# Guided Exercise - Online Bookstore - Backend - Express + MongoDB Backend Exercise: Online Bookstore API #192

akash-coded started this conversation in Tasks
Guided Exercise - Online Bookstore - Backend - Express + MongoDB Backend Exercise: Online Bookstore API#192
akash-coded
last week · 2 comments
Return to top

## akash-codedlast weekMaintainer

# Express + MongoDB Backend Exercise: Online Bookstore API

## Part-1

**This exercise will help you create a practical Express.js and MongoDB backend with features like authentication, authorization, logging, middleware, and security. You'll build an API for an online bookstore.**

## Prerequisites:

- Node.js and npm installed
- MongoDB installed or access to a MongoDB Atlas Cluster
- Postman or a similar API client for testing
- Basic understanding of JavaScript and RESTful API concepts

## Application Features:

1. User registration and login
2. JWT-based authentication and role-based authorization
3. CRUD operations on books
4. Middleware for logging requests
5. Input validation
6. Data encryption for passwords
7. Error-handling

---

## Step 1: Project Setup

**Initialize a new Node.js project and install the required packages.**

```
  npm init -y
npm install express mongoose jsonwebtoken bcryptjs dotenv morgan validator
```

**Create a `.env` file for environment variables like MongoDB URI, JWT secret, etc.**

```
  MONGODB_URI="your_mongodb_uri"
JWT_SECRET="your_secret_key"
```

---

## Step 2: Establish MongoDB Connection

**Use Mongoose to connect to MongoDB in your main `app.js` file.**

**Rationale: Mongoose provides a straight-forward, schema-based solution to model your application data.**

```
  const mongoose = require('mongoose');

  mongoose.connect(process.env.MONGODB_URI, {
    useNewUrlParser: true,
    useUnifiedTopology: true,
});
```

---

## Step 3: Basic Express Setup

**Create your main Express application.**

```
  const express = require('express');
  const app = express();

  app.use(express.json());  // For parsing JSON

  const PORT = process.env.PORT || 3000;
  app.listen(PORT, () => {
    console.log(`Server running on port ${PORT}`);
});
```

---

## Step 4: Model Definitions

**Define your Mongoose models for Users and Books in separate files.**

```
  // models/User.js
  const mongoose = require('mongoose');
  const bcrypt = require('bcryptjs');

  const userSchema = new mongoose.Schema({
    username: String,
    password: String,
    role: String,
  });

  // Password hashing middleware
  userSchema.pre('save', function(next) {
    this.password = bcrypt.hashSync(this.password, 12);
    next();
  });
```

```
module.exports = mongoose.model('User', userSchema);
```

**Rationale: We're using bcrypt.js to hash passwords before storing them, as plain text passwords are a security risk.**

---

## Step 5: Authentication

**Implement JWT-based authentication. Create a `/login` endpoint to issue tokens.**

**Rationale: JWT is stateless, meaning that you don't have to store the token on the server, making it scalable and easy to use.**

---

## Step 6: Authorization Middleware

**Create a middleware to protect routes based on user roles.**

```
  // middleware/authorization.js

  const jwt = require('jsonwebtoken');

  function authorize(role) {
    return (req, res, next) => {
      const token = req.header('x-auth-token');
      if (!token) return res.status(401).send('No token provided.');

      const decoded = jwt.verify(token, process.env.JWT_SECRET);
      if (!decoded || (role && decoded.role !== role)) return
  res.status(403).send('Forbidden.');

      req.user = decoded;
      next();
    }
  }

module.exports = authorize;
```

---

## Step 7: CRUD for Books

**Create CRUD operations for books and use the authorization middleware to protect the routes.**

**Rationale: CRUD operations form the backbone of most web APIs.**

---

## Step 8: Logging Middleware

**Create a middleware for logging using the morgan library.**

**Rationale: Logging helps in debugging and monitoring.**

```
  const morgan = require('morgan');
app.use(morgan('tiny'));
```

---

## Step 9: Error-handling

**Implement centralized error-handling middleware.**

**Rationale: This ensures that all errors are caught and handled appropriately, improving your application's robustness and user experience.**

---

## Step 10: Input Validation

**Use the `validator` library for basic input validation.**

**Rationale: Input validation is essential for application security to protect against malicious user data.**

---

**That's it! This exercise covers a lot of ground, but it provides a strong foundation in backend development with Node.js, Express, and MongoDB. Make sure to test all your endpoints using Postman or a similar API client.**

---

## Part-2

## Step 11: Securing the Application

**Add basic security measures to the application.**

**Rationale: Security is crucial in every application, especially when handling sensitive information like user credentials.**

Install `helmet` for setting security-related HTTP headers:

```
npm install helmet
```

In your main `app.js`:

```
  const helmet = require('helmet');
app.use(helmet());
```

---

## Step 12: Rate Limiting

To protect against brute-force attacks, implement rate limiting.

Rationale: Rate limiting ensures that a user cannot make unlimited API requests, thereby reducing the risk of brute-force attacks.

Install `express-rate-limit`:

```
npm install express-rate-limit
```

Then set it up:

```
  const rateLimit = require("express-rate-limit");

  const limiter = rateLimit({
    windowMs: 15 * 60 * 1000, // 15 minutes
    max: 100 // limit each IP to 100 requests per windowMs
  });

  //  apply to all requests
app.use(limiter);
```

---

## Step 13: User Registration and Validation

Add an endpoint for user registration and apply input validation and password hashing.

Rationale: User registration is a common feature in many apps, and proper validation and hashing ensure data integrity and security.

```
  // routes/users.js
  const { check, validationResult } = require('express-validator');

  // User registration
  app.post('/register',
    [
      check('username', 'Username is required').not().isEmpty(),
      check('password', 'Password should be at least 6 chars long').isLength({
  min: 6 }),
    ],
    async (req, res) => {
      const errors = validationResult(req);
      if (!errors.isEmpty()) {
        return res.status(400).json({ errors: errors.array() });
      }

      // Hash password and save user
      // ...
    }
);
```

---

## Step 14: Testing with Postman

**Now that your application is feature-rich, test all its functionalities using Postman.**

**Rationale: Testing ensures that the application works as expected and can handle edge cases.**

---

## Step 15: Logging and Monitoring

**Consider using a more advanced logging solution for production. You might use libraries like `winston` for this.**

**Rationale: Advanced logging solutions provide more features like logging to a file, logging levels, and more.**

```
npm install winston
```

---

## Step 16: Final Code Review

**Review your code to make sure everything is working as expected and that you've followed best practices for coding standards, file structure, and naming conventions.**

**Rationale: This ensures maintainability and readability of the codebase, making future updates easier.**

---

## Step 17: Deployment

Prepare your code for deployment. Check environment variables, and secure sensitive data.

Rationale: Proper deployment practices ensure that the application runs smoothly in a production environment.

---

## Testing Your Application

Now that your application is built, you should consider writing automated tests for it. Testing is a critical aspect of modern web development that ensures your application works as expected.

That's it! You've built a comprehensive and secure backend service with Node.js, Express, and MongoDB. Feel free to extend this application by adding more features, creating more robust validations, or even integrating more advanced authentication and authorization mechanisms.

---

## Part-3

## Step 18: Integrating Unit Testing with Jest

Now, let's add unit testing to ensure our Express + MongoDB application is working as expected.

Rationale: Unit testing helps to catch errors early in the development process, making it easier and cheaper to fix them.

Setup Jest and Required Packages

First, install Jest, `supertest` for HTTP assertions, and `mongodb-memory-server` to run an in-memory MongoDB instance:

```
npm install --save-dev jest supertest mongodb-memory-server
```

Update your `package.json`:

```
  "scripts": {
    "test": "jest --watchAll"
}
```

**Create a Jest Setup File**

**Create a file called `jest.config.js` in your root directory and add the following:**

```
  module.exports = {
    testEnvironment: 'node',
};
```

**Rationale: This configuration tells Jest to use a Node environment for testing.**

**Mock Test Cases**

**Let's consider some mock test cases for our application:**

1. **User registration should return 201.**
2. **User registration with an existing username should return 400.**
3. **User login should return a token.**
4. **Unauthorized route should return 401.**
5. **Create a new post should return 201.**
6. **Like a post should increment likes by 1.**

**Writing the Tests**

**Create a new folder in your root directory called `__tests__`.**

1. **User Registration**

**Create a file called `user.test.js`:**

```
  const request = require('supertest');
  const app = require('../app');  // Import your Express app

  describe('User Registration', () => {
    it('should create a new user and return 201', async () => {
      const res = await request(app)
        .post('/api/register')
        .send({
          username: 'john',
          password: 'doe12345'
        });
      expect(res.statusCode).toEqual(201);
    });

    it('should return 400 if username already exists', async () => {
      await request(app)
        .post('/api/register')
        .send({
          username: 'john',
          password: 'doe12345'
        });

      const res = await request(app)
        .post('/api/register')
        .send({
          username: 'john',
          password: 'doe12346'
        });
      expect(res.statusCode).toEqual(400);
    });
});
```

### 2. User Login

**Add these tests to `user.test.js`:**

```
  describe('User Login', () => {
    it('should return a token', async () => {
      // Make sure the user is registered first
      await request(app)
        .post('/api/register')
        .send({
          username: 'john',
          password: 'doe12345'
        });

      const res = await request(app)
        .post('/api/login')
        .send({
          username: 'john',
          password: 'doe12345'
        });
      expect(res.body).toHaveProperty('token');
    });
});
```

### 3. Unauthorized Routes

**Create a file called `auth.test.js`:**

```
  const request = require('supertest');
  const app = require('../app');

  describe('Unauthorized Routes', () => {
    it('should return 401', async () => {
      const res = await request(app)
        .get('/api/protected-route');
      expect(res.statusCode).toEqual(401);
    });
});
```

Run your tests:

```
npm test
```

Benefits of Unit Testing

1. Quick Feedback: Detects issues early in the development phase.
2. Simplifies Debugging: Easier to identify which change led to the error.
3. Enhanced Collaboration: Makes it easier for other developers to understand your code and contribute.
4. Code Quality: Encourages writing cleaner, more modular code.
5. Documentation: Test cases demonstrate how a system is intended to behave, serving as a form of documentation.

That's it! You've now integrated Jest for unit testing with your Express + MongoDB application, making it more robust and maintainable.

12

## Replies:

## 2 comments

Oldest

Newest

Top

akash-codedlast weekMaintainerAuthor

# Detailed Explanation

Let's delve into the details:

## Dependencies

1. Express:
   - What: It's a fast, unopinionated, minimalist web framework for Node.js.
   - Why: Makes setting up robust APIs and web servers much easier.
   - Example: `const app = express(); app.get('/', (req, res) => res.send('Hello World!'));`
2. MongoDB:
   - What: A NoSQL database.
   - Why: Allows you to store data in a flexible, JSON-like format which can evolve over time.
   - Example: `db.collection('users').insertOne({name: "John"})`
3. Mongoose:
   - What: An object data modeling (ODM) library for MongoDB and Node.js.
   - Why: It provides a straight-forward, schema-based solution to model your application data.
   - Example: Defining a schema and model: `const userSchema = new mongoose.Schema({ name: String }); const User = mongoose.model('User', userSchema);`
4. bcryptjs:
   - What: A library to hash passwords.
   - Why: Security concern to store passwords in hashed form.
   - Example: `bcrypt.hash("password", 10);`
5. jsonwebtoken:
   - What: A library to handle JSON Web Tokens.
   - Why: Useful for authentication.
   - Example: `jsonwebtoken.sign({ user: 'john' }, 'secretkey');`
6. Jest:
   - What: A JavaScript Testing Framework.
   - Why: Enables a range of testing methods including unit, integration, and end-to-end testing.
   - Example: `test('adds 1 + 2 to equal 3', () => { expect(1 + 2).toBe(3); });`
7. supertest:
   - What: A library for testing HTTP assertions.
   - Why: Used in conjunction with Jest to test Express routes.
   - Example: `await request(app).get('/').expect(200);`
8. mongodb-memory-server:
   - What: Spins up a real MongoDB Server programmatically from node for testing.
   - Why: Ideal for unit tests and CI.

- Example: `new MongoMemoryServer({ binary: { version: 'latest' }});`

## Web Development Concepts

1. RESTful APIs:
   - What: Architectural style that uses standard HTTP requests.
   - Why: Easy to use and scalable.
   - Example: `GET /users` to fetch all users.
2. CRUD Operations:
   - What: Stands for Create, Read, Update, and Delete.
   - Why: These are the basic operations that your application will perform on any data.
   - Example: `GET /users`, `POST /users`, `PUT /users/:id`, `DELETE /users/:id`
3. Middleware:
   - What: Functions that have access to request and response objects.
   - Why: Useful for adding headers, logging, etc.
   - Example: `app.use((req, res, next) => { console.log('Middleware'); next(); });`

## Security Concepts

1. Authentication:
   - What: Verifying the identity of a user.
   - Why: To restrict unauthorized access.
   - Example: Using JWT to confirm if a request is coming from a valid user.
2. Authorization:
   - What: Verifying if a user has permission to perform a given operation.
   - Why: To provide role-based access.
   - Example: Only admins can delete users.
3. Hashing:
   - What: Transforming a string of characters into a fixed-size string.
   - Why: Storing passwords securely.
   - Example: Using bcrypt to hash passwords.
4. HTTPS:
   - What: Secure version of HTTP.
   - Why: Encrypts the data sent between client and server.
   - Example: SSL certificates.
5. Data Validation:
   - What: The process of ensuring that user-supplied data is within expected boundaries.
   - Why: To prevent malformed data and security vulnerabilities like SQL injection, XSS, etc.
   - Example: Using `joi` or `express-validator` libraries.

## Development Process

1. Requirement Gathering: Understand what you are going to build. Is it an API, a frontend app, or both? What features do you need?
2. Design and Planning: Choose the architecture, database, and libraries.
3. Development: Start by setting up your development environment, followed by the databases and then begin actual coding.
4. Testing: Perform unit tests, integration tests, and end-to-end tests.
5. Deployment: Once everything looks good, you can deploy your application.
6. Maintenance: Even after deployment, you'll need to update your application regularly.

I hope this serves as a comprehensive guide to understanding the various elements, dependencies, and concepts involved in the complete

---

Now let's go deeper into each topic.

## Dependencies: Deeper Dive

1. Express:
   - Rationale: Express streamlines many HTTP server functionalities into simple methods and middleware, allowing for more readable, maintainable code.
   - Best Practices: Use middleware for tasks like logging, request parsing, or CORS configurations. Keep your routes modular by using `express.Router`.
2. MongoDB:
   - Rationale: Its schema-less architecture allows flexibility in your data structure, ideal for projects where quick iteration is necessary.
   - Best Practices: Use indexes for faster queries, and utilize its powerful Aggregation Pipeline for complex data manipulation.
3. Mongoose:
   - Rationale: Mongoose provides a strong, typed model to work from, ensuring data consistency.
   - Best Practices: Always define default values and validations in your schema. Also, leverage pre and post hooks for data integrity.
4. bcryptjs:
   - Rationale: Storing user passwords in plain text is a security risk. Bcrypt hashes passwords in a way that is computationally intensive, making brute-force attacks impractical.
   - Best Practices: Always salt your hashes and consider using asynchronous hashing functions.
5. jsonwebtoken:
   - Rationale: JSON Web Tokens are an open standard for securely transmitting information between parties.
   - Best Practices: Never store sensitive user information in the payload and always use a strong secret key.
6. Jest:

    - Rationale: Testing ensures your codebase stays error-free as it grows. Jest offers a feature-rich, well-documented environment for all types of JavaScript testing.
    - Best Practices: Use descriptive test case names and keep tests small and focused on a single functionality.
7. supertest:
    - Rationale: Supertest allows you to test RESTful endpoints with various HTTP assertions.
    - Best Practices: Always clean up any data you add into the database for testing purposes.
8. mongodb-memory-server:
    - Rationale: This package allows for fast, reliable testing that doesn't interfere with your actual database.
    - Best Practices: Clean up after tests to free up memory.

## Web Development Concepts: Deeper Dive

1. RESTful APIs:
    - Rationale: RESTful APIs use HTTP requests to perform CRUD operations, an intuitive and straightforward way to interact with your data.
    - Best Practices: Stick to the standard HTTP methods (`GET`, `POST`, `PUT`, `DELETE`) and keep your API stateless.
2. CRUD Operations:
    - Rationale: Nearly every application performs CRUD operations as it's the fundamental way to interact with databases.
    - Best Practices: Always validate data both client-side and server-side before performing CRUD operations.
3. Middleware:
    - Rationale: Middleware functions can execute any code, make changes to request and response objects, or end the request-response cycle.
    - Best Practices: Keep your middleware functions small and focused on a single task.

## Security Concepts: Deeper Dive

1. Authentication:
    - Rationale: Knowing who is interacting with your application is the first step towards security.
    - Best Practices: Always use HTTPS for authentication and use token-based authentication for RESTful APIs.
2. Authorization:
    - Rationale: Not every user should have the same permissions, and some data should remain restricted.
    - Best Practices: Implement role-based access control and always re-check permissions server-side.
3. Hashing:

    - Rationale: Passwords should never be stored in plain text for security reasons.
    - Best Practices: Use a strong hashing algorithm like bcrypt and always salt your hashes.
4. HTTPS:
    - Rationale: Encryption is a fundamental aspect of data security.
    - Best Practices: Always use strong SSL certificates and keep them updated.
5. Data Validation:
    - Rationale: Incorrect or malformed data can lead to security vulnerabilities and bugs.
    - Best Practices: Never trust user input; always validate and sanitize data that comes from outside systems.

## Development Process: Deeper Dive

1. Requirement Gathering:
    - Rationale: The more clearly you understand what is needed, the more efficient your development process will be.
    - Best Practices: Document requirements meticulously and make sure all stakeholders agree.
2. Design and Planning:
    - Rationale: A good design will serve as a strong foundation for your project.
    - Best Practices: Use design patterns where appropriate and consider future scalability.
3. Development:
    - Rationale: This is where the actual product begins to take shape.
    - Best Practices: Follow coding standards and best practices. Keep your code modular and well-commented.
4. Testing:
    - Rationale: A well-tested application is easier to debug and maintain.
    - Best Practices: Write tests as you go (Test-Driven Development) and aim for high code coverage.
5. Deployment:
    - Rationale: Your application needs to be accessible to end-users.
    - Best Practices: Use CI/CD pipelines for automated testing and deployments.
6. Maintenance:
    - Rationale: Software needs to be updated and improved continuously.
    - Best Practices: Keep dependencies up to date, and always monitor for and fix security vulnerabilities.

This deeper dive aims to provide a comprehensive understanding of the various elements involved in building a secure, efficient, and scalable web application.

akash-codedlast weekMaintainerAuthor

Here's a detailed guide to run and test all functionalities of the hypothetical Express + MongoDB backend application, focusing on features like authentication, authorization, logging, middlewares, and security. Since we haven't actually created a specific application, the following examples are to be used as a generic guide.

## How to Run the Application

### Step 1: Install Dependencies

First, navigate to your project directory and install the required dependencies using npm or yarn.

```
npm install
```

### Step 2: Start MongoDB

Ensure that MongoDB is running.

```
mongod
```

### Step 3: Start the Server

Start the server by running:

```
npm start
```

## How to Test Functionalities

### User Registration:

- **Endpoint:** `POST /api/users/register`
- **Payload:**

```
  {
    "username": "john_doe",
    "password": "123456",
    "email": "john@example.com"
}
```

- Expected: Should return a 201 Created and a JWT token if successful.

### User Login:

- **Endpoint:** `POST /api/users/login`
- **Payload:**

```
  {
    "username": "john_doe",
    "password": "123456"
}
```

- **Expected: Should return a 200 OK and a JWT token if successful.**

**Fetch User Profile (Authorized):**

- **Endpoint:** `GET /api/users/profile`
- **Headers:**

```
  {
    "Authorization": "Bearer <Your_JWT_Token>"
}
```

- **Expected: Should return a 200 OK and the user's profile details if the token is valid.**

**Update User Profile (Authorized):**

- **Endpoint:** `PUT /api/users/profile`
- **Headers:**

```
  {
    "Authorization": "Bearer <Your_JWT_Token>"
}
```

- **Payload:**

```
  {
    "email": "john_new@example.com"
}
```

- **Expected: Should return a 200 OK and the updated user details.**

**Create Post (Authorized):**

- **Endpoint:** `POST /api/posts/`
- **Headers:**

```
  {
    "Authorization": "Bearer <Your_JWT_Token>"
}
```

- **Payload:**

```
  {
    "content": "This is a new post."
}
```

- **Expected: Should return a 201 Created and the post details.**

**Fetch Posts:**

- **Endpoint:** `GET /api/posts/`
- **Expected: Should return a 200 OK and an array of posts.**

**Like a Post (Authorized):**

- **Endpoint:** `POST /api/posts/<Post_ID>/like`
- **Headers:**

```
  {
    "Authorization": "Bearer <Your_JWT_Token>"
}
```

- **Expected: Should return a 200 OK and the updated post details with incremented likes count.**

**Add a Comment to Post (Authorized):**

- **Endpoint:** `POST /api/posts/<Post_ID>/comment`
- **Headers:**

```
  {
    "Authorization": "Bearer <Your_JWT_Token>"
}
```

- **Payload:**

```
  {
    "comment": "This is a comment."
}
```

- **Expected: Should return a 200 OK and the updated post details with the new comment.**

## Testing with Jest

1. **Run Jest Tests**
   **Run the Jest test suite with the following command:**

```
npm test
```

2. **Test Scenarios to Cover**
   - **Test successful registration and login.**
   - **Test fetching and updating profile with valid and invalid JWT tokens.**
   - **Test creating, fetching, liking, and commenting on posts with authorized and unauthorized requests.**

Each test should cover these basic scenarios, plus edge cases like invalid data formats, unauthorized access, and so forth. Make sure to mock your database using a package like `mongodb-memory-server` to isolate the test environment.

This guide aims to provide a comprehensive methodology to run and test the backend application thoroughly, covering all major scenarios.

12

0 replies
Write a reply

Write Preview

Attach files by dragging & dropping, selecting or pasting them.

Comment

Remember, contributions to this repository should follow our GitHub Community Guidelines.
Category

📋
Tasks

Labels

None yet

1 participant

Notifications

Subscribe

You’re not receiving notifications from this thread.

Create issue from discussion

Create issue from discussion
The original post will be copied into a new issue, and the discussion will remain active.
OK, got it!

## Footer

## Footer navigation

- Terms
- Privacy
- Security