FullStackers

Team Members & Roles:

- Dumistracel Eduard-Costin SRIC1 - Member
- Alexandra Cornea SRIC1 – Member
- Alexandru Pascu SRIC1 - Member
- Alexandru Maxim SRIC1 - Member
- Alex Puflene SRIC1 - Member

2. Project Summary

- Project Overview:

SS_Project is a full-stack application that enables real-time image transmission from IoT devices (such as Android mobile devices with camera functionality) to a backend server using MQTT with mutual TLS (mTLS) encryption. The images are stored in a PostgreSQL database and can be viewed through a web interface.

- Key Feature

Android Client App

  - Captures images using the device's camera
  - Connects securely to the MQTT broker using mTLS
  - Sends base64-encoded image data to a specific MQTT topic with metadata

Secure MQTT Communication

  - Uses Eclipse Mosquitto as the MQTT broker configured with mutual TLS authentication inside a container
  - Verifies client identity using certificates signed by a custom Certificate Authority (CA)
  - The backend is composed of two main services:

Backend - Node.js + Express (TypeScript):

  - Provides the REST API for device management and user authentication.
  - User credentials and authentication data are stored in **MongoDB**.
  - Device metadata is retrieved from **PostgreSQL**.
  - Uses JWT for secure session management.

Backend – Java Spring Boot

  - Subscribe to MQTT topics to receive base64-encoded image data.
  - Decodes and stores the image content along with metadata (e.g., resolution, timestamp, device ID) in **PostgreSQL**.

Image Processing and Storage

- A Python-based MQTT client receives image messages inside a container
- Decodes and stores the image along with metadata into the PostgreSQL database

Web Frontend (React)

- Displays connected devices and their latest captured image
- Retrieves data from the backend using REST API with JWT authentication
- Display a login interface with registration

Docker Deployment

- All services are containerized using Docker Compose, including:
  - Mosquitto broker
  - PostgreSQL database
  - Python MQTT-to-PostgreSQL client
  - MQTT

Default OSSF Criticality Score Result:

Our project currently holds a criticality score of approximately **0.20**, which is typical for a young and actively developing open-source initiative.

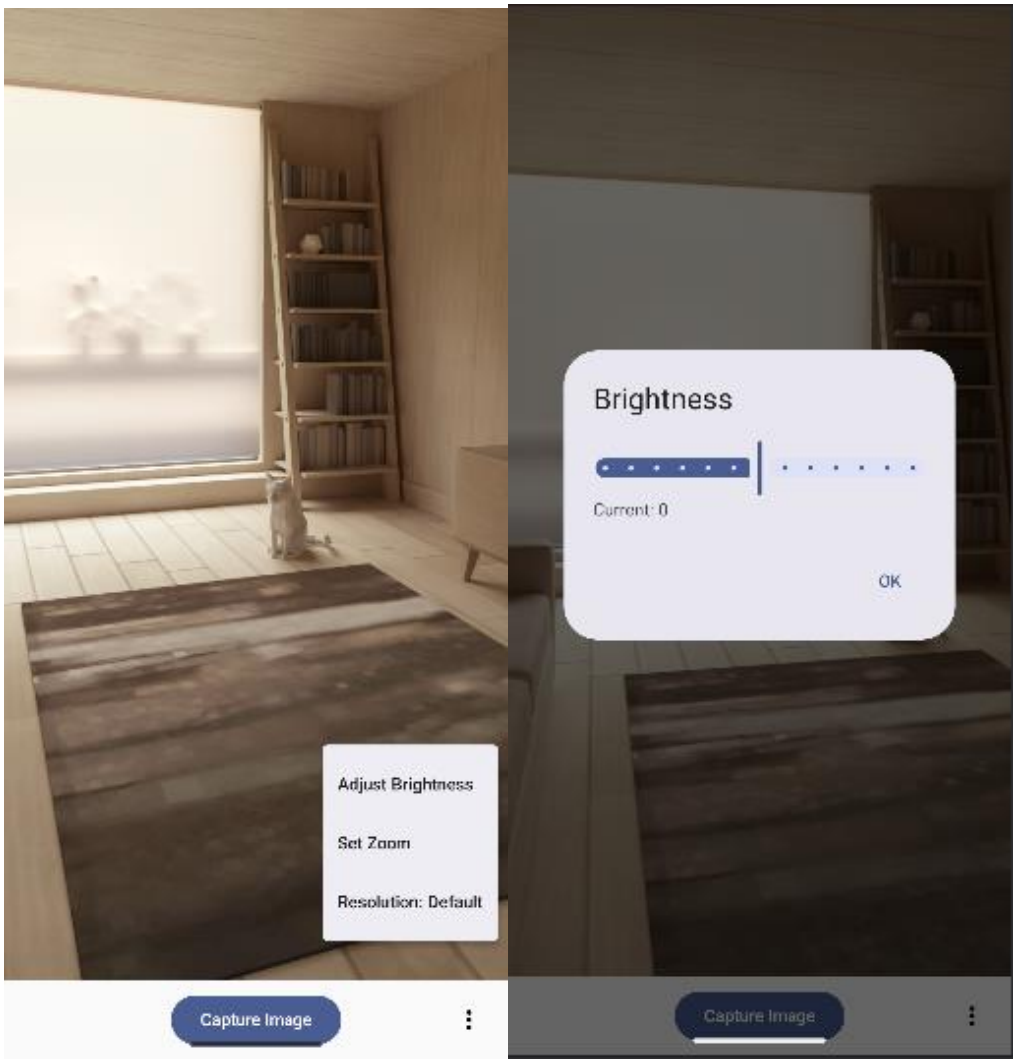
This score reflects our consistent development efforts and the valuable contributions made by our team. As we continue to expand the project, we remain committed to enhancing its visibility, stability, and adoption within the open-source community.

Metric	Value	Explanation
Star count	0	The project hasn't received any GitHub stars yet, suggesting limited visibility or discovery.
Created at	2025-03-03	The project is **very new**, about 2 months old.
Updated at	2025-05-26	Actively maintained, with a recent update.
Contributor count	5	A good number of contributors for an early-stage project.
Organization count	1	Indicates the project is associated with a GitHub organization or a team.
Commit frequency	0.54	Roughly one commit every two days – solid development pace.
Recent release count	0	No official releases yet – adding one can improve credibility.
Updated issues count	0	No updated issues – might indicate limited external usage.
Closed issues count	0	No issue resolutions – consider tracking tasks using GitHub Issues.
Issue comment frequency	0	No discussions around issues – expected for a young project.
GitHub mention count	2	Project has been mentioned twice on GitHub – promising sign of attention.

3. Functionality, Documentation, Execution

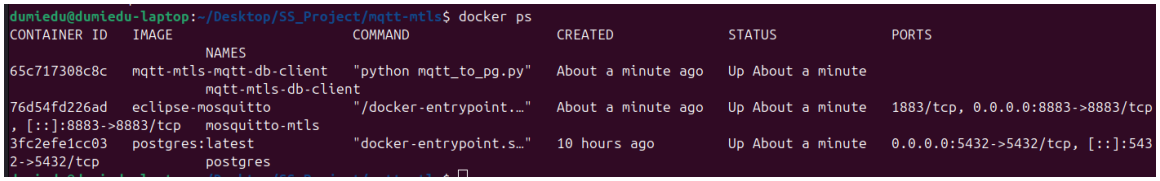
Android Client (Mobile App):

- The user captures an image using the device's camera.
- The image is encoded to a Base64 string.
- The app connects securely to the MQTT broker using mutual TLS (mTLS), presenting a client certificate (.p12).
- The encoded image is published to a predefined topic, such as camera/image.
- The app also has a menu from where different camera parameters can be adjusted: brightness, zoom and resolution



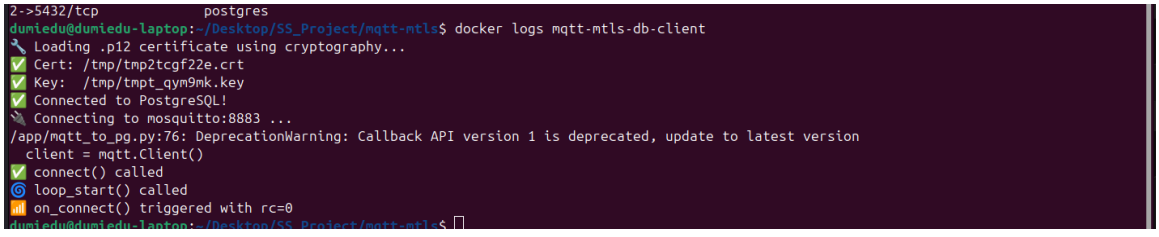
MQTT Broker (Mosquitto)

- The broker is configured to accept only authenticated clients with valid TLS certificates.
- Once the image is published to the topic, the broker routes the message to all subscribers of that topic.
- In this setup, the Python MQTT client is the main subscriber to camera/image.



Python MQTT Client

- Runs continuously and subscribes to the camera/image topic.
- Upon receiving a new message:
  - It decodes the Base64 string into a binary image (bytes).
  - Extracts any relevant metadata (e.g., timestamp, resolution, device ID if included).
  - Connects to the PostgreSQL database using credentials provided via environment variables or Docker secrets.
  - Inserts a new row into the image_messages table with:
    - the raw image (BYTEA)
    - the image dimensions
    - the topic name
    - the timestamp



PostgreSQL Database

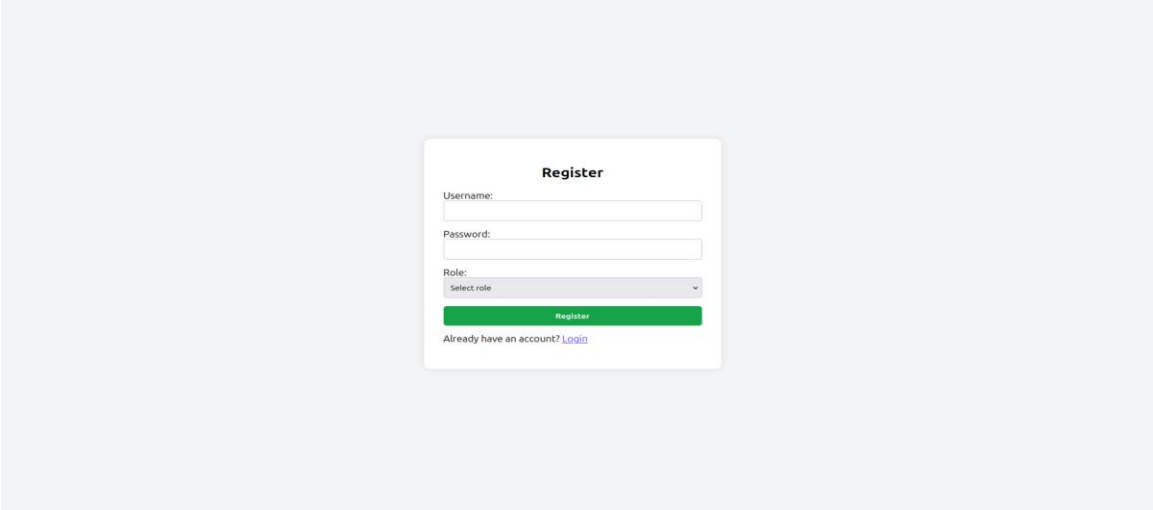
- Stores all incoming image data and associated metadata in the image_messages table.
- This data can later be queried by:
    - A REST API (Node.js backend)
    - A web frontend (React) or directly via tools like pgAdmin or psql.

```
mqttdb=# SELECT id, topic, width, height, timestamp FROM image_messages;
 id |      topic       | width | height |         timestamp
----+------------------+-------+--------+---------------------------
  1 | test/topic/image |  1280 |    960 | 2025-05-25 21:26:13.559044
  2 | test/topic/image |  1280 |    960 | 2025-05-25 21:26:37.407891
  3 | test/topic/image |  1280 |    960 | 2025-05-26 05:11:50.399316
(3 rows)

mqttdb=#
```

User Authentication – Login and Registration with MongoDB

- The authentication system is implemented using **Node.js with Express** and **MongoDB and** supports user roles for access control. Users can register, log in, and receive a JWT token that includes their role.
    1. Registration (POST /api/register)
        - The client sends a request with username, password, and optionally a role (defaulting to user).
        - The password is hashed using **bcrypt**.
        - The user is stored in the user's collection with fields:



2. Login (POST /api/login)
    - The client submits login credentials.
    - The server verifies the password using bcrypt.compare().
    - On success, a **JWT token** is generated, containing the user's ID and **role**.

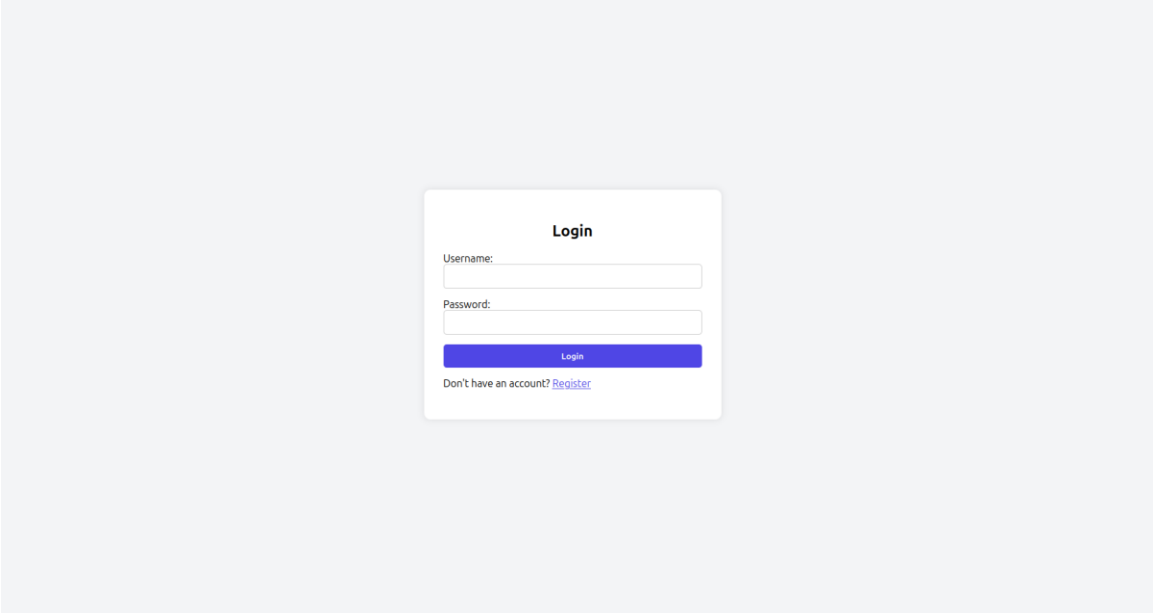
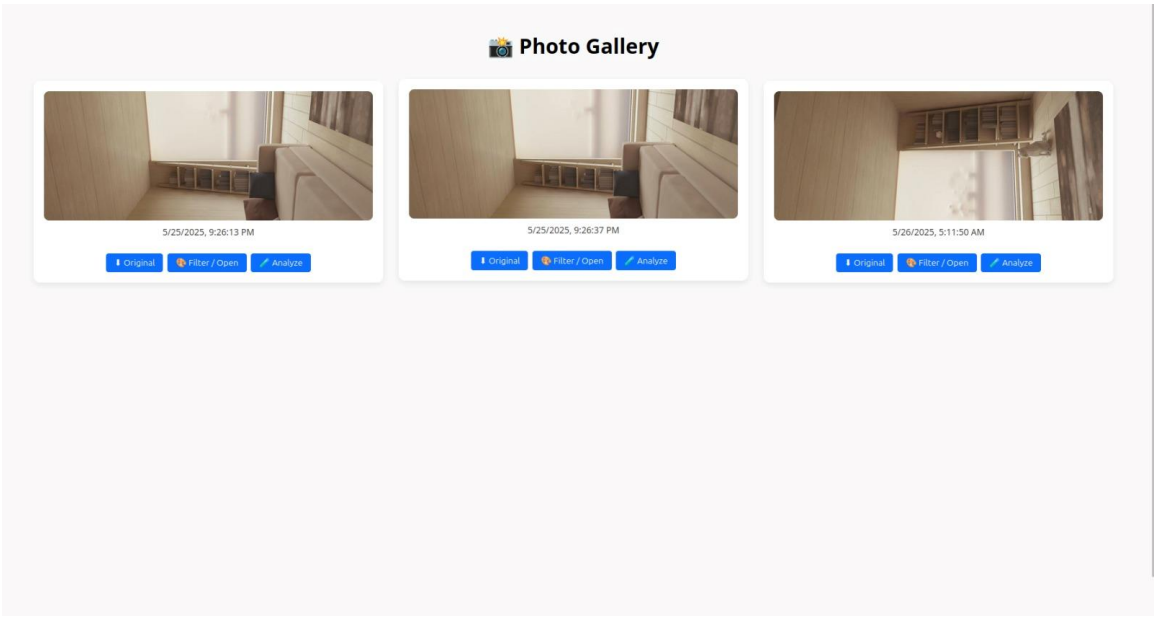
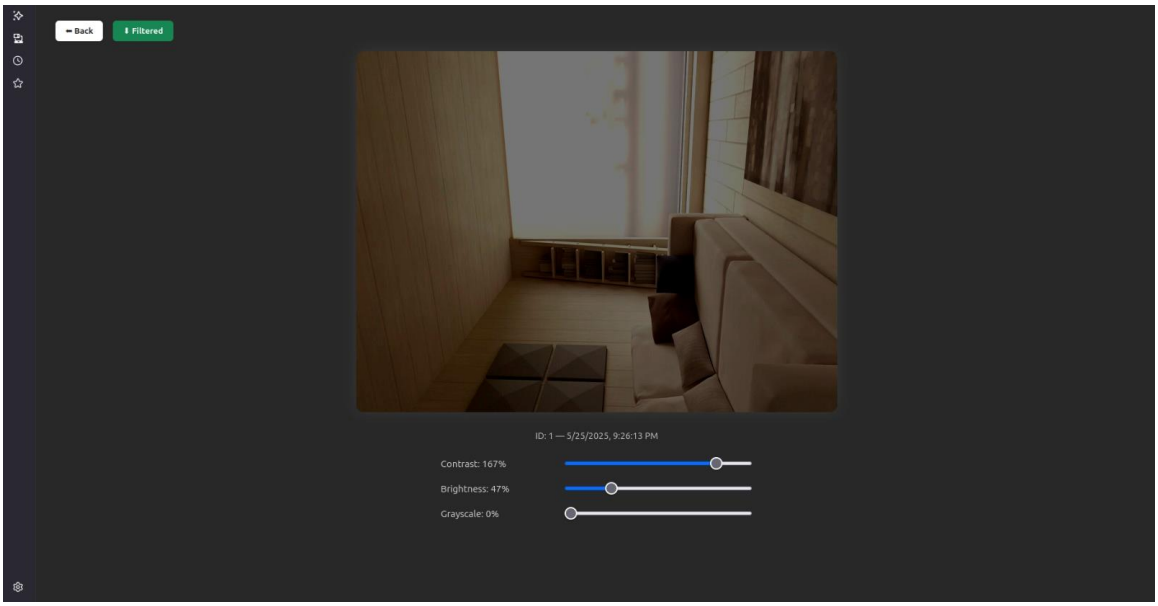


Image Gallery – Upload, View & Analyze Images

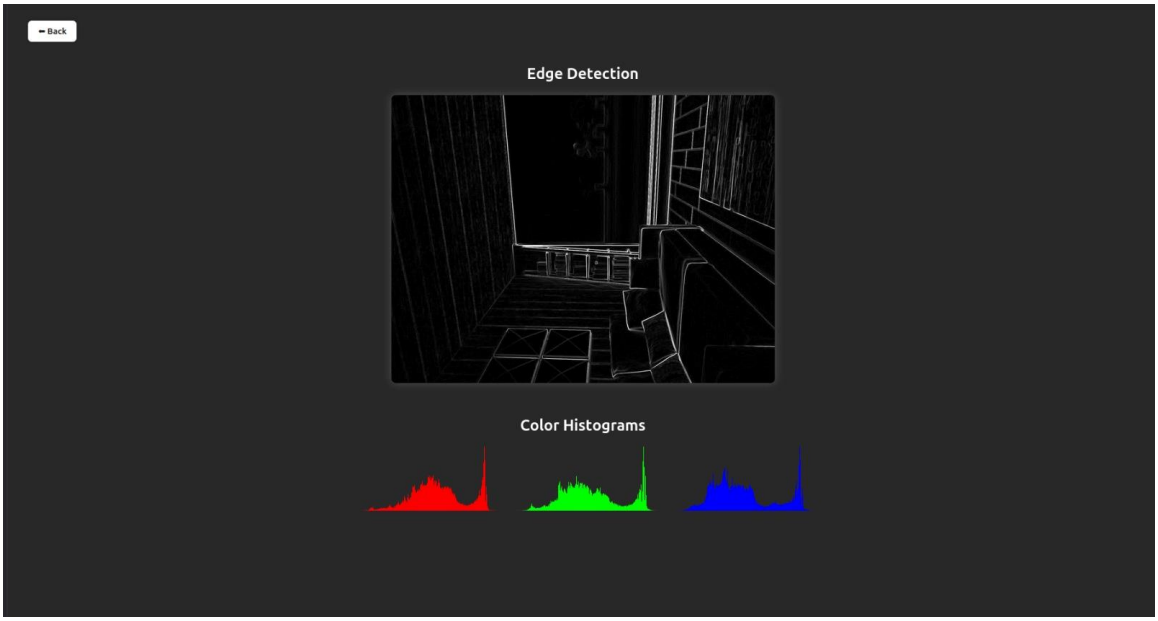
- The Image Gallery is a full-stack application built with Spring Boot (Java) on the backend and React on the frontend. It enables users to store, view, filter, and analyze images using a modern interface and PostgreSQL for persistence.



1. Upload & Storage
    - Images are uploaded via external ingestion or backend APIs and stored as binary (byte[]) in the database.
    - Each image is saved with metadata (auto-generated id, timestamp, width and height)
2. View Gallery (GET /api/images)
    - Returns a list of image metadata as JSON (ImageMetadataDTO).
    - Used by the frontend to render the gallery grid.
3. Fetch Image Data (GET /api/images/{id}/data)
    - Returns the actual image binary (byte[]) by ID.
    - Response is served as image/jpeg and rendered by the frontend <img>.
4. Filter & Open Features – users can click on "Filter / Open" to:
    - View the image in fullscreen.
    - Apply live filters using CSS: contrast, brightness, grayscale.
    - Download the filtered version as a new image.



5. Image Analysis – clicking "Analyze" runs client-side:
    - Edge Detection (Sobel filter) using HTML canvas.
    - Color Histogram displayed for red, green, and blue channels.



- Backend Technologies
  - Spring Boot for REST APIs.
  - Spring Data JPA with PostgreSQL.
  - DTO projection for clean metadata exposure.
  - Image bytes stored in a @Lob field.
- Frontend Features
  - Built with React + TypeScript.
  - Lazy-loaded gallery with styled image cards.
  - Filter controls and fullscreen preview using dynamic CSS.
  - Client-side canvas logic for analysis and download.

Device Dashboard – Lists all known devices and their statuses

4. Security & Compliance

- Threat Modeling & Mitigations

We identified potential risks related to unauthorized access to the MQTT broker, exposure of sensitive image data, and denial-of-service (DoS) attacks. To address these risks, we applied mutual TLS (mTLS) authentication for all MQTT

communications, ensuring encrypted data transmission and certificate-based client verification. A simplified threat diagram illustrates the secure communication flow between IoT clients, the broker, and the backend services.

- MISRA / CERT Compliance

SLint is a static code analysis tool that helps identify and report problems in JavaScript and TypeScript code. Its main purpose is to catch syntax errors, potential bugs, code quality issues, and security risks before the code is executed. By enforcing coding standards and best practices, ESLint helps developers maintain a clean and consistent codebase.

In our project, we implemented ESLint for both the frontend and backend. For the frontend, which uses React and TypeScript, we created a configuration file named .eslintrc.json inside the WebApp/frontend/ directory. For the backend, which is built with Node.js and TypeScript, we used an eslint.config.mjs file placed inside the WebApp/backend/ directory. These config files include recommended ESLint rules as well as specific plugins like @typescript-eslint and eslint-plugin-react, which are tailored for the technologies used.

To run the analysis manually, we used the command npx eslint "src//*.{js,jsx,ts,tsx}", which scans all relevant source files and reports issues such as undefined variables, improper typing, or bad JSX syntax. This process ensures that errors are caught early in development and that code follows the agreed standards.

Additionally, we integrated ESLint into the CI/CD pipeline using GitHub Actions. We created a workflow file called lint.yml in the .github/workflows/ directory. This workflow automatically runs ESLint every time new code is pushed or a pull request is opened. It installs dependencies and runs the linter separately for both frontend and backend, providing feedback directly in the GitHub interface.

Through this setup, ESLint helps us maintain code quality, prevent regressions, and enforce consistency across the project, both during development and in automated pipelines.

- Testing & Coverage

Unit Test Overview – ImageService & ImageController

This section documents the unit testing strategy for the core business logic and web layer of the Image Gallery application, focusing on the ImageService and ImageController components.

Purpose

The goal of these tests is to validate the correctness, stability, and independence of the service and controller layers, ensuring that:

- The service layer correctly handles image metadata and image data operations.
- The controller exposes accurate and reliable REST endpoints that behave as expected.

What's Being Tested:

1. **ImageService:**
    a. Retrieval of all image metadata.
    b. Retrieval of image data (as byte[]) by image ID.

```
33      @Test
34      void testGetAllImageMetadata() {
35          List<ImageMetadataDTO> mockList = Arrays.asList(
36                  new ImageMetadataDTO(1L, 100, 200, null),
37                  new ImageMetadataDTO(2L, 150, 250, null)
38          );
39
40          when(repo.findAllMetadata()).thenReturn(mockList);
41
42          List<ImageMetadataDTO> result = service.getAllImageMetadata();
43
44          assertEquals(2, result.size());
45          assertEquals(100, result.get(0).width());
46          verify(repo, times(1)).findAllMetadata();
47      }
```

```
49      @Test
50      void testGetImageById() {
51          byte[] mockData = new byte[]{1, 2, 3};
52
53          when(repo.findImageById(42L)).thenReturn(mockData);
54
55          byte[] result = service.getImageById(42L);
56
57          assertNotNull(result);
58          assertEquals(3, result.length);
59          verify(repo, times(1)).findImageById(42L);
60      }
```

2. **ImageController:**
    a. HTTP GET /api/images returns metadata list.
    b. HTTP GET /api/images/{id}/data returns image data or 404 if not found.

```
30      @Test
31      @DisplayName("GET /api/images should return image metadata list")
32      void testListImages() throws Exception {
33          List<ImageMetadataDTO> mockMetadata = Arrays.asList(
34                  new ImageMetadataDTO(1L, 800, 600, Timestamp.valueOf("2025-05-25 12:00:00")),
35                  new ImageMetadataDTO(2L, 1024, 768, Timestamp.valueOf("2025-05-26 15:00:00"))
36          );
37
38          when(service.getAllImageMetadata()).thenReturn(mockMetadata);
39
40          mockMvc.perform(get("/api/images"))
41                  .andExpect(status().isOk())
42                  .andExpect(jsonPath("$.length()").value(2))
43                  .andExpect(jsonPath("$[0].id").value(1))
44                  .andExpect(jsonPath("$[0].width").value(800))
45                  .andExpect(jsonPath("$[0].height").value(600));
46      }
47
48      @Test
49      @DisplayName("GET /api/images/{id}/data should return image data")
50      void testGetImageSuccess() throws Exception {
51          byte[] mockImage = new byte[]{1, 2, 3};
52          when(service.getImageById(42L)).thenReturn(mockImage);
53
54          mockMvc.perform(get("/api/images/42/data"))
55                  .andExpect(status().isOk())
56                  .andExpect(content().contentType(MediaType.IMAGE_JPEG))
57                  .andExpect(content().bytes(mockImage));
58      }
59
60      @Test
61      @DisplayName("GET /api/images/{id}/data should return 404 if not found")
62      void testGetImageNotFound() throws Exception {
63          when(service.getImageById(99L)).thenReturn(null);
64
65          mockMvc.perform(get("/api/images/99/data"))
66                  .andExpect(status().isNotFound());
67      }
```

3. Test Characteristics
    a. **Fast Execution:** Tests are isolated from the database and run quickly.
    b. **Behavior Verification:** Each test checks both the output and the interaction between components (e.g., repository or service methods being called).
    c. **Error Handling:** Includes validation for null cases, empty results, and HTTP status correctness (like 200 OK and 404 Not Found).
4. Benefits
    a. Confirms that the core layers of the application behave as expected under various conditions.
    b. Enables early detection of regressions or API contract violations.
    c. Simplifies development by allowing focused debugging of individual layers.

- Fixing Own Vulnerabilities

During the development of the SS_Project platform, we identified a security concern with the default settings of the Eclipse Mosquitto MQTT broker. By default, Mosquitto permits unauthenticated clients to connect, publish, and subscribe to topics, which is inappropriate for applications handling sensitive data such as base64-encoded images transmitted by Android devices.

To secure the system, we configured the broker to enforce **mutual TLS (mTLS)**, which requires both the broker and the client to present valid certificates during the TLS handshake. Here's how we mitigated the issue:

1. **Client authentication through certificates**: Only clients presenting a valid certificate, signed by our Custom Certificate Authority (CA), can connect to the MQTT broker. Unauthorized clients are rejected during the handshake phase.
2. **Disabled anonymous access**: We explicitly disabled allow_anonymous in the mosquitto.conf file, ensuring that all connections must be authenticated via certificate.
3. **Controlled access through certificate trust**: Since we do not use usernames, passwords, or ACLs, the system's trust model relies entirely on possession of a valid client certificate. Only devices we provision with a certificate (such as our Android app and backend processor) are able to connect and interact with topics.
4. **Topic access by convention**: Rather than enforcing topic-level permissions, we ensure that each trusted client follows a well-defined communication pattern. For example, the Android client is configured to only publish to a fixed topic (camera/image), and the backend is configured to only subscribe to that topic.

This setup ensures that the entire communication pipeline remains encrypted, authenticated, and isolated from unauthorized actors—without requiring additional layers such as ACL files or password authentication.

5. Team Contributions

Team Member	Lines Added	Lines Removed	Number of Commits
Dumistracel Eduard-Costin	1230	452	11
Cornea Alexandra	1280	0	1
Pascu Alexandru	25380	214	9
Puflene Alex-Costin	198	45	3
Maxim Alexandru	4546	1162	5