Security Analysis

Security is crucial for any website, this section will outline the measures our platform takes to ensure the protection of user information and prevent unauthorized access.

When a user signs into the platform, a JSON Web Token (JWT) is generated specifically for them. This token acts as a digital credential and contains essential user details, including their unique ID and the time of token creation. To establish the authenticity and integrity of the JWT, the server signs it using a private key. This cryptographic signature enables the server to validate the JWT and ascertain that the communication is with an authenticated user. Once generated, the token is sent back to the user, who must present it for subsequent requests. Without a valid token, attempting to access any page or API endpoint will result in a 401 unauthorized error, preventing unauthorized access.
To further enhance security, our platform employs a robust authorization mechanism. All pages within the platform are categorized into three distinct categories: staff, student, and company. When a user requests access to a particular page, the server conducts two crucial checks. First, it verifies the validity of the JWT token, ensuring that it has not been tampered with or expired. Once the token is deemed valid, the server proceeds to verify whether the user is authorized to access the category to which the page belongs. If access is denied, a 403 forbidden error is returned, preventing unauthorized entry and maintaining strict access control.

In addition to authentication and authorization practices, our platform addresses several security vulnerabilities to protect against potential threats. We have identified three primary concerns and implemented effective solutions.

The first concern centers around the possibility of a database leak. To mitigate risks, our platform employs a password hashing mechanism. Every password undergoes a one-way hashing process using bcrypt with 12 rounds. Bcrypt is a proven and secure hashing algorithm based on the blowfish cipher, which incorporates variable keys to slow down brute force attacks. By hashing passwords, we ensure that even if the database is compromised, user passwords remain protected and resistant to decryption.

The second concern we address is the risk of SQL injection attacks. To prevent such attacks, our platform utilizes prepared statements. Prepared statements offer a secure approach to interact with the database by separating code from data. By keeping code and data separate, we eliminate the potential for unintended execution of SQL code and effectively mitigate the risk of SQL injection vulnerabilities.

The third concern involves protecting against cross-site scripting (XSS) attacks. Our platform adopts a two-fold approach to tackle XSS vulnerabilities. On the server side, we implement thorough input validation and employ escaping techniques. This ensures that any user-supplied data is properly sanitized and rendered harmless before being processed and displayed. On the client side, our application utilizes the **innerText** property instead of

innerHTML when rendering dynamic content. This prevents the execution of any HTML code that might have slipped through the server-side validation, further strengthening our defense against XSS attacks.

Through the implementation of these comprehensive security measures, including JWT authentication, stringent authorization checks, password hashing, prepared statements, and XSS prevention techniques, our platform maintains a robust security mechanism. These measures work cohesively to safeguard user data, protect against unauthorized access, and fortify our platform against potential security threats.

Software Testing

To properly test the software developed throughout this project, we employed both unit and user-story testing. The following will shortly discuss the purpose of the testing method, how it was realized and how it helped identifying problems in the system.

Unit Testing Approach

The unit testing focused on verifying the functionality and behavior of the DAO classes. The implications of unit testing helped identify and fix bugs early in the development cycle and improve overall code quality. To approach unit testing, the JUnit testing framework was used. The unit tests were designed to cover various scenarios and edge cases to ensure comprehensive test coverage.

The unit tests for the DAO classes aimed to verify the following aspects:

1. Correct execution and results of database queries: The tests validated that the SQL queries constructed by the DAO classes were correct and returned the expected results when interacting with the database.
2. Proper handling of exceptions: The tests verified that exceptions, such as SQLExceptions, were appropriately handled by the DAO classes, ensuring graceful error handling and preventing any unexpected behavior or application crashes.
3. Integration with other components: The tests ensured that the DAO classes interacted correctly with other components, such as the **GenericDAO** class, by setting up necessary dependencies and validating the interactions.
4. Business logic and data manipulation: The tests covered the various methods and functionalities of the DAO classes, verifying that the business logic and data manipulation operations were executed correctly, without any inconsistencies or unexpected behavior.

Throughout the unit testing process, test cases were carefully designed to cover both positive and negative scenarios, including boundary cases, error conditions, and exceptional situations. This approach helped uncover hidden bugs, validate the correctness of the code, and ensure robustness.

User Story Testing Approach

Mainly, user story testing was performed to validate the system behavior from an end-user perspective. This testing approach focused on verifying that the application fulfilled the intended functionalities and requirements, as defined by user stories.
The user story testing process involved the following steps:

1. Interaction with the UI: Testers interacted with the user interface of the web application, performing actions and operations as a typical user would. This included navigating through different screens, entering input data, and triggering various functionalities.
2. Validating expected outputs: After performing actions, the testers compared the outputs and results against the expected behavior defined in the user stories. This verification ensured that the application responded correctly and produced the intended outcomes.
3. Error code printing: The application was configured to generate error codes or messages when encountering errors or exceptions. Testers monitored these error codes during user story testing to identify any issues or unexpected behaviors. This approach helped identify potential issues, allowing for debugging and resolution.
4. Monitoring the database: Testers closely monitored the database to ensure that the application correctly stored and retrieved data. They verified that data modifications, such as adding, updating, or deleting records, were reflected accurately in the database.

By combining UI interaction, output validation, error code monitoring, and database monitoring, the user story testing approach provided a comprehensive assessment of the application's functionality, usability, and data integrity.

Overall, the combination of unit testing for code-level verification and user story testing for end-user validation ensures a robust and reliable application. It enabled early bug detection, and improved the overall quality of the software.