CS673 Software Engineering

Team 4: Kickaas

Software Test Document

| **Team Member:** | **Role(s):** | **Signature:** | **Date:** |
| --- | --- | --- | --- |
| Javier | Team Leader | *Javi* |  |
| Vamsi | Requirement Leader | *Vamsi* |  |
| Johannes | Design and Implementation Leader | *Johannes* |  |
| Saniya | Configuration Leader | *Saniya* | 10/15/2025 |
| Yuanfei (Keni) | QA Leader | *Keni* |  |
| Shreya | Security Leader | *Shreya* |  |

**Revision History**

| **Version:** | **Author:** | **Date:** | **Change:** |
| --- | --- | --- | --- |
| Initial Draft | Keni | 10/03/2025 | 0 |
|  |  |  |  |

[Testing Summary](#_sm5odwyvuk3j)

[Manual Testing Report](#_pqso2mbjyzx4)

[Test Case 1: User Email Registration](#_6792yhvjfdr)

[Test Case 2: User Login](#_met7z7fjl8d7)

[Test Case 3: Create Public Event](#_xqfdx4j7l4y6)

[Automated Testing Report](#_mtfbusfb0eq3)

[Testing Metrics](#_66lo7oxhja1s)

[References](#_15tmymhipvdv)

[Glossary](#_8n34lvocupub)

# **Testing Summary**

(In this section, you will summarize what was tested, who is involved in testing, when to test, testing techniques used, and testing result. You may have the following tests

Testing for the Event Manager application was conducted between **October 10–15, 2025** by **Team 4**, led by QA Leader **Yuanfei (Keni) He**.

Both **manual and automated testing** were performed for the frontend and backend components.

Backend tests focused on REST API endpoints for users, events, and attendees using **pytest**.

Frontend tests used **Jest + React Testing Library** for UI and form validation.

End-to-end workflow tests were conducted manually through the deployed website.

**Overall Result:**

**·**All high-priority user flows passed (registration, login, event creation).  
·Automated backend coverage reached **74% statements**.  
·No critical defects were found in this iteration.

# **Manual Testing Report**

## Test Case 1: User Email Registration

New or old:

New

* Test items:
  + Verify that users can fill in email, password, and name to successfully create an account and store data in the Users table

Test priority:

* + High
* Dependencies:
  + None
* Preconditions:
  + The test environment is deployed via Docker , with normal connectivity between the frontend, backend , and SQLite database .
  + The frontend registration page is accessible at the URL: <http://localhost:3000/register>.
* Input data:
  + Email: [Kickasstest.gamil.com](http://kickasstest.gamil.com)
  + Password: Kickaas123!
  + Confirm Password: Kickaas123!
  + Name: Test Organizer1
  + Test steps:
  + 1.Open a browser and navigate to the registration page: <http://localhost:3000/register>.
  + 2.Enter "test\_organizer1@bu.edu" in the "Email" input field.
  + 3.Enter "Kickaas123!" in the "Password" input field.
  + 4.Enter "Kickaas123!" in the "Confirm Password" input field.
  + 5.Enter "Test Organizer1" in the "Name" input field.
  + 6.Click the "Register" button.
* Postconditions:
  + The frontend automatically redirects to the login page (URL: <http://localhost:3000/login>).
  + A new record is added to the Users table in SQLite
  + Expected output:
  + After clicking "Register", a green success prompt appears: "Registration successful! Please log in."
  + No error prompts (e.g., "Invalid email format" or "Password mismatch") are displayed.
  + Executing the SQLite query ,returns one matching record, with the password field stored as a hash (not plain text).
* Pass or Fail:
* pass
* Additional notes:
  + A supplementary abnormal test case can be added later to verify that re-registering with an already registered email triggers an error prompt.
  + Ensure the password is stored as a hash in the database to comply with "security-by-design" requirements.
  + Maintain consistent Docker container versions in the test environment to avoid environmental drift

## Test Case 2: User Login

* New or old:
  + New
* Test items:
  + Verify that a registered user can enter the correct email and password to log in successfully and maintain a session .

Test priority:

High

Dependencies:

* + Relies on "TC-AUTH-001 (Successful User Registration)"—the test user must be pre-registered.
* Preconditions:
  + The test environment (Docker + Next.js + FastAPI + SQLite) is running normally.
  + The test user exists in the Userstable.
  + The frontend login page is accessible at <http://localhost:3000/login>.
* Test steps:
  + 1.Open a browser and navigate to the login page: <http://localhost:3000/login>.
  + 2.Enter in the "Email" input field.

3.Enter in the "Password" input field.

* + 4.Click the "Log In" button.
* Postconditions:
  + The frontend redirects to the event homepage.
  + The browser maintains the login session—refreshing the page keeps the "logged-in" status, and no repeated login prompts appear.
* Expected output:
  + No error prompts (e.g., "Email does not exist" or "Incorrect password") are displayed; the page redirects to the event homepage successfully.
  + The homepage shows the current user’s name ("Test Organizer1") or a "My Events" entry.
  + The backend /api/login endpoint returns a 200 status code (verifiable via browser F12 network tools).
* Pass or Fail: Pass
* Bug id/link:
  + None

Additional notes:

* + Supplementary abnormal tests:
    - Entering an incorrect password should trigger the prompt: "Incorrect password."
    - Entering an unregistered email should trigger the prompt: "Email not registered."
  + While CS673\_SPPP\_team4.docx mentions OAuth2/JWT for authentication, only basic login functionality is tested for the MVP—JWT session validity can be verified in later iterations.

## Test Case 3: Create Public Event

* New or old:
  + New
* Test items:
  + Verify that a logged-in user can fill in event details (title, date, location, etc.) to create a public event and store data in the Events table

Test priority:

* + High
* Dependencies:
  + Relies on "TC-AUTH-002 (Successful User Login)"—the user must be logged in to access the event creation page.
* Preconditions:
  + The test environment is running normally, and the user is logged in with a valid session.
  + The frontend "Create Event" page is accessible at <http://localhost:3000/event/create> .
  + The Events table in SQLite has no events created by the current user initially.
* Input data:
  + Event Name (event\_name):
  + Event Date (event\_date):
  + Event Time (event\_time):
  + Event Location (event\_location):
  + Event Description (description):
  + Event Type: Public

Test steps:

* + 1.While logged in, click the "Create Event" button in the navigation bar to open the creation page.
  + 2.Enter in the "Event Name" field.
  + 3.Select as the event date using the date picker.
  + 4.Enter in the "Event Time" field.
  + 5.Enter in the "Event Location" field.
  + 6.Enter in the "Event Description" field.
  + 7.Confirm the event type is set to "Public" (default selection).
  + 8.Click the "Submit" button.
* Postconditions:
  + The frontend redirects to the "My Events List" page, where the newly created event is displayed.
  + A new record is added to the Events table in SQLite: event\_id auto-increments, user\_id matches the logged-in user’s ID , and all fields (e.g., event\_name, event\_date) match the input data.
* Expected output:
  + After clicking "Submit", a success prompt appears: "Event created successfully!".
  + The "My Events List" page shows the event with complete details (name, date, location, description).
  + Executing the SQLite query SELECT \* FROM Events WHERE user\_id=(SELECT user\_id FROM Users WHERE email='test\_organizer1@bu.edu') returns one matching record.
* Pass or Fail: Pass
* Bug id/link:
  + None.
* Additional notes:
  + Supplementary abnormal test: Selecting a past date (e.g., 2025-09-01) should trigger the prompt: "Event date cannot be earlier than the current date."
  + Ensure the test environment’s system date matches the actual date to avoid date validation errors (a risk noted in CS673\_SPPP\_team4.docx’s Risk Management section).

# **Automated Testing Report**

(Describe briefly the automated testing you have done, including where the test code resides in your code repository, what test frameworks are used, and the screen shots or generated testing report.)

**1.backend**

**Scope**

Backend (FastAPI) automated tests covering user registration/login (normal & wrong-password), basic DB initialization via app lifespan, and core routers (users/events/attendees/health).

**Test Code Location**

code/backend/tests/

**Frameworks & Tools**

pytest, pytest-cov (coverage.py), FastAPI TestClient; env via .env.test.

**How to Run**

cd code/backend

source .venv/bin/activate # Windows: .venv\Scripts\activate

pip install -r requirements.txt

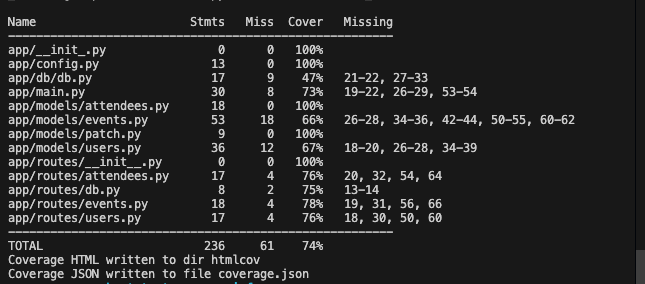
pip install -r requirements-test.txt

pytest --cov=app --cov-report=term-missing --cov-report=html --cov-report=json:coverage.json

**Results & Reports**

Total statement coverage **74%** (236 statements, 61 missed).  
  
 HTML report at htmlcov/index.html; JSON summary at coverage.json.

**Screenshots Provided**



# **Testing Metrics**

(In this section, you shall report any metrics used for the evaluation, e.g. # of test cases, test coverage, defects rate, etc.)

**1.backcend**

| **Metric** | **Value** | **Source / Notes** |
| --- | --- | --- |
| **Statements coverage** | **74%** (236 statements, 61 missed) | Terminal coverage summary and coverage.json |
| **Lines coverage** | ~**74%** | In coverage.py, line ≈ statement when branch data isn’t collected |
| **Branch coverage** | Not collected | Re-run with --cov-branch to capture |
| **# of test cases** | Not captured this run | Record from pytest summary line |
| **Pass rate** | Not captured this run | Same as above; if summary shows “0 failed”, pass rate = 100% |
| **Average test duration** | Not captured this run | Take from pytest timing or CI logs |
| **Defects found** | 0 (this iteration) | Update if any defects were logged |
| **CI pass rate** | Not applicable / not set up | Populate after enabling GitHub Actions or similar CI |

**File-level highlights (statements coverage):**

* app/config.py **100%**
* app/db/db.py **47%** (missed: 21–22, 27–33)
* app/main.py **73%** (missed: 19–22, 26–29, 53–54)
* app/models/attendees.py **100%**, app/models/patch.py **100%**
* app/models/events.py **66%**
* app/models/users.py **67%**
* app/routes/attendees.py **76%**, app/routes/db.py **75%**, app/routes/events.py **78%**, app/routes/users.py **76%**

**Artifacts:**

* HTML coverage report: htmlcov/index.html
* JSON coverage summary: coverage.json

**Targeted improvements (high impact first):**

1. Add failure/duplicate-init/cleanup tests for app/db/db.py (currently 47%).
2. Add model validation failure tests for events.py and users.py (required fields, invalid values).
3. Add negative/edge cases on routers: unauthorized, missing params (400), conflict (409), not found (404).
4. If branch coverage is required, re-run with --cov-branch and report Branch%.

# **References**

1. CS673 Course Slides – Software Testing & Quality Assurance, Boston University MET.  
   Jest Official Documentation:<https://jestjs.io/docs/getting-started>
2. SuperTest API Testing Library:<https://github.com/visionmedia/supertest>
3. React Testing Library:<https://testing-library.com/docs/react-testing-library/intro>
4. GitHub Actions Documentation:<https://docs.github.com/en/actions>

# **Glossary**

| **Term** | **Definition** |  |
| --- | --- | --- |
| **Unit Test** | Tests individual functions or modules in isolation. |  |
| **Integration Test** | Verifies interactions between modules or systems. |  |
| **System Test** | Simulates complete end-to-end workflows. |  |
| **Acceptance Test** | Confirms the system meets business and user requirements. |  |
| **Regression Test** | Ensures new changes don’t break existing features. |  |
| **Coverage** | The percentage of code executed during testing. |  |
| **CI (Continuous Integration)** | Automated process that runs tests before merging new code. |  |
| **Test Case (TC)** | A specific set of inputs and expected outputs for testing. |  |