# Secure Collaborative Document Editor

Master's in Segurança Informática

June 3, 2025

Diogo Teixiera, up202408889

Guilherme Coutinho, up202108872

Leandro Costa, up202408816

Group **Salpicos De Aquarela**

# Contents

## 1 Project Overview

The developed application is a **secure, collaborative document editor** that enables multiple authenticated users to create, manage, and edit shared documents in real-time. The project prioritizes both functionality and security by combining real-time collaboration features with strict access control mechanisms.

The system adopts a modular architecture, built on a Node.js backend with the Express framework and SQLite for data persistence. The frontend leverages EJS templates to render dynamic views. Core functionalities include user registration and authentication, role-based document collaboration, invitation management, and real-time document editing using WebSockets. The application enforces distinct roles for document owners, editors, and read-only collaborators, ensuring secure and structured access to content.

Security is a cornerstone of the project. Several layers of protection have been implemented:

- **Input sanitization** is performed using **DOMPurify** on both the client and server sides to mitigate **XSS (Cross-Site Scripting** risks.
- **Passwords** are securely hashed using cryptographic algorithms before storage.
- **WebSocket communication** includes authentication and permission checks to prevent unauthorized access during live editing sessions.

To support secure development practices, the project integrates a CI/CD pipeline using GitHub Actions. This pipeline automates static code analysis, linting, and dependency auditing using tools such as ESLint, Horusec, Semgrep, and npm audit. Additionally, Docker hardening is enforced through Checkov to ensure secure container configurations.

Overall, this application demonstrates a comprehensive approach to building a secure, collaborative web platform, adhering to modern software engineering and security best practices.

# 2 Architecture Documentation

## 2.1 System Architecture

### 2.1.1 Project Structure (Site Map)

The project features a modular directory structure that separates server-side logic, client-side views, utilities, and configuration files. The src directory contains core backend logic, including route definitions, middleware, database operations, and utility modules. The views directory holds EJS templates for rendering dynamic HTML pages, while static assets like client-side JavaScript are in the static folder. The db.sqlite file serves as the embedded database, and package.json defines project dependencies and scripts for development and deployment.

```
project/
 src/
    index.js               # Main application entry point
    database.js            # Database configuration and queries
    middlewares.js         # Express middleware functions
    text_editor.js         # Text editor component
    routes/
       auth.js            # Authentication routes
       documents.js       # Document management routes
       invites.js         # Invitation management routes
       main.js            # Main application routes
    utils/
        auth.js            # Authentication utilities
        crypto.js          # Cryptographic functions
        sanitize.js        # Input sanitization
 views/
    layouts/               # Layout templates
    errors/                # Error pages
    document.ejs           # Document view
    documents.ejs          # Documents list
    document_options.ejs   # Document settings
    invites.ejs            # Invitations view
    login.ejs              # Login page
```

```
   register.ejs        # Registration page
   home.ejs            # Home page
static/                 # Static assets
   js/                 # Client-side JavaScript
db.sqlite               # SQLite database
package.json            # Project dependencies
```

#### 2.1.2 Component Relationships

**Server-Side Components**

- **Main Application (index.js)**: Initializes the Express application, configures middleware, sets up routes, and launches the WebSocket server.
- **Database Layer (database.js)**: Manages SQLite configuration and CRUD operations for users, documents, collaborations, and invitations.
- **Route Handlers**:
  - `auth.js`: Handles user authentication and session management.
  - `documents.js`: Manages documents and collaboration logic.
  - `invites.js`: Manages sending and receiving invitations.
  - `main.js`: Handles general routes (e.g., homepage, fallback).
- **Utilities**:
  - `auth.js`: Helper functions for authentication.
  - `crypto.js`: Password hashing and cookie generation.
  - `sanitize.js`: Input sanitization and XSS prevention.

**Client-Side Components**

- **EJS Views**:
  - `document.ejs`: Real-time editor interface.
  - `documents.ejs`: Lists owned and collaborated documents.
  - `document_options.ejs`: Document configuration page.
  - `invites.ejs`: Interface to manage invitations.
  - `login.ejs` / `register.ejs`: Authentication forms.

- **WebSocket Events**: Used for document synchronization, live updates, and collaborator presence tracking.

### 2.1.3 Security Layers

**Authentication Layer:** The application uses sessions and cookies to authenticate users, ensuring that only valid sessions can access protected features. Passwords are securely stored using cryptographic hashing functions, preventing direct retrieval even if the database is compromised.

**Authorization Layer:** Access to documents and features is enforced through middleware that checks user permissions. The system distinguishes between owners, editors, and read-only collaborators, maintaining a consistent and secure permission model.

**Input Validation Layer:** All client inputs are validated and sanitized. DOMPurify is used to prevent XSS attacks, and SQL queries are parameterized to mitigate injection risks. This protects both the integrity of the system and the users' data.

**WebSocket Security:** Connections require authentication during the initial handshake and are subject to ongoing permission checks during document editing. This prevents unauthorized users from accessing or modifying shared content in real-time.

### 2.1.4 Dependencies

**Server-Side:** The server relies on several core dependencies to provide its functionality. The `express` framework is used to handle routing and middleware in a modular and efficient way. For data persistence, the application uses `better-sqlite3`, an embedded database engine offering synchronous and performant SQLite access. Real-time communication is handled by `socket.io`, which enables bi-directional WebSocket support. The server-side rendering of views is done through the `ejs` template engine. Session management is facilitated by `cookie-parser`, while input sanitization and XSS prevention are ensured using `dompurify`.

**Client-Side:** On the client side, the application uses `socket.io-client` to establish and maintain WebSocket connections with the server, enabling real-time collaborative editing. Views are rendered using EJS templates, allowing dynamic data to be injected directly into HTML on the server before being sent to the client.

### 2.1.5 Configuration Files

The project includes several important configuration files. The `package.json` file outlines the project dependencies and defines the scripts for development and deployment. A `Dockerfile` is used to specify the environment and instructions for building the application into a containerized image. Finally, the `.gitignore` file excludes sensitive or unnecessary files, such as the database, local environment settings, and node modules, from version control to maintain repository cleanliness and security.

## 2.2 Database Design

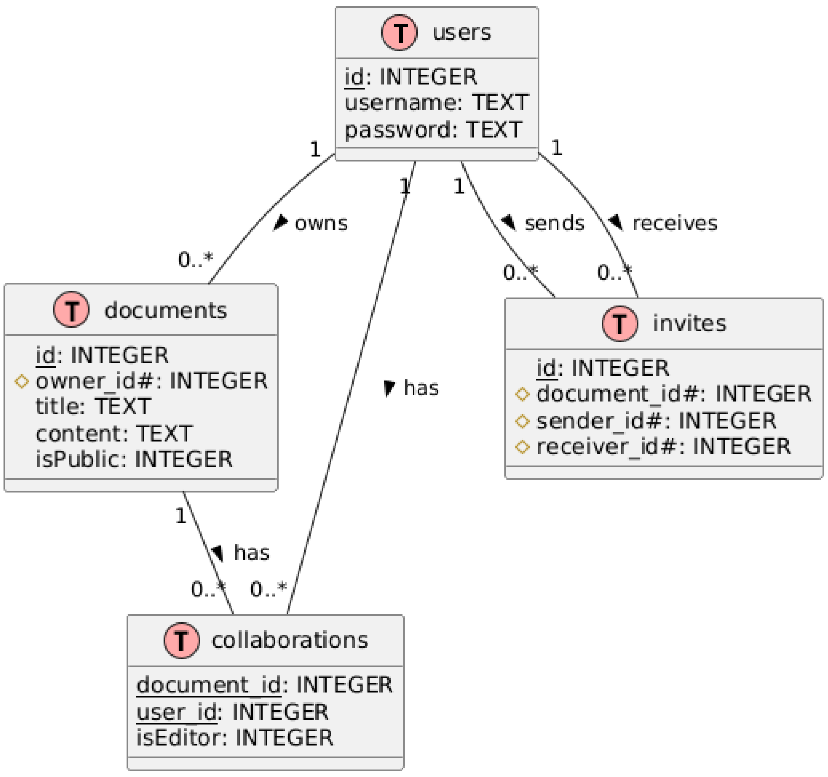


Figure 1: Database Design

The database schema supports secure collaboration on shared documents and consists of four main entities: `users`, `documents`, `invites`, and `collaborations`. Each user has a unique identifier, `username`, and a hashed `password`. The `documents` table includes an `owner_id`, `title`, `content`, and a flag `isPublic` for accessibility. There is a one-to-many relationship from users to their owned documents.

The `collaborations` table establishes a many-to-many relationship between users and documents, linking `user_id` to `document_id` and including an `isEditor` flag for permission levels.

The `invites` table manages collaboration requests, recording `document_id`, `sender_id`, and `receiver_id` for tracking invitations.

## 2.3 API Documentation

### 2.3.1 Authentication Endpoints

This section documents the RESTful API provided by the system, organized by functionality. All endpoints follow a clear authentication model and return appropriate HTTP status codes. Unless otherwise noted, endpoints require the user to be authenticated.

**GET /login** Renders the login page.

**POST /login** Authenticates a user.

- **Request Body:**
  - `username`: string
  - `password`: string
- **Responses:**
  - Success: Redirects to `/documents`
  - Error: Renders login page with error message
- **Status Codes:**
  - `401`: Invalid credentials

**GET /register** Renders the user registration page.

**POST /register** Registers a new user.

- **Request Body:**
  - `username`: string
  - `password`: string
- **Responses:**
  - Success: Redirects to `/login`
  - Error: Renders registration page with error message
- **Status Codes:**
  - `401`: Username already exists

**GET /logout** Logs out the current user and redirects to the login page.

### 2.3.2 Document Endpoints

**GET /documents** Lists all documents owned by or shared with the user.

- **Response:** Renders `documents.ejs` with:
  - `documents`: Array of owned documents
  - `collaboratedDocuments`: Array of shared documents

**POST /documents/create** Creates a new document.

- **Request Body:**
  - `title`: string
- **Response (JSON):**
  - Success: { `documentId: number, message: string` }
  - Error: { `error: string` }
- **Status Codes:**
  - `400`: Invalid title

**GET /documents/:id** Retrieves a specific document.

- **Response:** Renders `document.ejs`
- **Status Codes:**
  - `400`: Invalid document ID
  - `403`: Access denied
  - `404`: Document not found

**GET /documents/:id/options** Shows document settings and collaborators.

- **Response:** Renders `document_options.ejs`
- **Status Codes:**
  - `400`: Invalid document ID
  - `403`: Not document owner
  - `404`: Document not found

**POST /documents/:id/delete** Deletes a document.

- **Response (JSON):**
  - Success: { `message: string` }

  - Error: { `error: string` }

- **Status Codes:**
  - `400`: Invalid document ID
  - `403`: Not document owner
  - `404`: Document not found

**POST /documents/:id/invite** Sends a collaboration invite.

- **Request Body:**
  - `username`: string
- **Response (JSON):**
  - Success: { `message: string` }
  - Error: { `error: string` }
- **Status Codes:**
  - `400`: Invalid input
  - `403`: Not document owner
  - `404`: Document or user not found

**POST /documents/:id/visibility-change** Toggles public visibility of a document.

- **Response (JSON):**
  - Success: { `message: string, isPublic: boolean` }
  - Error: { `error: string` }
- **Status Codes:**
  - `400`: Invalid document ID
  - `403`: Not document owner
  - `404`: Document not found

**POST /documents/:id/collaborator-permission-change** Updates a collaborator's permissions.

- **Request Body:**
  - `username`: string
- **Response (JSON):**
  - Success: { `message: string` }

  - Error: { `error: string` }
- **Status Codes:**
  - `400`: Invalid input
  - `403`: Not document owner
  - `404`: Document not found

### 2.3.3 Invitation Endpoints

**GET /invites** Lists all pending collaboration invitations.

**POST /invites/accept** Accepts a pending invite.

- **Request Body:**
  - `invite_id`: number
- **Responses:**
  - Success: Redirects to the corresponding document
  - Error: Renders error page
- **Status Codes:**
  - `400`: Invalid invite ID
  - `403`: Not invite recipient
  - `404`: Invite not found

**POST /invites/decline** Declines a pending invite.

- **Request Body:**
  - `invite_id`: number
- **Response (JSON):**
  - Success: { `message: string` }
  - Error: { `error: string` }
- **Status Codes:**
  - `400`: Invalid invite ID
  - `403`: Not invite recipient
  - `404`: Invite not found

### 2.3.4 Security Considerations

The web application implements several essential security measures to ensure strong protection of user data and functionality.

First, access to the application is strictly controlled: only the login and registration endpoints are publicly available, while all other routes require proper authentication. To reduce the risk of injection attacks, all user input is carefully sanitized before processing. Furthermore, user passwords are never stored in plain text; instead, they are securely hashed using industry-standard cryptographic algorithms, which enhances protection against credential theft. The application also enforces strict access control policies, ensuring that users can only interact with documents they are authorized to access. Lastly, safeguard against Cross-Site Request Forgery (CSRF) attacks is provided through the use of session tokens, which validate the origin of requests and help maintain the integrity of user sessions.

Collectively, these measures contribute to a secure and resilient application architecture.

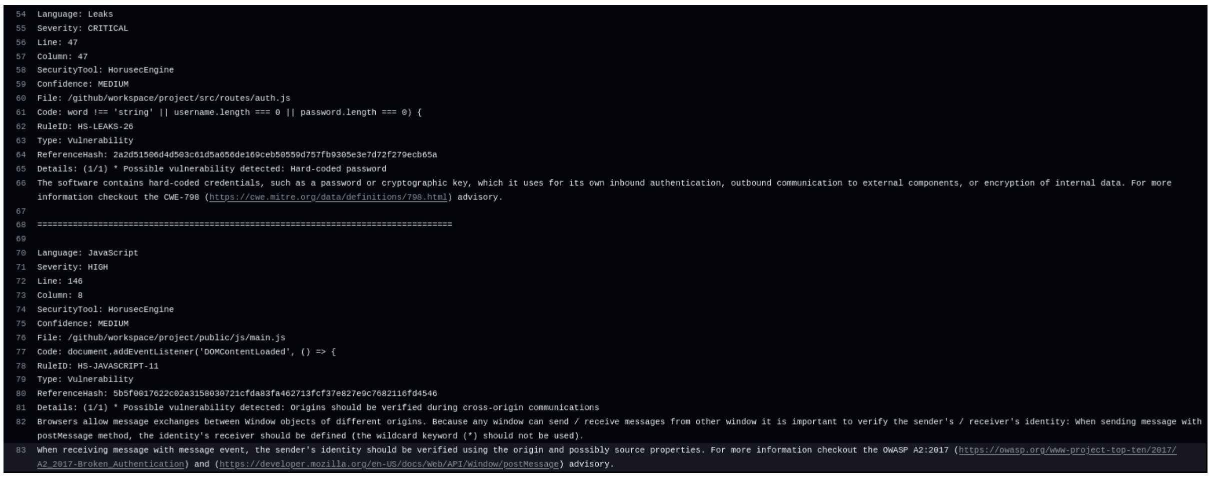
# 3 Security Analysis

## 3.1 GitHub Workflows

As part of a robust Secure Software Development Lifecycle (SSDLC), this project integrates GitHub Actions to enforce continuous security and code quality validation throughout the development process. The security.yml workflow is automatically triggered on every push or pull request to the main branch and orchestrates a set of independent jobs, each targeting a specific aspect of software assurance. Its core objective is proactively identifying and mitigating potential security and quality issues before code reaches production.

### 3.1.1 Jobs Breakdown

**Horusec** A static code analysis tool that helps identify security vulnerabilities in source code across multiple languages. In our workflow, we execute Horusec using the ‘fike/horusec-action‘ GitHub Action, which is configured via a dedicated ‘horusec-config.json‘ file and applied to the entire project directory. By identifying issues such as hardcoded secrets and insecure functions early in development, Horusec enabled us to refactor insecure logic, particularly regarding critical, high and medium vulnerabilities.

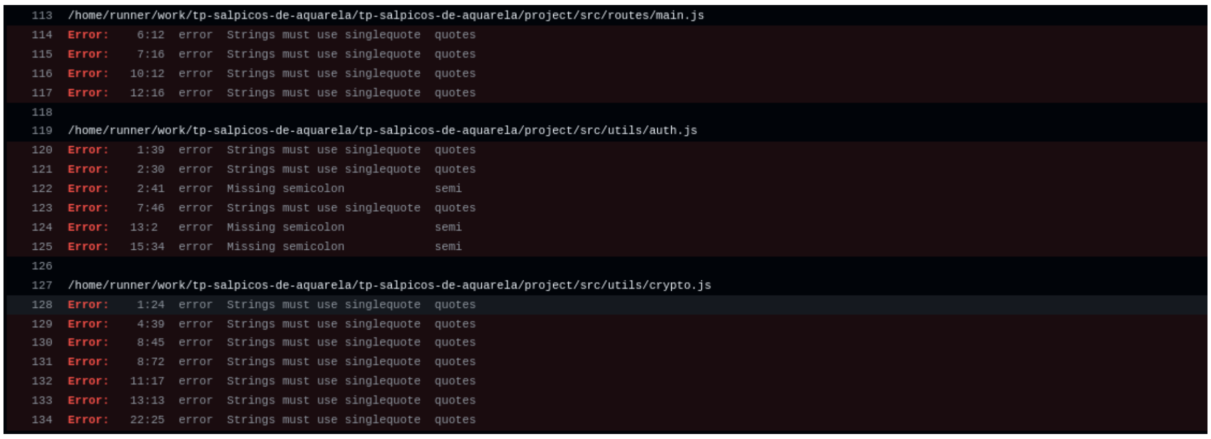


```
Language: Leaks
Severity: CRITICAL
Line: 47
Column: 47
SecurityTool: HorusecEngine
Confidence: MEDIUM
File: /github/workspace/project/src/routes/auth.js
Code: word !== 'string' || username.length === 0 || password.length === 0) {
RuleID: HS-LEAKS-26
Type: Vulnerability
ReferenceHash: 2a2d51506d4d503c61d5a656de169ceb50559d757fb9305e3e7d72f279ecb65a
Details: (1/1) * Possible vulnerability detected: Hard-coded password
The software contains hard-coded credentials, such as a password or cryptographic key, which it uses for its own inbound authentication, outbound communication to external components, or encryption of internal data. For more
information checkout the CWE-798 (https://cwe.mitre.org/data/definitions/798.html) advisory.

================================================================================

Language: JavaScript
Severity: HIGH
Line: 146
Column: 8
SecurityTool: HorusecEngine
Confidence: MEDIUM
File: /github/workspace/project/public/js/main.js
Code: document.addEventListener('DOMContentLoaded', () => {
RuleID: HS-JAVASCRIPT-11
Type: Vulnerability
ReferenceHash: 5b5f0017622c02a3158030721cfda83fa462713fcf37e827e9c7682116fd4546
Details: (1/1) * Possible vulnerability detected: Origins should be verified during cross-origin communications
Browsers allow message exchanges between Window objects of different origins. Because any window can send / receive messages from other window it is important to verify the sender's / receiver's identity: When sending message with
postMessage method, the identity's receiver should be defined (the wildcard keyword (*) should not be used).
When receiving message with message event, the sender's identity should be verified using the origin and possibly source properties. For more information checkout the OWASP A2:2017 (https://owasp.org/www-project-top-ten/2017/
A2_2017-Broken_Authentication) and (https://developer.mozilla.org/en-US/docs/Web/API/Window/postMessage) advisory.
```

Figure 2: Horusec Analysis - High and Critical vulnerabilities

**eslint** ESLint is a linter for JavaScript and TypeScript that enforces coding standards and helps detect common bugs. In our project, we run ESLint after installing dependencies to ensure consistent style rules across the codebase and to avoid unsafe coding patterns. ESLint has been instrumental in maintaining code quality, catching potential logic errors, and promoting best coding practices.

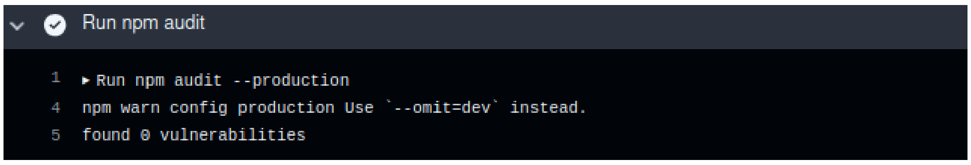
It identified a significant number of code style violations across several project files. The most common issues included the inconsistent use of double quotes (") instead of the configured standard of single quotes (') for string literals, as well as several missing semicolons at the end of statements. These violations were found in core files such as index.js, route handlers like auth.js and documents.js, and utility modules including auth.js and crypto.js.



```
/home/runner/work/tp-salpicos-de-aquarela/tp-salpicos-de-aquarela/project/src/routes/main.js
Error:    6:12  error  Strings must use singlequote  quotes
Error:    7:16  error  Strings must use singlequote  quotes
Error:   10:12  error  Strings must use singlequote  quotes
Error:   12:16  error  Strings must use singlequote  quotes

/home/runner/work/tp-salpicos-de-aquarela/tp-salpicos-de-aquarela/project/src/utils/auth.js
Error:    1:39  error  Strings must use singlequote  quotes
Error:    2:30  error  Strings must use singlequote  quotes
Error:    2:41  error  Missing semicolon             semi
Error:    7:46  error  Strings must use singlequote  quotes
Error:   13:2   error  Missing semicolon             semi
Error:   15:34  error  Missing semicolon             semi

/home/runner/work/tp-salpicos-de-aquarela/tp-salpicos-de-aquarela/project/src/utils/crypto.js
Error:    1:24  error  Strings must use singlequote  quotes
Error:    4:39  error  Strings must use singlequote  quotes
Error:    8:45  error  Strings must use singlequote  quotes
Error:    8:72  error  Strings must use singlequote  quotes
Error:   11:17  error  Strings must use singlequote  quotes
Error:   13:13  error  Strings must use singlequote  quotes
Error:   22:25  error  Strings must use singlequote  quotes
```

Figure 3: Some ESLint errors flagged

**code-quality** The ‘npm audit‘ command is utilized in this job to scan project dependencies for known vulnerabilities, comparing them against the official npm advisory database. This process provides actionable insights into security issues stemming from third-party packages. By swiftly identifying and patching vulnerable packages, some with high-severity issues, we effectively reduced risks associated with the software supply chain and enhanced the security of our application stack.

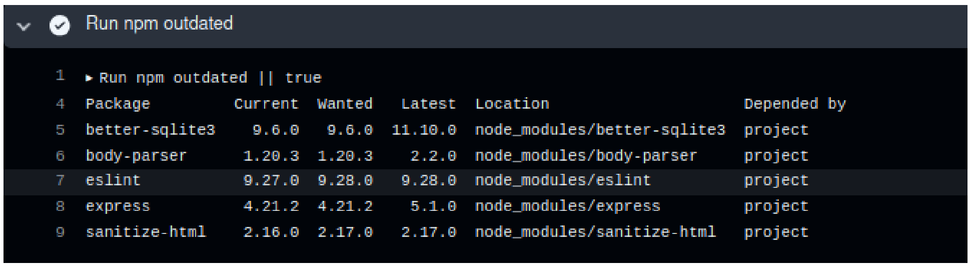


Run npm audit

```
▸ Run npm audit --production
npm warn config production Use `--omit=dev` instead.
found 0 vulnerabilities
```

Figure 4: nmp audit run

**dependency-check** We used ‘npm outdated‘ in the dependency-check job to identify lagging packages. This is crucial for both leveraging new features and ensuring critical security fixes are applied. The job helps us maintain long-term code health by alerting us to outdated dependencies, allowing for proactive planning of updates that prevent long-term technical debt or potential security liabilities.



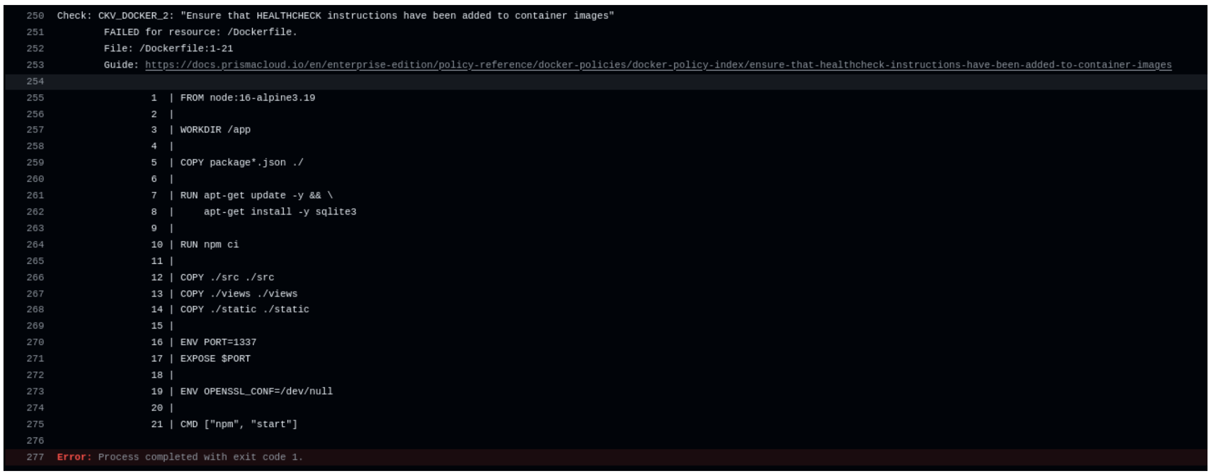
```
Run npm outdated
▸ Run npm outdated || true
Package          Current  Wanted   Latest  Location                     Depended by
better-sqlite3     9.6.0   9.6.0  11.10.0  node_modules/better-sqlite3  project
body-parser       1.20.3  1.20.3    2.2.0  node_modules/body-parser     project
eslint            9.27.0  9.28.0   9.28.0  node_modules/eslint          project
express           4.21.2  4.21.2    5.1.0  node_modules/express         project
sanitize-html     2.16.0  2.17.0   2.17.0  node_modules/sanitize-html   project
```

Figure 5: nmp outdated run

**Semgrep/ci** Employs semantic pattern matching to uncover security issues, anti-patterns, and logic bugs in our code. It provided insights beyond those of traditional linters, identifying complex security issues such as hardcoded credentials and unvalidated inputs. This enabled us to detect non-trivial bugs that may not have been caught otherwise, enhancing our confidence in the application's resilience.

**Checkov** A static analysis tool focused on infrastructure as code (IaC) that helps identify misconfigurations in Dockerfiles. By running Checkov against the 'project/' directory, we were able to detect insecure deployment settings early on.

It triggered the policy CKV_DOCKER_2, which ensures that every container image includes a HEALTHCHECK instruction. This is crucial for reducing downtime and improving observability, allowing us to detect failed containers early and helping orchestrators like Kubernetes respond to unhealthy services. Additionally, the application was running with the default root user, which needed to be fixed.



```
Check: CKV_DOCKER_2: "Ensure that HEALTHCHECK instructions have been added to container images"
        FAILED for resource: /Dockerfile.
        File: /Dockerfile:1-21
        Guide: https://docs.prismacloud.io/en/enterprise-edition/policy-reference/docker-policies/docker-policy-index/ensure-that-healthcheck-instructions-have-been-added-to-container-images

                1  | FROM node:16-alpine3.19
                2  |
                3  | WORKDIR /app
                4  |
                5  | COPY package*.json ./
                6  |
                7  | RUN apt-get update -y && \
                8  |     apt-get install -y sqlite3
                9  |
                10 | RUN npm ci
                11 |
                12 | COPY ./src ./src
                13 | COPY ./views ./views
                14 | COPY ./static ./static
                15 |
                16 | ENV PORT=1337
                17 | EXPOSE $PORT
                18 |
                19 | ENV OPENSSL_CONF=/dev/null
                20 |
                21 | CMD ["npm", "start"]

Error: Process completed with exit code 1.
```

Figure 6: Checkov Analysis on HEALTHCHECK

```
RUN addgroup --system appgroup && adduser --system --ingroup appgroup appuser
...
HEALTHCHECK --interval=30s --timeout=10s --start-period=5s --retries=3 \
CMD wget --quiet --spider http://localhost:1337/ || exit 1

RUN chown -R appuser:appgroup /app

USER appuser
```

Listing 1: Proceeded fixes onto the Dockerfile

## 3.2 Security Development Tools

### 3.2.1 DOMPurify

**DOMPurify** is a widely-used JavaScript library that sanitizes HTML to prevent XSS (Cross-Site Scripting) attacks. It accomplishes this by parsing and cleaning HTML input to eliminate any potentially harmful code, such as script tags or malicious event handlers.

We used the tool in the code like this:

```
DOMPurify.sanitize(input,
    {
        ALLOWED_TAGS: [],
        ALLOWED_ATTR: []
    }
);
```

We set **ALLOWED_TAGS** and **ALLOWED_ATTR** as an empty array, saying that we want **DOMPurify** to remove every HTML tag and attribute that it encounters. This way, no kind of XSS is going to be possible in our application.

### 3.2.2 The EJS template framework

DOMPurify only removes HTML tags, so it does not prevent **SSTI (Server Side Template Injection)** attacks. By default, **EJS** escapes user input by default using

$$< \% = \% >$$

So, we made sure that every template generated with user input used this tag.

### 3.2.3 better-sqlite3

We used the **Node.js** package **better-sqlite3** as our database manager. This package helps us perform safe, synchronous SQL parameterized queries (also called prepared statements).

## 3.3 Pentest Excecuted

As the developers of this project, we have full access to the entire codebase and a deep understanding of its logic and data flow. In this section, we will describe how we conducted a **White Box Penetration Test** on our application.

### 3.3.1 Client Side

#### XSS - Cross-Site Scripting

To test for XSS, we used a simple payload like in Figure 7 in the create document form as the document's title.

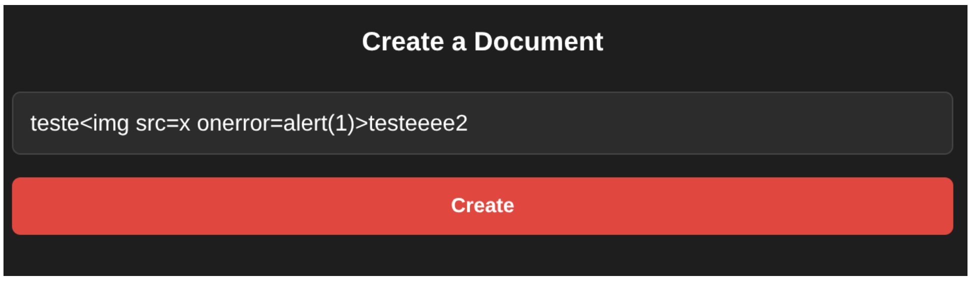
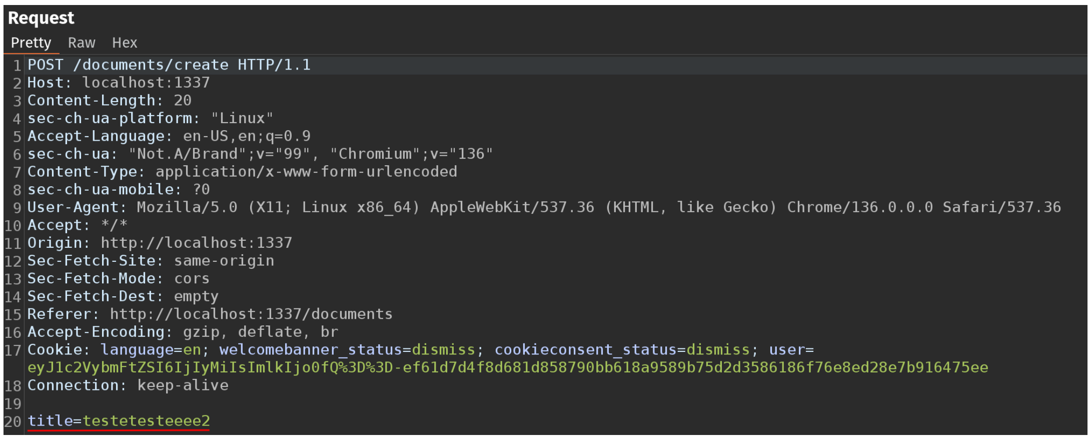


Figure 7: XSS attempt on Create Document

To analyze better the request being sent, we used **Burp Suite's Interceptor** to capture it and analyze it. As we can observe in Figure 8, the title field does not have the HTML tag with the malicious event handler, because **DOMPurify** was able to clean it.

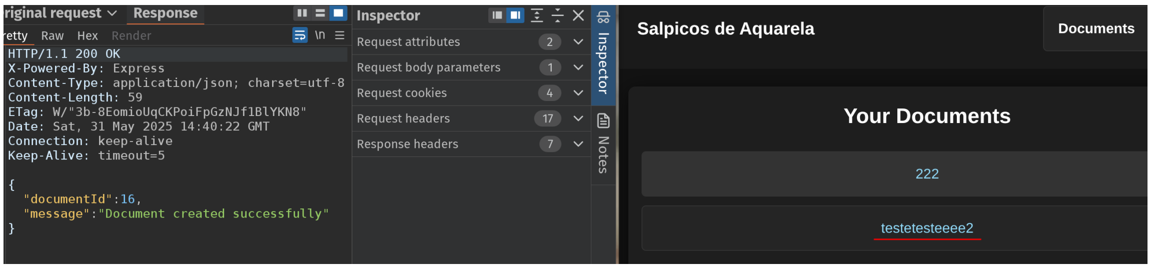


```
POST /documents/create HTTP/1.1
Host: localhost:1337
Content-Length: 20
sec-ch-ua-platform: "Linux"
Accept-Language: en-US,en;q=0.9
sec-ch-ua: "Not.A/Brand";v="99", "Chromium";v="136"
Content-Type: application/x-www-form-urlencoded
sec-ch-ua-mobile: ?0
User-Agent: Mozilla/5.0 (X11; Linux x86_64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/136.0.0.0 Safari/537.36
Accept: */*
Origin: http://localhost:1337
Sec-Fetch-Site: same-origin
Sec-Fetch-Mode: cors
Sec-Fetch-Dest: empty
Referer: http://localhost:1337/documents
Accept-Encoding: gzip, deflate, br
Cookie: language=en; welcomebanner_status=dismiss; cookieconsent_status=dismiss; user=
eyJ1c2VybmFtZSI6IjIyMiIsImlkIjo0fQ%3D%3D-ef61d7d4f8d681d858790bb618a9589b75d2d3586186f76e8ed28e7b916475ee
Connection: keep-alive

title=testetesteeee2
```

Figure 8: Intercepted Request of XSS attempt on Create Document

Even though, the payload was already wiped before the request was sent, we can change the data being sent to the server again and put the same payload on the title field and forward the intercepted request.

Good thing that we also use **DOMPurify** on the backend and the HTML tag is also going to cleaned up again as we can see by the server's response and the title of the created document on Figure 9:



```
HTTP/1.1 200 OK
X-Powered-By: Express
Content-Type: application/json; charset=utf-8
Content-Length: 59
ETag: W/"3b-8EomioUqCKPoiFpGzNJf1BlYKN8"
Date: Sat, 31 May 2025 14:40:22 GMT
Connection: keep-alive
Keep-Alive: timeout=5

{
  "documentId":16,
  "message":"Document created successfully"
}
```

Figure 9: Response of XSS attempt on Create Document

### 3.3.2 Server Side

#### SSTI

Even though we previously mentioned that we are using EJS's special tag that escapes tags, we obviously still needed to test if we missed something and if it was indeed vulnerable to SSTI attacks. These attacks are on the server side because the template is going to be generated on the server side and then sent in the response.

We followed the same behavior as in the XSS attacks and sent payloads like the one in Figure 10 in every field that was going to be reflected on the page.

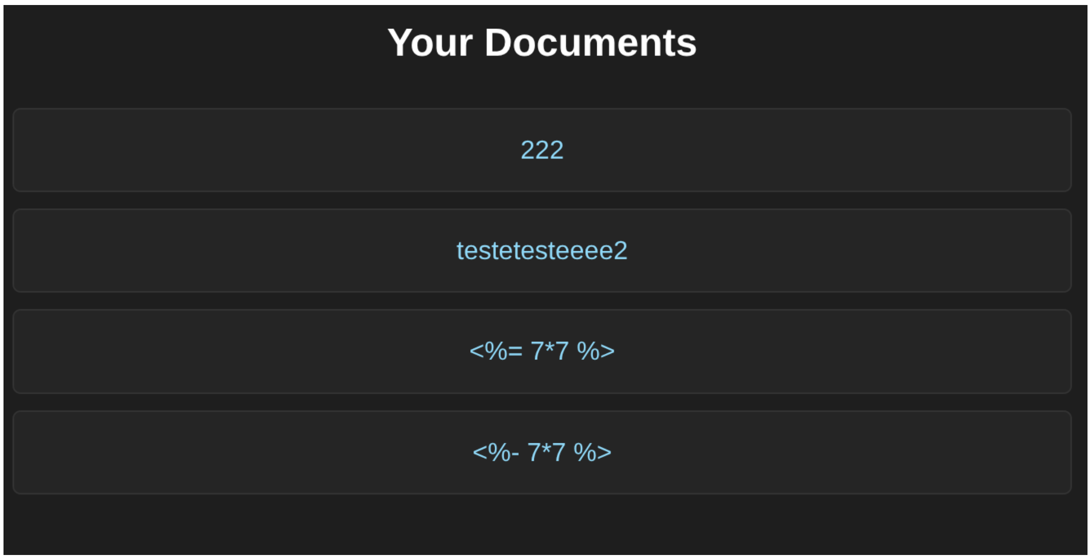
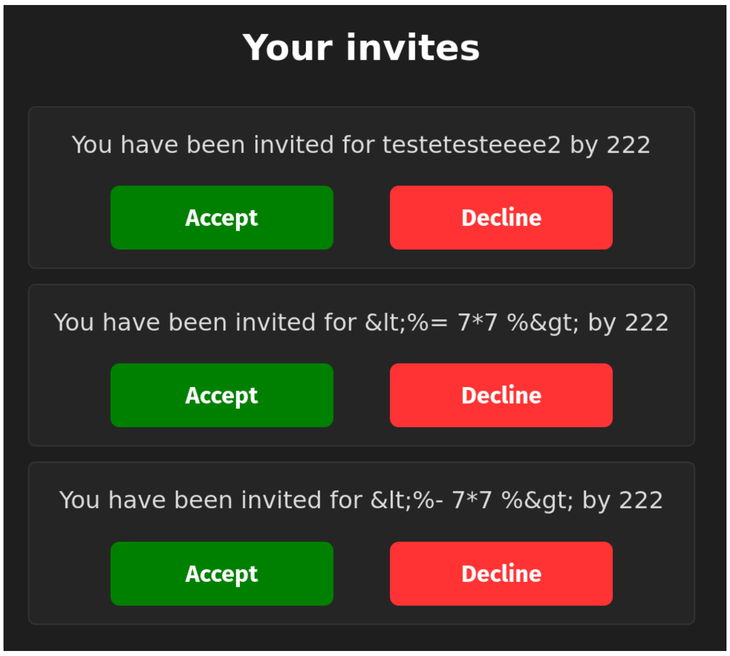


Figure 10: SSTI attempt on Create Document

Another place where this input is reflected is in the user's invites page, where we tell the user the title of the document he has been invited to. As we can see by looking at Figure 11, the payload was also sanitized.



Your invites

You have been invited for testetesteeee2 by 222

You have been invited for &lt;%= 7*7 %&gt; by 222

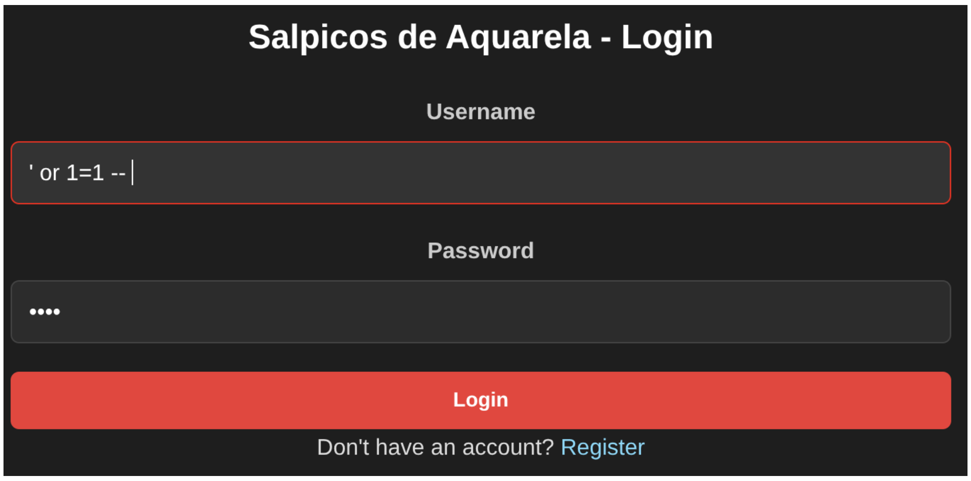
You have been invited for &lt;%- 7*7 %&gt; by 222

Figure 11: SSTI attempt reflected on the Invites page

**SQL Injection**

An SQL Injection occurs when user input is unsafely included in a SQL query, meaning that the input is interpreted as part of the SQL command, not just as data.

We tried to attempt this on our Login page sending the payload seen in Figure 12.



Salpicos de Aquarela - Login

Username: ' or 1=1 --

Password: ••••

Figure 12: SQLi attempt on Login page

As expected, we received the response "Invalid username".

We already mentioned the usage of **better-sqlite3**, that allows us to perform safe parametrized queries like this one:

```
db.prepare('SELECT * FROM users WHERE username = ?').get(username);
```

This way, SQLite treats the provided input by the user as a literal string, not as part of the SQL logic.

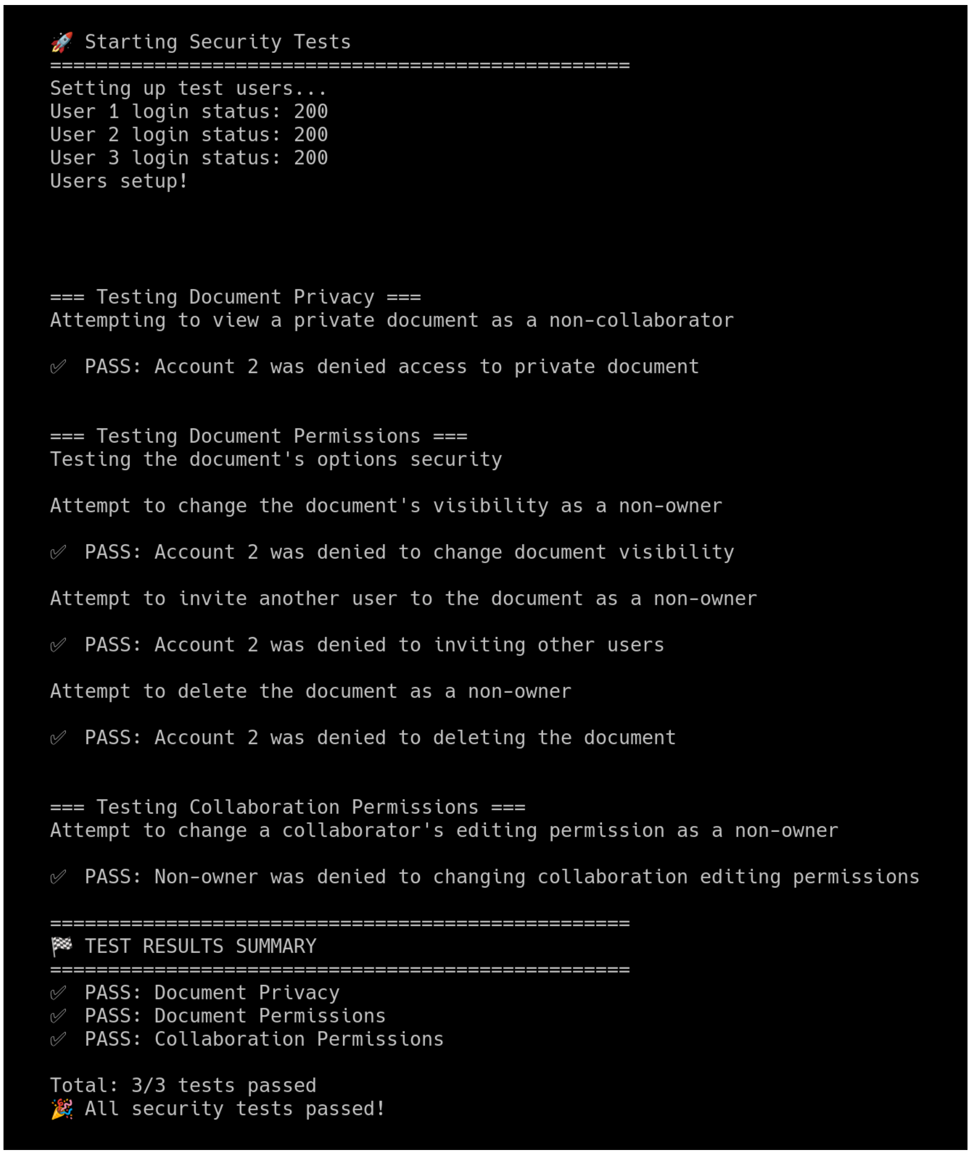
Every developer should implement their code like this, and never interpolate user input directly into SQL strings.

### 3.3.3 Business/Code Logic Flaws

One of the main causes for vulnerabilities in the web and reports on Bug Bounty programs are **Business/Code Logic flaws**. An application must have a strong data flow logic to prevent attacks, a simple attack like sending 2 requests in opposite order might be enough to break the system's logic.

On our github repository is a Python script on the **tests** folder that tests for IDORs (Insecure Direct Object Reference) and Broken Access Controls. The tests consist of creating three accounts on the application and performing multiple actions that should not be done, like trying to change a document's visibility or deleting the document as a user that is not the owner, attemping to access private documents as a user that is not a collaborator, etc.

In Figure 13 is a print of the output of the performed security tests showing that all tests passed:



```
🚀 Starting Security Tests
==================================================
Setting up test users...
User 1 login status: 200
User 2 login status: 200
User 3 login status: 200
Users setup!




=== Testing Document Privacy ===
Attempting to view a private document as a non-collaborator

✅ PASS: Account 2 was denied access to private document


=== Testing Document Permissions ===
Testing the document's options security

Attempt to change the document's visibility as a non-owner

✅ PASS: Account 2 was denied to change document visibility

Attempt to invite another user to the document as a non-owner

✅ PASS: Account 2 was denied to inviting other users

Attempt to delete the document as a non-owner

✅ PASS: Account 2 was denied to deleting the document


=== Testing Collaboration Permissions ===
Attempt to change a collaborator's editing permission as a non-owner

✅ PASS: Non-owner was denied to changing collaboration editing permissions

==================================================
🏁 TEST RESULTS SUMMARY
==================================================
✅ PASS: Document Privacy
✅ PASS: Document Permissions
✅ PASS: Collaboration Permissions

Total: 3/3 tests passed
🎉 All security tests passed!
```

Figure 13: Security tests output

# 4 Conclusion and Future Work

The development of this secure collaborative document editor showcased the combination of real-time functionality with robust security practices. We focused on modularity, maintainability, and secure software engineering. The application features role-based access, real-time collaboration via WebSockets, and strong input validation. We successfully implemented a CI/CD pipeline with automated security tools to uphold best practices throughout the development lifecycle.

Built with Node.js, Express, SQLite, and EJS templates, the architecture is efficient and easy to deploy. Key security elements, including XSS protection with DOMPurify, secure password hashing, CSRF protection, and WebSocket permission enforcement were validated through internal analysis and penetration testing. GitHub Actions, using tools like ESLint, Horusec, and Semgrep, helped maintain code quality.

## Future Work

Although the project achieved its main goals, there are opportunities for improvement and the addition of new features:

- **Scalability**: Transitioning to PostgreSQL for better performance with more users.
- **Authorization Granularity**: Introducing finer access control options.
- **Version Control**: Adding document versioning and edit history for rollback capabilities.

In summary, this project provides a solid foundation for secure collaborative platforms, with potential to evolve into a production-ready solution for various environments.