Test Plan for Concierge

Change Log

Version	Change Date	By	Description
1.0.0	October 6th, 2024	All Members	Initial Testing Plan
2.0.0	November 3rd, 2024	All Members	Testing Plan for Sprint 2

Introduction

Scope

This is our testing scope for Sprint 1:

1. Amenities
    a. View all Amenities
    b. View Specified Amenity
    c. Create Amenity
    d. Update Amenity
    e. Delete Amenity
2. Incident Reports
    a. View all Incident Reports
        i. With filters
        ii. With no filter
    b. View Specified Incident Reports
    c. Create Incident Reports
    d. Update Incident Reports
    e. Delete Incident Reports
3. Task System
4. Account Management
    a. Server
        i. Verify get /index
        ii. Verify account login
        iii. Verify account creation
    b. Auth Manager
        i. Validate hashed passwords
        ii. Authenticate user login
        iii. Validate genuine hashes
    c. Database controller
        i. Test user retrieval
        ii. Test user addition
        iii. Test user deletion
    d. User dto
        i. Verify creation
        ii. Verify equality test
    e. User Service
        i. User creation
        ii. User deletion
    f. Validation Manager
        i. Validate password

        ii. Validate username
5. Permissions
    a. Group permissions
    b. Permissions
    c. Session keys
6. Guest WebApp
    a. Tasks
        i. Create task
        ii. View task creation dashboard
    b. Amenities
        i. View amenities
    c. Incident Reports
        i. Submit a report
7. Staff WebApp
    a. Tasks
        i. Create task
        ii. Update task
        iii. View task management dashboard
    b. Amenities
        i. View amenities management dashboard
        ii. Update amenity
        iii. Delete amenity
        iv. Create amenity
    c. Incident Reports
        i. View incident report management dashboard
        ii. Update incident report
        iii. Delete incident report
        iv. Create incident report
    d. Permissions
        i. View permissions management dashboard
        ii. Edit permissions

Roles and Responsibilities

Name	Net ID	Github Username	Role
Nhat Anh	nguyen74	nateng98	Full-stack Developer, Staff Front-end Supervisor
		rainclouded	Full-stack Developer, Database Supervisor
Mykola		mykolabesarab	Full-stack Developer, Guest Front-end Supervisor
Josh			Full-stack Developer, Back-end Supervisor, DevOps Supervisor
Leeroy	diliml1	LeeroyDilim	Full-stack Developer, Back-end Supervisor

Role Details

1. Full-stack Developer
    - A developer focused on implementing features across the presentation, logic, and data layer
2. Guest Front-end Supervisor
    - An analyst responsible for code reviewing pull requests of changes relating to the guest front end, ensuring additions meet requirements and follow good practices.
3. Staff Front-end Supervisor
    - An analyst responsible for code reviewing pull requests of changes relating to the staff front end, ensuring additions meet requirements and follow good practices.
4. Database Supervisor
    - An analyst responsible for code reviewing pull requests of changes relating to the SQL/Mongo Databases, ensuring additions meet requirements and follow good practices.
5. Back-end Supervisor
    - An analyst responsible for code reviewing pull requests of changes relating to the microservice architecture, ensuring additions meet requirements and follow good practices.
6. DevOps Supervisor
    - An analyst responsible for supervising the CI/CD pipeline. OBJ

Test Methodology

Test Levels

Core Feature: Amenities

Unit Tests and Logic/Persistence Integration Tests

1. The test ensures that the GetAmenities endpoint returns a response with a 200 OK status code and valid data when called.
2. When provided with a valid amenity ID, the GetAmenityByID endpoint returns a response with a 200 OK status code.
3. The test confirms that when an invalid amenity ID is used, the GetAmenityByID endpoint returns a 404 Not Found response.
4. When valid data is provided, the AddAmenity endpoint returns a 201 Created response, indicating that the amenity has been successfully added.
5. After successfully adding an amenity, the test verifies that the amenity can be fetched, and the response contains a 200 OK status code.
6. The test ensures that if invalid data is submitted to the AddAmenity endpoint, the response returns a 400 Bad Request status code.
7. After attempting to add an invalid amenity, the test checks that the amenity cannot be fetched, resulting in a 404 Not Found response.
8. The test confirms that trying to add a duplicate amenity returns a 400 Bad Request response, indicating a failure due to duplication.
9. If a null amenity is submitted, the AddAmenity endpoint returns a 400 Bad Request response, indicating invalid input.
10. When valid data is submitted to update an amenity, the test ensures that the response contains a 200 OK status code, indicating successful modification.
11. After updating an amenity, the test confirms that the updated data can be retrieved with a 200 OK response.
12. The test verifies that attempting to update an amenity with invalid data returns a 404 Not Found response.
13. After attempting an invalid update, the test ensures that the amenity cannot be fetched, resulting in a 404 Not Found response.
14. The test confirms that attempting to update a non-existing amenity returns a 404 Not Found response.
15. When a null value is submitted for updating an amenity, the endpoint returns a 404 Not Found response, indicating a failure.
16. The test ensures that when a valid amenity is deleted, the response contains a 200 OK status code.
17. After successfully deleting an amenity, the test checks that attempting to fetch it returns a 404 Not Found response.

18. When attempting to delete an amenity with an invalid ID, the response is a 404 Not Found.
19. After attempting to delete an invalid amenity, the test confirms that trying to fetch it results in a 404 Not Found response.

Integration Tests (Front End)

1. **Get and View All Amenities**
    - This test verifies that the application can successfully retrieve and display a list of all amenities. The test checks the API response to ensure that the correct data is being returned in an expected format, and the list is rendered properly on the frontend.
2. **Create New Amenity**
    - This test ensures that a new amenity can be created by sending a POST request to the server with the appropriate data. It validates the API's response to confirm the amenity was created successfully and checks if the newly added amenity appears in the list of amenities on the frontend.
3. **Delete Amenity**
    - This test confirms the ability to delete an existing amenity. It checks the API response for the deletion request and validates that the amenity is removed from the server's data as well as the frontend display.
4. **Edit Amenity**
    - This test ensures that an existing amenity can be edited. It sends a PUT request to update the amenity's information, validates the API response to confirm the changes were applied, and verifies that the updated amenity is reflected on the frontend.

Acceptance Tests

1. **Scenario:** Guest views hotel amenity information.
    - Upon opening the amenities dashboard, the guest should see a comprehensive list of all amenities offered by the hotel.
    - Each amenity listed must display the following information:
    - Title of the amenity
    - Description of the amenity
    - Operating hours for the amenity
    - The list must be presented in a clear and user-friendly format for easy navigation.
2. **Scenario:** Staff updates an amenity.
    - The staff member must be logged into the staff dashboard.
    - The staff member can access the amenity details for editing.

- When the staff member clicks the "Save" button after updating the amenity details:
- The system processes the update in real-time.
- The updated amenity details are successfully saved in the system.
- A confirmation message is displayed to the staff member indicating the update was successful.

Core Feature: Incident Reports

Unit Tests and Logic/Persistence Integration Tests

1. When filtering incident reports by a specified severity, the system should respond appropriately.
2. When filtering incident reports by a specific status, the system should respond appropriately.
3. When filtering incident reports by a range of dates, the system should respond appropriately.
4. When no filters are applied to the incident reports request, the system should respond appropriately.
5. When an invalid severity is used for filtering incident reports, the system should respond with an error.
6. When an invalid status is used for filtering incident reports, the system should respond with an error.
7. When invalid dates are provided for filtering incident reports, the system should respond with an error.
8. When retrieving an incident report by a valid identifier, the system should respond with the correct data.
9. When attempting to retrieve an incident report by an invalid identifier, the system should respond with an error.
10. When a valid incident report is submitted, the system should confirm successful creation.
11. After submitting a valid incident report, it should be possible to retrieve it successfully.
12. When submitting an incident report with a missing title, the system should respond with an error.
13. When submitting a null incident report, the system should respond with an error.
14. When submitting an incident report with a missing description, the system should respond with an error.
15. When submitting an incident report with an invalid filing person ID, the system should respond with an error.

16. When submitting an incident report with an invalid reviewer ID, the system should respond with an error.
17. When updating a valid incident report, the system should confirm the update.
18. After updating a valid incident report, it should be retrievable successfully.
19. When updating an incident report with a missing title, the system should respond with an error.
20. When attempting to update a non-existing incident report, the system should respond with an error.
21. When attempting to update an incident report with a null value, the system should respond with an error.
22. When deleting a valid incident report, the system should confirm the deletion.
23. After deleting a valid incident report, it should not be retrievable.
24. When attempting to delete an invalid incident report, the system should respond with an error.
25. After attempting to delete an invalid incident report, it should still be untraceable.

Integration Tests

1. **Get and View All Reports**
   - This test ensures that the system can successfully retrieve and display a list of all incident reports. It checks that a GET request to the server fetches the correct data and verifies that the reports are properly rendered in the user interface.
2. **Edit Incident Report**
   - This test verifies the ability to edit an existing incident report. It sends a PUT request with updated data, checks the server's response to confirm that the changes were applied, and confirms that the edited report is correctly reflected on the frontend.
3. **Delete Incident Report**
   - This test ensures that an incident report can be successfully deleted. It sends a DELETE request to the server, verifies the correct response from the backend, and checks that the deleted report is no longer visible in the list on the frontend.

Acceptance Tests

1. **Scenario:** Hotel Manager Views Past and Current Incident Reports
   - Given the hotel manager is logged in,
   - When the hotel manager accesses the incident reports dashboard,
   - Then the system displays a list of past and current incident reports.

    - And each report includes details such as severity, status, and description.
2. **Scenario:** Hotel Manager Updates an Incident Report
    - Given the hotel manager is viewing an incident report,
    - When the hotel manager updates the incident details and clicks "Save,"
    - Then the system updates the report in real-time.
    - And the system displays a confirmation message indicating the update was successful.

Core Feature: Accounts

Unit Tests and Logic/Persistence Integration Tests

- Confirm successful access to accounts with a GET request.
- Create a guest account successfully and handle duplicate guest creation.
- Create a staff account successfully and handle duplicate staff creation.
- Handle account creation failure due to empty username.
- Successfully log in with correct credentials for two different users.
- Handle login failure for an incorrect username and password.
- Confirm that the service uses the correct port.
- Delete an existing user successfully.
- Handle failure when deleting a non-existent user.
- Handle deletion failure with an expired token.
- Update an existing user's information successfully.
- Handle update failure for a non-existent user.
- Prepares a mock database with test user data for authentication tests.
- Confirms that the password hash is generated correctly for various user IDs and passwords.
- Validates that the password verification function accurately identifies correct and incorrect passwords for different users.
- Ensures that valid user credentials authenticate successfully while invalid credentials do not.
- Ensure that all users retrieved from the database are of the correct type and match expected test data.
- Confirm that all retrieved staff users match the first three entries of the expected test data.
- Verify that all retrieved guest users match the last three entries of the expected test data.
- Check that a new guest user can be created, added to the database, and that the number of guests has increased.

- Ensure that a new staff user can be created, added to the database, and that the number of staff has increased.
- Confirm that the largest user ID updates correctly after adding new staff members.
- Verify that deleting a staff user removes them from the database and updates the total number of staff accordingly.
- Ensure that the database connection is established with the correct URI.
- Check that the method for retrieving all users correctly queries the database.
- Confirm that the method for retrieving staff users correctly queries for users of type "staff."
- Verify that the method for retrieving guest users correctly queries for users of type "guest."
- Ensure that adding a staff user correctly calls the appropriate function to insert their details into the database.
- Confirm that adding a guest user correctly calls the appropriate function to insert their details into the database.
- Check that updating a user's details calls the appropriate function to replace their information without creating a new record.
- Verify that deleting a user correctly calls the function to remove their details from the database.
- Ensure that staff deletion permissions work correctly with valid and expired tokens, and that invalid tokens raise the right errors.
- Confirm that guest deletion permissions work similarly to staff permissions, checking for valid token functionality and error handling for invalid tokens.
- Verify that guest update permissions accept valid tokens, return false for expired tokens, and raise errors for invalid tokens.
- Ensure that staff update permissions accept valid tokens, return false for expired tokens, and raise errors for invalid tokens.
- Check that all service getters return no values before the setup process.
- Call the setup function to initialize the services.
- Confirm that all service getters return values after the setup process, indicating successful initialization.
- Verify that a user can be created with valid details.
- Ensure a user can be created with None attributes.
- Check that two identical users are considered equal.
- Confirm that users with the same attributes in a different order are equal.
- Validate that a user with extra fields is still considered equal.
- Assert that users with different usernames are not equal.
- Verify that a new guest can be created with valid attributes.

- Ensure a new staff member is created correctly with a valid password.
- Check that attempting to create a staff member with no attributes returns None.
- Confirm that the user deletion function correctly removes a guest from the user list.
- Assert that the user deletion function correctly removes a staff member from the user list.
- Ensure the list of remaining valid users is accurate after deletions.
- Check that various valid passwords for staff are accepted as correct.
- Confirm that various invalid passwords for staff are rejected.
- Ensure that various valid usernames for staff are accepted as correct.
- Validate that various invalid usernames for staff are rejected.
- Verify that new staff members with valid attributes are accepted.
- Confirm that new staff members with invalid attributes are rejected.
- Ensure that various valid usernames for guests are accepted as correct.
- Confirm that various invalid usernames for guests are rejected.
- Validate that new guests with valid attributes are accepted.
- Ensure that new guests with invalid attributes are rejected.

Integration Tests (Front end)

- Create a staff member successfully.
- Try creating the same staff member again but fail.
- Create a guest successfully.
- Try creating the same guest again but fail.
- Successfully log in with correct staff credentials.
- Try logging in as a guest without a password, but fail.
- Try logging in with an incorrect password, but fail.
- Delete a user successfully with valid permissions.
- Attempt to delete a user with an expired token, but fail.
- Attempt to delete a user with an invalid token, but fail.
- Successfully update a user's information with valid permissions.
- Attempt to update a user with an expired token, but fail.
- Attempt to update a user with an invalid token, but fail.
- Attempt to update a staff user without sufficient permissions, but fail.

Acceptance Tests

1. **Scenario:** User Logs In
    - Given the user has valid credentials,
    - When the user enters their username and password and clicks the "Login" button,

    - Then the system directs the user to their role-specific dashboard (guest, staff, or manager).
2. **Scenario:** Creating a New Account
    - Given the staff member has permission to create accounts,
    - When the staff member fills in the necessary fields for user creation and submits the form,
    - Then a new account is created successfully.
3. **Scenario:** Editing Permissions
    - Given the staff member is logged in and has permission to edit permissions,
    - When the staff member creates a new group, configures its permissions, and assigns accounts to it,
    - Then those accounts gain access to the specified tools.
4. **Scenario:** Modifying Preferences
    - Given the staff member is logged in,
    - When the staff member updates settings (e.g., default dashboard views, notification preferences, initial landing pages),
    - Then the system saves the changes and reflects them in the staff member's next login.

Core Feature: Task System

Unit Tests and Logic/Persistence Integration Tests

- Returns a successful response with a list of tasks when tasks are available.
- Returns a "not found" response when no tasks are available.
- Returns a successful response with the task details when a specific task is found.
- Returns a "not found" response when a specific task is not found.
- Returns a created response when a new task is successfully added.
- Returns a successful response when a task is updated successfully.
- Returns a successful response when a task is deleted successfully.
- Returns a "not found" response when attempting to delete a task that does not exist.
- Returns an error response when there is an issue with the request data for getting tasks.
- Returns an error response when there is an issue with the request data for retrieving a specific task.
- Returns an error response when there is an issue with the request data for adding a task.
- Returns an error response when there is an issue with the request data for updating a task.

- Returns an error response when there is an issue with the request data for deleting a task.

Integration Tests (Front End)

- Guest Frontend
    - Guest can view task creation dashboard
    - Guest can create task
- Staff Frontend
    -

Acceptance Tests

Scenario: Staff completes a task.
Given I'm a staff member, when I view an existing task I have completed on the staff dashboard, I can click the "Completed" button, and the system updates the task in real-time, giving a confirmation alert.

Scenario: Staff completes a task.
Given I'm a staff member, when I view an existing task I have completed on the staff dashboard, I can click the "Completed" button, and the system updates the task in real-time, giving a confirmation alert.
Scenario: Guest creates a service request.
Given I'm a guest, when I open the guest task dashboard and choose to create a new service request, I can enter the request details (type of service, description, priority). When I submit the request, it appears in my task list with a status of "pending," and the relevant staff members are notified.

Scenario: Staff views a list of incomplete tasks.
Given I'm a staff member, when I open the staff task dashboard, I can see a list of all incomplete tasks with details such as request type, room number, and priority.

Core Feature: Permissions

Unit Tests and Logic/Persistence Integration Tests

- Verifies that a request with missing required fields in the permission group fails.
- Confirms that a valid permission group with no members to remove is created successfully.
- Checks that an invalid permission ID in the group request results in a bad request.
- Ensures that a full update with specific permissions and members is applied to a group.

- Verifies that a patch request with an empty body still succeeds without changes.
- Confirms that an unauthorized request returns an unauthorized status.
- Verifies that a "guest" role cannot patch a permission group.
- Checks that a "viewer" role cannot patch a permission group.
- Verifies that only the group name is updated successfully.
- Ensures that an empty name retains the original group name without issues.
- Confirms that only the group description is updated successfully.
- Checks that an empty description retains the original group description without issues.
- Ensures that updating a single permission in the group works as expected.
- Verifies that an empty permission array does not alter existing permissions.
- Confirms that a nil permission list keeps current permissions intact.
- Checks that an invalid permission ID in a patch request fails with a bad request.
- 4o
- Checks that health check endpoint returns success.
- Verifies health check works without authentication.
- Tests retrieving permission groups with admin access returns OK.
- Verifies unauthorized access when fetching permission groups without authentication.
- Confirms unauthorized access for guests trying to fetch permission groups.
- Ensures viewers can successfully retrieve permission groups.
- Checks server error response with database failure when fetching permission groups.
- Tests successful retrieval of a specific permission group by ID with admin access.
- Confirms unauthorized access for guests when fetching a specific permission group.
- Verifies unauthorized response for viewers trying to access a specific permission group.
- Ensures server error with database failure when retrieving a specific permission group by ID.
- Verifies correct response when a specific permission group ID is not found.
- Checks bad request handling for invalid permission group ID format.
- Ensures server error on database failure when creating permission groups.
- Confirms bad request response for malformed data when creating permission groups.
- Creates a new permission group with full permissions as admin and verifies creation.

- Verifies unauthorized response for guests attempting to create permission groups.
- Ensures viewers get unauthorized response when trying to create permission groups.
- Tests creation of a permission group with no permissions set to false, and verifies creation.
- Creates a permission group with empty permissions and members, confirms success.
- Tests successful creation of a permission group without permissions or members set.
- Ensures creation of a permission group with no members returns success.
- Confirms bad request when attempting to create a permission group without a name.
- Successfully creates permission group with a missing description.
- Tests fetching permission groups with a valid database connection.
- Tests fetching permission groups with a bad database connection.
- Tests fetching a specific permission group by ID with a valid database connection.
- Tests fetching a specific permission group by ID with a bad database connection.
- Tests fetching a non-existent permission group by ID.
- Tests fetching a permission group with an invalid ID format.
- Tests creating a new permission group with a bad database connection.
- Tests creating a new permission group with an invalid request.
- Tests creating a permission group with all permissions set to true.
- Tests creating a permission group with all permissions set to false.
- Tests creating a permission group with no permissions and members.
- Tests creating a permission group with nil permissions and members.
- Tests creating a permission group with permissions but no members.
- Tests creating a permission group without a name.
- Tests creating a permission group without a description.
- Tests creating a permission group with members to remove.
- Tests creating a permission group with an empty member removal list.
- Tests creating a permission group with an invalid permission ID.
- Tests updating a permission group by adding/removing permissions and members.
- Checks if an admin can retrieve all permissions.
- Confirms a viewer can retrieve all permissions.
- Verifies if a guest can retrieve all permissions.
- Tests retrieval of permissions without authentication.

- Ensures response is correct when there are no permissions.
- Verifies error response when there is no database connection.
- Checks if an admin can retrieve a specific permission by ID.
- Confirms if a viewer can retrieve a specific permission by ID.
- Tests if a guest is denied access when retrieving a specific permission by ID.
- Verifies unauthorized access without authentication when retrieving by ID.
- Checks response for a nonexistent permission ID.
- Tests response for an invalid permission ID format.
- Verifies error response when there is no database connection.
- Confirms if an admin can create a new permission.
- Checks if a viewer is denied access when creating a permission.
- Tests if a guest is denied access when creating a permission.
- Verifies unauthorized access without authentication when creating a permission.
- Ensures proper response when creating a duplicate permission.
- Confirms error response for a permission request without a name.
- Verifies error response when creating a permission without a database connection.
- Validates successful login attempt with correct credentials.
- Confirms error response when database is unavailable during login.
- Checks for error when account client is unavailable during login.
- Verifies error response when JWT context is missing during login.
- Tests response for login attempt with missing or incorrect credentials.
- Confirms successful parsing of session key with admin permissions.
- Tests error response when JWT context is missing while parsing session key.
- Ensures unauthorized response when session key (API key) is missing.
- Verifies error response for invalid session key format.
- Confirms successful retrieval of public key.
- Checks error response when JWT context is missing during public key retrieval.