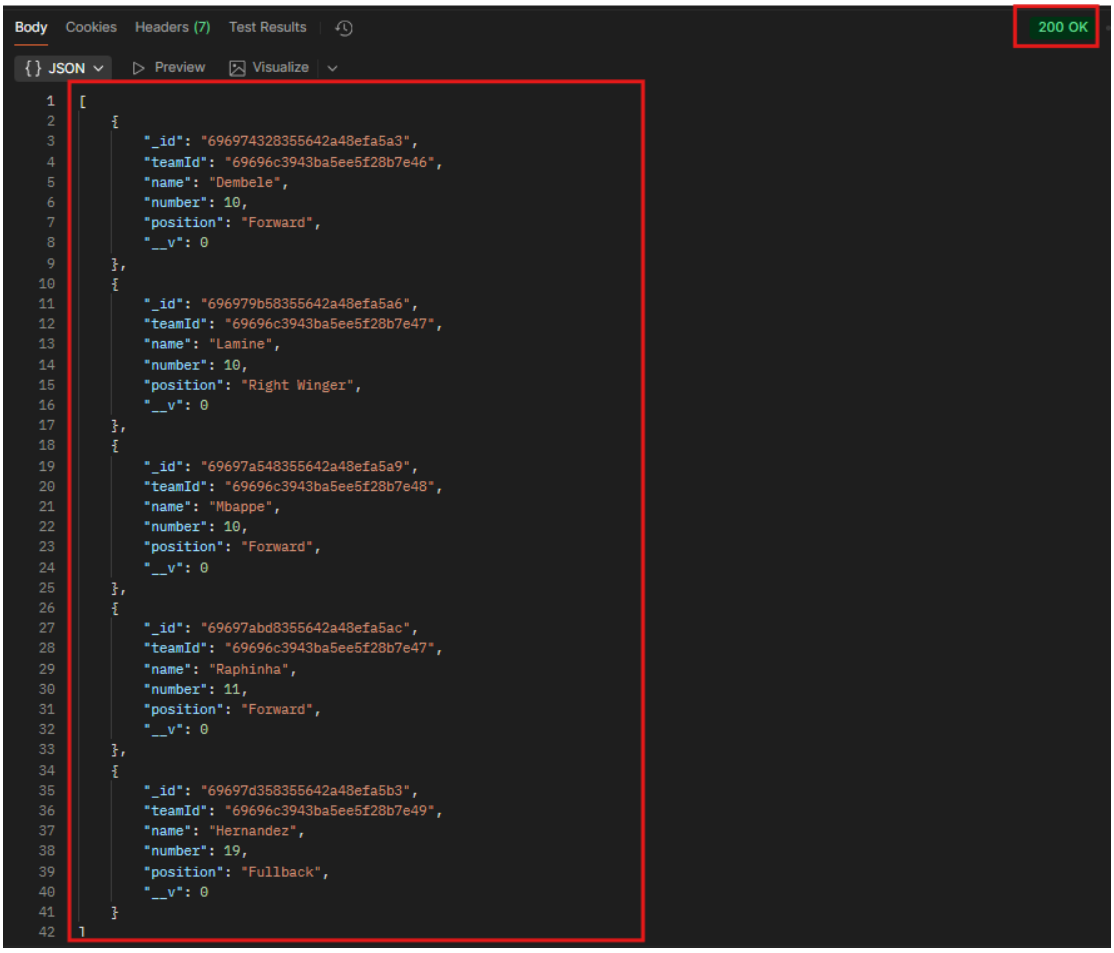

Fig 6.2.5b: All Players retrieved from database

2. DELETE PLAYER:

➔ INPUT:

Fig 6.2.6a Delete a player request

➔ OUTPUT:

Fig 6.2.6b:Player deleted

3. GET PLAYER BY ID:

➔ **INPUT**:

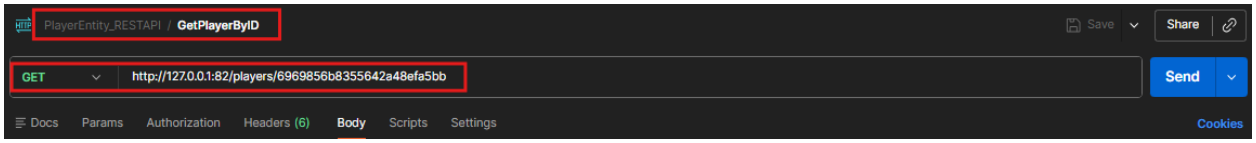

Fig 6.2.7a: Get Player by ID request

➔ **OUTPUT:**

Fig 6.2.7b: Get Player by ID success output

4. UPDATE PLAYER:

➔ **INPUT**:

Fig 6.2.8a: Update Player number request

➔ **OUTPUT:**

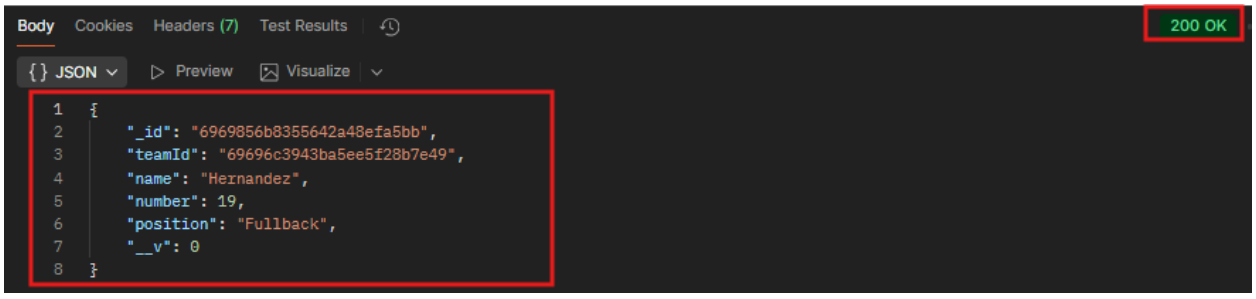

Fig 6.2.8b: Player number updated successfully

5. **GET ALL PLAYERS IN TEAM**

➔ **INPUT:**

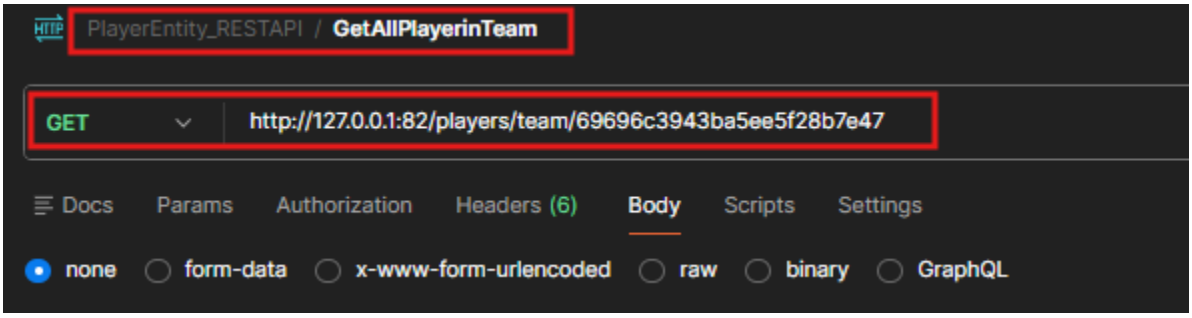

Fig 6.2.9a: Get Players in team request

➔ **OUTPUT:**

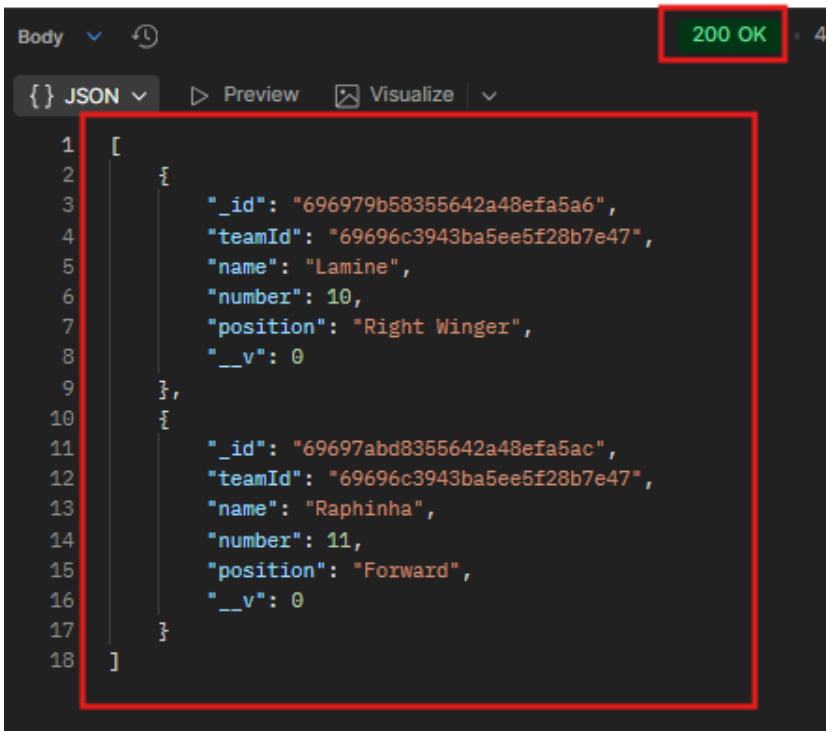

Fig 6.2.9b: Players in team successfully retrieved

6. **GET TEAM OF PLAYER**

➔ **INPUT:**

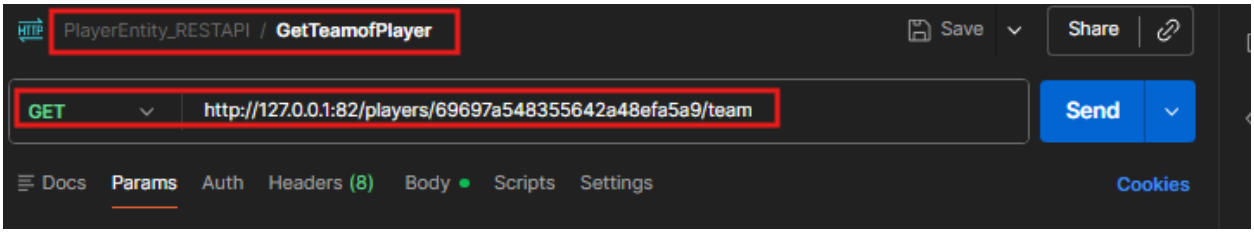

Fig 6.2.10a: Get Players team request

➔ **OUTPUT:**

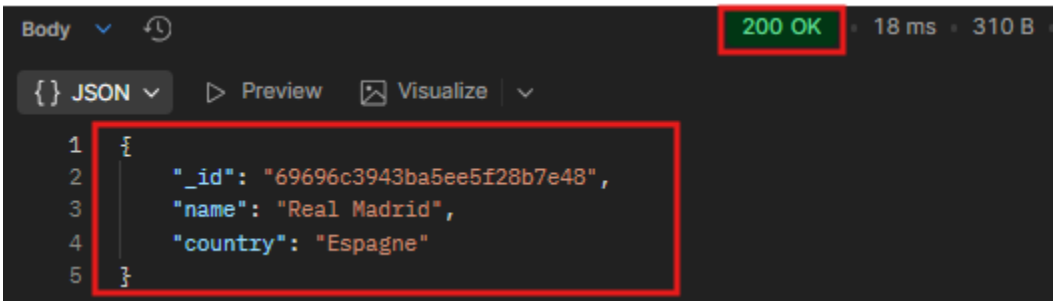

Fig 6.2.10b: Player team successfully retrieved

7. **Search by Name**

➔ **INPUT:**

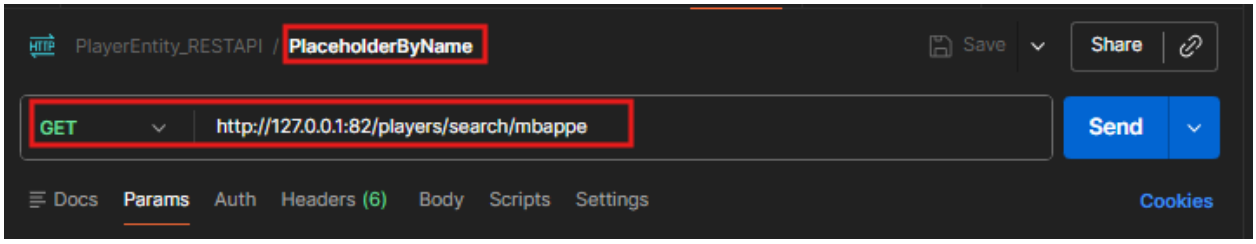

Fig 6.2.11a: Placeholder Search by Name Sample Request

➔ **OUTPUT:**

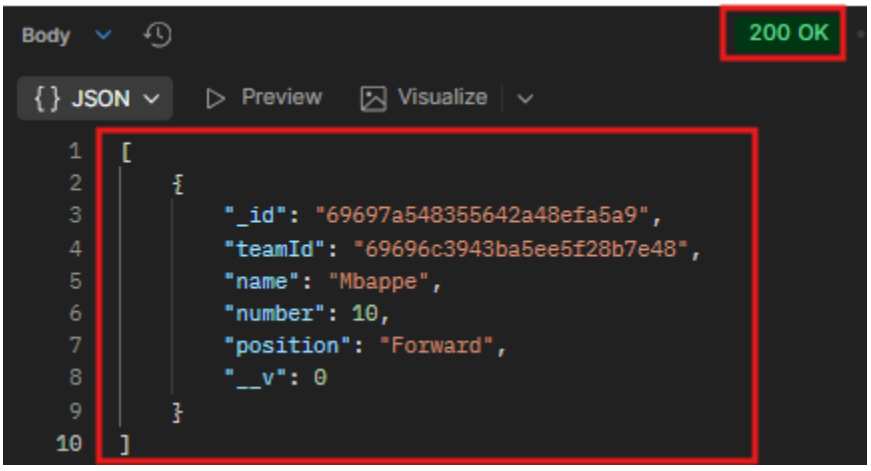

Fig 6.2.11a: Placeholder Search by Name Result

6.3 Detailed Flow of Request

A. GETALLPLAYERS

Step 1 - Client Request

- The client sends an HTTP GET request to /players

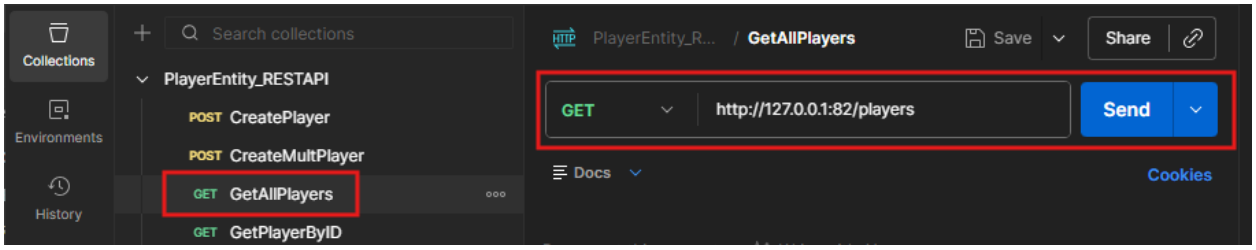

Fig 6.3.1: Postman configured with GET /players

Step 2 - Routing

- Express receives the request
- The /players route matches the request

```
router.get('/', auth, playerController.getAllPlayers);
```

Fig 6.3.2: routes/players.js showing GET route

Step 3 - Middleware Execution (if enabled)

- JWT middleware checks authorization header
- Request is allowed or blocked

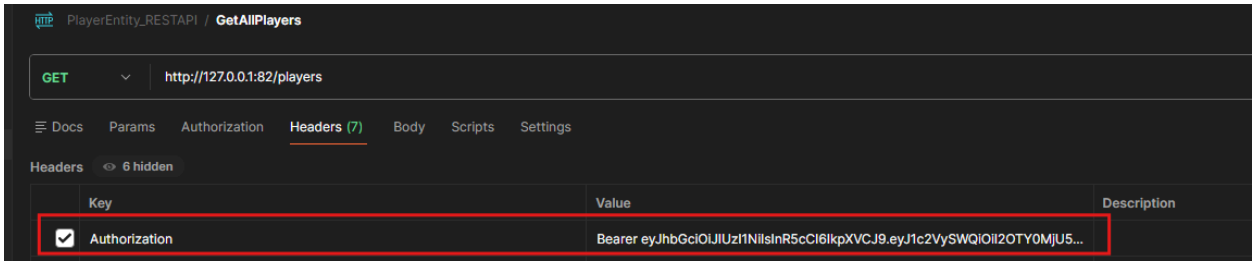

Fig 6.3.3: Authorization header in Postman

Step 4 - Controller Execution

- The controller function getAllPlayers() is called.
- MongoDB query find() is executed

```
// Get all players
exports.getAllPlayers = async (req, res) => {
  const players = await Player.find();
  res.status(200).json(players);
};
```

Fig 6.3.4: playerController.js logic for Get all players

Step 5 - Database Response

- MongoDB returns all player documents

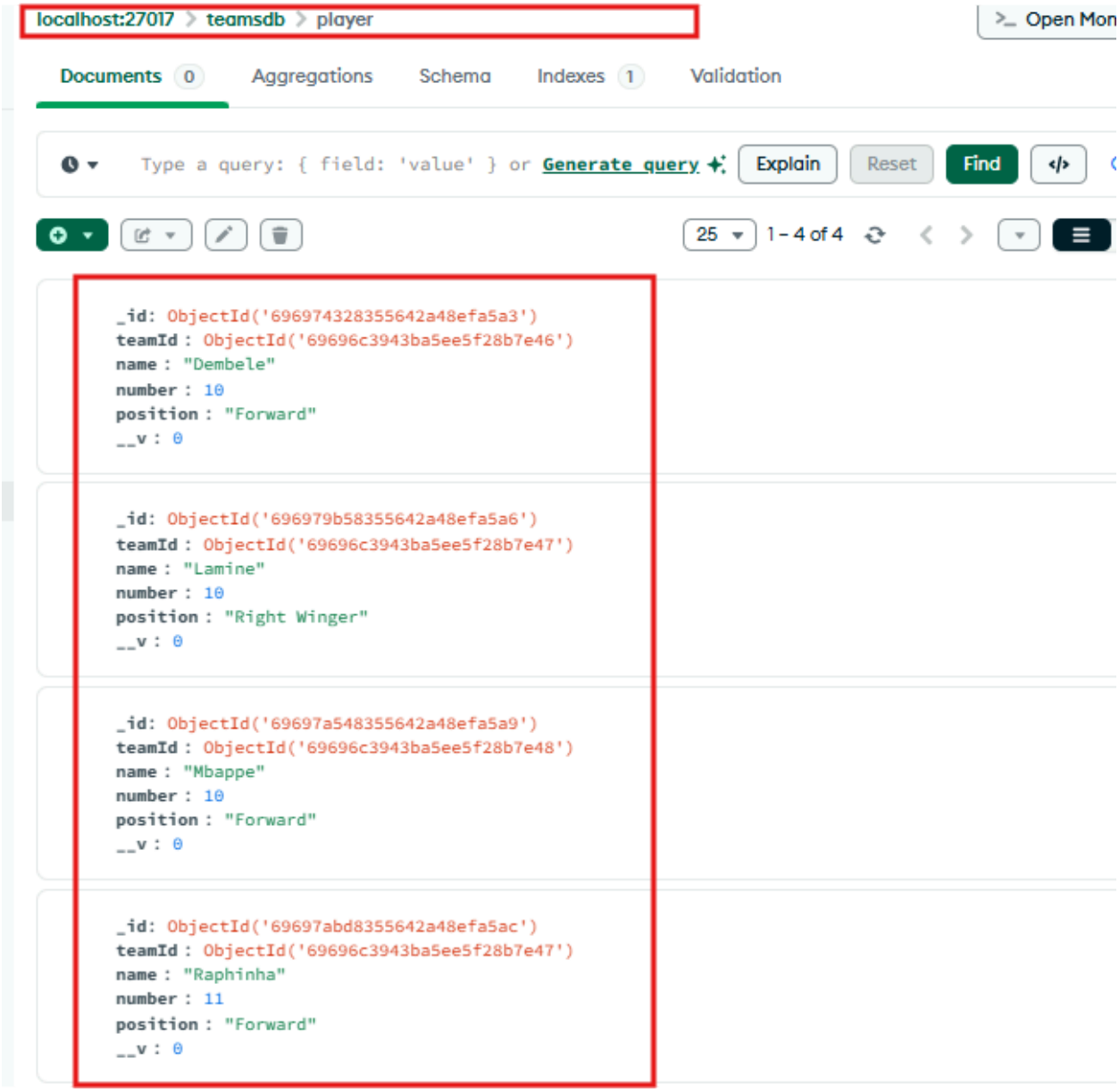

Fig 6.2.5: MongoDB Compass showing player collection

Step 6.6 - API Response

- Server returns HTTP 200
- JSON array of players is sent

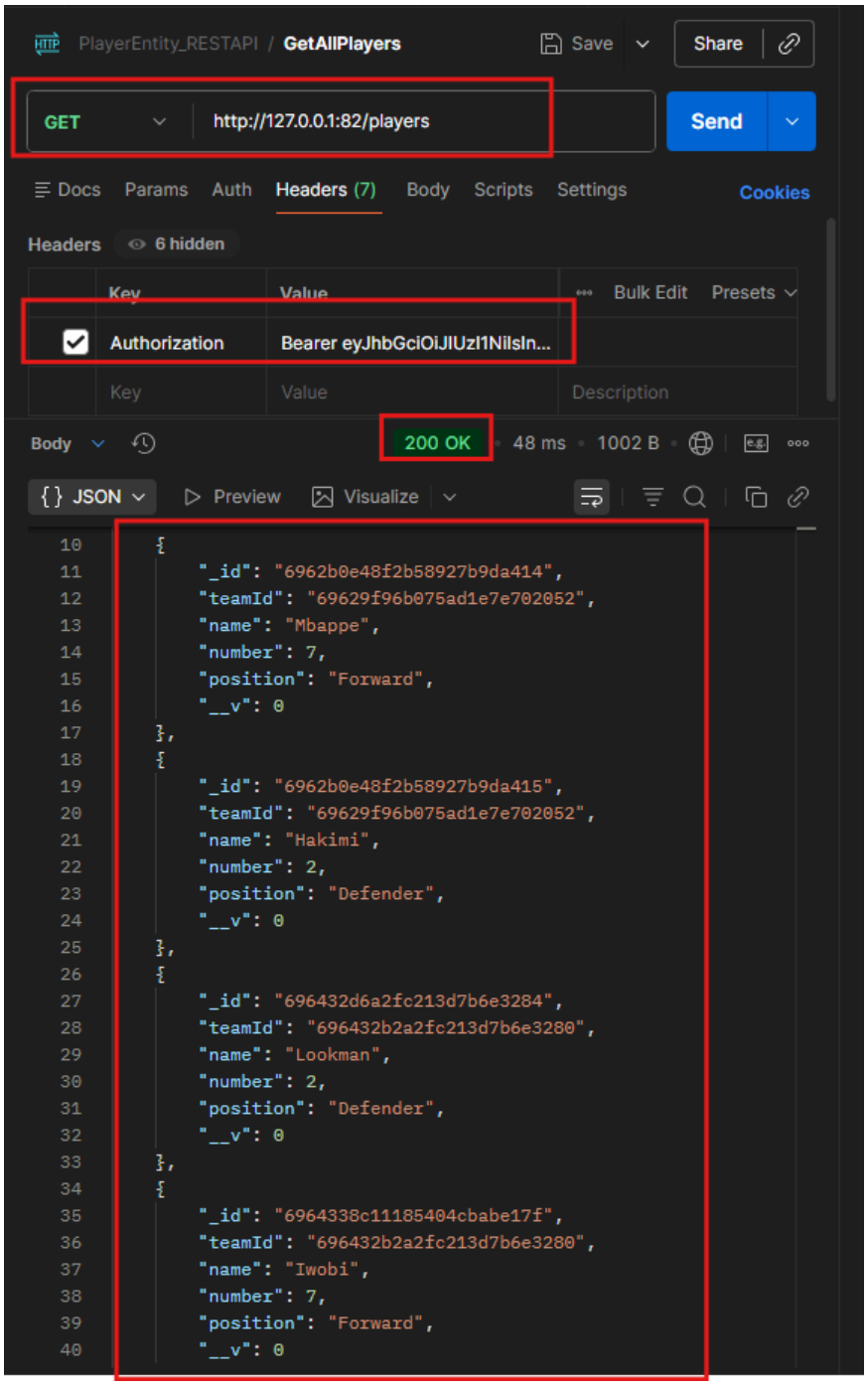

Fig 6.2.6: Postman response body

Conclusion:

This flow shows the complete lifecycle of a REST request from client to database and back to client. This explains how a single API request is processed end-to-end, and how all components function together as designed.

FINAL CONCLUSION

This step-by-step implementation shows how a secure, modular REST API is built from the ground up. Each component plays a clearly defined role, making the system maintainable, scalable, and secure.

The implementation was done using Node.js and Express for server-side logic, MongoDB for data storage, and JSON Web Tokens (JWT) for authentication and authorization. The workflow covered environment setup, database configuration,data modeling, REST API design, controller logic implementation, route definition, middleware integration, and systematic testing with Postman.

The resulting system supports:

- ➔ Persistent data storage using MongoDB.
- ➔ Full CRUD operations for related entities (Equipe and Players)
- ➔ Secure operations using JWT-based authentication.
- ➔ Clear separation of concerns through models, controllers, routes, and middleware.
- ➔ Scalable architecture

The project reinforced concepts like entity relationships, stateless HTTP communication, middleware-based security, and structured API workflows.

Finally, this report provides a step-by-step guide that allows the system to be reproduced reliably.

APPENDIX

A. API Endpoint Reference

Method	Endpoint	Description	Authentication
POST	/auth/signup	Create user	No
POST	/auth/login	Authenticate user	No
GET	/players	Get all players	Yes
GET	/players/:id	Returns one specific player	No
POST	/players	Create player	Yes
PUT	/players/:id	Updates data of an existing player using its ID	Yes
DELETE	/players/:id	Deletes a player from the database	Yes
GET	/players/:id/team	Get the team of specified player	No
GET	/players/team/:teamId	Display players of a Team	No
GET	/players/search/name?name=value	Name-based search	No

B. Collection and Requests Created on Postman

Add requests for each endpoint using HTTP method + URL:

- - POST http://127.0.0.1:82/auth/signup -> create user (password hashed)

- - POST http://127.0.0.1:82/auth/login -> login, returns JWT

- - GET http://127.0.0.1:82/players ->list all players

- - GET http://127.0.0.1:82/players/:id -> Get player using their ID

- - POST http://127.0.0.1:82/players -> create one player

- - PUT http://127.0.0.1:82/players/:id -> update player

- - DELETE http://127.0.0.1:82/players/:id ->delete player

- - GET http://127.0.0.1:82/players/team/:teamId -> Get players of a team

- - GET http://127.0.0.1:82/players/:id/team -> Get team of a player

- - GET http://127.0.0.1:82/players/search/:name -> Search by name placeholder

Repository Link:
https://github.com/JanieAbutu/ECE_Distributed-Systems-Security/tree/main/Projects/Lab%201_PlayerEntity_RestAPI_Project