1. Testing Functional Requirements

Use Case: Apply for Scholarship

Functional Requirements:

1. **User Login**
    - **Test Case 1:** Verify student can log in using BGU credentials.
        - **Steps:**
            1. Navigate to the login page.
            2. Enter valid BGU credentials.
            3. Click the login button.
        - **Expected Result:** Student dashboard is displayed.
2. **View Scholarships**
    - **Test Case 2:** Verify student can view the list of available scholarships.
        - **Steps:**
            1. Log in as a student.
            2. Navigate to the scholarships page.
        - **Expected Result:** List of available scholarships is displayed.

Use Case: Create & Delete Scholarships

Functional Requirements:

3. **Create Scholarships**
    - **Test Case 3:** Verify admin can create a new scholarship.
        - **Steps:**
            1. Log in as an admin.
            2. Navigate to the scholarship management section.
            3. Click "Create New Scholarship."
            4. Fill in the required scholarship details.
            5. Click "Submit."
        - **Expected Result:** Scholarship is created and visible in the list of available scholarships.
4. **Delete Scholarships**

- **Test Case 4:** Verify admin can delete an existing scholarship.
    - **Steps:**
        1. Log in as an admin.
        2. Navigate to the scholarship management section.
        3. Select a scholarship to delete.
        4. Click "Delete."
        5. Confirm the deletion.
    - **Expected Result:** Scholarship is deleted and no longer visible in the list of available scholarships.

Use Case: Automated Notifications

Functional Requirements:

5. **Send Automatic Updates**
    - **Test Case 5:** Verify system sends automatic updates to students about their application status via email.
        - **Steps:**
            1. Submit a scholarship application.
            2. Change the application status (e.g., from "Pending Review" to "Reviewed").
        - **Expected Result:** Student receives an email notification about the status change.

Use Case: Store All User Information

Functional Requirements:

6. **Store User Information**
    - **Test Case 6:** Verify system stores all user information correctly.
        - **Steps:**
            1. Create a new user account.
            2. Log in as the new user.
            3. Navigate to the user profile section.
            4. Verify the displayed information matches the entered data.
        - **Expected Result:** User information is stored and displayed correctly.

Use Case: Data Security

Functional Requirements:

7. **Ensure Data Security**
    - **Test Case 7:** Verify system ensures that all sensitive student data is securely stored.
        - **Steps:**
            1. Register a new user with sensitive information.
            2. Access the database directly to verify that sensitive data is encrypted.
        - **Expected Result:** Sensitive data is encrypted in the database and complies with security standards.

2. Testing Non-Functional Requirements

Non-Functional Requirements: These include aspects such as speed, safety, security, and data integrity.

Speed and Performance

- **Requirement:** System should respond within 2 seconds under normal load conditions.
    - **Test Case:** Verify system response time under different user loads.
        - **Steps:**
            1. Simulate various user loads using appropriate testing tools.
            2. Measure response times for key operations.
        - **Expected Result:** Response times should not exceed 2 seconds under normal load conditions.

Safety

- **Requirement:** Ensure the system prevents unauthorized access and misuse.
    - **Test Case:** Verify system safety features.
        - **Steps:**
            1. Verify that safety mechanisms (e.g., input validation, access controls) are in place and effective.
        - **Expected Result:** System effectively prevents unauthorized access and misuse.

Security

- **Requirement:** All sensitive data should be encrypted and securely stored.
    - **Test Case 1:** Verify encryption and secure storage of sensitive data.
        - **Steps:**
            1. Verify encryption standards for data at rest and in transit.
        - **Expected Result:** All sensitive data is encrypted and securely stored.
    - **Test Case 2:** Verify communication security using JWT tokens with expiration dates.
        - **Steps:**

        1. Monitor the communication between the backend and frontend to ensure JWT tokens are used.
        2. Validate the tokens to ensure they are correctly implemented, include expiration dates, and are secure.
    - **Expected Result:** JWT tokens with expiration dates are used securely for communication between backend and frontend.
- **Test Case 3:** Verify password encryption in the database.
    - **Steps:**
        1. Check the database to ensure passwords are stored in an encrypted format.
        2. Attempt to retrieve and decrypt passwords to ensure they are securely stored.
    - **Expected Result:** Passwords are encrypted in the database.
- **Test Case 4:** Verify email verification for new users using only BGU emails.
    - **Steps:**
        1. Register a new user.
        2. Ensure the registration uses a BGU email address.
        3. Check the email for a verification link containing a token.
        4. Verify that clicking the link activates the user's account.
    - **Expected Result:** New users receive a verification email, and their account is activated upon clicking the verification link. Only BGU emails are accepted for registration.
- **Test Case 5**: Verify that user email is encoded into the JWT token.
    - **Steps**:
        1. Monitor the JWT token generation process.
        2. Ensure the user's email is encoded into the JWT token.
        3. Decode the JWT token to verify the presence of the email.
    - **Expected Result**: User email is encoded into the JWT token and can be successfully decoded.

Data Integrity

- **Requirement:** Ensure data integrity during data storage and retrieval.
    - **Test Case:** Verify data integrity for stored user and application data.
        - **Steps:**
            1. Insert, update, and retrieve data from the database.
            2. Check for data consistency and integrity.
        - **Expected Result:** Data remains consistent and accurate during storage and retrieval.
    - **Note:** Over 50 unit and integration tests have been used to test database stability, ensuring robust data handling and system reliability.

Usability

- **Requirement:** System should be easy to use and navigate.
    - **Test Case:** Verify usability through user testing.
        - **Steps:**
            1. Conduct usability testing sessions with representative users.
            2. Collect and analyze feedback on system usability.
        - **Expected Result:** Users find the system easy to use and navigate.

3. Test-Driven Development (TDD)

While the traditional Test-Driven Development (TDD) approach involves writing tests before writing the actual code, in this project, a slightly modified approach was used. The code was written first, followed immediately by writing the tests to check its validity. This approach helped in several ways:

- **Bug Detection:** Writing tests immediately after coding helped catch numerous bugs early in the development process. This immediate feedback loop ensured that issues were identified and resolved quickly, improving the overall quality of the code.
- **Learning Experience:** Through this process, it became clear that Python's dynamic nature makes it particularly prone to bugs. The absence of static type checking in Python can lead to runtime errors that are often difficult to predict without thorough testing.
- **Improved Code Quality:** The practice of writing tests right after coding enforced a discipline that led to better code quality. By validating the functionality immediately, it ensured that the code met the requirements and behaved as expected under various scenarios.
- **Type Checking with Pydantic:** To mitigate issues arising from Python's dynamic typing, the Pydantic library was used to enforce type checks for classes. This added a layer of validation, ensuring that the data structures adhered to the expected types and formats, further reducing runtime errors.

Benefits Realized:

1. **Early Bug Detection:** Immediate testing after coding led to the early identification and resolution of bugs.
2. **Enhanced Code Quality:** Continuous validation of the code helped maintain a high standard of quality.
3. **Learning and Adaptation:** Recognized the importance of testing in a dynamic language like Python, leading to more cautious and mindful coding practices.
4. **Type Safety with Pydantic:** Using Pydantic for type checking ensured that data adhered to the expected types, mitigating issues related to Python’s dynamic nature.

Examples of Test Cases Written Immediately After Code

Example 1: User Login

- **Test Case:** Verify student can log in using BGU credentials.
    - **Steps:**

    1. Write the login functionality code.
    2. Write the test case to check if the login is successful with valid credentials.
  - **Expected Result:** Student dashboard is displayed.

Example 2: Create Scholarship

- **Test Case:** Verify admin can create a new scholarship.
  - **Steps:**
    1. Write the code for creating a scholarship.
    2. Write the test case to check if the scholarship creation is successful.
  - **Expected Result:** Scholarship is created and visible in the list of available scholarships.

By integrating tests immediately after coding and using Pydantic for type checking, the project benefited significantly in terms of stability, reliability, and maintainability of the codebase.

In this project, although full-fledged random and automatically-generated tests were not utilized, random data was used to create test cases for the database. This approach helped ensure the robustness and reliability of the system by testing with a variety of inputs.

Approach:

- **Random Data for Test Cases:** Random dates and other random data were generated to create diverse test cases for the database. These test cases helped simulate real-world scenarios and ensured that the database could handle a variety of inputs.

5. Testing the User Interface

The user interface (UI) is a critical component of the application as it directly interacts with the users. The UI was thoroughly tested to ensure functionality, usability, and user experience. Here's a detailed outline of the approach taken:

UI Testing Process

- **Method:** The UI was tested by navigating through the application and performing various user actions. This included logging in, viewing scholarships, applying for scholarships, and verifying user notifications.
- **Scope:** The testing covered all major functionalities accessible through the UI, ensuring that each feature worked as intended and provided a satisfactory user experience.

UI Testing Process:

1. **Login Functionality:**
    - **Steps:**
        1. Navigate to the login page.
        2. Enter valid BGU credentials.
        3. Click the login button.
    - **Expected Result:** The user is redirected to the student dashboard.
2. **Viewing Scholarships:**
    - **Steps:**
        1. Navigate to the scholarships page after logging in.
        2. Verify the list of available scholarships is displayed.
    - **Expected Result:** Scholarships are listed with correct details and are accessible.
3. **User Notifications:**
    - **Steps:**
        1. Perform actions that trigger notifications (e.g., application submission, recommendation request).
        2. Check the user's email for corresponding notifications.
    - **Expected Result:** Appropriate notifications are received in the user's email.

Findings and Improvements

- **Usability:** The testing process highlighted several areas for improvement in terms of user experience. Feedback from testers was used to refine and enhance the UI.
- **Functionality:** Key functionalities were verified to work as expected, ensuring that users could complete their tasks without issues.
- **Responsiveness:** The UI was checked for responsiveness to ensure it works well on different devices and screen sizes.

By performing thorough testing and planning for future automated testing, the project ensures that the UI is functional, user-friendly, and ready for real-world use.

6. Testing Build, Integration & Deployment

To ensure the project builds correctly and operates as intended, several measures were implemented focusing on the build process, integration of components, and deployment. Here's a detailed outline of the approach taken:

Build Process

- **Python 3.9 Interpreter:** The project is built using the Python 3.9 interpreter to maintain consistency and compatibility across development and production environments.
- **Virtual Environment:** A virtual environment (venv) is utilized to manage dependencies and isolate the project environment. This ensures that the project is not affected by external package changes.
- **Requirements File:** A **requirements.txt** file is maintained to specify the exact versions of the packages required. This file ensures that the virtual environment can be rebuilt smoothly with all the necessary libraries at the correct versions.

Example of requirements.txt:

```
Flask==1.1.2

SQLAlchemy==1.3.23

Pydantic==1.8.2

pytest==6.2.2

.....
```

Integration Testing

- **Component Integration:** Integration tests are conducted to verify that different components of the system work together seamlessly. This includes interactions between the backend, database, and frontend components.
- **Automated Tests:** Automated tests are run to validate the functionality and sanity of the integrated system. These tests help identify issues that might arise from the integration of different modules.

Deployment

- **Smooth Deployment:** The use of a virtual environment and **requirements.txt** file ensures that deployment is smooth and all dependencies are correctly installed.

- **Version Control:** By specifying the exact versions of packages, it ensures that the same environment can be replicated in production, avoiding issues caused by version discrepancies.