**Social Media App Documentation:**

**Project Part 2:**

The backend of the Social Media App has been developed to support various functionalities such as user authentication, post management, real-time messaging, and community engagement through channels. The backend leverages Node.js, Express, MongoDB, and other relevant technologies to ensure a robust and scalable infrastructure.

**Technology Stack:**

Backend Framework: Node.js, Express.js

Database: MongoDB

Authentication: JWT, bcrypt

Real-time Communication: Socket.io

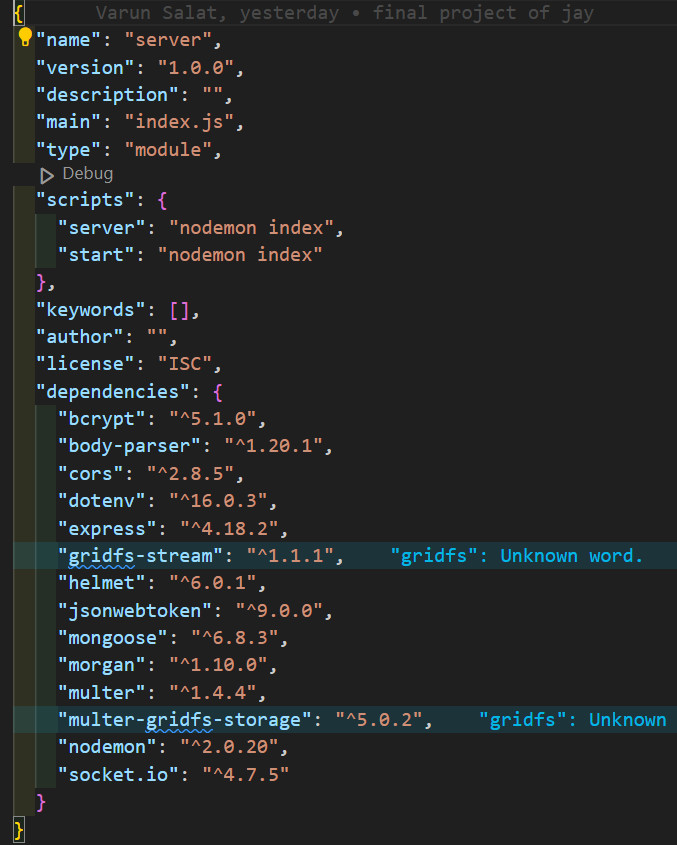
File Handling: Multer

Validation: Yup

**PROGRESS REPORT:**

**1: Dependencies Documentation:**

The following is a brief description of the dependencies installed for the back-end(Part-2) in the social media app:



**2: index.js file Documentation:**

**Overview:**

The index.js file serves as the main entry point for the backend of the Social Media App. It sets up the Express server, configures middleware, connects to MongoDB, and defines the API routes. Below is a detailed overview of its key components and functionalities:



Screenshot of multer, CORS, and socket integration:

**Explanation:**

1. **Configuration and Setup:**

Imports necessary modules such as express, body-parser, cors, mongoose, dotenv, multer, helmet, morgan, path, and url.

Configures environment variables using dotenv.

Sets up a static file server to serve files from the public/assets directory.

1. **Middleware Configuration:**

Security: Uses helmet for securing HTTP headers.

Configures helmet.crossOriginResourcePolicy to allow cross-origin requests.

Logging and Parsing:

Uses morgan for HTTP request logging.

Configures body-parser to parse incoming request bodies.

CORS:

Configures CORS to allow requests from specified origins.

1. **File Storage Configuration:**

Multer Configuration:

Configures multer for handling file uploads.

Sets up storage destination and filename settings.

1. **Socket.io Integration:**

Real-time Communication:

Sets up a Socket.io server for real-time communication.

Handles events for joining rooms and sending/receiving messages.



Routes and Mongoose setup

1. **Route Definitions:**

File Upload Routes:

Defines routes for registering users, creating posts, changing profile images, and creating channels, all with file upload capabilities.

API Routes:

Defines and uses routes for authentication (/auth), user operations (/users), post operations (/posts), channel operations (/channel), and communication (/communication).

1. **Database Connection:**

MongoDB Setup:

Configures and connects to MongoDB using Mongoose.

Sets Mongoose to use the unified topology option and disables strict query mode.

**G. Server Initialization:**

Server Start:

Starts the Express server and listens on the specified port.

Logs the server status and MongoDB connection status to the console.

**H. Test Route:**

Root Route:

Defines a simple test route (/) to verify server functionality.

1. **API Routes:**

Defines and uses routes for authentication (/auth), user operations (/users), post operations (/posts), channel operations (/channel), and communication (/communication).

1. **Auth.js file Documentation:**

**Overview:**

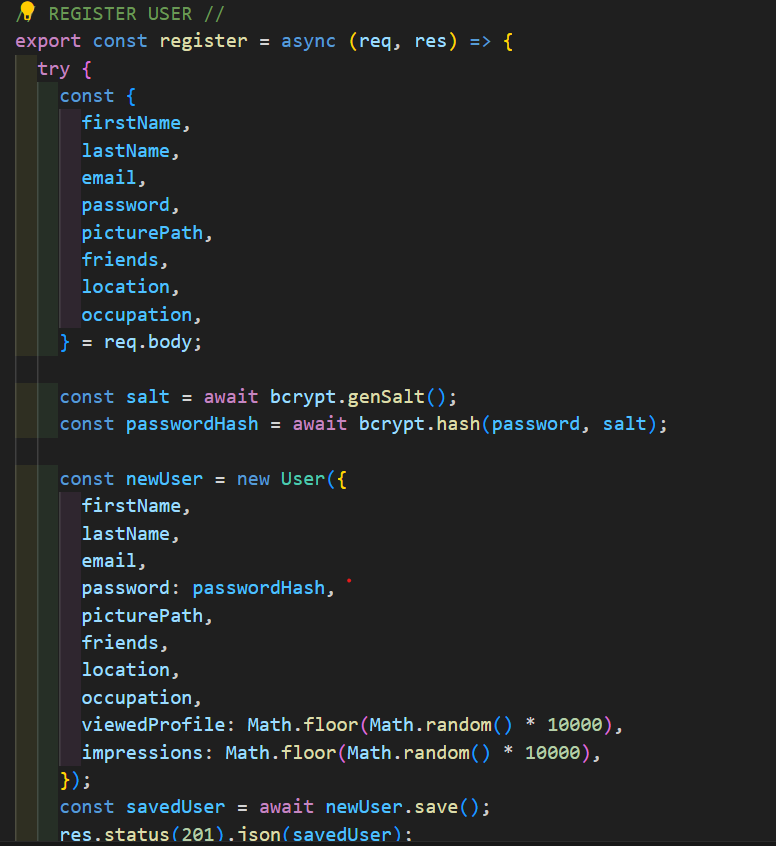
The auth.js file in the routes directory defines the routes related to authentication and user account management. It sets up the Express router and specifies the endpoints for login, password change, and username change. Here's a detailed overview:



**Route Definitions:**

1. **Register Route:**

The register function is an asynchronous Express.js controller function responsible for handling user registration. This function processes incoming registration requests, hashes the user's password, saves the new user to the MongoDB database, and returns the saved user data in the response.



Screenshot of register-user function

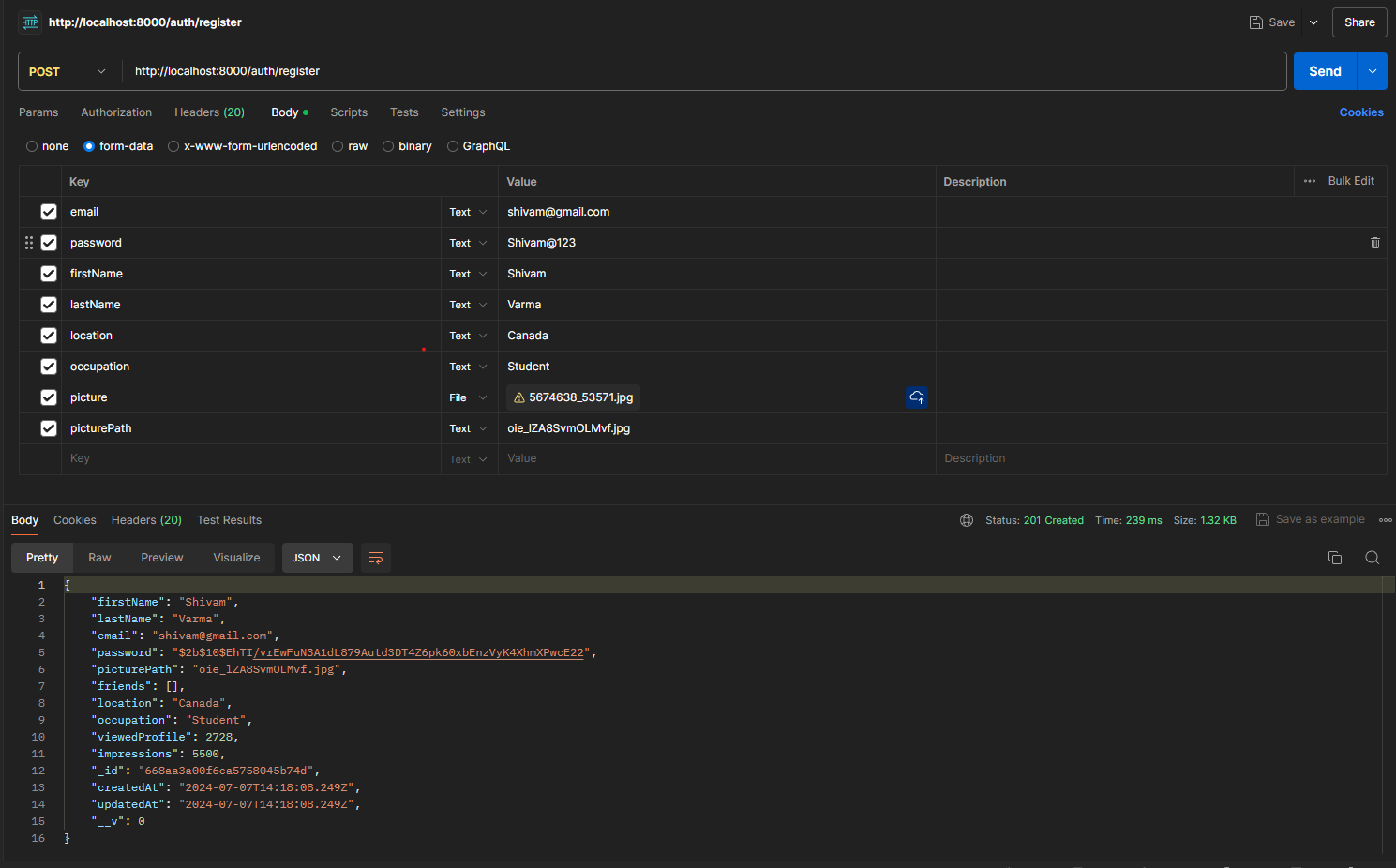
**Postman Screenshot of login API:**

**Description:**

The Postman screenshot demonstrates how to test the user registration API endpoint (/auth/register) using Postman, a popular API development tool.

**HTTP Method and URL:**

The URL is set to the API endpoint: http://localhost:8000/auth/register.

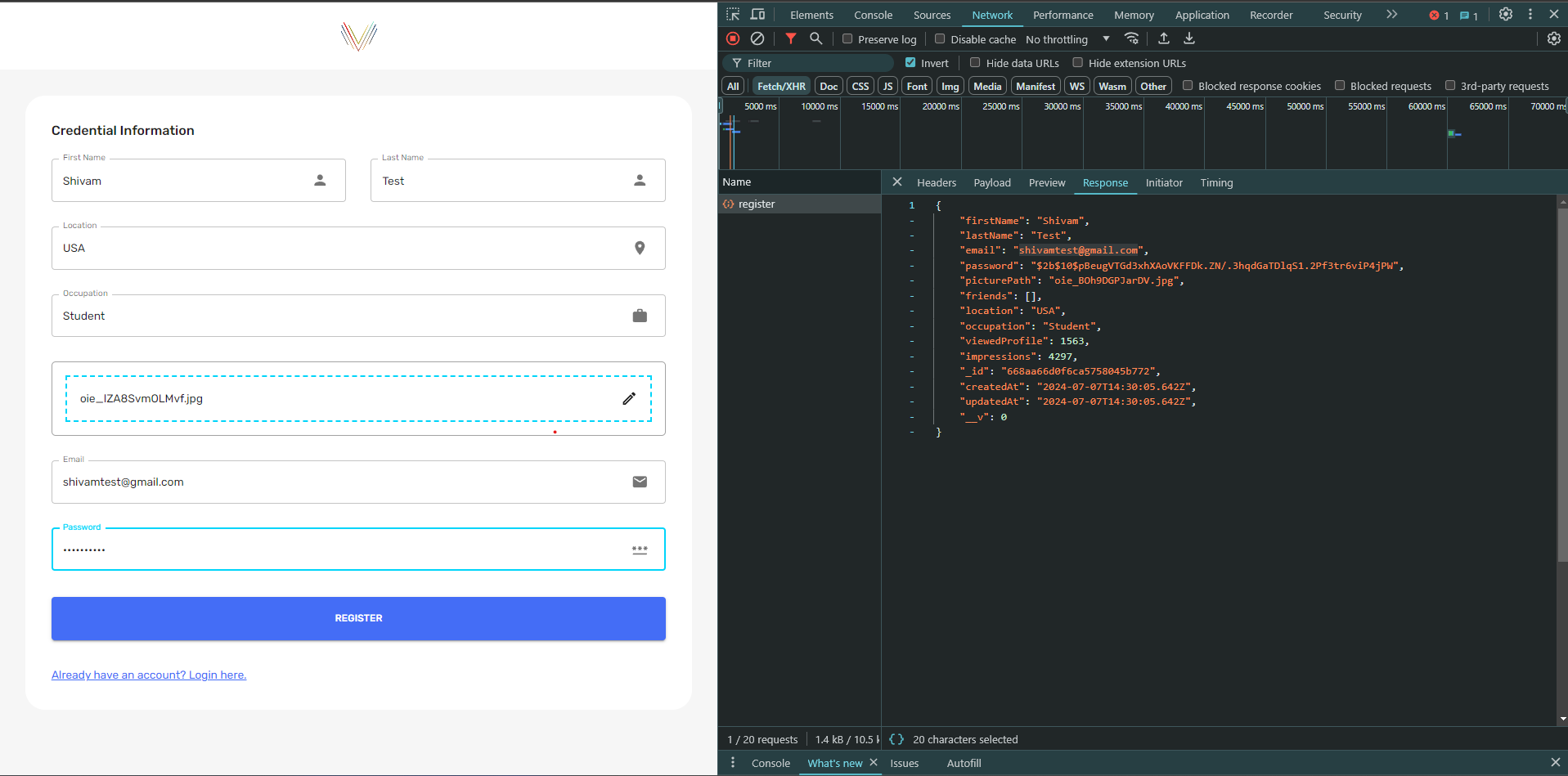


Postman screenshot of register-user function

**Browser/UI Screenshot of login API:**

**Description:**

The provided screenshot captures the browser UI along with the network panel, showcasing the interaction between the frontend application and the backend API.



Browser screenshot of register-user function

1. **login Route:**

The login function is an asynchronous function designed to handle user login requests in an Express application.

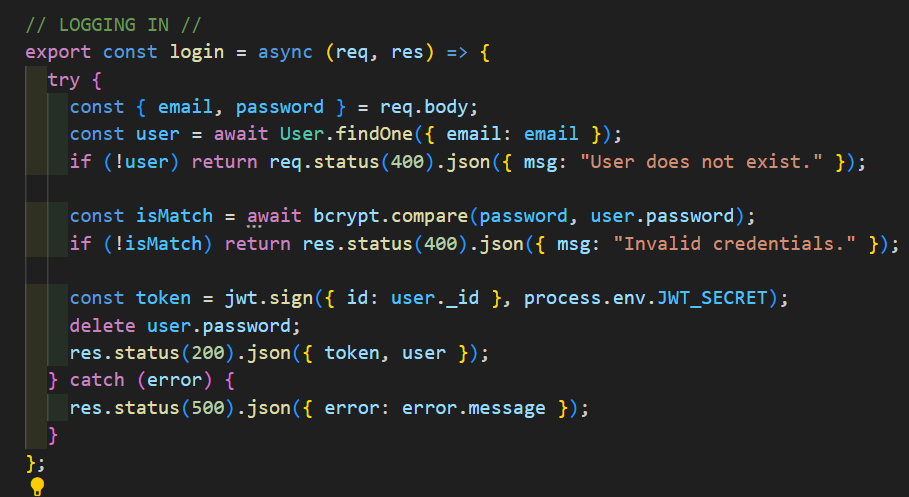
**Import Dependencies**

The function relies on several key modules, although they are not shown in the provided code snippet, these include:

User model from Mongoose.

bcrypt for password hashing and comparison.

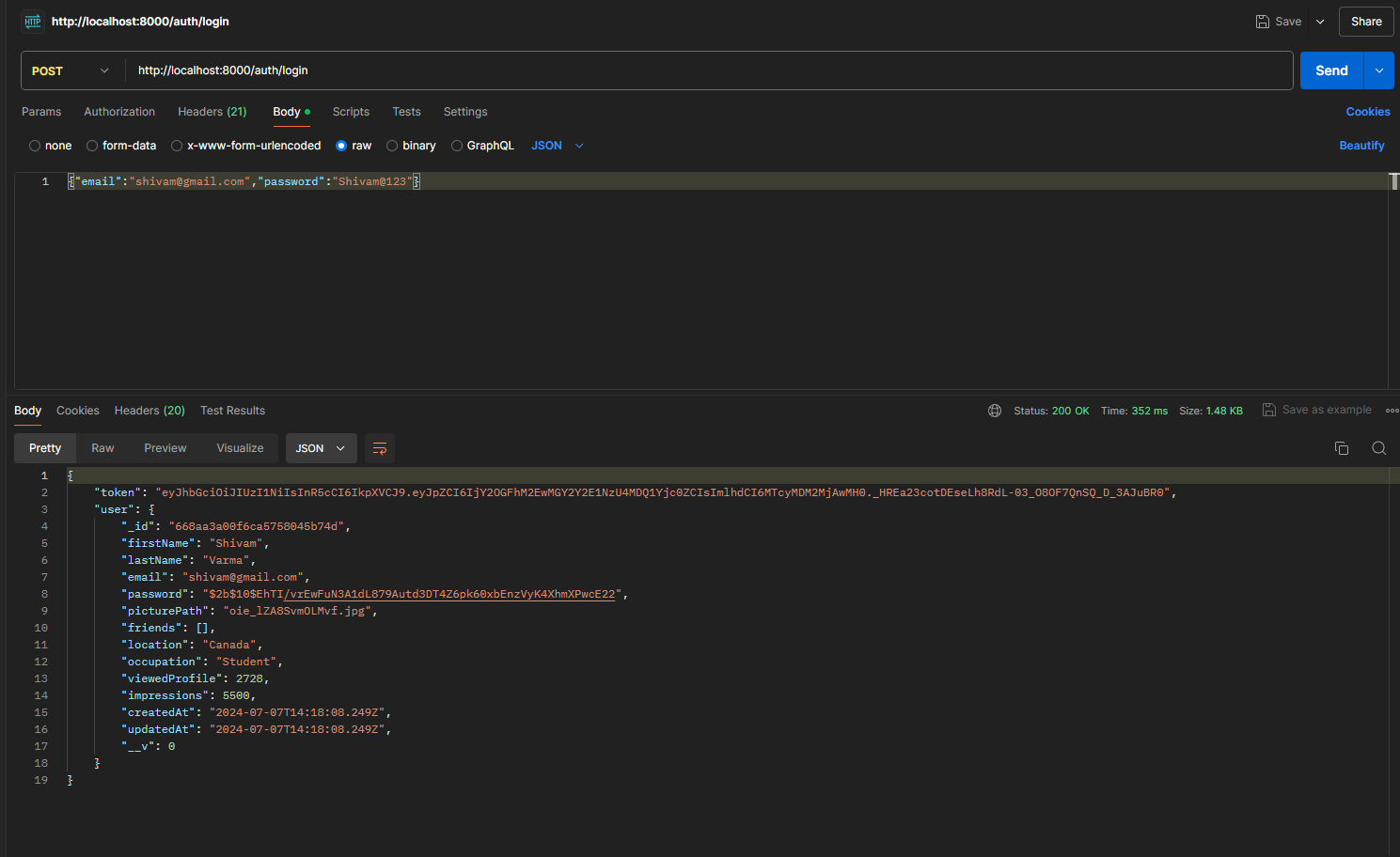
jwt for generating JSON Web Tokens (JWT).



**Postman Screenshot of login API:**

**Description:**

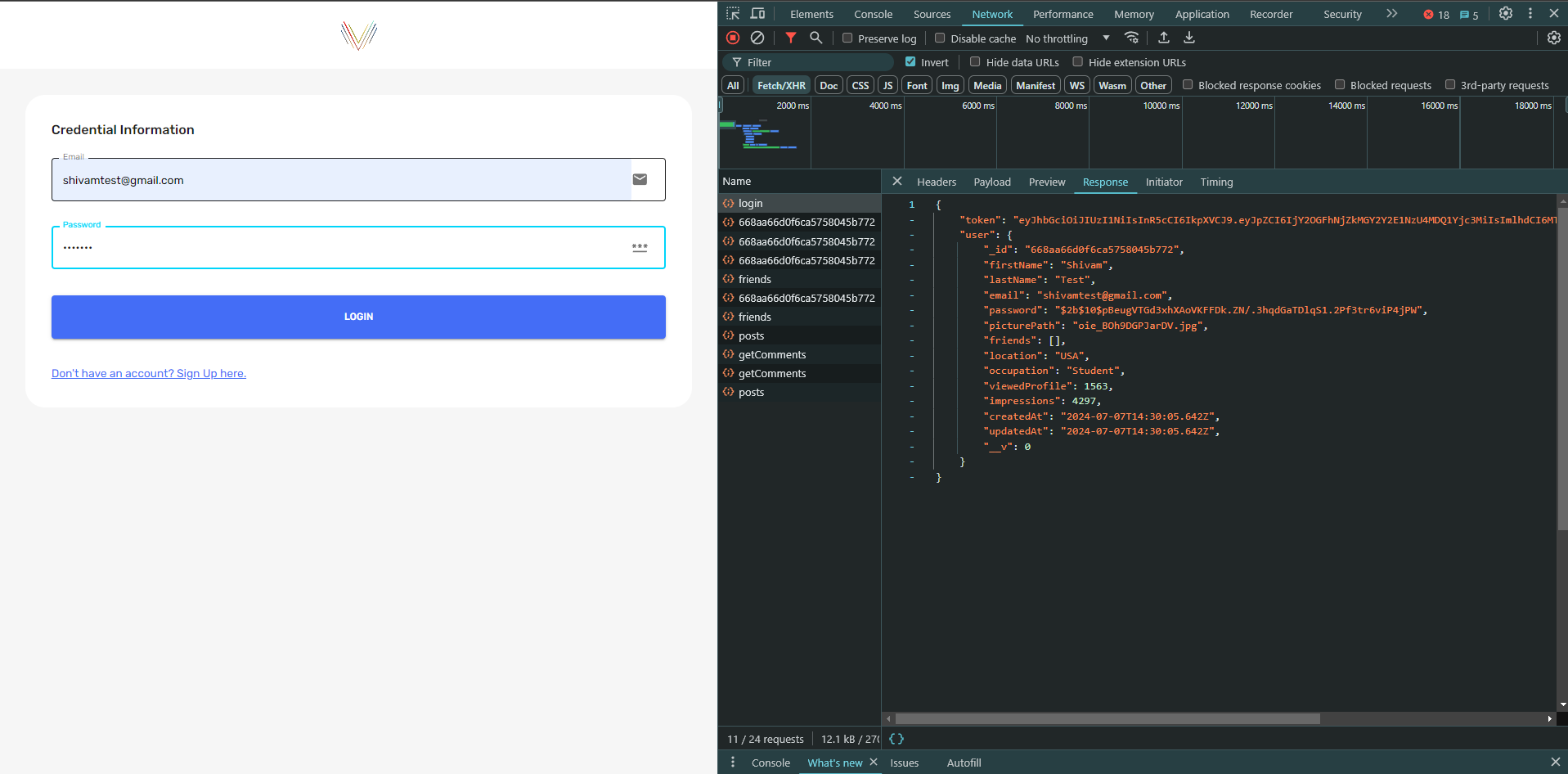
This endpoint allows a user to log in to the application by providing their email and password. Upon successful authentication, a JSON Web Token (JWT) and the user details (excluding the password) are returned.



**Browser/UI Screenshot of login API:**

**Description:**

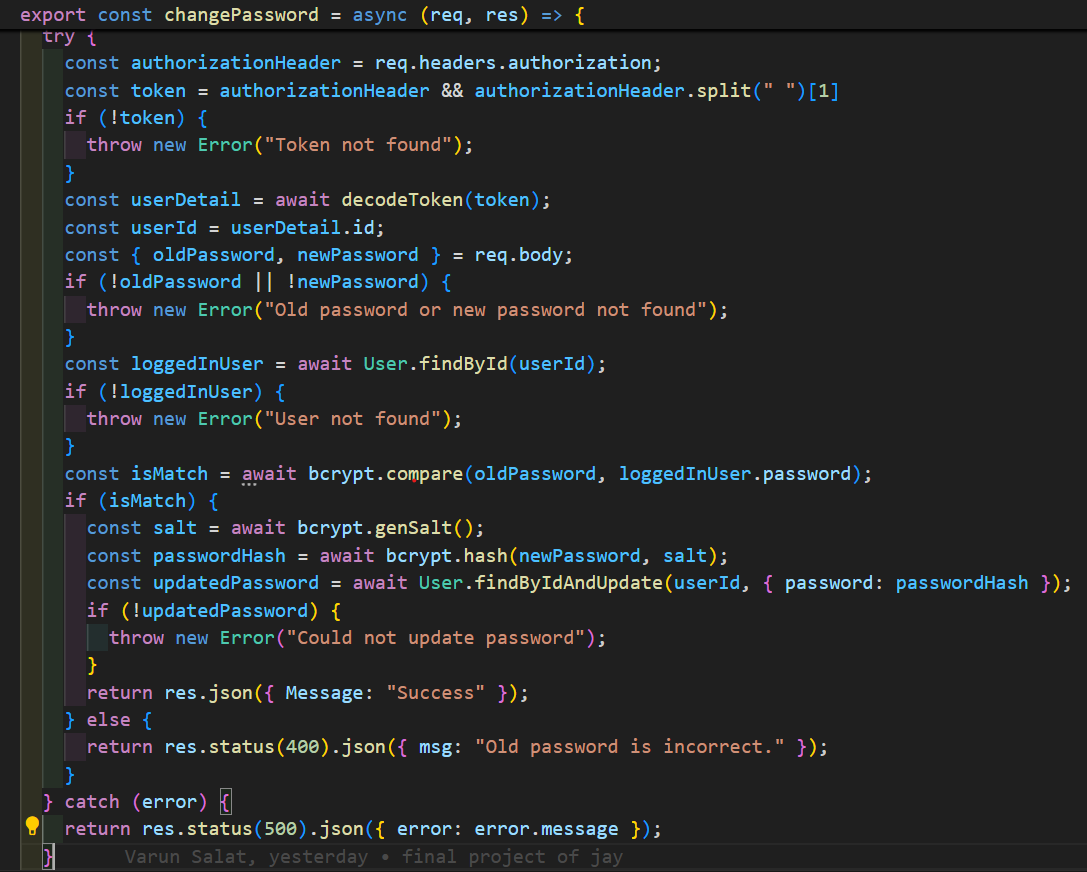
The screenshot captures the process of calling the login API endpoint from the user interface of the application, along with the Network panel of the browser's developer tools to monitor the request and response.



Postman screenshot of login-user function

1. **Change Password Route:**

The changePassword function verifies the user's identity through a JWT token, compares the provided old password with the stored hashed password in the database, and updates the password if the old password matches.



Screenshot of ChangePassword function

**Token Validation:**

The function first extracts the JWT token from the Authorization header of the incoming request.

It decodes the token to obtain the user's ID.

Request Body Handling:

It retrieves the oldPassword and newPassword from the request body.

Checks if both oldPassword and newPassword are present; if not, it throws an error.

**Password Comparison:**

Compares the oldPassword provided in the request with the hashed password stored in the database for the user.

If the passwords match, it proceeds to update the user's password.

**Password Update:**

Generates a new salt and hashes the newPassword.

Updates the user's password in the database with the newly hashed password.

**Postman Screenshot:**

**Description:**

The Postman screenshot captures the API call made to change a user's password in a social media application's backend. Here are the details highlighted in the screenshot:

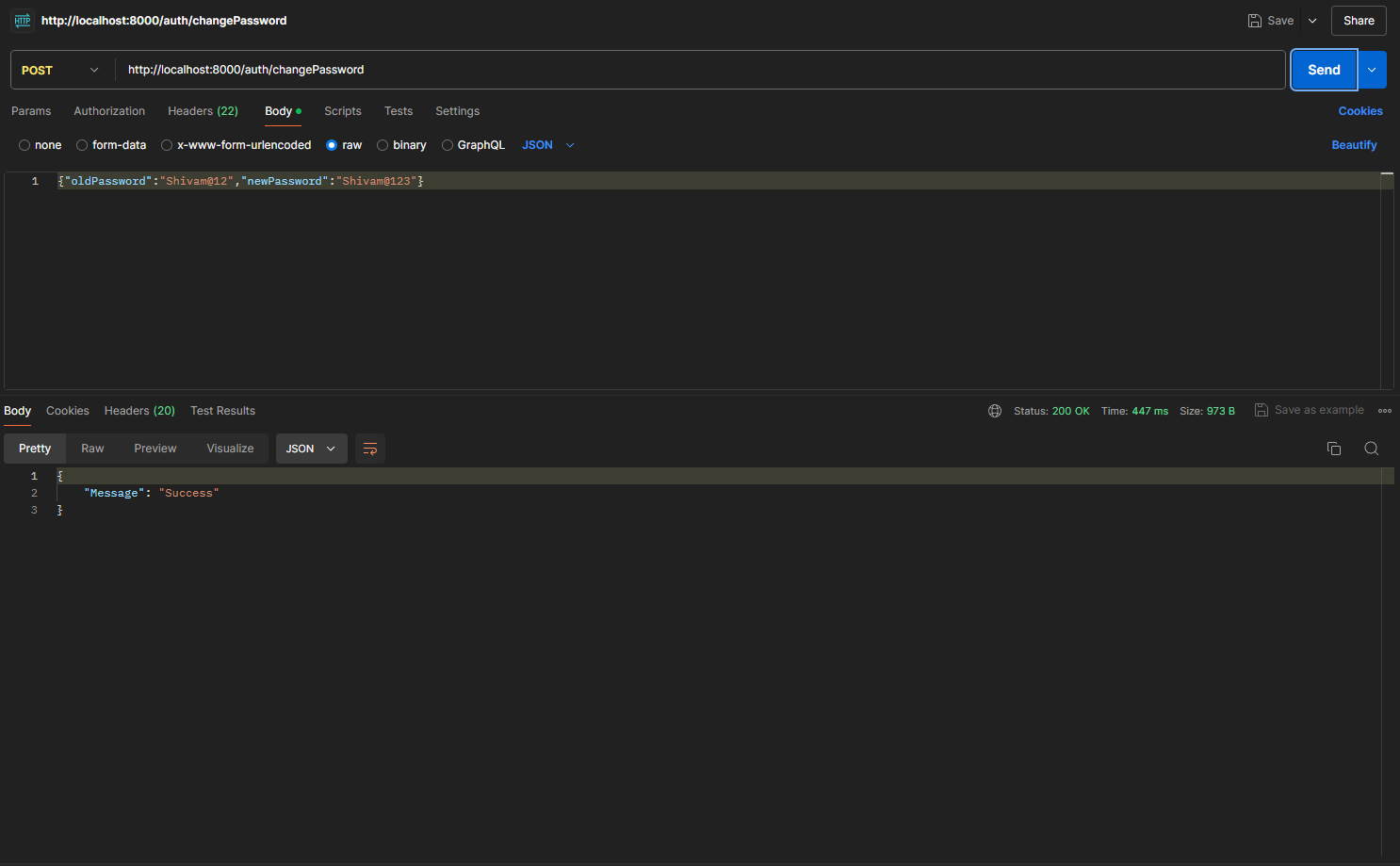
**Request Configuration:**

HTTP Method: POST

Endpoint: /auth/changePassword

Headers: Typically includes Content-Type: application/json and Authorization: Bearer [token]

Body: JSON payload containing oldPassword and newPassword.

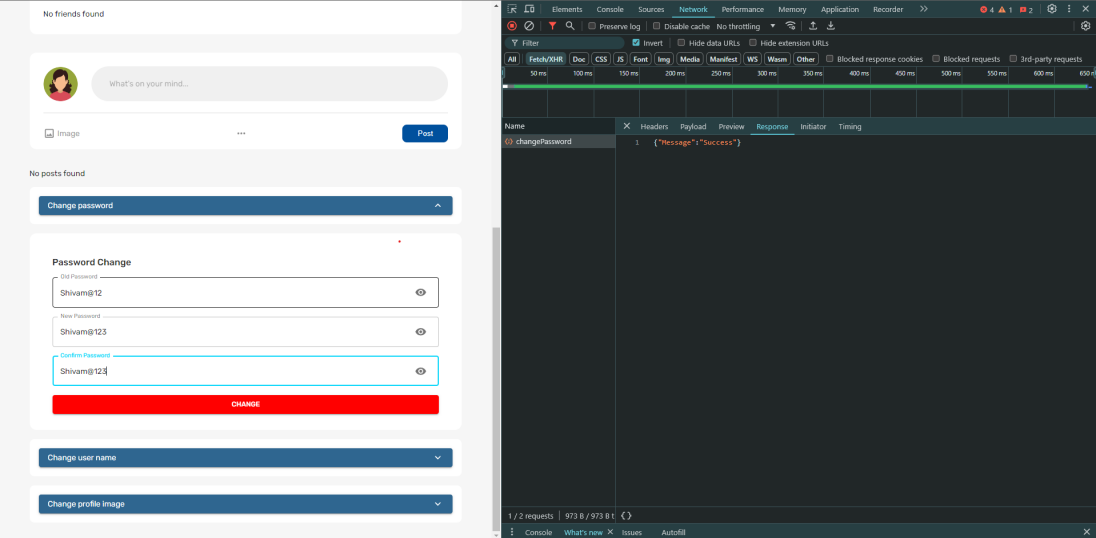


Postman Screenshot of ChangePassword function

**Browser/UI Screenshot of login API:**

**Description:**

The changePassword functionality allows users to update their passwords securely. This feature is typically accessible through the user settings or profile management section of the application, ensuring ease of access for users to maintain account security.



Browser Screenshot of ChangePassword function