

Midterm Project Report by Narmin Valiyeva, Vafa Gafarzade, Nargiz Bakshaliyeva, Shahana Huseynzade, Sevinj Rustamova

1. Advanced Session Management

Enhance session management features to include:

- Remember Me functionality using secure cookies.
- Session activity logging to detect anomalies.

Our solution:

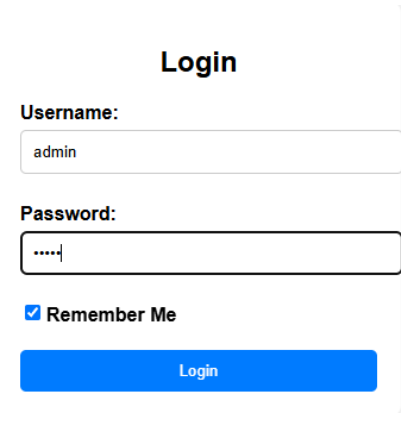
We have implemented the Remember Me feature where a secure token is generated if the user chooses to be remembered on future visits. We have put a “Remember Me” tick box in the login page.

```
<div class="login-form">
  <h1>Login</h1>
  <form action="login" method="POST">
    <label for="username">Username:</label>
    <input type="text" id="username" name="username" required>

    <label for="password">Password:</label>
    <input type="password" id="password" name="password" required>

    <div class="remember-me">
      <input type="checkbox" id="rememberMe" name="rememberMe">
      <label for="rememberMe">Remember Me</label>
    </div>

    <button type="submit">Login</button>
  </form>
</div>
```



At login, a token is created with the combination of a username, one **UUID**, and a hash made through **SHA-256**.

This token is stored in a cookie with the **HttpOnly** and **Secure** flags, which means it is accessible only over HTTPS and inaccessible via JavaScript, thus increasing security.

The cookie is set to **expire after 7 days**, allowing the user to remain logged in without re-entering credentials.

```
if ("on".equals(rememberMe)) {
    String authToken = UUID.randomUUID().toString();
    storeAuthToken(conn, username, authToken); // Save token to DB

    Cookie userCookie = new Cookie( name: "authToken", authToken);
    userCookie.setMaxAge(7 * 24 * 60 * 60); // 7 days
    userCookie.setHttpOnly(true);
    response.addCookie(userCookie);
}
```

The token is also stored in the database, and if the user revisits the site, the server checks the token against the stored value to authenticate the user.

We can see the cookies using **Browser Dev Tools**, by clicking inspect -> application -> **cookies**:

Name	Value	Domain	Path	Expires / ...	Size	HttpOnly
JSESSIONID	1DB323239B8E669DCDE639B339038AF0	localhost	/	Session	42	✓
JSESSIONID	77536530573E39DCDAD8861920D762F7	localhost	/SecureW...	Session	42	✓
authToken	b86ba8d8-d936-48f6-bc29-2339f8850459	localhost	/SecureW...	2024-12-...	45	✓
rememberMe	ILR9wDyLaChnRWIARv1ZBNsxVt03Ryl	localhost	/	2025-01-...	42	✓

When a user logs in, a secure session cookie is created and stored in the server's database. On subsequent visits, the server retrieves **the session cookie** from the user's request to **validate** their session and identify them, ensuring a seamless and secure experience without requiring **re-login**.

```
mysql> select * from SessionLogs ;
```

session_id	username	ip_address	login_time	logout_time
CE3C0F00FE21E8F6A5498034D7EF9B8A	admin	0:0:0:0:0:0:1	2024-12-11 20:53:13	NULL

We store "Remember Me" cookies securely in the user_token table in the database. When a user opts for "Remember Me," a unique token is generated, stored with the user's ID and expiration time, and set in a secure cookie. On future visits, the server checks the token against the database to automatically authenticate the user, ensuring both security and convenience.

```
private void storeAuthToken(Connection conn, String username, String authToken) throws SQLException {
    String sql = "UPDATE Users SET auth_token = ? WHERE username = ?";
    PreparedStatement stmt = conn.prepareStatement(sql);
    stmt.setString(1, authToken);
    stmt.setString(2, username);
    stmt.executeUpdate();
}
```

When the user revisits the site, we retrieve the "Remember Me" token from the cookie and validate it against the stored token in the user_token table. If the token is valid and has not expired, the user is automatically authenticated, granting them access without needing to re-enter their credentials.

```
if (session == null || session.getAttribute(s:"username") == null) {
    Cookie[] cookies = request.getCookies();
    String username = null;

    // Check if the "username" cookie exists
    if (cookies != null) {
        for (Cookie cookie : cookies) {
            if ("username".equals(cookie.getName())) {
                username = cookie.getValue();
                break; // If cookie found, exit the loop
            }
        }
    }

    // If no username cookie is found, redirect to login
    if (username == null) {
        response.sendRedirect(s:"Login.jsp");
        return;
    }

    // Set session attributes from cookie
    session = request.getSession(b:true); // Create a new session if it doesn't exist
    session.setAttribute(s:"username", username);
}
```

2. User Role-Based Access Control (RBAC)

Implement role-based access control:

- Roles: Admin, Moderator, and User.
- Admin: Full access to all features.
- Moderator: Manage uploaded files.
- User: Limited to upload files.

Our solution:

2. User Role-Based Access Control (RBAC)

We implemented the use of Role-Based Access Control to ensure that users in different roles can have access to certain features in the application. The three defined roles are Admin, Moderator, and User. Each has different privileges of access.

Roles Defined:

Admin: Full access in all features, including user management and file upload, among other administrative tasks.

Moderator: Can manage and approve uploaded files, ensuring that they meet the application's standards.

User: Restricted to uploading files only, with no access to other functionalities within the system.

In the LoginServlet, based on the role it is redirected jsp files accordingly:

```
if ("admin".equals(role)) {  
    response.sendRedirect(location: "adminWelcome");  
} else if ("moderator".equals(role)) {  
    response.sendRedirect(location: "moderatorWelcome");  
} else {  
    response.sendRedirect(location: "welcome"); // Default for regular users
```

The names and profile photos of users are retrieved from the **Users table**, accessible to **admin and moderator roles**, as they are authorized to view this information.

```
// If user is an admin or moderator, fetch all user profiles  
if ("moderator".equalsIgnoreCase(userRole) || "admin".equalsIgnoreCase(userRole)) {  
    String allUsersQuery = "SELECT username, profile_picture FROM users";  
    PreparedStatement allUsersStmt = conn.prepareStatement(allUsersQuery);  
    ResultSet allUsersRs = allUsersStmt.executeQuery();  
  
    while (allUsersRs.next()) {  
        Map<String, String> userProfile = new HashMap<>();  
        userProfile.put("username", allUsersRs.getString(columnLabel: "username"));  
        String pic = allUsersRs.getString(columnLabel: "profile_picture");  
        userProfile.put("profilePicture", pic != null && !pic.isEmpty() ? "profile_pics/" + pic : "profile_pics/default.jpg");  
        userProfiles.add(userProfile);  
    }  
}
```

In the **web.xml**, we define three distinct **welcome.jsp** pages, each corresponding to a different user role. These pages are mapped to the **WelcomeServlet**, ensuring users are redirected based on their specific roles.

```

<servlet>
  <servlet-name>WelcomeServlet</servlet-name>
  <servlet-class>com.secure_web.servlets.WelcomeServlet</servlet-class>
</servlet>
<servlet-mapping>
  <servlet-name>WelcomeServlet</servlet-name>
  <url-pattern>/welcome</url-pattern>
  <url-pattern>/moderatorWelcome</url-pattern>
  <url-pattern>/adminWelcome</url-pattern>
</servlet-mapping>

```

Both **moderators** and **admins** can view users' usernames and profile pictures, as these are displayed on the JSP pages through a **request** sent to the **userProfiles** endpoint.

```

<div class="profile-gallery">
  <!-- Assuming 'userProfiles' is a list of objects containing 'username' and 'profilePicture' URLs -->
  <%
    List<Map<String, String>> userProfiles = (List<Map<String, String>>) request.getAttribute("userProfiles");
    if (userProfiles != null) {
      for (Map<String, String> userProfile : userProfiles) {
        <%
          <div class="profile-item">
            " alt="<%= userProfile.get("username") %>'s Profile Picture">
            <p><%= userProfile.get("username") %></p>
          </div>
        <%
          }
        <%
          } else {
            <p>No user profiles available.</p>
          <%
            }
          <%
        </div>

```

In the **adminWelcome.jsp**, when the "Delete" button is clicked, the request is redirected to the **deleteProfilePictureServlet** for processing.

```

<form action="/deleteProfilePicture" method="post" onsubmit="return confirm('Are you sure you want to delete this profile picture?');">
  <input type="hidden" name="username" value="<%= userProfile.get("username") %>" />
  <button type="submit" class="delete-btn">Delete Profile Picture</button>
</form>

```

When the **DeleteProfilePictureServlet** receives the request, it updates the user's profile photo to the default default.jpg and removes the previous profile picture from the system.

```

public class DeleteProfilePictureServlet extends HttpServlet {
  protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException {
    try {
      // Load environment variables for database credentials
      Dotenv dotenv = Dotenv.load();
      String dbUrl = dotenv.get("DB_URL");
      String dbUsername = dotenv.get("DB_USERNAME");
      String dbPassword = dotenv.get("DB_PASSWORD");

      // Establish a connection to the database
      try (Connection conn = DriverManager.getConnection(dbUrl, dbUsername, dbPassword)) {
        // SQL query to update the profile picture to default
        String sql = "UPDATE users SET profile_picture = ? WHERE username = ?";
        PreparedStatement stmt = conn.prepareStatement(sql);
        stmt.setString(1, "default.jpg"); // Set the profile picture
        stmt.setString(2, usernameToDelete); // Set the username for which
        int rowsUpdated = stmt.executeUpdate();
        if (rowsUpdated > 0) {
          // Profile picture was successfully updated to the default
          response.sendRedirect(location: "adminWelcome"); // Redirect back to the admin page
        } else {
          // Handle the case where no rows were updated
          response.getWriter().println("Error: User not found.");
        }
      } catch (SQLException e) {
        e.printStackTrace();
        response.getWriter().println("Database error. Please try again later.");
      }
    } catch (Exception e) {
      e.printStackTrace();
      response.getWriter().println("Error processing the request.");
    }
  }
}

```

The user roles are stored in the database in the Users table. Every record of a user includes a field for the role, which is set to either 'Admin', 'Moderator', or 'User'. This role information is used in enforcing access control during the time of login and feature checks.

Upon successful login, the user's role is stored in the session. This session attribute then is used to manage the access control by checking the user's role before allowing them into certain pages or actions. The session ensures that users can only interact with features they are authorized to use.

3. Enhanced File Upload Security

Improve the file upload feature:

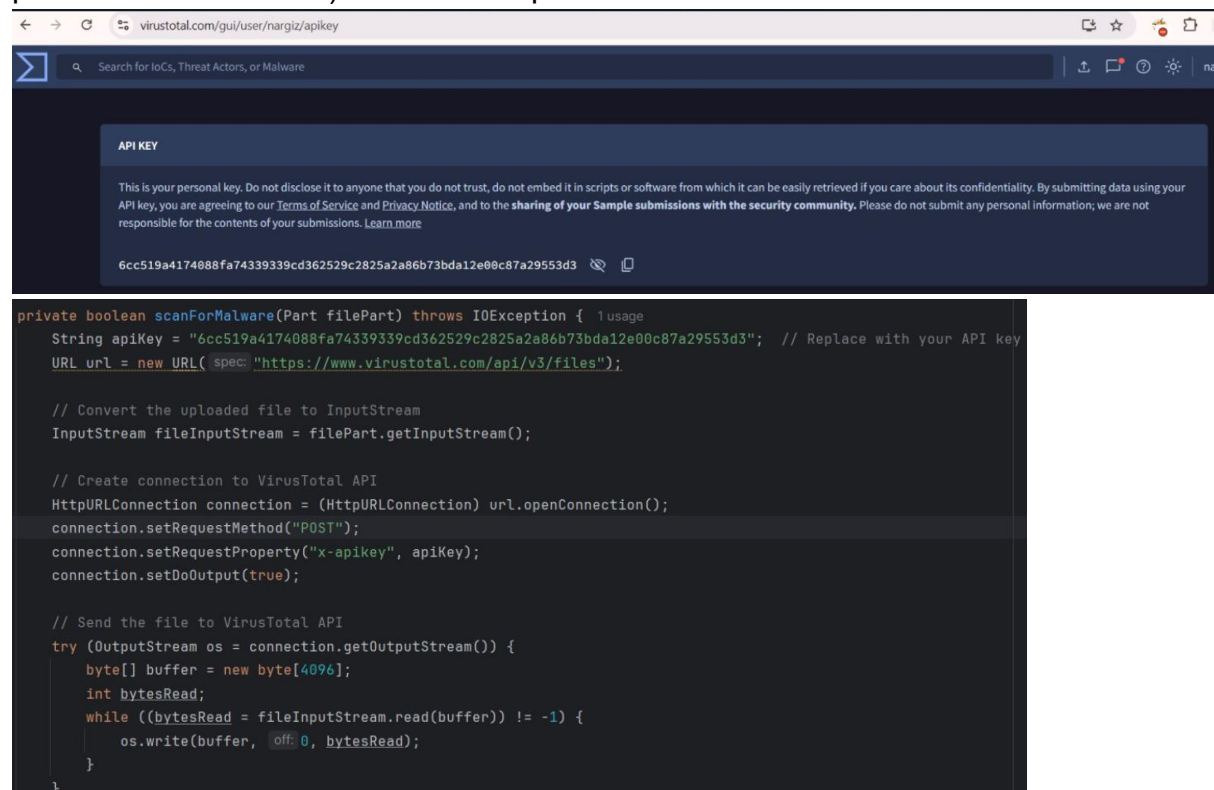
- Restrict file types to images only (e.g., JPG, PNG).
- Scan uploaded files for potential malware.
- Implement file size restrictions (max 5MB).

Our solution:

We restricted file uploads to only **image files (JPG, PNG)** by checking the file's MIME type and extension. This ensures that users cannot upload potentially dangerous files like executable files. We use regex to filter out the filename.

```
// Validate file type (only .jpg, .jpeg, .png)
if (!fileName.toLowerCase().matches( regex: ".+\\.?(jpg|jpeg|png)$")) {
    response.getWriter().println("Invalid file type. Only .png, .jpg, and .jpeg are allowed.");
    logger.warn( message: "Invalid file type uploaded by user: {}", session.getAttribute( s: "username"));
    return;
}
```

To **prevent** the upload of **malicious files**, we integrated a malware scanning process. For this, we used an external antivirus scanning tool like **VirusTotal** (or any preferred antivirus API) to scan the uploaded files.



The image shows a screenshot of the VirusTotal API key page and a corresponding Java code snippet. The top part of the image displays the VirusTotal website interface, including the search bar and the API key section. The API key is highlighted as a long alphanumeric string. Below this, the Java code is shown, which implements a method to scan uploaded files for malware using the VirusTotal API. The code includes comments and uses the `HttpURLConnection` class to send the file data to the API.

```
private boolean scanForMalware(Part filePart) throws IOException {
    usage
    String apiKey = "6cc519a4174088fa74339339cd362529c2825a2a86b73bda12e00c87a29553d3"; // Replace with your API key
    URL url = new URL( spec: "https://www.virustotal.com/api/v3/files");

    // Convert the uploaded file to InputStream
    InputStream fileInputStream = filePart.getInputStream();

    // Create connection to VirusTotal API
    HttpURLConnection connection = (HttpURLConnection) url.openConnection();
    connection.setRequestMethod("POST");
    connection.setRequestProperty("x-apikey", apiKey);
    connection.setDoOutput(true);

    // Send the file to VirusTotal API
    try (OutputStream os = connection.getOutputStream()) {
        byte[] buffer = new byte[4096];
        int bytesRead;
        while ((bytesRead = fileInputStream.read(buffer)) != -1) {
            os.write(buffer, 0, bytesRead);
        }
    }
}
```


After saving the file temporarily, we run it through a malware scanner. If the file is detected as malicious, it is rejected.

```
// Scan the uploaded file for malware
if (scanForMalware(filePart)) {
    response.getWriter().println("File contains malware. Please upload a clean file.");
    logger.warn( message: "Malware detected in file uploaded by user: {}", session.getAttribute( s: "username"));
    return;
}
```

We set a **file size limit** to prevent users from uploading excessively large files, which could strain the server. The maximum file size allowed is set to **5MB**.

Implementation: We check the file size before proceeding with the upload process.

```
public class UploadProfilePictureServlet extends HttpServlet {
    private static final Logger logger = LogManager.getLogger(UploadProfilePictureServlet.class); // 7 usages
    private static final String UPLOAD_DIR = "profile_pics"; // 1 usage
    private static final long MAX_FILE_SIZE = 5 * 1024 * 1024; // 5MB 1 usage
}
```

4. Secure API Development

Expose a RESTful API for the application:

- Endpoints for user login and file uploads.
- Apply rate limiting to prevent abuse.

Our solution:

The core of secure API development for this application is to expose endpoints for user login and file uploads. These endpoints will be utilized to log in users and upload files, ensuring that each is done securely.

1) User Login Endpoint

The API will receive user credentials, usually a username and password, authenticate the user, and generate a JWT that will be sent back to the client; this token will be used in subsequent requests for authentication.

2) File Upload Endpoint

The file upload endpoint makes it possible for users to upload files onto the server, such as images, documents, and so on. This gets received as an `InputStream`; thus, this stream could be saved onto the server or processed.

```
@Path("/user") // no usages
public class UserApi {
    // Simulated user authentication (replace with actual DB logic)
    private boolean authenticateUser(String username, String password) { // 1 usage
        return "validUser".equals(username) && "validPassword".equals(password);
    }

    @POST
    @Path("/login")
    @Consumes(MediaType.APPLICATION_JSON)
    @Produces(MediaType.APPLICATION_JSON)
    public Response login(UserCredentials credentials, @Context HttpServletResponse response) {
        // Authenticate the user
        if (authenticateUser(credentials.getUsername(), credentials.getPassword())) {
            // Generate JWT token (simplified for example)
            String token = generateJwtToken(credentials.getUsername());
            // Set the JWT token in the response header
            response.setHeader( name: "Authorization", value: "Bearer " + token);
            // Return success response
            return Response.status(Response.Status.OK).entity("Login successful").build();
        }
        // Invalid credentials response
        return Response.status(Response.Status.UNAUTHORIZED).entity("Invalid credentials").build();
    }

    // Method to generate a JWT (simplified for the example)
    private String generateJwtToken(String username) { // 1 usage
        // In a real application, use a library like JWT or Auth0 to generate the token
        return "jwt-token-for-" + username;
    }
}
```

```
@Path("/user") // no usages
public class FileUploadApi {
    @POST // no usages
    @Path("/upload")
    @Consumes(MediaType.MULTIPART_FORM_DATA)
    @Produces(MediaType.APPLICATION_JSON)
    public Response uploadFile(@FormParam("file") InputStream fileInputStream) {
        // Process the file (save it, analyze it, etc.)
        // For example, save the file to disk (you need to implement the actual saving logic)
        // Simulate saving the file
        String filePath = "uploads/uploaded_file";
        saveFile(fileInputStream, filePath);
        // Return a success response
        return Response.status(Response.Status.OK).entity("File uploaded successfully").build();
    }

    private void saveFile(InputStream fileInputStream, String filePath) { // 1 usage
        // Implementation for saving file to disk
        // For example: Files.copy(fileInputStream, Paths.get(filePath), StandardCopyOption.REPLACE_EXISTING);
    }
}
```

3) Abuse Prevention by Rate Limiting

The rate limit filter will monitor the number of requests from each client based on its IP or authentication token and may allow or reject based on the limit.

Max Requests: We are only allowing a certain number of requests in a time window.

Example: 100 requests in a minute.

Tracking Requests: We maintain a record of requests coming from each client (which can be identified by IP or authentication token) in the past minute.

```
import javax.ws.rs.*;
import javax.ws.rs.core.*;
import java.util.concurrent.*;

@Provider
public class RateLimitingFilter implements ContainerRequestFilter {

    private static final int MAX_REQUESTS_PER_MINUTE = 100; // Max requests per minute 1 usage
    private static final long TIME_WINDOW = TimeUnit.MINUTES.toMillis(1); // Time window (1 minute) 1 usage
    private static final Map<String, RequestTracker> requestMap = new HashMap<>(); // 2 usages

    @Override
    public void filter(ContainerRequestContext requestContext) throws WebApplicationException {
        String clientId = requestContext.getUriInfo().getRequestUri().getHost(); // Get client IP

        // Get or create a RequestTracker for the client
        RequestTracker tracker = requestMap.getOrDefault(clientId, new RequestTracker());

        // Check if the client has exceeded the rate limit
        if (!tracker.isRequestAllowed()) {
            requestMap.put(clientId, tracker); // Update the tracker
        } else {
            // Reject the request if rate limit exceeded
            Response rateLimitExceededResponse = Response.status(Response.Status.TOO_MANY_REQUESTS)
                .entity("Rate limit exceeded. Try again later.")
                .build();
            requestContext.abortWith(rateLimitExceededResponse);
        }
    }
}

// Inner class to track request count and time
private static class RequestTracker {
    private long lastRequestTime;
    private int requestCount;

    public RequestTracker() {
        this.lastRequestTime = System.currentTimeMillis();
        this.requestCount = 0;
    }

    // Check if the request is allowed based on rate limit
    public boolean isRequestAllowed() {
        long currentTime = System.currentTimeMillis();

        // Reset request count if the time window has passed
        if (currentTime - lastRequestTime > TIME_WINDOW) {
            requestCount = 0;
            lastRequestTime = currentTime;
        }

        // Allow the request if under the limit
        if (requestCount < MAX_REQUESTS_PER_MINUTE) {
            requestCount++;
            return true;
        }
    }
}
```

If the client exceeds the allowed number of requests within the time window, the request is rejected with a HTTP 429 Too Many Requests status. To register the Rate Limiting Filter in the application's JAX-RS configuration (ApplicationConfig.java or equivalent).

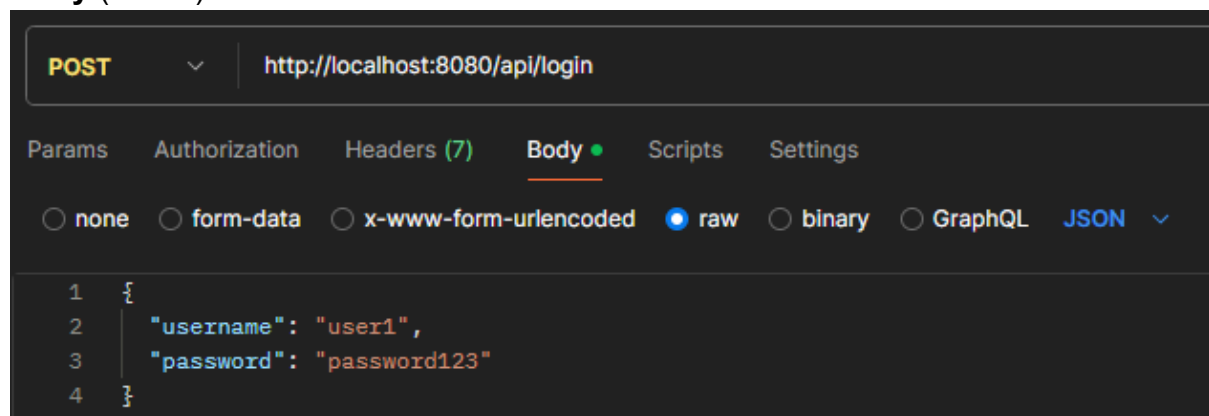
Let's send request to our **REST API** using the **Postman App**:

First test out **User Login Endpoint** by entering the credentials in **JSON** format.

URL: http://localhost:8080/api/login

Method: POST

Body (JSON):



If the credentials are correct, the response will contain a success message or **authentication token**:

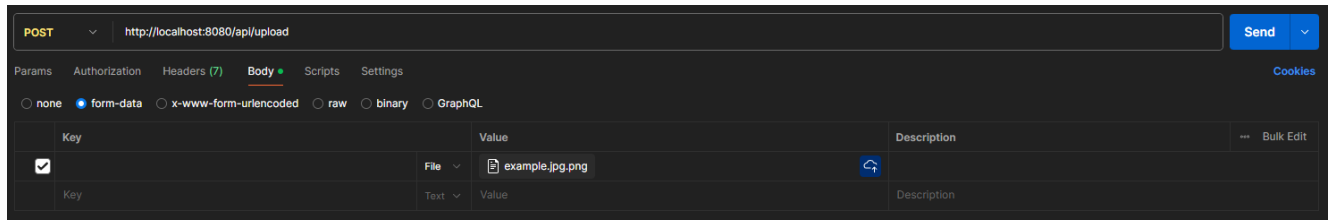
```
{
  "status": "success",
  "token": "abcdef1234567890"
}
```

Now let's test the **File Upload Endpoint** by sending a jpg file:

URL: <http://localhost:8080/api/upload>

Method: POST

Body: Select form-data and add a file under the "file" key.



If the file is valid according to restrictions on image type (**JPG, PNG**) and size (**5MB**) we get a **success message** and **upload the image**:

```
{
  "status": "success",
  "message": "File uploaded successfully."
}
```

Rate Limiting Applied: Rate limiting is applied to each request to the API.

User Login: A post request with credentials can be sent to /user/login, which authenticates the user and sends a response with a JWT token.

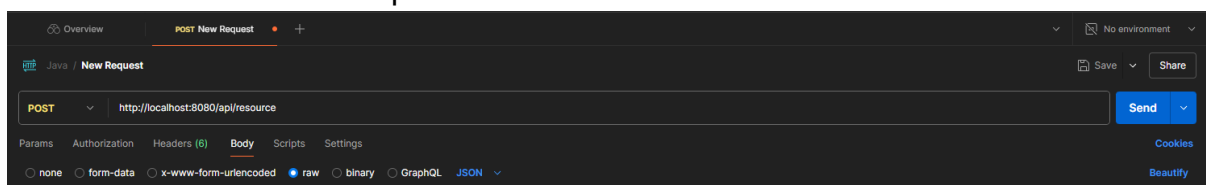
File Upload: A post request can be made to /user/upload with the attached file in form data of multipart media type. If the rate limit for a user exceeds, the requests will be rejected with a 429 Too Many Requests error.

Let's test the **Rate Limiting** feature:

URL: <http://localhost:8080/api/resource>

Method: GET or POST

Then we click to send multiple times to limit the rate:



If rate limiting is implemented, you should see a response with a status code **429 Too Many Requests** after making the allowed number of requests (e.g., 5 requests).

```
{
  "error": "Rate limit exceeded. Try again later."
}
```

5. Comprehensive Logging and Monitoring

Implement an advanced logging system:

- Log all significant user actions, including failed login attempts and file uploads.

- Send alerts for suspicious activities (e.g., multiple failed login attempts).
- Store logs securely in a database or a file system with restricted access.

Our solution:

In this solution, an advanced logging and monitoring system is implemented to improve the security of user login attempts, as well as the application's general behavior. It provides tracking for major actions such as failed login attempts and successful logins. The system also identifies suspicious activities like multiple failed attempts at logging in, suspected to be brute-force attacks, and stores logs securely to ensure that unauthorized access is prevented.

The **isUserLockedOut** method queries the **LoginAttempts** table to fetch the **failed_attempts** and **last_failed_attempt** for a given username. If the number of failed attempts exceeds the limit (**MAX_FAILED_ATTEMPTS**), and the time since the last failed attempt is less than the lockout duration (**LOCKOUT_TIME**), it returns true to indicate the user is locked out.

```
private boolean isUserLockedOut(Connection conn, String username) throws SQLException {
    String sql = "SELECT failed_attempts, last_failed_attempt FROM LoginAttempts WHERE username = ?";
    PreparedStatement stmt = conn.prepareStatement(sql);
    stmt.setString(1, username);
    ResultSet rs = stmt.executeQuery();

    if (rs.next()) {
        int failedAttempts = rs.getInt("failed_attempts");
        long lastFailedAttempt = rs.getLong("last_failed_attempt");
        long currentTime = System.currentTimeMillis();

        return failedAttempts >= MAX_FAILED_ATTEMPTS && (currentTime - lastFailedAttempt) < LOCKOUT_TIME;
    }
    return false;
}
```

When a failed login attempt is detected, the method checks if the username exists in the **LoginAttempts** table. If the user doesn't exist, it inserts a new record with **failed_attempts = 1** and the current time as **last_failed_attempt**. If the user exists, it increments the **failed_attempts** count and updates the **last_failed_attempt** time with the current timestamp.

```
private void incrementFailedAttempts(Connection conn, String username) throws SQLException {
    String sql = "INSERT INTO LoginAttempts (username, failed_attempts, last_failed_attempt) VALUES (?, 1, ?) " +
        "ON DUPLICATE KEY UPDATE failed_attempts = failed_attempts + 1, last_failed_attempt = ?";
    PreparedStatement stmt = conn.prepareStatement(sql);
    stmt.setString(1, username);
    stmt.setLong(2, System.currentTimeMillis());
    stmt.setLong(3, System.currentTimeMillis());
    stmt.executeUpdate();
}
```

When the user successfully logs in, this method resets the **failed_attempts** count to **0** in the **LoginAttempts** table for the specified username.

```
private void resetFailedAttempts(Connection conn, String username) throws SQLException {
    String sql = "UPDATE LoginAttempts SET failed_attempts = 0 WHERE username = ?";
    PreparedStatement stmt = conn.prepareStatement(sql);
    stmt.setString(1, username);
    stmt.executeUpdate();
}
```

The solution implements login attempt tracking, a lockout mechanism after multiple failed attempts, and secure logging with alerting for suspicious activity to enhance security and enable rapid response to potential threats.