Securing Rest Api

Bocsa Georgiana-Maria

Ensuring the security of RESTful APIs is essential to protect sensitive data and maintain user trust. One effective way to achieve this is by using JSON Web Tokens (JWT) for authentication and authorization. JWTs are compact, URL-safe tokens that securely transmit information between two parties, usually a client and a server. These tokens are digitally signed, ensuring their authenticity and integrity, making them a reliable choice for securing APIs.

A JSON Web Token consists of three parts:

1. **Header**: Specifies the type of token and the signing algorithm used.
2. **Payload**: Contains the claims, which are statements about an entity (typically, the user) and additional data.
3. **Signature**: Ensures the token's integrity by signing the header and payload with a secret key.

This structure allows the server to verify the authenticity of the token and trust the information it contains.

**Implementing Secured REST API with JWT Authentication in an application:**

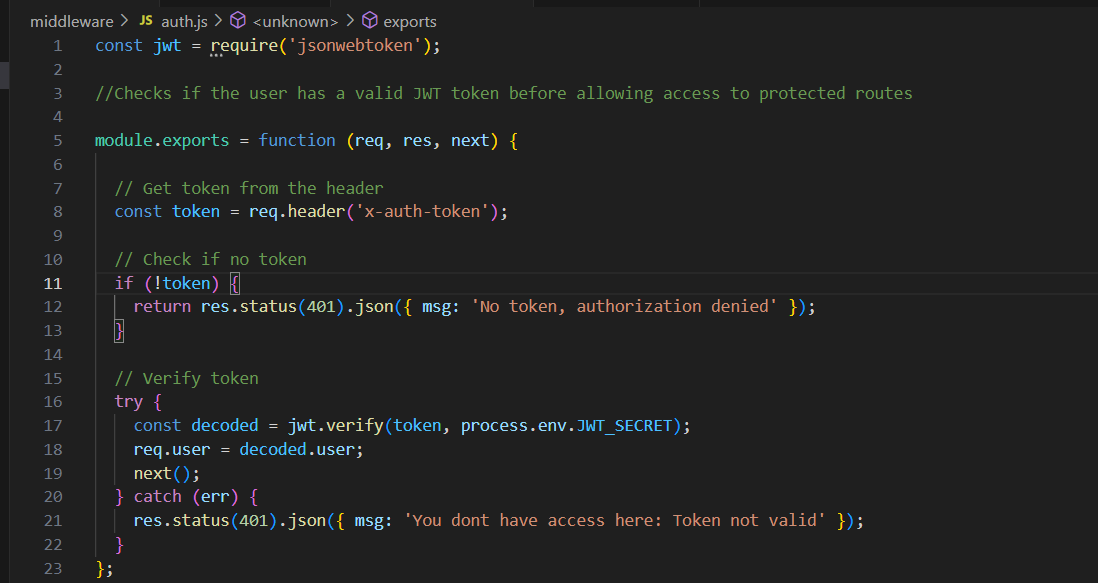
In this tutorial, we’ll build a simple REST API with the following features:

* **User Registration**: Users can register by providing a username, email, and password.
* **User Login**: Registered users can log in and receive a JWT (JSON Web Token).
* **Protected Routes**: Only authenticated users with a valid JWT can access certain routes.

We'll use Node.js and Express to build the API, MongoDB to store user data, and JWT for authentication. Let's dive into the implementation.

*1. Middleware: Authentication Middleware (auth.js)*

To protect our API routes, we created middleware that verifies the presence and validity of a JWT token. If a token is missing or invalid, access is denied. This middleware is used to guard protected endpoints.



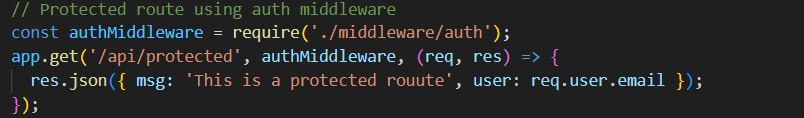
*2. Authentication Routes*

We created routes to handle user registration and login. The registration endpoint checks if a user already exists, hashes their password, and saves them in the database. The login endpoint verifies the credentials and issues a JWT upon successful authentication.



3. Server Configuration

The main server file initializes the application, connects to MongoDB, and sets up API routes. It also includes authentication middleware to protect specific endpoints. We also created a protected route that requires a valid JWT to access:



*4. Environment Configuration*

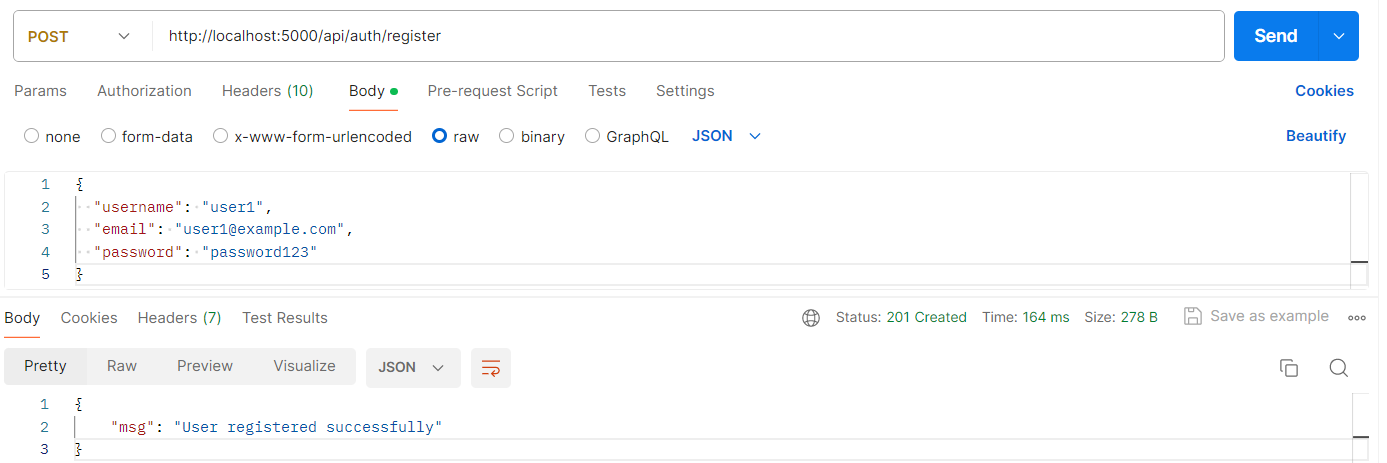
Sensitive information, such as database credentials and JWT secrets, are stored in environment variables using an .env file.

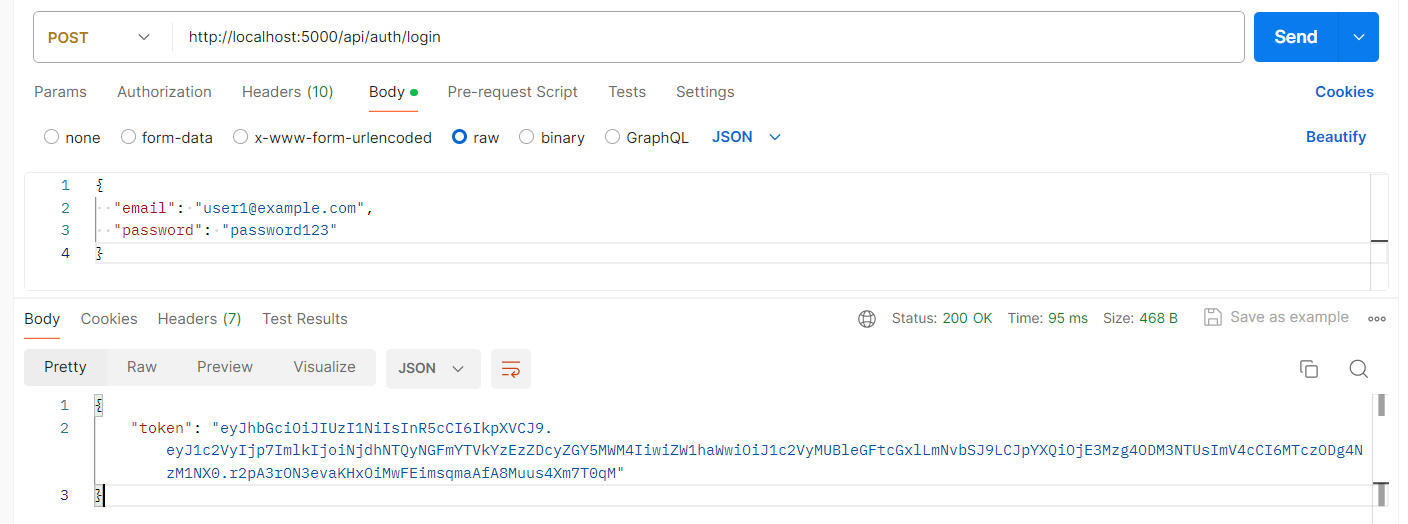


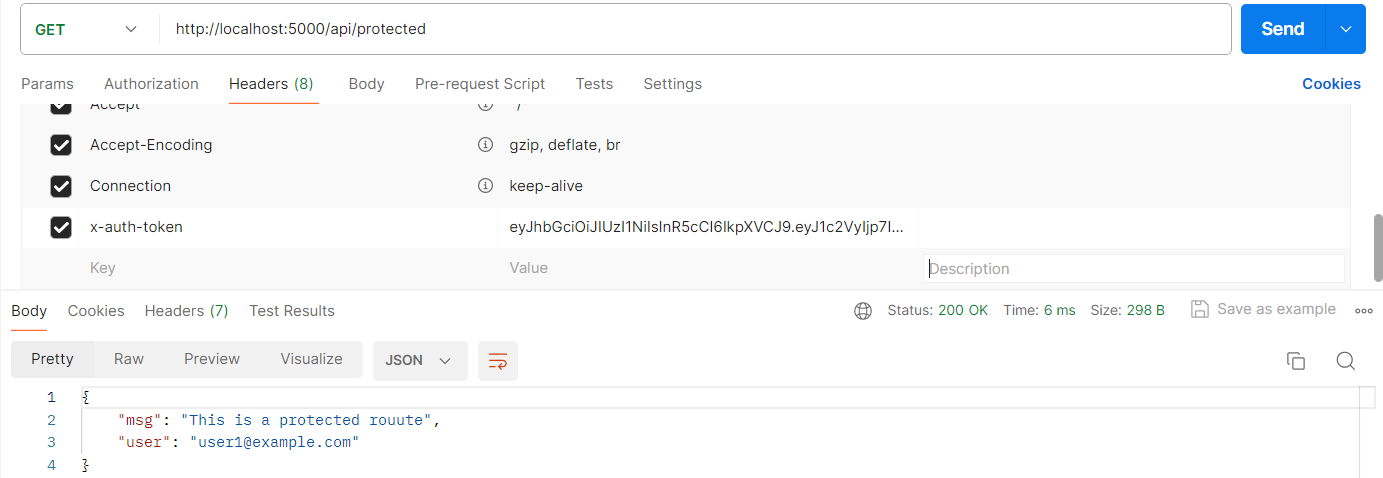
*5. Docker Configuration*

To simplify deployment, we set up MongoDB using Docker. The docker-compose.yml file defines the database container and required environment variables.

To check the REST APIs, we will use Postman. Here are some examples for checking these APIs



Now, we use this token to acces the protected route



If the token is not correct, the route cannot be accessed

