Test Plan - Album Atlas

Abdullah Abdullah, Naufil Ansari, Jinthushan Sutharman, Izaan Syed, Thomas Petkovic

Introduction

The tests for this product will focus on ensuring clean, uninterrupted, and intuitive functionality. Many advanced features have been added since the last iteration, such as image previewing and lyrical previews, which will need to be thoroughly tested and verified, so that they can be shipped out in a working state. Additionally, basic components such as item management and login/logout functionality will also need to be tested under adverse input conditions, so that any edge conditions can be ironed out before the application is open for public use.

Throughout testing, we plan on using pytest, a Python framework that allows unit testing for classes, objects, and properties. With that, the program's functionality can be tested using assertions and mock objects. Additionally, the tests can be automated through GitHub, auto-running whenever a commit is pushed so that we can assess the functionality of that commit and catch any issues or bugs.

There are three types of tests we plan to employ:

1. Unit Tests
2. Integration Tests
3. System Tests

The unit testing will be performed from a clear box view, as it will evaluate function outputs and attributes, and so will need to access all parts of the application. The integration tests, on the other hand, will utilise a translucent box view, as not all parts of the application will be relevant, only the interaction between major parts of the system will be considered. Finally, system tests will make use of an opaque box approach, as only the functionality and behaviour of the overall application will be considered; access to the internal components would be irrelevant.

Unit Tests (Clear Box)
These are tested internally using Pytest only.

Test ID	Components Involved	Preconditions	Steps	Expected Result	Testing Approach	Assigned Team Member
UT-01-CB	Registration System	None	Call validate_register with valid and invalid inputs	Valid inputs return True, invalid inputs return False	Pytest	Thomas
UT-02-CB	Login System	None	Call validate_login with valid and invalid inputs	Valid inputs return True, invalid inputs return False	Pytest	Thomas
UT-03-CB	Song Input	None	Call validate_inputs with various song details	Valid inputs return True, invalid return False	Pytest	Naufil
UT-04-CB	Dupe Username	Duplicate usernames must be present	Call is_username_taken with different usernames	Existing usernames return True, new ones return False	Pytest	Izaan
UT-05-CB	Song Search	Song data available	Call search_item with search queries	Matching song found or error message shown	Pytest	Naufil
UT-06-CB	Lyrical Preview	Lyrics available or missing	Call lyrical_preview with song names	Returns correct lyrics or "No lyrics available"	Pytest	Jin

Integration Tests (Opaque Box):
These employ the frontend and the backend together to test interaction between the two.

Test ID	Components Involved	Preconditions	Steps	Expected Result	Assigned Team Member
IT-01-TB	Window and UI Rendering	Application is launched	Open the application. Observe if all UI components are displayed properly.	Main window loads successfully. Search bar, buttons, and song list are visible and functional.	Abdullah
IT-02-TB	Search Bar, Song Database	Application is launched	Type a valid name in the search bar and press Enter. Observe if the results update correctly. Clear the search bar and press Enter. Type a non-existent name and press Enter.	Matching results appear in the song list. Clearing the search restores the full list. Non-existent name shows an error message or empty results.	Naufil
IT-03-TB	"Add Item" Button, Entry Fields, Save Button,	Application is launched	Click the "Add Item" button. Enter valid data in the input fields. Click the "Save"	With valid data: Song appears in the song list. With invalid data: Warning or error message	Jin

			button. Repeat with missing/invalid data.	appears.	
IT-04-TB	"Edit Item" Button, Entry Fields, Save Button,	Application is launched, there is a pre-existing song to edit	Select an item from the song list. Click the "Edit" button. Modify the fields. Click "Save" and check if changes apply. Try saving without making changes.	Changes are reflected in the song list. Saving without changes does not modify data.	Izaan

System Tests (Opaque Box):
These are tested in a full system environment. Tests conducted on both frontend and backend, focusing on user operation.

Test ID	Components Involved	Preconditions	Steps	Expected Result	Assigned Team Member
ST-01-OB	UI, Login System, Sort System, Add functionality, Edit functionality, Delete functionality, Logout system,	The application is installed and running. A test user account exists (Thomas1 / 1234).	1. Login System Open the application. Enter valid credentials (testuser / testpass). Click the Login button.	Login should work with no error message Songs should only be added to the current users' database. Search should only find songs in the current users'	Izaan

			2. Adding a New Item Click the Add Item button. Fill in the form with valid details (e.g., Item Name: "Test Song", Category: "Music"). Click Save. 3. Searching for an Item: Type "Test Song" into the Search Bar. Press Enter. 4. Editing an Item: Select "Test Song" from the UI. Click Edit. Modify the category from "Music" to "Favorite Music". Click Save. 5. Sorting Items: Click on the Category Column Header in the UI. Observe the item order. Click again to	database. Editing a song should only affect the specific song in the current user's database Item sort should work regardless of database chosen Deleting a song should only affect the specific song in the current user's database Logout should reset chosen database to default Application should not crash or hang	

			change sorting direction. 6. Deleting an Item: Select "Test Song" in the UI. Click Delete and confirm. 7. Logging Out: Click the Logout button.		
ST-02-OB	UI, Login System, Add functionality, Lyrical Preview	The application is installed and running.	1. Create new user account 2. Log into new account 3. Add new (real) song 4. Check lyrical preview	Registration should create new account, login should be performed, song should be added to DB, lyrical preview should show real lyrics	Jin
ST-03-OB	UI, Login System, Add functionality, Image addition/preview	The application is installed and running. An admin account exists. A sample image exists.	1. Log in to admin account 2. Add new (real) song 3. View details and upload image of the album 4. View details and verify image	Login should work with no error message Song should be added to all databases, including guest database Image should upload with no problem (admin privilege) Song should have cover art of the album visible to all users and guests	Naufil

Demonstration of Coverage

Mapping stories to tests:

User Story	**Test Case**
Login	UT-02-LS
Logout	ST-01-A
Search Albums	IT-02-SB
Edit Albums	IT-04-EI
Delete Albums	ST-01-A
Add Albums	IT-03-AI
Beautiful UI	ST-01-A
Lyrical Preview	UT-06-LP

High risk components:

- File I.O.: imperative not to have data corrupted when saving.
- User Data Handling: User data operations like registration, login, and data saving must be handled properly. Mixups can be very bad, and are a security risk.
- GUI Interactions: GUI elements must interact correctly with backend logic. Otherwise, program will be nonfunctional or worse - malfunctioning.

Test Case Distribution:

Unit Tests	6
Integration Tests	4
System Tests	3

Currently, there are a few gaps that are uncovered in test cases.

- Boundary Testing for Input Validation
    - Right now, any input can be put in any text field

    - Can be fixed by implementing input sanitization tests
- UI Adaptiveness
    - UI is not tested on different screen sizes
    - Can be solved by creating test cases that launch application with different screen sizes
- Error Handling in UI
    - No test for backend failure (e.g. missing database file)
    - Can create a test that mocks a backend failure mid-test
- Concurrent User Operations
    - No test for multiple users using the same data
    - Can create a test involving multiple instances running
- Performance Testing
    - No test for extremely large datasets
    - Can create a test that creates an extremely large mock dataset

Our tests employ both clear, translucent, opaque testing. Clear box testing is employed in the backend unit tests, to ensure that each file produces correct output within its own context. Translucent box testing is employed in the integration tests, to ensure that the outputs of each system are valid in the shared context of multiple systems. Opaque box testing is employed in the system tests, to ensure that the output of the program overall is usable by the client.

Overall, the testing plan aims to test the outputs, functionality, and behaviours of the program, in parts and as a whole, before shipping it out for public use.