Lucky Charms

SOEN 341 Project – LuckyCharms: Campus Events & Ticketing Web Application Testing Documentation

Testing Scope and Types

In this sprint, testing was done to make sure all main features and new changes worked properly and did not break anything that was already working. Testing covered both small isolated components and complete end-to-end user flows.

Acceptance Testing
Acceptance testing was done for each user story to confirm that the feature met its acceptance criteria. This included both core and extra features from Sprint 3 and Sprint 4.

Regression Testing
Used to verify that new features from Sprint 4 did not break existing features from earlier sprints. Most regression checks were done manually as part of functional testing, where we retested key flows such as login, browsing events, buying tickets, organizer tools, and admin actions. We also included light non-functional checks to ensure that performance, responsiveness, and overall stability were not negatively affected by recent updates.

Integration (API) Testing
Integration tests checked that the API and database worked correctly together. The tests implemented make actual HTTP requests to our API endpoints and verify the responses, ensuring that our controllers, routing, and database interactions all work together correctly. The tests covered actions such as searching and filtering events, logging in, buying a ticket, and using a QR code to redeem a ticket. These tests helped make sure information was saved and returned correctly.

Unit Testing
Unit tests were added as a plus, to verify the core logic of the system. These tests checked individual classes such as Event, Ticket, and Review, ensuring their properties and behaviors worked correctly before they interacted with controllers or the database.

Acceptance Testing

For acceptance testing, we focused on each user story and verified that every feature behaved the way a real user would expect. Even though we tested the stories individually, the goal was to confirm that the full feature worked properly from end to end. We walked through the actual user actions for students, organizers, and admins and checked that each story met all of its acceptance criteria.

This included testing flows such as browsing events, purchasing tickets, creating and managing events, submitting feedback, and using admin tools. We also tried edge cases and invalid inputs to see how the system handled real-world usage. All acceptance-testing scenarios and results were documented, and any issues were flagged for follow-up. The full documentation for these tests can be found in our GitHub repository.

Regression Testing

For this sprint, regression testing was our main approach. After adding new features and updates, we retested the entire web application to make sure nothing that previously worked was affected. We went through all the main user flows again, such as logging in, browsing events, buying tickets, using organizer tools, and performing admin actions, to confirm that everything still behaved as expected.

Each feature was checked through functional testing against its original acceptance criteria, and we created specific scenarios to ensure that recent changes did not introduce new issues. We also included basic non-functional tests to identify any performance or security concerns. All regression scenarios and results were documented in our GitHub repository to maintain clear traceability across sprints. Any failures were flagged immediately, while the specific defects and fixes are described in a later section.

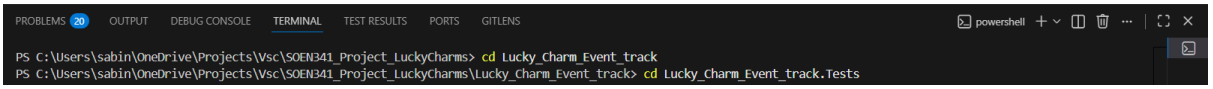
Integration (API) Testing

Integration tests were implemented using ASP.NET core's WebApplicationfactory pattern. This pattern creates an in-memory test server for each test run, allowing the application to be exercised end-to-end without hosting a real web server.
Each test is configured with its own isolated SQLite test database. The database is separate from the application's production SQLite database to ensure that no real data is affected when tests are run. The test database is created and seeded automatically for each test class using a custom factory that initializes data matching our application production schema.

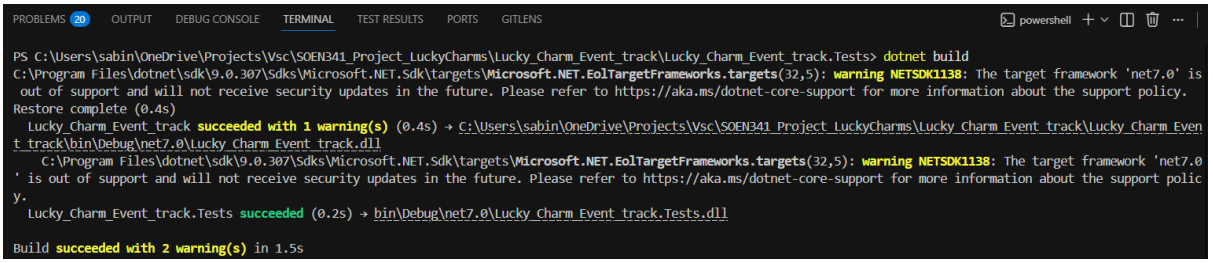
When the tests are run , the test runner (XUnit) builds both the application project and the test project , however it does not run the application like a hosted web app. The tests call the application code programmatically and the test project directly calls methods needed from the application's controllers. WebApplicationfactory spins up an in-memory version of the web server, and XUnit issues simulated HTTP requests through an HttpClient instance created by the factory. There are no real HTTP server listens on ports; the test runner (XUnit) makes HTTP requests after creating a client to test.
Integration tests make HTTP requests to the API endpoints and validate both the HTTP responses and the database state.

In order to run the tests the following steps should be taken :

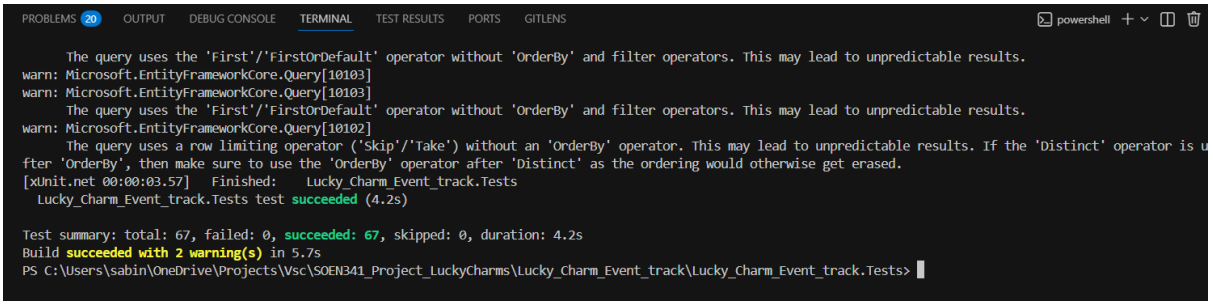
1- since the application program is separate from the test program , start by navigating to the test project directory using the following commands:



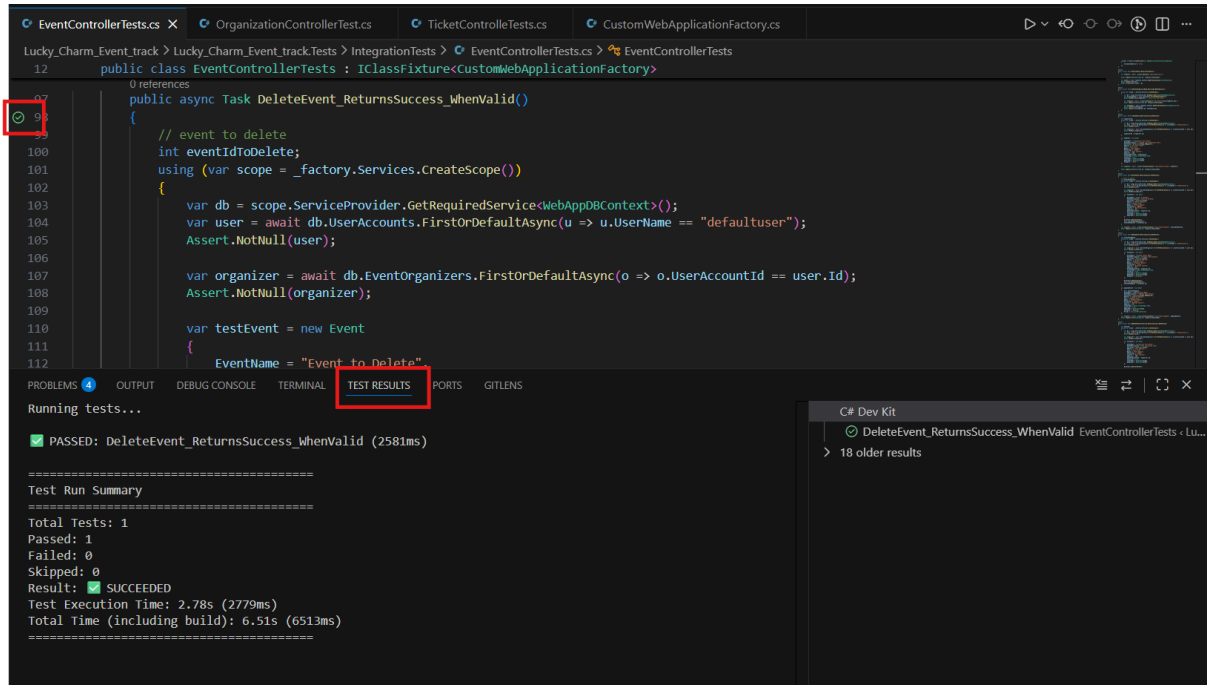
2- Build the test project using "dotnet build" :



3- Run tests using “dotnet test” .The terminal should display a summary of the results of all tests created in the project. In our case 67 tests were created to cover all possible cases:



4- After building the test project, each test method in the controller test files will show a run (play) button next to it. Clicking this button runs that individual test, and the detailed results will appear in the TEST RESULTS panel, as shown in the image below:



Continuous Integration & Unit Testing

Continuous Integration (CI) is handled through GitHub Actions. Every pull request must pass the automated checks before it can be merged, so the workflow runs automatically on each PR. The pipeline restores the project, builds it, and runs our unit tests to ensure nothing breaks when new code is added.

Since our full integration testing was already completed, we added unit tests as an additional layer of validation. As part of Sprint 4, 10 unit tests were created to verify the core logic of our main models (Event, Ticket, and Review). These unit tests run automatically through our CI pipeline, and GitHub displays the pass or fail results on every pull request.

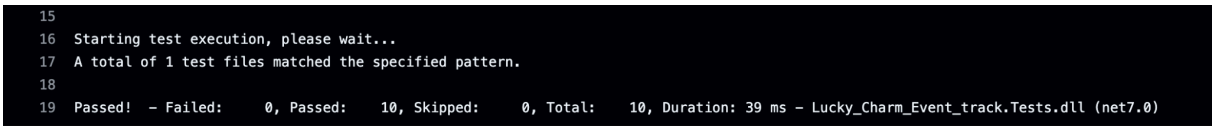


Figure 1: Continuous Integration Workflow and Automated Unit Test Results.

Defects, Fixes, and Final Quality Assessment

Fix for Issue #155 – Event Capacity and Ticket Limit Inconsistency

We found that organizers could lower an event's capacity or ticket limits even when tickets were already created or redeemed, which led to inconsistent metrics and allowed extra redemptions. To fix this, we added backend validation so capacity and ticket MaxQuantity can only be increased, not decreased, and we capped the event capacity at 2000 to avoid performance issues and slow .

Fix for Issue #152 – Favourite Events Visibility Across Users

We found that favourited events were being stored on the event itself, which meant that if one user favourited an event, it appeared as favourites for everyone. To fix this, we moved favourites into the UserAccount model so each user keeps their own list of favourited event IDs. Now the system checks the logged-in user's list only, so favourites are private and no longer shared across accounts.

Fix for Issue #95 - Event Points & Rewards

The "Pay with Points" button wasn't submitting anything because it had no form action and didn't trigger a post request. We fixed this by adding a small JavaScript handler that sets the correct MockPointPay endpoint and submits the form with the event ID. Now point payments are processed properly.

Fix – Blank Metrics Page for Seeded User

A seeded test user was seeing an empty metrics page because no default metrics were created for their account. This caused the dashboard to load with no data. We resolved the issue by generating the required metric entries during user seeding. Now all seeded users have the proper baseline metrics, and the metrics page loads correctly.

Fix – Invalid Date of Birth During Account Creation

During account creation, users could select a date of birth in the future, which caused validation errors and incorrect account data. We added a maximum date constraint set to today's date so future dates cannot be selected. This ensures cleaner data and prevents accidental invalid submissions.

Fix – Duplicate Feedback Submission

We discovered that students were able to submit feedback more than once for the same event, which created repeated reviews and messy data for organizers. To fix this, we added a proper check on the backend that blocks any second attempt once a student has already submitted feedback. Now the system correctly allows only one feedback entry per event per student, and the reviews stay clean and accurate.

Fix – Character Limit on Feedback Text Field

The feedback text box had no limit, so students could type extremely long messages that sometimes caused layout problems or very large entries in the database. We fixed this by adding a clear character limit and simple validation on both the page and the backend. This keeps feedback at a reasonable length, prevents errors, and makes sure all comments follow the same format.

Fix – Database Migration for Feedback Updates

Some new changes to the feedback feature weren't working on older database versions because the tables and fields were outdated. This caused issues during testing and prevented the new rules from applying. To fix this, we created and ran a new migration so the database matches the updated structure. With the migration applied, all environments now use the correct fields and everything runs smoothly with the new feedback logic.