

Assignment 3 - User Authentication

Instructions

In this assignment, You are required to build a backend application to manage a multiple-choice exam question bank. Authenticated users can create, update, and delete questions, while anyone can view available questions by subject or topic. Each question belongs to a specific topic and subject, and includes four answer options and a correct answer key. The system uses JWT authentication and provides RESTful API endpoints.

Assignment Overview

- **Main Technologies:** Node.js, Express.js, MongoDB, Mongoose, JWT, bcrypt, dotenv, cors
- **Tools:** Postman, MongoDB Compass, Visual Studio Code, Curl

Collections:

You will create 4 collections:

1. `Users` – to store user credentials (teacher accounts)
2. `Subjects` – high-level domains (e.g., Math, Physics)
3. `Topics` – sub-categories under each subject (e.g., Algebra)
4. `Questions` – multiple-choice questions with 4 options and 1 correct answer

At the end of this assignment, you will have developed a secure backend system for a Multiple-Choice Exam Question Bank that supports user authentication and allows only **logged-in users** to manage exam questions. Specifically, you will achieve the following:

- Implement user registration and login using secure password hashing and JSON Web Tokens (JWT).
- Allow anyone (public users) to perform GET operations such as:
 - View all questions
 - View all subjects and topics
- Restrict POST, PUT, and DELETE operations to only authenticated (logged-in) users, including:
 - Submitting new questions
 - Updating or deleting their own submitted questions
- Prevent unauthenticated users from performing any data modification.
- Ensure that users can only update or delete questions they have created. No other user, including other authors, can modify someone else's questions.
- (Optional) Provide an endpoint for viewing all registered users — accessible only to authenticated users if implemented

Assignment Requirements

1. Task 1: Project Initialization and Configuration

◆ Description:

- Create a new Node.js project.
- Install required packages: `express`, `mongoose`, `dotenv`, `cors`, `bcryptjs`, `jsonwebtoken`
- Configure environment variables using `.env` file:
- Create `.env`:

```
PORT=6000
MONGO_URI=mongodb://localhost:27017/assignment3
JWT_SECRET=your_jwt_abc
```

✓ **Test:** Start server → <http://localhost:6000>

Expected Outcome:

- The server starts successfully on `http://localhost:6000`
- MongoDB connects without error

2. Task 2: Register User

Endpoint: `POST /users/register`

◆ Description:

- Implement `POST /users/register`
- Receive username and password in the request body.
- Check if the username is unique.
- Hash the password using `bcrypt` before saving.
- Save the new user to the `Users` collection.

✓ **Example Request:**

POST /users/register

Content-Type: `application/json`

```
{
  "username": "teacher01",
  "password": "exam1234"
}
```

Expected Response:

```
{
  "message": "User registered successfully"
}
```

Error Cases:

- Username already exists → return `409 Conflict` with appropriate message.

3. Task 3: Login (JWT)

Endpoint: `POST /users/login`

◆ Description:

- Implement `POST /users/login`
- Receive username and password in body.
- Validate credentials.
- If correct, generate a JWT token including `userId` and `username`.

✓ **Example Request:**

Body:

`POST /users/login`

```
{
  "username": "teacher01",
  "password": "exam1234"
}
```

Expected Response:

```
{
  "token": "eyJhbGciOiJIUzI1NiIsInR..."
}
```

Error Cases:

- Wrong username or password → return 401 Unauthorized

4. Task 4: Logout

Endpoint: POST /users/logout

◆ Description:

- Implement POST /users/logout
- This is a simulated route. It does not delete the token.
- Simply return a confirmation message.

✓ Example Request:

POST /users/logout

Expected:

```
{
  "message": "You are logged out"
}
```

5. Task 5: Get All Questions (Public)

Endpoint: GET /questions

◆ Description:

- Implement GET /questions
- Return all questions with:
 - questionText, options, correctAnswer
 - Subject and Topic populated by their names
- No authentication required

Expected Response:

```
[
  {
    "questionText": "What is the solution to  $2x + 3 = 7$ ",
    "options": {
      "A": "x = 1",
      "B": "x = 2",
      "C": "x = 3",
      "D": "x = 4"
    },
    "correctAnswer": "B",
    "topic": { "_id": "64aee210c438b927a9cfa333", "name": "Algebra" },
    "subject": { "_id": "64aee200c438b927a9cfa222", "name": "Mathematics" }
  }
]
```

6. Task 6: Create New Question (Auth Required)

Endpoint: POST /questions

Headers: Authorization: Bearer <token>

◆ Description:

- Implement POST /questions
- Only authenticated users can create
- Require fields:
 - questionText, options (A–D), correctAnswer, subjectId, topicId
- Save createdBy as the current user ID

✓ Example Request:

POST /questions

Header:

Authorization: Bearer <JWT>

Body:

```
{
  "questionText": "What is the derivative of x^2?",
  "options": {
    "A": "2x",
    "B": "x",
    "C": "x^2",
    "D": "1"
  },
  "correctAnswer": "A",
  "topicId": "64aee210c438b927a9cfa333",
  "subjectId": "64aee200c438b927a9cfa222"
}
```

 **Expected Response:**

```
{
  "message": "Question created successfully",
  "question": {
    "_id": "...",
    "questionText": "What is the derivative of x^2?",
    ...
  }
}
```

7. Task 7: Update Question (Auth + Ownership) (1.5 points)

Endpoint: PUT /questions/:id

 **Description:**

- Implement PUT /questions/:id
- Only the user who created the question can update it
- Allow updating questionText, options, or correctAnswer

 **Example Request:**

PUT /questions/64aee300...

Headers: Authorization: Bearer <token>

Body:

```
{
  "questionText": "Updated question text"
}
```

Expected (Success):

```
{
  "message": "Question updated successfully"
}
```

Error Case:

- Not the owner → return 403 Forbidden

8. Task 8: Delete Question (Auth + Ownership)

Endpoint: DELETE /questions/:id

◆ Description:

- Implement `DELETE /questions/:id`
- Only allow deletion by creator

✓ Example Request:

`DELETE /questions/64aee300...`

Headers: `Authorization: Bearer <token>`

Expected:

```
{
  "message": "Question deleted successfully"
}
```

9. Task 9: Get All Subjects and Topics

◆ Description:

- Implement:
 - `GET /subjects` → list all subjects
 - `GET /topics` → list all topics, optionally by subject

✓ Example Response:

```
[
  { "_id": "64aee200c438b927a9cfa222", "name": "Mathematics" },
  { "_id": "64aee201c438b927a9cfa223", "name": "Physics" }
]

[
  { "_id": "64aee210...", "name": "Algebra", "subjectId": "64aee200..." },
  { "_id": "64aee211...", "name": "Mechanics", "subjectId": "64aee201..." }
]
```

🔒 JWT Middleware Example

```
// middleware/auth.js
const jwt = require('jsonwebtoken');
module.exports = function (req, res, next) {
  const token = req.headers['authorization']?.split(' ')[1];
  if (!token) return res.status(401).json({ error: 'Access denied' });

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (err) {
    return res.status(400).json({ error: 'Invalid token' });
  }
};
```

Data structure

1. Collection: users

Stores user information (e.g., teachers who create questions).

| Field | Data Type | Description |
|------------------|-----------|---|
| _id | ObjectId | Automatically generated by MongoDB |
| username | String | Unique account name |
| password | String | Password hashed using <code>bcrypt</code> |
| createdAt | Date | Timestamp when the account was created |

 Passwords **must be hashed** before storage.

2. Collection: subjects

Stores **academic subjects**, such as *Mathematics*, *Physics*, etc.

| Field | Data Type | Description |
|-------------|-----------|---------------------------|
| _id | ObjectId | Auto-generated by MongoDB |
| name | String | Name of the subject |

3. Collection: topics

Stores **specific topics** under each subject (e.g., *Algebra* under *Mathematics*).


| Field | Data Type | Description |
|------------------|-----------|---|
| _id | ObjectId | Auto-generated |
| name | String | Topic name (e.g., Calculus, Mechanics) |
| subjectId | ObjectId | References the <code>_id</code> field of the <code>subjects</code> collection |

 `subjectId` is a **foreign key** linking each topic to a subject.

4. Collection: questions

Stores multiple-choice questions.

| Field | Data Type | Description |
|----------------------|-----------|---|
| _id | ObjectId | Auto-generated |
| questionText | String | The question prompt |
| options | Object | 4 answer options: { "A": "...", "B": "...", "C": "...", "D": "..." } |
| correctAnswer | String | One of "A", "B", "C", or "D" as the correct answer |
| subjectId | ObjectId | References <code>_id</code> in the <code>subjects</code> collection |
| topicId | ObjectId | References <code>_id</code> in the <code>topics</code> collection |
| createdBy | ObjectId | References <code>_id</code> in the <code>users</code> collection (who created the question) |
| createdAt | Date | Creation timestamp |

 You can later expand `options` to support random shuffling or metadata.

5. Relationship Summary:

- `topics.subjectId` → references `subjects._id`
- `questions.subjectId` & `questions.topicId` → references `subjects._id` and `topics._id`
- `questions.createdBy` → references `users._id`

6. Sample Data:

```
sample_data/
├── subjects.json
├── topics.json
└── questions.json
```



subjects.json

```
[
  { "_id": { "$oid": "64aee200c438b927a9cfa222" }, "name": "Mathematics" },
  { "_id": { "$oid": "64aee201c438b927a9cfa223" }, "name": "Physics" }
]
```



topics.json

```
[
  {
    "_id": { "$oid": "64aee210c438b927a9cfa333" },
    "name": "Algebra",
    "subjectId": { "$oid": "64aee200c438b927a9cfa222" }
  },
  {
    "_id": { "$oid": "64aee211c438b927a9cfa334" },
    "name": "Mechanics",
    "subjectId": { "$oid": "64aee201c438b927a9cfa223" }
  }
]
```



questions.json

```
[
  {
    "_id": { "$oid": "64aee300c438b927a9cfa444" },
    "questionText": "What is the solution to  $2x + 3 = 7$ ?",
    "options": {
      "A": "x = 1",
      "B": "x = 2",
      "C": "x = 3",
      "D": "x = 4"
    },
    "correctAnswer": "B",
    "topicId": { "$oid": "64aee210c438b927a9cfa333" },
    "subjectId": { "$oid": "64aee200c438b927a9cfa222" },
    "createdBy": { "$oid": "64aee188c438b927a9cfa111" },
    "createdAt": { "$date": "2025-07-01T10:00:00.000Z" }
  },
  {
    "_id": { "$oid": "64aee301c438b927a9cfa445" },
    "questionText": "What is the acceleration due to gravity on Earth?",
    "options": {
      "A": "8.9 m/s2",
      "B": "10.0 m/s2",
      "C": "9.8 m/s2",
      "D": "9.0 m/s2"
    },
    "correctAnswer": "C",
    "topicId": { "$oid": "64aee211c438b927a9cfa334" },
    "subjectId": { "$oid": "64aee201c438b927a9cfa223" },
    "createdBy": { "$oid": "64aee188c438b927a9cfa111" },
    "createdAt": { "$date": "2025-07-01T11:00:00.000Z" }
  }
]
```

4. HÌNH THỨC NỘP BÀI

- Gửi bài tập dưới dạng file nén (.zip/.rar), gồm:
 - Chụp screenshot toàn màn hình: Thư mục, comment tác giả đầu mỗi file source code, cấu trúc vscode, kết quả thực thi đầy đủ.
 - source code
 - File tài liệu báo cáo (.PDF hoặc .DOCX).
- Đặt tên file nén theo format:
 - [MãSV]_[Tên]_assignment3.zip

Ví dụ: SE12345_NguyenVanA_assignment3.zip
- Hạn chót nộp bài: theo lịch Edunext

5. LƯU Ý QUAN TRỌNG

- X Bài nộp không đầy đủ hoặc thiếu file báo cáo sẽ bị trừ điểm.**
- X Mọi hành vi sao chép code sẽ bị xử lý theo quy định của nhà trường.**
- X Sinh viên cần kiểm tra kỹ lưỡng trước khi nộp bài.**
 - Được sử dụng AI để phân tích và thực hiện bài.