Test Document

for

Lostify

Version 1.0

Prepared by

Group #6 — **Group Name: hAPPy Developers**

Aayush Kumar	230027	aayushk23@iitk.ac.in
Aman Raj	230116	amanraj23@iitk.ac.in
Anirudh Cheriyachanaseri Bijay	230140	anirudhcb23@iitk.ac.in
Ayush Patel	220269	ayushpatel22@iitk.ac.in
Krishna Kumayu	230576	kkrishna23@iitk.ac.in
Marada Teja Satvik	230636	maradateja23@iitk.ac.in
Satwik Raj Wadhwa	230937	satwikraj23@iitk.ac.in
Shaurya Johari	230959	shauryaj23@iitk.ac.in
Somaiya Narayan Aniruddh	231019	snarayana23@iitk.ac.in
Vinay Chavan	231155	vinay23@iitk.ac.in

Course: CS253

Mentor TA: Jeswaanth Gogula

Date: 05.04.2025

CONTENTS

Revision

Version	Primary Author(s)	Description of Version	Date Completed
v1.0	Shaurya Johari Ayush Patel Somaiya Narayan Aniruddh Satwik Raj Wadhwa Marada Teja Satvik Krishna Kumayu Vinay Chavan Aayush Kumar Anirudh Cheriyachanaseri Bijay Aman Raj	The Final version of the Test Document	06/04/25

1 INTRODUCTION

1.1 Test Strategy

We performed manual testing to test the frontend and used a testing framework to test the backend.

1.2 Testing Timeline

We mainly tested our software after completing the implementation phase for backend automation. However, we conducted widget testing for the frontend during the implementation process.

1.3 Testers

Throughout the week, we sat together for several hours, focusing on testing each feature such as unit testing, implementation testing and system testing. Testing was done by all.

1.4 Coverage Criteria

We have ensured that testing leads to branch coverage, statement coverage and function coverage. Coverage of automated tests in the backend was ensured by a test coverage measurement framework.

1.5 Tools Used for Testing

We have used the **pytest** unit testing framework to perform unit testing on the API endpoints of the backend. Further, we have used **Coverage** and **pytest-cov** to ensure test coverage.

We have also used **Dart DevTools** for debugging the frontend. It comes packaged with **Flutter inspector**, which analyses the widget tree hierarchy and exposes data stored in each node of the widget tree for inspection.

2 UNIT TESTING

The backend and frontend components were treated as separate units during this phase. Their integration was subsequently validated through integration testing. Backend unit testing involved providing inputs in the form of JSON objects and performing API calls, followed by comparing the actual responses with the expected results. Frontend unit testing focused on verifying that inputs were correctly converted into JSON format, validation checks functioned as intended, and alerts were displayed appropriately.

We conducted exhaustive unit testing for each section using a wide range of test cases to cover all possible scenarios. For documentation purposes, these sections have been clearly listed, with each section corresponding to an individual unit comprising multiple detailed test cases.

2.1 Authentication

2.1.1.a Registering a User

API Endpoint: `auth/signup/get_otp`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to check if a user can register successfully.

Test Results: User is able to register successfully. If a user with the username already exists, an appropriate error response is relayed. A 4-digit OTP is sent to the user's email; if the OTP service is down, an internal server error response is relayed.

i) Valid Input

```python
def test_signup_get_otp(client: FlaskClient, app: Flask):
    response = client.post(
        '/auth/signup/get_otp',
        json = {
            'username': 'uname',
            'password': 'pass',
            'profile': {
                'name': 'First M. Last',
                'email': 'uname25@example.com',
                'phone': '+91 12345 67890',
                'address': '123 Main St., City, Country',
                'designation': 'Student',
                'roll': 124
            }
        }
    )

    assert response.status_code == 201
    assert response.json['email'] == 'uname@iitk.ac.in'
    assert response.headers['Location'] == '/auth/signup/verify_otp'

    with app.app_context():
        assert get_db().execute(
            "SELECT * FROM awaitOTP WHERE username = 'uname'",
        ).fetchone() is not None
```

ii) Invalid Input

```python
@pytest.mark.parametrize(('username', 'password', 'profile', 'status'), (
    ('', '', {}, 400),
    ('a', '', {}, 400),
    ('a', 'a', {}, 400),
    ('test', 'test', {
        'name': 'First M. Last',
        'email': 'uname25@example.com',
        'phone': '+91 12345 67890',
        'address': '123 Main St., City, Country',
        'designation': 'Student',
        'roll': 124
    }, 409)
))
def test_signup_get_otp_validate_input(client: FlaskClient, username, password, profile, status):
    response = client.post(
        '/auth/signup/get_otp',
        json = {'username': username, 'password': password, 'profile': profile}
    )

    assert response.status_code == status
```

2.1.1.b Registering a User

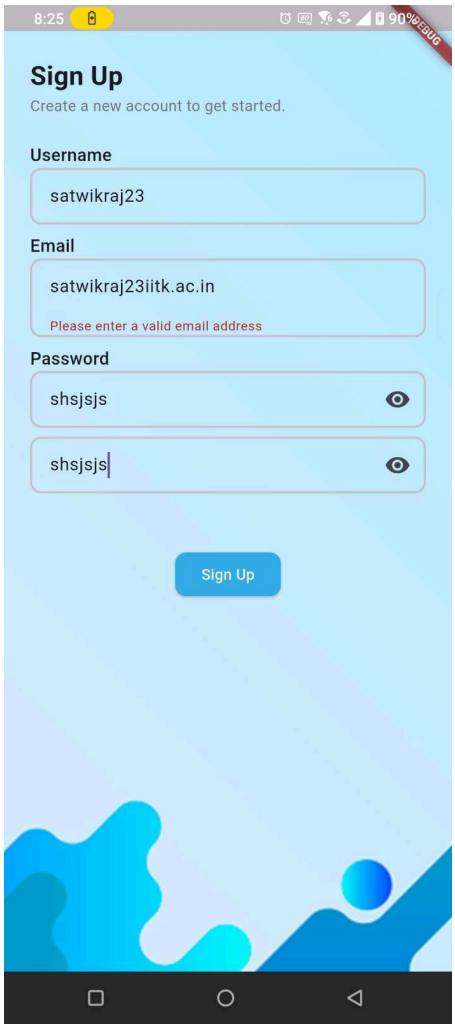
Test Owner: Krishna Kumayu

Test Date: 18/03/2025

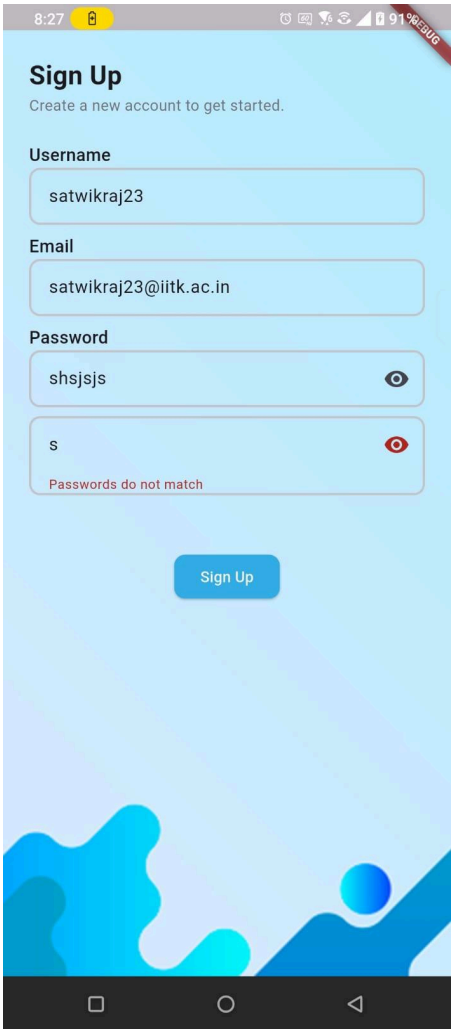
Test Description: The signup page's validation logic was exercised to verify the handling of invalid email formats, confirmation of password matching, and proper error message displays for failed validations. Additionally, the logic ensured that the signup button remained inactive until all inputs were valid.

Test Results: The system displayed appropriate alerts for invalid email formats and mismatched passwords, ensuring users were informed of the specific input errors. Valid credentials were successfully formatted into the correct JSON structure for signup requests.

Check for a valid email address:



Check for password mismatch:



2.1.2.a Logging in as a User

API Endpoint: `auth/login`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to check if a user can login successfully.

Test Results: User is able to login successfully upon supplying valid credentials. If the username does not exist or the password is incorrect, an appropriate error response is relayed. An encrypted session cookie is transmitted to the client through the response header on success.

i) Valid Input

```
56  def test_login(client: FlaskClient, app: Flask):
57      response = client.post(
58          '/auth/login',
59          json = {
60              'username': 'test',
61              'password': 'test'
62          }
63      )
64
65      assert response.status_code == 200
66      assert response.headers['Set-Cookie'].startswith('session=')
67      user_id, user_role = response.json['id'], response.json['role']
68
69      with app.app_context():
70          row = get_db().execute(
71              "SELECT id, role, counter FROM users WHERE username = 'test'",
72          ).fetchone()
73
74          assert row is not None
75          assert row['id'] == user_id
76          assert row['role'] == user_role
```

ii) Invalid Input

```python
@pytest.mark.parametrize(('username', 'password', 'status'), (
    ('', '', 400),
    ('a', '', 400),
    ('a', 'a', 404),
    ('test', 'wrong', 401),
))
def test_login_validate_input(client: FlaskClient, app: Flask, username, password, status):
    response = client.post(
        '/auth/login',
        json = {'username': username, 'password': password}
    )

    assert response.status_code == status

    if username == 'test' and password == 'wrong':
        with app.app_context():
            db = get_db()

            for i in range(4):
                assert client.post(
                    '/auth/login',
                    json = {'username': username, 'password': password}
                ).status_code == 401

                assert db.execute(
                    "SELECT counter FROM users WHERE username = 'test'",
                ).fetchone()[0] == i + 2

            assert client.post(
                '/auth/login',
                json = {'username': username, 'password': password}
            ).status_code == 429

            assert db.execute(
                "SELECT counter FROM users WHERE username = 'test'",
            ).fetchone()[0] == 5

            sleep(LOGIN_COUNTER_RESET_DELAY.seconds + 1)
            assert client.post(
                '/auth/login',
                json = {'username': username, 'password': password}
            ).status_code == 401

            assert db.execute(
                "SELECT counter FROM users WHERE username = 'test'",
            ).fetchone()[0] == 1
```

2.1.2.b Logging in as a User

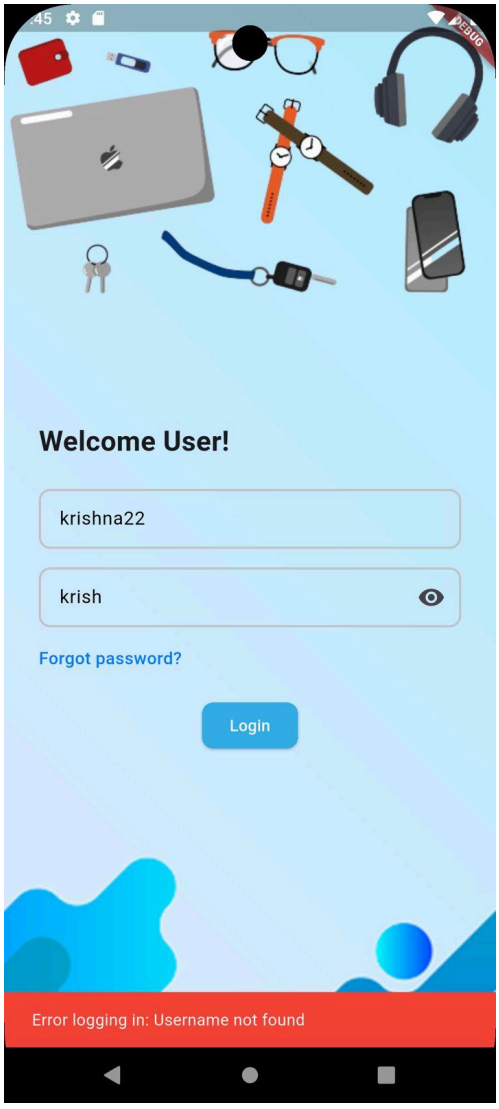
Test Owner: Krishna Kumayu

Test Date: 19/03/2025

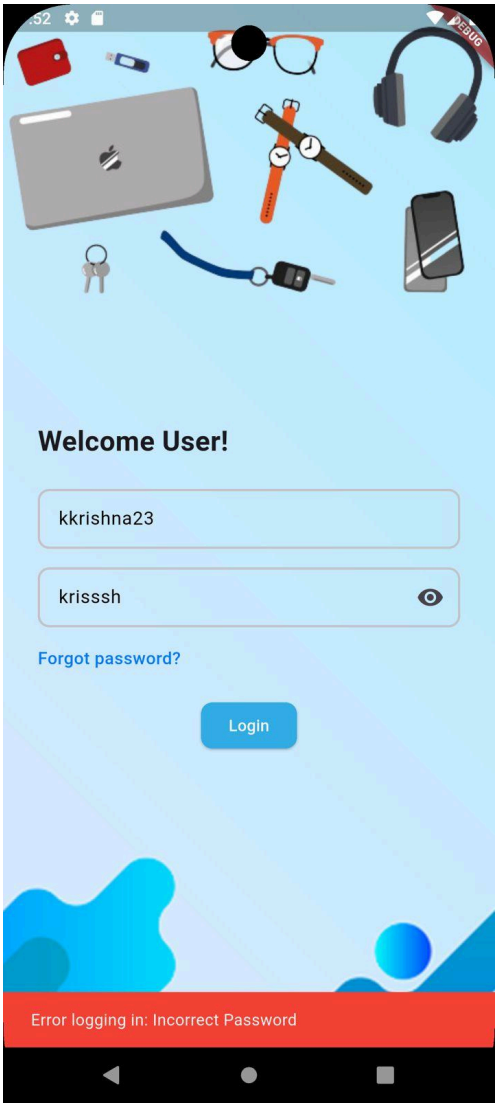
Test Description: The login page's validation logic was exercised to verify presence checks for username and password, database lookup for username existence, secure password-hash comparison, and proper JSON payload formation for successful logins.

Test Results: The system displayed appropriate alerts for missing inputs, unknown usernames, and incorrect passwords, and it packaged valid credentials into the correct JSON format.

Check for username existence:



Check for password correctness:



2.1.3.a Logging in as a Admin

API Endpoint: `auth/login`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to check if an admin can login successfully.

Test Results: Admin is able to login successfully upon supplying valid credentials. If the username does not exist or the password is incorrect, an appropriate error response is relayed. An encrypted session cookie is transmitted to the client through the response header on success.

i) Valid Input

```python
def test_login(client: FlaskClient, app: Flask):
    response = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    )

    assert response.status_code == 200
    assert response.headers['Set-Cookie'].startswith('session=')
    user_id, user_role = response.json['id'], response.json['role']

    with app.app_context():
        row = get_db().execute(
            "SELECT id, role, counter FROM users WHERE username = 'test'",
        ).fetchone()

        assert row is not None
        assert row['id'] == user_id
        assert row['role'] == user_role
```

ii) Invalid Input

```python
@pytest.mark.parametrize(('username', 'password', 'status'), (
    ('', '', 400),
    ('a', '', 400),
    ('a', 'a', 404),
    ('test', 'wrong', 401),
))
def test_login_validate_input(client: FlaskClient, app: Flask, username, password, status):
    response = client.post(
        '/auth/login',
        json = {'username': username, 'password': password}
    )

    assert response.status_code == status

    if username == 'test' and password == 'wrong':
        with app.app_context():
            db = get_db()

            for i in range(4):
                assert client.post(
                    '/auth/login',
                    json = {'username': username, 'password': password}
                ).status_code == 401

                assert db.execute(
                    "SELECT counter FROM users WHERE username = 'test'",
                ).fetchone()[0] == i + 2

            assert client.post(
                '/auth/login',
                json = {'username': username, 'password': password}
            ).status_code == 429

            assert db.execute(
                "SELECT counter FROM users WHERE username = 'test'",
            ).fetchone()[0] == 5

            sleep(LOGIN_COUNTER_RESET_DELAY.seconds + 1)
            assert client.post(
                '/auth/login',
                json = {'username': username, 'password': password}
            ).status_code == 401

            assert db.execute(
                "SELECT counter FROM users WHERE username = 'test'",
            ).fetchone()[0] == 1
```

2.1.3.b Logging in as a Admin

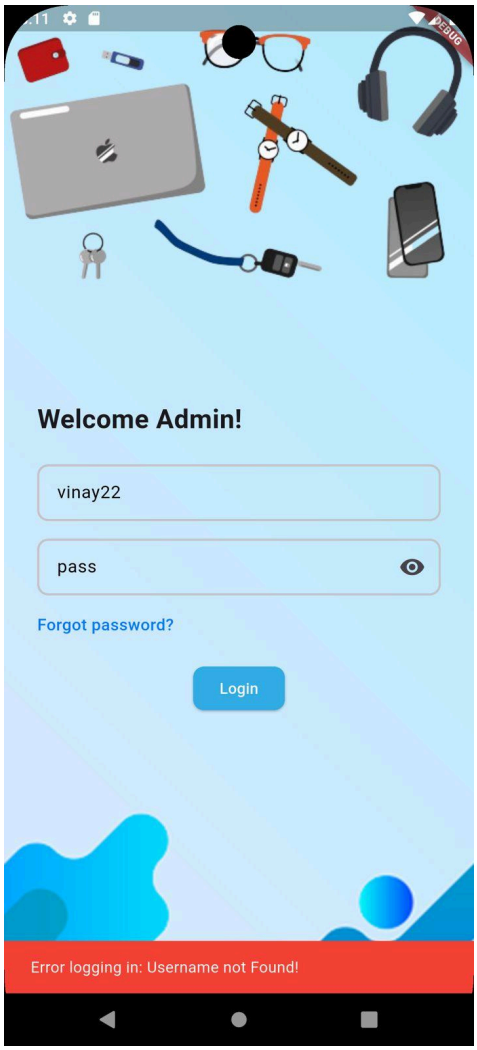
Test Owner: Krishna Kumayu

Test Date: 22/03/2025

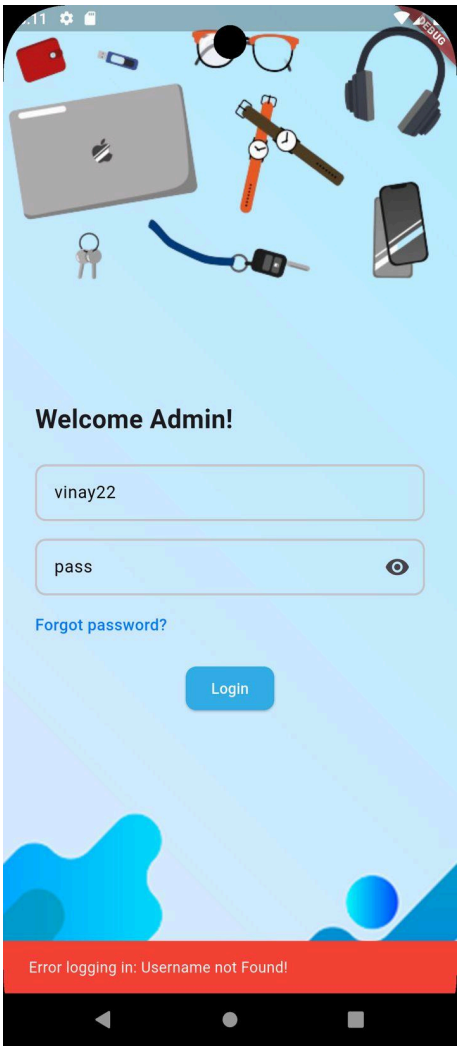
Test Description: The login page's validation logic was exercised to verify presence checks for admin's username and password, database lookup for admin's username existence, secure password-hash comparison, and proper JSON payload formation for successful logins.

Test Results: The system displayed appropriate alerts for missing inputs, unknown admin's usernames, and incorrect passwords, and it packaged valid credentials into the correct JSON format.

Check for username existence:



Check for password correctness:



2.1.4.a Logging out

API Endpoint: `auth/logout`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to check if a user can log out successfully.

Test Results: Session cookies are cleared, if they exist. The action cannot fail unless an internal server error occurs.

```
125  def test_logout(client: FlaskClient):
126      response = client.get('/auth/logout')
127      assert response.status_code == 205
```

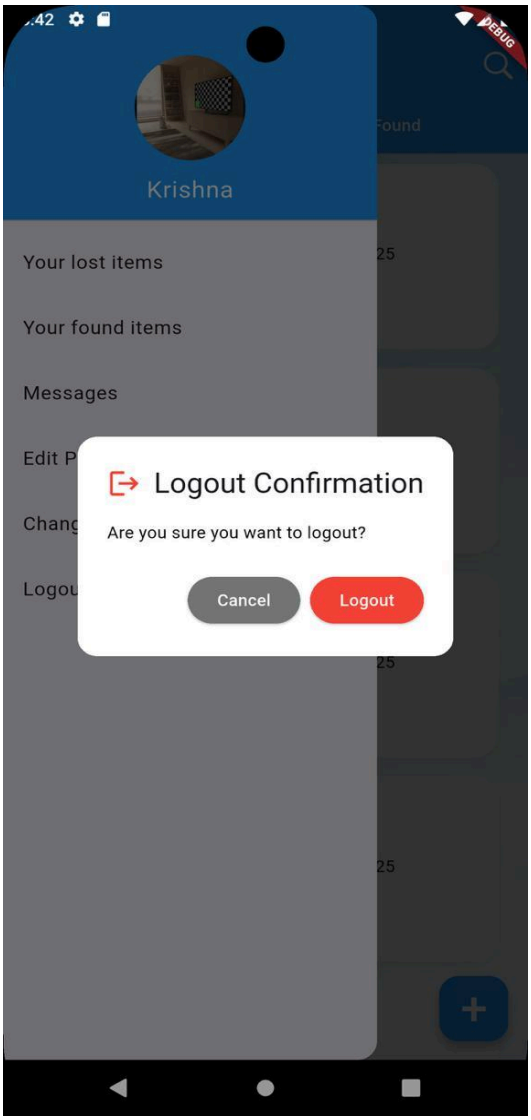
2.1.4.b Logging out

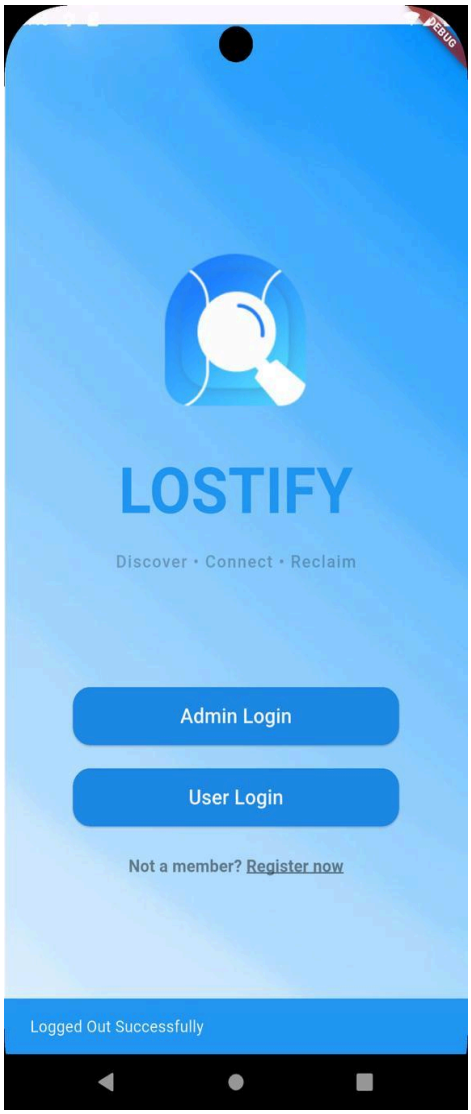
Test Owner: Krishna Kumayu

Test Date: 22/03/2025

Test Description: Verify that the logout functionality correctly invalidates the user's session or token, preventing further access to protected resources.

Test Results: The logout feature is working as expected. All scenarios resulted in correct and secure behavior, ensuring proper session termination and access control.





2.1.5.a Change Password

API Endpoint: `auth/change_password`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to check if a user can change his/her password successfully.

Test Results: Password is updated in the database if the old password is correct and the new password is not empty. Errors (non-existent user, incorrect password, missing fields) are informed through relevant HTTP responses.

```python
def test_change_password(client: FlaskClient, app: Flask):
    # Test without session cookie
    response = client.post(
        '/auth/change_password',
        json = {
            'old_password': 'test',
            'new_password': 'new_test'
        }
    )

    assert response.status_code == 401

    # Set session cookie
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test with session cookie
    response = client.post(
        '/auth/change_password',
        json = {
            'old_password': 'test',
            'new_password': 'new_test'
        },
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 204

    # Check if password is changed in the database
    with app.app_context():
        assert check_password_hash(get_db().execute(
            "SELECT password FROM users WHERE username = 'test'",
        ).fetchone()[0], 'new_test')

    # Test with incorrect password
    response = client.post(
        '/auth/change_password',
        json = {
            'old_password': 'a',
            'new_password': 'new_test'
        },
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 401
```

2.1.5.b Change Password:

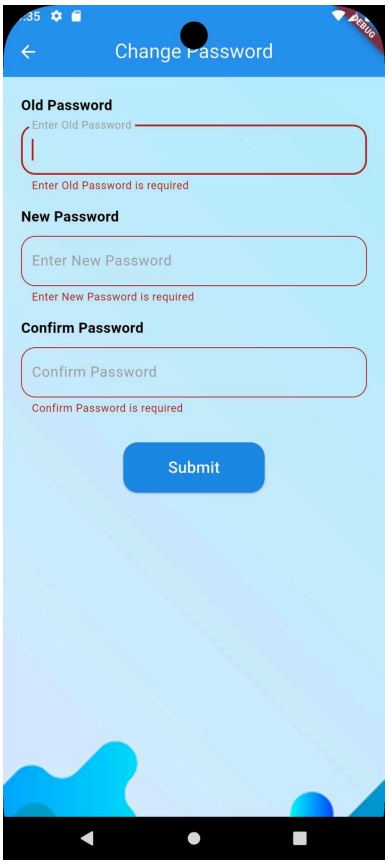
Test Owner: Aayush Kumar

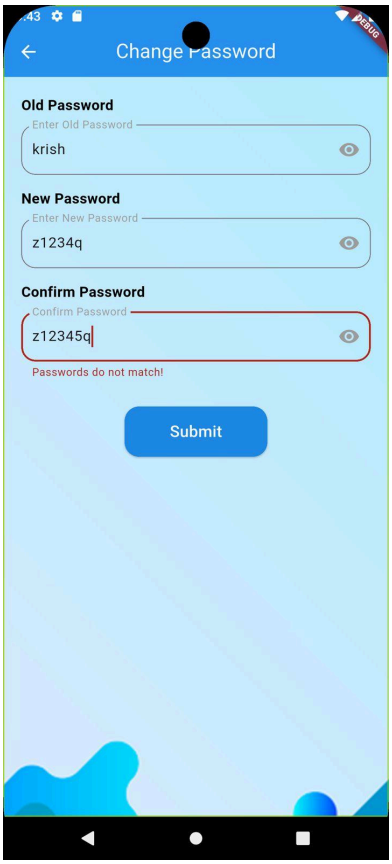
Test Date: 23/03/2025

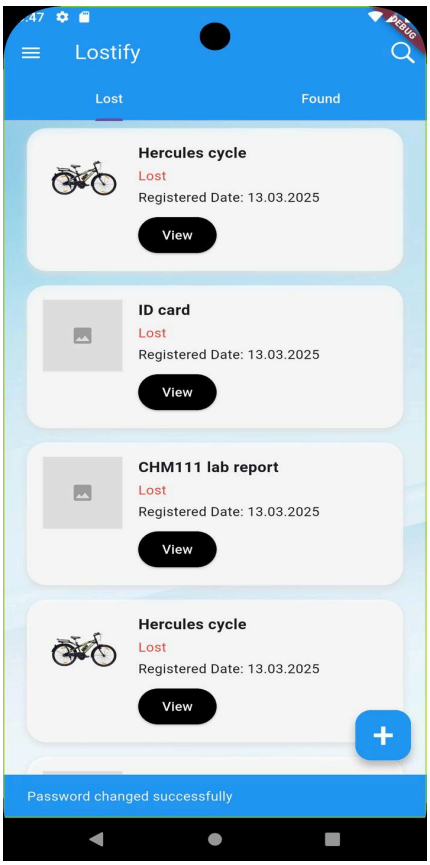
Test Description: The change password feature verifies that the current password, new password, and confirm password fields cannot be left blank; that the supplied current password matches the stored credential; that the new and confirm password entries are identical; and that a correctly formatted JSON payload is generated for the backend update request.

Test Results: The system displayed alerts for any missing fields, incorrect current passwords, or mismatched entries, and when all inputs were valid it successfully updated the password and produced the correct JSON object.

Check for changed password status:







2.1.6.a Reset Password

API Endpoint: `auth/reset_password`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to reset the user's password if he/she forgets his/her login credentials.

Test Results: A random 16-character password is generated and emailed to the user's iitk.ac.in email address if the username exists. The sent email advises the user to change the password upon login with the new password. An error response is relayed if the username does not exist.

i) Success

```
184 def test_reset_password(client: FlaskClient):
185     response = client.post(
186         '/auth/reset_password',
187         json = {
188             'username': 'test'
189         }
190     )
191
192     assert response.status_code == 204
```

ii) Failure

```
202 def test_reset_password_validate_input(client: FlaskClient):
203     response = client.post(
204         '/auth/reset_password',
205         json = {
206             'username': ''
207         }
208     )
209
210     assert response.status_code == 400
211
212     response = client.post(
213         '/auth/reset_password',
214         json = {
215             'username': 'non_existent_user'
216         }
217     )
218
219     assert response.status_code == 404
```

2.1.6.b Reset Password:

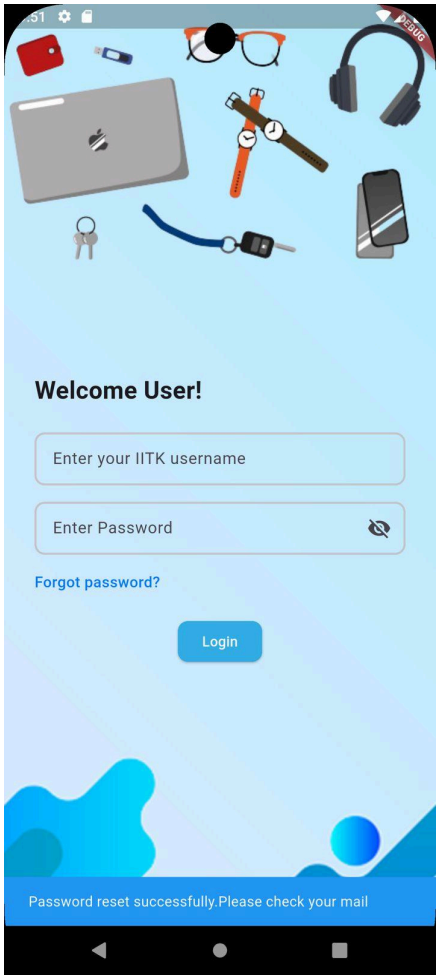
Test Owner: Aayush Kumar

Test Date: 23/03/2025

Test Description: The change-password feature was exercised to verify that the current password, new password, and confirm-password fields cannot be left blank; that the supplied current password matches the stored credential; that the new and confirm-password entries are identical; and that a correctly formatted JSON payload is generated for the backend update request.

Test Results: The system displayed alerts for any missing fields, incorrect current passwords, or mismatched entries, and when all inputs were valid it successfully updated the password and produced the correct JSON object.

Check for fetch profile (dashboard):



2.2 Profile Management

2.2.1.a Fetch Profile

API Endpoint: `users/<id>/profile`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to fetch a user's profile.

Test Results: An authenticated user can fetch profile details of any user; the details are passed in the response body in JSON format. If the user id does not exist or the user is unauthenticated, an appropriate error response is relayed.

i) Valid Input

```
6   def test_fetch_profile(client: FlaskClient):
7       # Authenticate
8       cookie = client.post(
9           "/auth/login",
10          json = {
11              "username": "test",
12              "password": "test"
13          }
14      ).headers["Set-Cookie"]
15
16      # Test fetching a user profile with authentication
17      response = client.get(
18          "/users/1/profile",
19          headers = {
20              "Cookie": cookie
21          }
22      )
23
24      assert response.status_code == 200
25      assert response.json["userid"] == 1
```

ii) Invalid Input

```
27  def test_fetch_profile_validate_input(client: FlaskClient):
28      # Test fetching a user profile without authentication
29      response = client.get("/users/1/profile")
30
31      assert response.status_code == 401
32
33      # Authenticate
34      cookie = client.post(
35          "/auth/login",
36          json = {
37              "username": "test",
38              "password": "test"
39          }
40      ).headers["Set-Cookie"]
41
42      # Test fetching a non-existent user profile
43      response = client.get(
44          "/users/100/profile",
45          headers = {
46              "Cookie": cookie
47          }
48      )
49
50      assert response.status_code == 404
```

2.2.1.b Fetch Profile

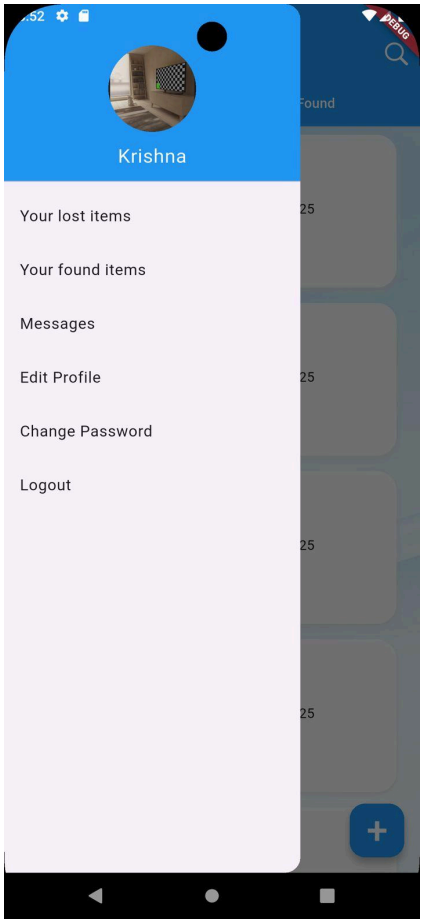
Test Owner: Satwik Raj Wadhwa

Test Date: 27/03/2025

Test Description: The fetch profile feature was tested to verify that the application correctly retrieves and displays the user's stored profile data, including name, profile picture, and associated items. The test ensured that API requests for profile data are authenticated and formatted correctly, and that the response is parsed and rendered properly in the UI.

Test Results: The system successfully fetched and displayed the profile information of the logged-in user, including the profile image and username, without any errors or delays, confirming correct backend integration and UI rendering.

Check for fetch profile (dashboard):



2.2.2.a Edit Profile

API Endpoint: `users/<id>/profile`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is used to edit a user's profile.

Test Results: An authenticated user can edit his/her own profile details. If the user is unauthenticated or the user attempts to edit someone else's profile, an appropriate error response is relayed.

i) Valid Input

```
52  def test_update_profile(client: FlaskClient, app: Flask):
53      # Authenticate
54      cookie = client.post(
55          "/auth/login",
56          json = {
57              "username": "test",
58              "password": "test"
59          }
60      ).headers["Set-Cookie"]
61
62      # Test updating own user profile
63      response = client.put(
64          "/users/0/profile",
65          json = {
66              "name": "Updated Name"
67          },
68          headers = {
69              "Cookie": cookie
70          }
71      )
72
73      assert response.status_code == 204
74      with app.app_context():
75          assert get_db().execute(
76              "SELECT name FROM profiles WHERE userid = 0"
77          ).fetchone()[0] == "Updated Name"
```

ii) Invalid Input

```python
def test_update_profile_validate_input(client: FlaskClient, app: Flask):
    # Test updating a user profile without authentication
    response = client.put(
        "/users/1/profile",
        json = {
            "name": "Updated Name"
        }
    )

    assert response.status_code == 401

    # Authenticate
    cookie = client.post(
        "/auth/login",
        json = {
            "username": "test",
            "password": "test"
        }
    ).headers["Set-Cookie"]

    # Test updating someone else's user profile
    response = client.put(
        "/users/1/profile",
        json = {
            "name": "Updated Name 2"
        },
        headers = {
            "Cookie": cookie
        }
    )

    assert response.status_code == 403

    # Empty update request
    response = client.put(
        "/users/0/profile",
        json = {},
        headers = {
            "Cookie": cookie
        }
    )

    assert response.status_code == 400

    # Test updating own user profile with invalid data
    response = client.put(
        "/users/0/profile",
        json = {
            "name": None
        },
        headers = {
            "Cookie": cookie
        }
    )

    assert response.status_code == 400

    # Test emptying required fields
    response = client.put(
        "/users/0/profile",
        json = {
            "name": ""
        },
        headers = {
            "Cookie": cookie
        }
    )

    assert response.status_code == 400
```

2.2.2.b Edit Profile

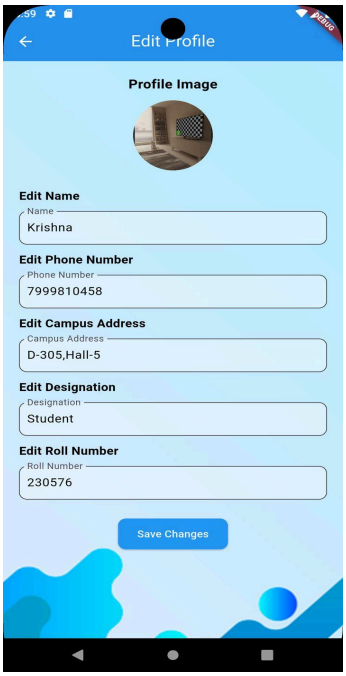
Test Owner: Satwik Raj Wadhwa

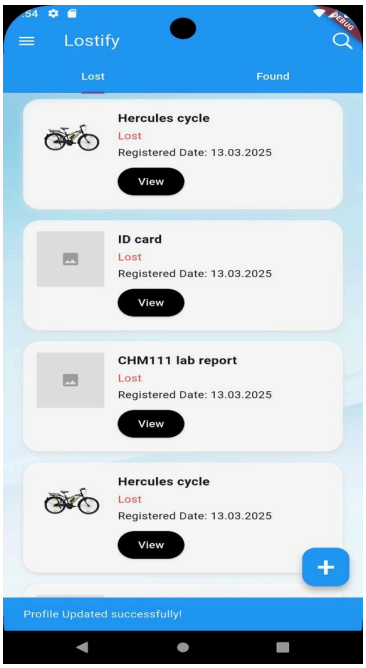
Test Date: 25/03/2025

Test Description: The Edit Profile feature was tested to verify that users can view and update their existing profile information seamlessly. Upon accessing the Edit Profile section, all previously saved user details—including name, email, phone number, and profile picture—were pre-populated in the corresponding input fields. The test focused on confirming that users could modify this information by adding, updating, or deleting text, and then save the changes successfully. Validation checks were performed to ensure only valid data could be submitted, and appropriate error messages were shown for invalid or incomplete inputs. The test also checked whether updates were correctly reflected in the database and user interface.

Test Results: The system allowed users to edit and save their profile information accurately. All form fields were validated correctly, and appropriate error messages were shown for invalid inputs. Upon successful submission, a confirmation message was displayed, and the updated profile data was saved in the database and reflected in the user interface without issues. The Edit Profile functionality performed as expected with proper UI feedback and backend integration.

Check for edited profile messages:





2.3 Post Handling

2.3.1.a Create a Post

API Endpoint: `items/<id>`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is to create a user's post.

Test Results: An authenticated user can create a post. If the post format is invalid, the request misses required fields, or the user is unauthenticated, an appropriate error response is generated.

i) Valid input

```python
def test_create(client: FlaskClient, app: Flask):
    # Test adding an item without authentication
    response = client.post(
        '/items/post',
        json = {
            'type': 0,
            'title': 'Test Item',
            'description': 'This is a test item.',
            'location1': 'Test Location'
        }
    )

    assert response.status_code == 401

    # Authenticate
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test adding an item
    response = client.post(
        '/items/post',
        json = {
            'type': 0,
            'title': 'Test Item',
            'description': 'This is a test item.',
            'location1': 'Test Location'
        },
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 201
    assert type(response.json['id']) is int
    assert response.headers['Location'] == f"/items/{response.json['id']}"

    with app.app_context():
        assert get_db().execute(
            "SELECT creator FROM posts WHERE id = ?",
            (response.json['id'],)
        ).fetchone()[0] == 0    # ID of user 'test'
```

ii) Invalid input

```python
@pytest.mark.parametrize(('type', 'title', 'location1'), (
    (2, '', 'Test Location'),
    (1, 'Title', ''),
    (0, None, None)
))
def test_create_validate_input(client: FlaskClient, type, title, location1):
    # Authenticate
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test adding an item with invalid input
    response = client.post(
        '/items/post',
        json = {
            'type': type,
            'title': title,
            'location1': location1
        },
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 400
```

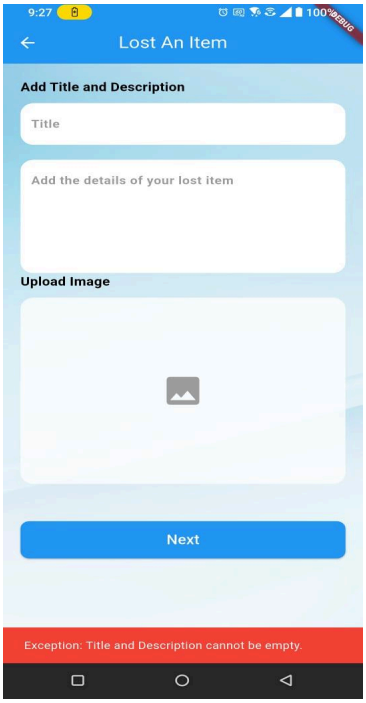
2.3.1.b Create a Post

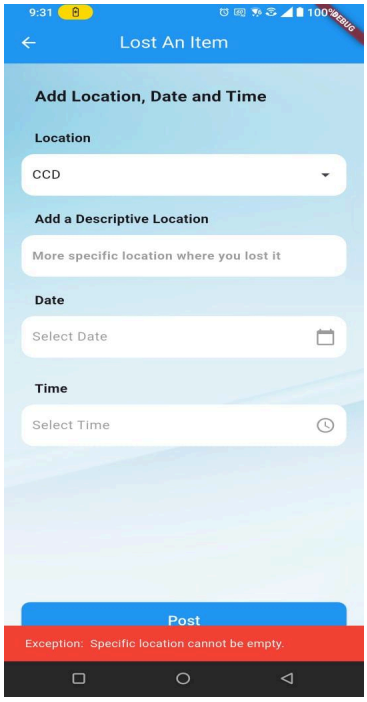
Test Owner: Aman Raj
Test Date: 02/04/2025

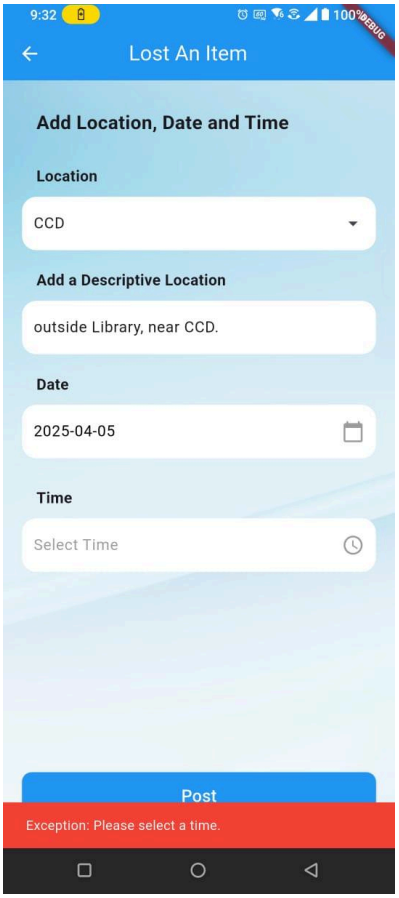
Test Description: The create post feature was tested to verify that the application allows users to submit both lost and found item posts. The functionality included a form that captures essential information, such as title, image, description, location, date, and time. The test ensured that the submitted data was formatted correctly, validated for completeness, and displayed appropriate confirmation alerts upon successful submission. Additionally, the application correctly categorized the posts as either lost or found in the database, ensuring an organized retrieval process for users.

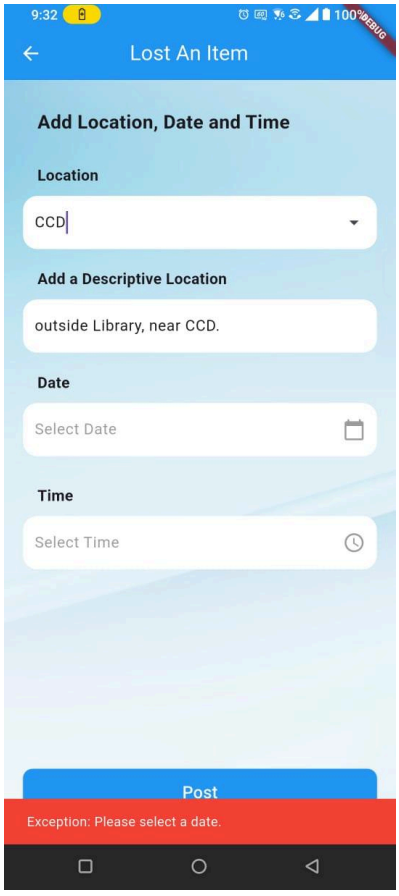
Test Results: The system successfully created and displayed both lost and found item posts with all necessary details, including title, image, description, location, date, and time, without any errors. The posts were categorized correctly in the database, confirming proper form validation, backend integration, and UI rendering for user submissions.

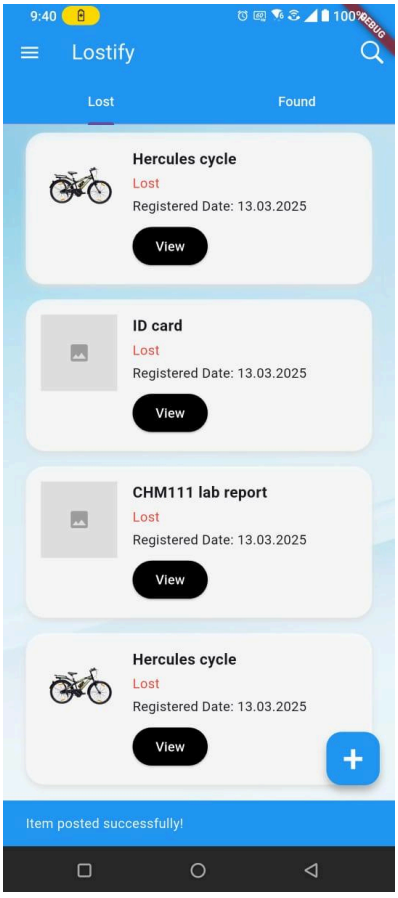
Post a Lost Item :



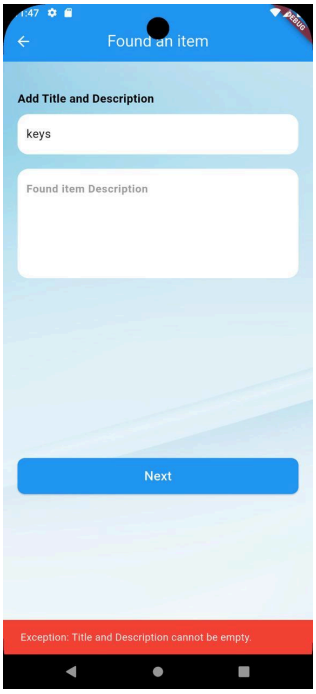


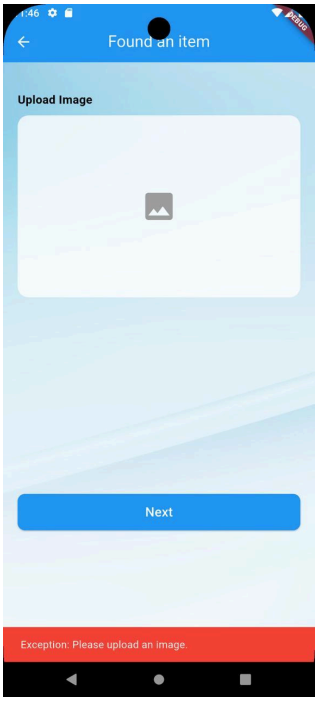


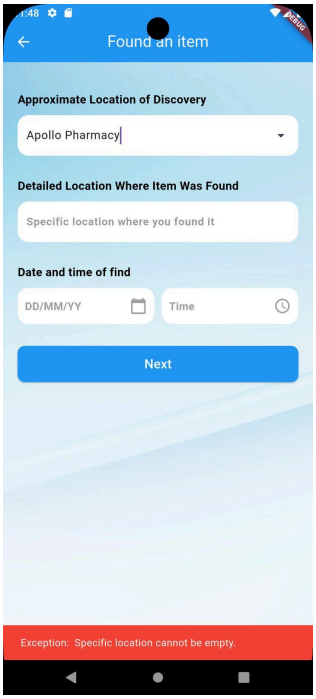


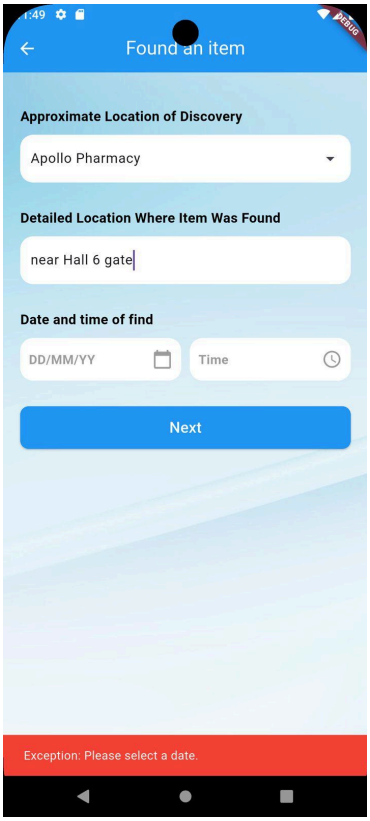


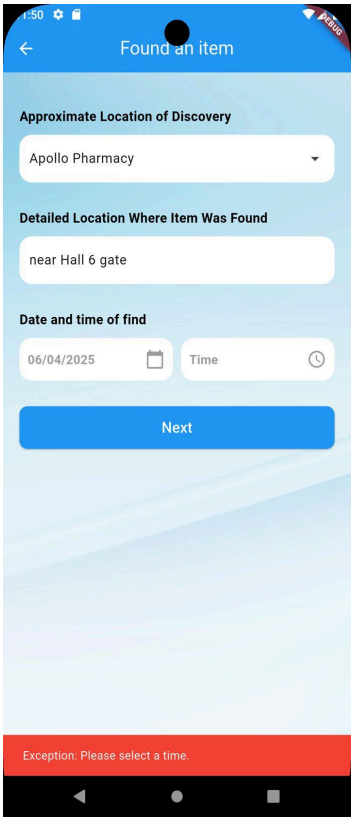
Post a Found Item :

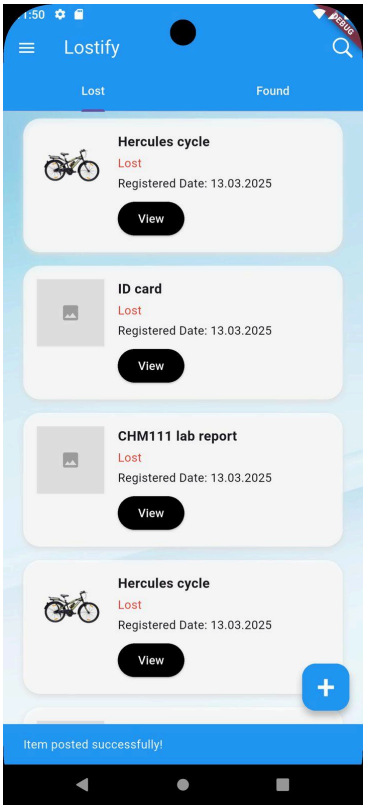












2.3.2.a Delete a Post

API Endpoint: `items/<id>`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is to delete a user's post.

Test Results: Users can delete their post. Admins can delete anyone's post. Satisfaction of permission requirements is verified correctly. Unauthorised requests are handled using error responses.

```python
def test_delete(client: FlaskClient, app: Flask):
    # Test deleting an item without authentication
    response = client.delete('/items/1')

    assert response.status_code == 401

    # Authenticate
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test deleting an item
    response = client.delete(
        '/items/1',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 204

    with app.app_context():
        row = get_db().execute(
            "SELECT * FROM posts WHERE id = ?",
            (1,)
        ).fetchone()
        assert row is None  # Item should be deleted

    # Test deleting a non-existent item
    response = client.delete(
        '/items/9999',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 404

    # Test deleting another user's post
    response = client.delete(
        '/items/6',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 403
```

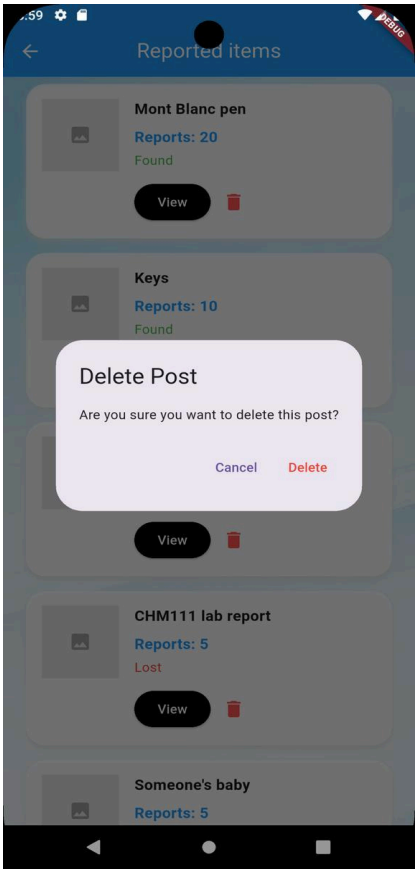
2.3.2.b Delete a Post

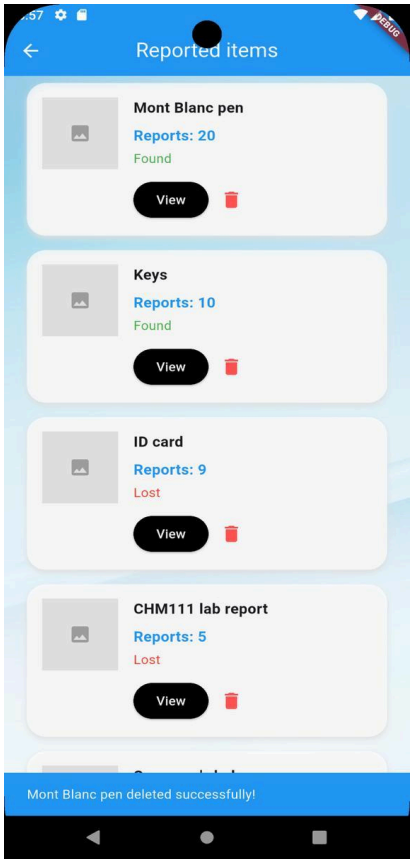
Test Owner: Aman Raj

Test Date: 04/04/2025

Test Description: The delete post feature was tested to ensure that users and admins could successfully remove lost or found item posts as per their administration . The functionality prompted a confirmation dialog before deletion to prevent accidental removal. Upon confirmation, the post was removed from both the user interface and the database.

Test Results: The system successfully deleted lost and found item posts upon user confirmation without any errors.





2.3.3.a Fetch all posts

API Endpoint: `items/all`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is to retrieve all posts.

Test Results: Authenticated users can fetch all posts at once and transfer them to the client as an array. If authentication fails, a suitable error response is relayed.

```python
def test_retrieve_all(client: FlaskClient):
    # Test retrieving all items without authentication
    response = client.get('/items/all')

    assert response.status_code == 401

    # Authenticate
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test retrieving all items
    response = client.get(
        '/items/all',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 200
    assert type(response.json['posts']) is list  # Should return a list of items
```

2.3.4.a Report a post

API Endpoint: `items/<id>/report`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is to report a post.

Test Results: Any authenticated users can put any other user post as inappropriate. Reports are idempotent. If the user is unauthenticated , an appropriate error message is relayed.

```python
def test_report_post(client: FlaskClient, app: Flask):
    # Test reporting a post without authentication
    response = client.put('/items/2/report')

    assert response.status_code == 401

    # Authenticate
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test reporting a post
    response = client.put(
        '/items/2/report',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 204

    with app.app_context():
        assert get_db().execute(
            "SELECT reportCount FROM posts WHERE id = ?",
            (2,)
        ).fetchone()[0] == 1  # Post should be reported

        assert get_db().execute(
            "SELECT * FROM reports WHERE postid = ? AND userid = ?",
            (2, 0)
        ).fetchone() is not None

    # Test idempotence
    response = client.put(
        '/items/2/report',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 204
    with app.app_context():
        assert get_db().execute(
            "SELECT reportCount FROM posts WHERE id = ?",
            (2,)
        ).fetchone()[0] == 1  # Report count should not increase
        assert get_db().execute(
            "SELECT * FROM reports WHERE postid = ? AND userid = ?",
            (2, 0)
        ).fetchone() is not None

    # Test reporting a non-existent item
    response = client.put(
        '/items/9999/report',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 404
```

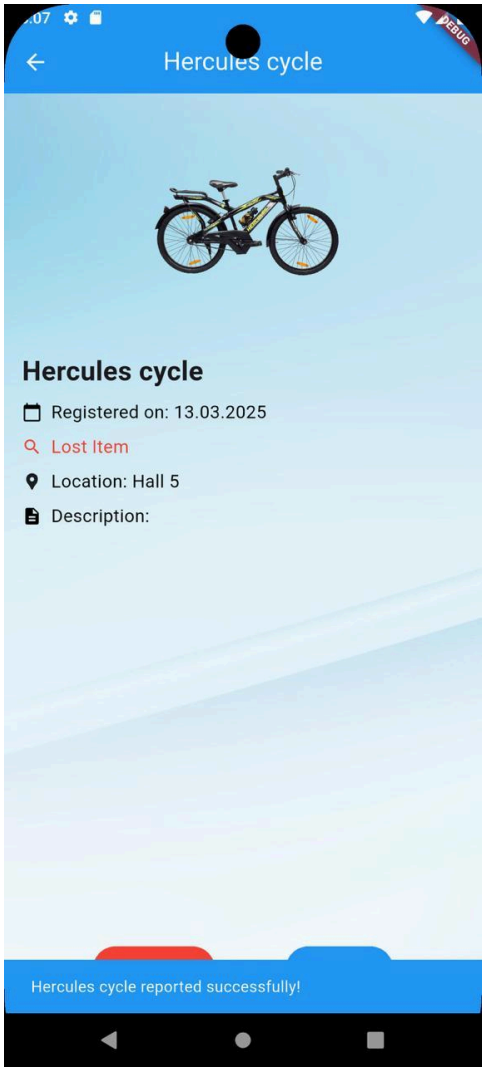
2.3.4.b Report a Post

Test Owner: Aman Raj

Test Date: 04/04/2025

Test Description: The report post feature was tested to ensure that users could flag inappropriate or suspicious lost and found item posts.By that, the report was recorded in the database and flagged the post for admin review. The test verified that the system correctly handled report submissions and updated the post status accordingly.

Test Results: The system successfully recorded reported posts upon user confirmation without any errors.



2.3.5.a Get Report count

API Endpoint: `items/<id>/report`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is to get the report count of a post.

Test Results: An admin can get the report count of any post.

```python
def test_get_report_count(client: FlaskClient, app: Flask):
    # Test getting report count without authentication
    response = client.get('/items/4/report')

    assert response.status_code == 401

    # Authenticate
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test getting report count for a post as a non-admin user
    response = client.get(
        '/items/4/report',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 403

    # Authenticate as admin
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'other',
            'password': 'other'
        }
    ).headers['Set-Cookie']

    # Test getting report count for a post
    response = client.get(
        '/items/4/report',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 200
    assert response.json['reportCount'] == 5  # Report count should be 5

    # Test getting report count for a non-existent item
    response = client.get(
        '/items/9999/report',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 404
```

2.3.6.a Claim a post

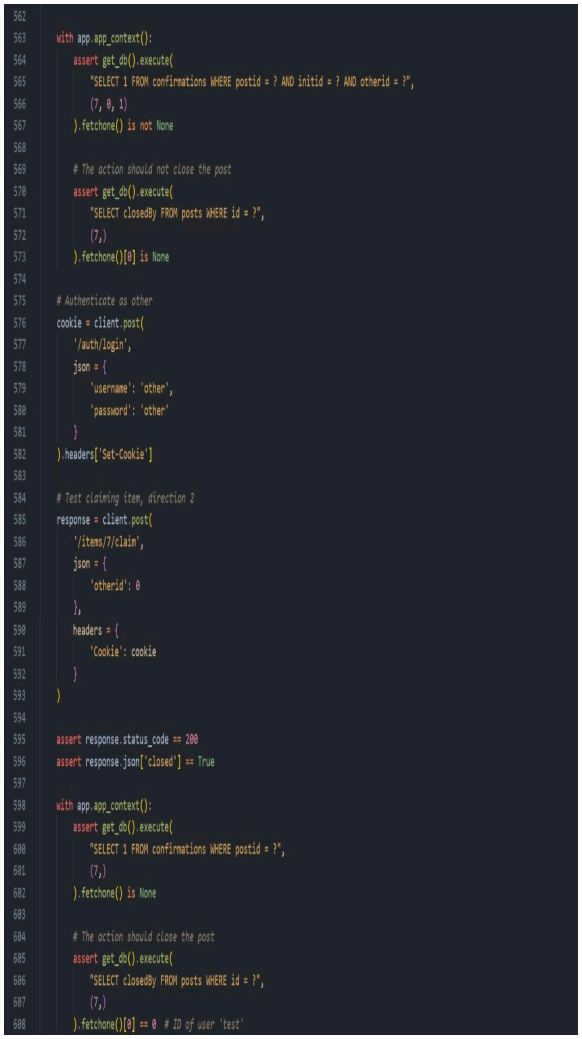
API Endpoint: `items/<id>/claim`

Test Owner: Anirudh Cheriyachanaseri Bijay

Test Description: This test case is to claim a post.

Test Results:

```python
def test_clain(client: FlaskClient, app: Flask):
    # Test claiming an item without authentication
    response = client.post('/items/1/claim')

    assert response.status_code == 401

    # Authenticate
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'test',
            'password': 'test'
        }
    ).headers['Set-Cookie']

    # Test claiming own item, direction 1
    response = client.post(
        '/items/1/claim',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 415

    with app.app_context():
        assert get_db().execute(
            "SELECT 1 FROM confirmations WHERE postid = ? AND initid = ? AND otherid = ?",
            (1, 0, 0)
        ).fetchone() is None

        # The action should not close the post
        assert get_db().execute(
            "SELECT closedBy FROM posts WHERE id = ?",
            (1,)
        ).fetchone()[0] is None

    # Test claiming own item, direction 2
    response = client.post(
        '/items/1/claim',
        json = {
            'otherid': 0
        },
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 200

    with app.app_context():
        assert get_db().execute(
            "SELECT 1 FROM confirmations WHERE postid = ? AND initid = ? AND otherid = ?",
            (1, 0, 0)
        ).fetchone() is not None

        # The action should not close the post
        assert get_db().execute(
            "SELECT closedBy FROM posts WHERE id = ?",
            (1,)
        ).fetchone()[0] is None

    # Test claiming a non-existent item
    response = client.post(
        '/items/9999/claim',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 404

    # Test claiming item, direction 1
    response = client.post(
        '/items/7/claim',
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 200
    assert response.json['closed'] == False
```



```python

    with app.app_context():
        assert get_db().execute(
            "SELECT 1 FROM confirmations WHERE postid = ? AND initid = ? AND otherid = ?",
            (7, 0, 1)
        ).fetchone() is not None

        # The action should not close the post
        assert get_db().execute(
            "SELECT closedBy FROM posts WHERE id = ?",
            (7,)
        ).fetchone()[0] is None

    # Authenticate as other
    cookie = client.post(
        '/auth/login',
        json = {
            'username': 'other',
            'password': 'other'
        }
    ).headers['Set-Cookie']

    # Test claiming item, direction 2
    response = client.post(
        '/items/7/claim',
        json = {
            'otherid': 0
        },
        headers = {
            'Cookie': cookie
        }
    )

    assert response.status_code == 200
    assert response.json['closed'] == True

    with app.app_context():
        assert get_db().execute(
            "SELECT 1 FROM confirmations WHERE postid = ?",
            (7,)
        ).fetchone() is None

        # The action should close the post
        assert get_db().execute(
            "SELECT closedBy FROM posts WHERE id = ?",
            (7,)
        ).fetchone()[0] == 0  # ID of user 'test'
```

3- INTEGRATION TESTING

Integration testing involves testing whether the backend and frontend interact the right way and produce the desired behaviour. Both backend and frontend units have been individually tested by the unit testing team.

We have performed integration testing as follows: we gave the input to the frontend and observed the final changes that occur after the backend processes the request sent by the frontend. Correct final behaviour implies that the interaction between backend and frontend was proper.

Apart from this, integration testing also involves the interaction between different components, redirection of pages, and updation of pages whenever a change is made to the database from anywhere.

3.1 Authentication

Module Details:
This module encompasses testing procedures for user registration through OTP verification, login for already registered users, and the forgot password feature. Additionally, it includes validating the redirection to respective pages upon completion of these actions.

Test Date: 03/04/2025

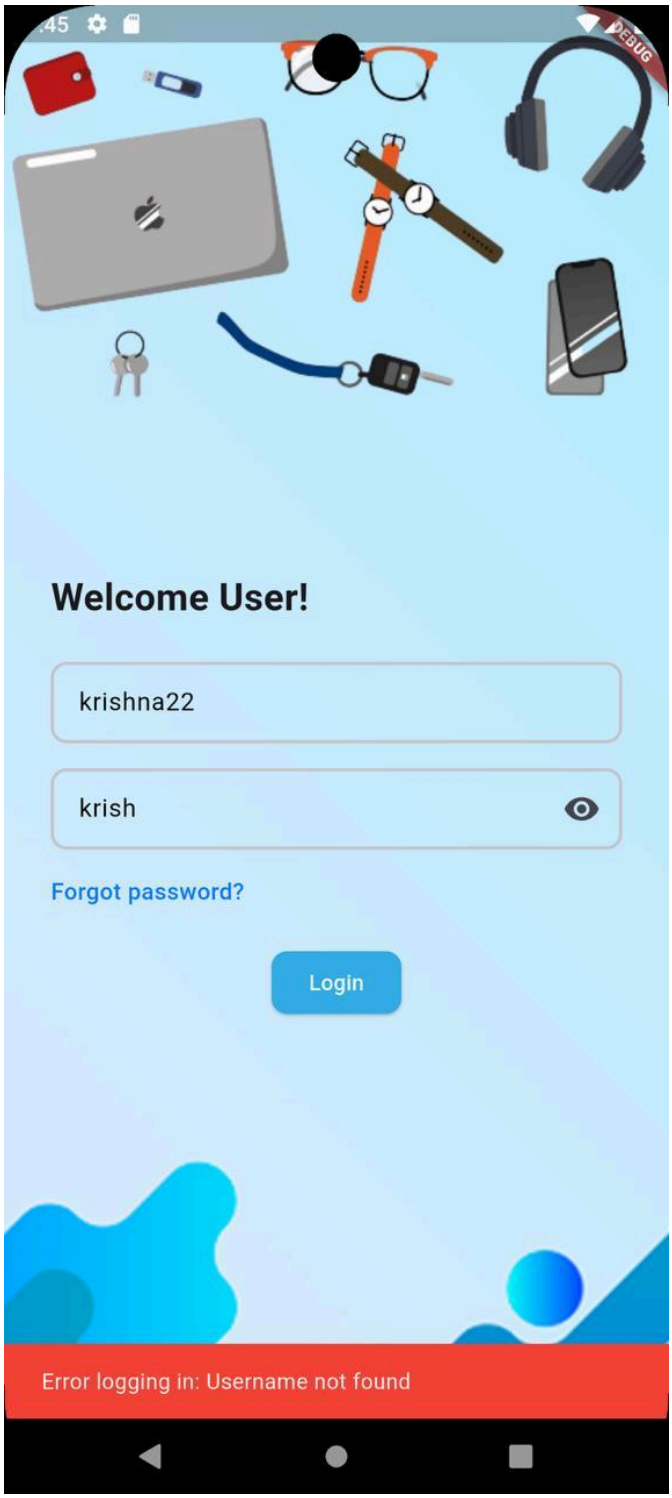
Test Results:
In this test, we validated the successful integration of backend and frontend components of Sign Up, Login, Email Verification, and Forgot Password features. We also validated that the database is updated upon registration of a new user, and the password is stored in encrypted format to maintain user security.
The tested APIs include:

- login API
- signup API
- forgot password API
- generate OTP API

Throughout the testing process, we also scrutinized the redirection of users to the correct pages upon completion of these actions.

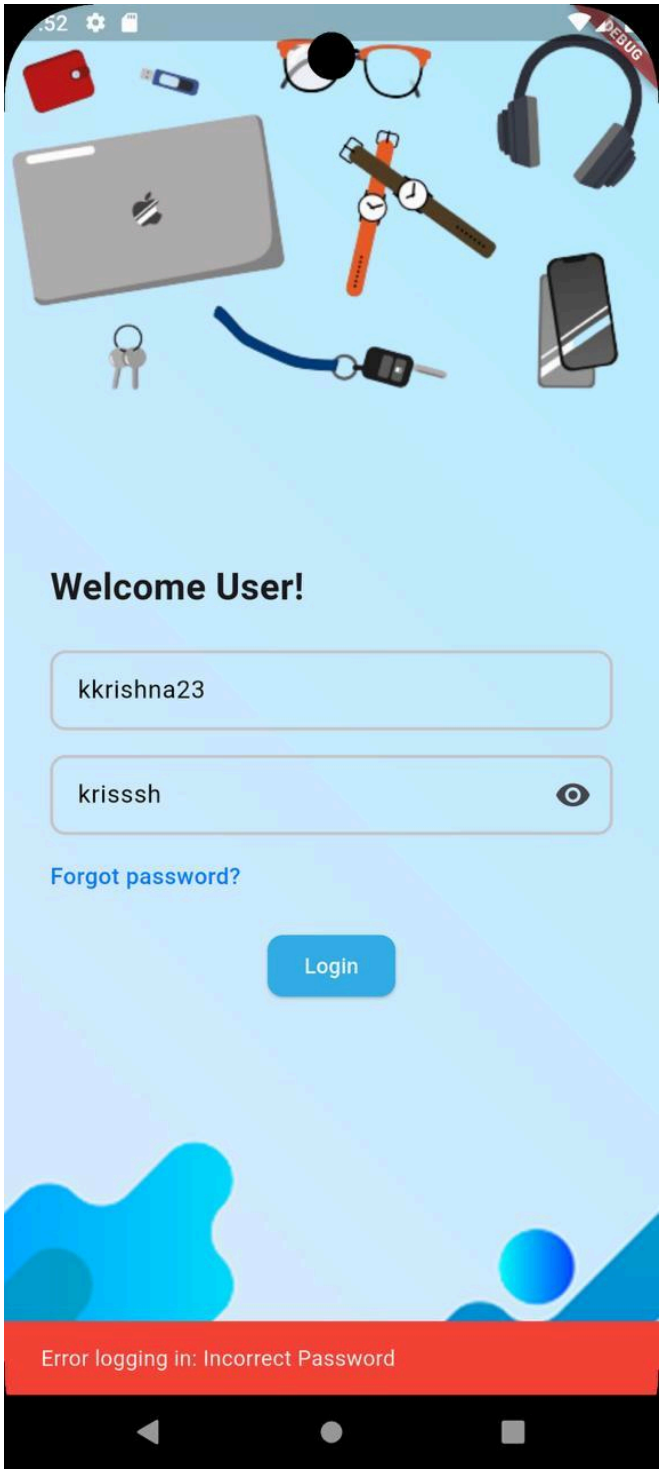
For the following tests, Correct username and password is kkrishna23 and krish

TEST #1
Details: Invalid Username entered while trying to login.



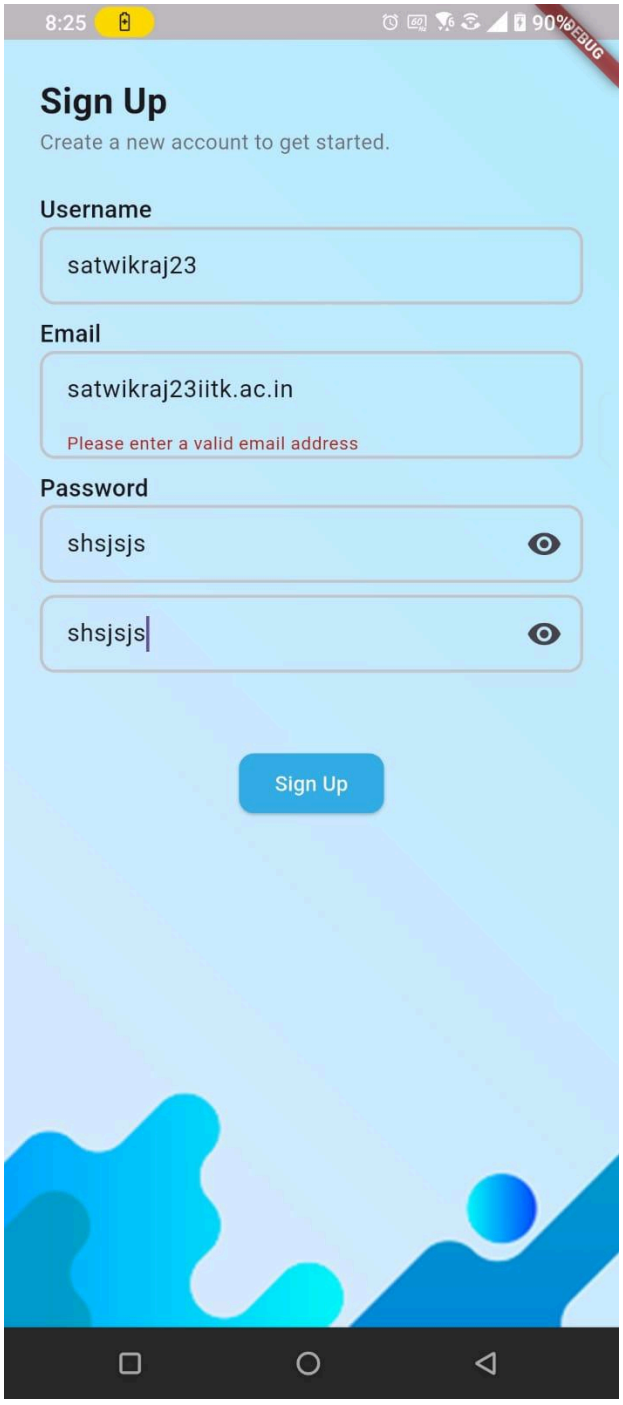
Result: Pop-up displayed that entered username is not found. Login is denied.

TEST #2
Details: Incorrect password entered while trying to login.



Result: Pop-up displayed that entered password is invalid. Login is denied.

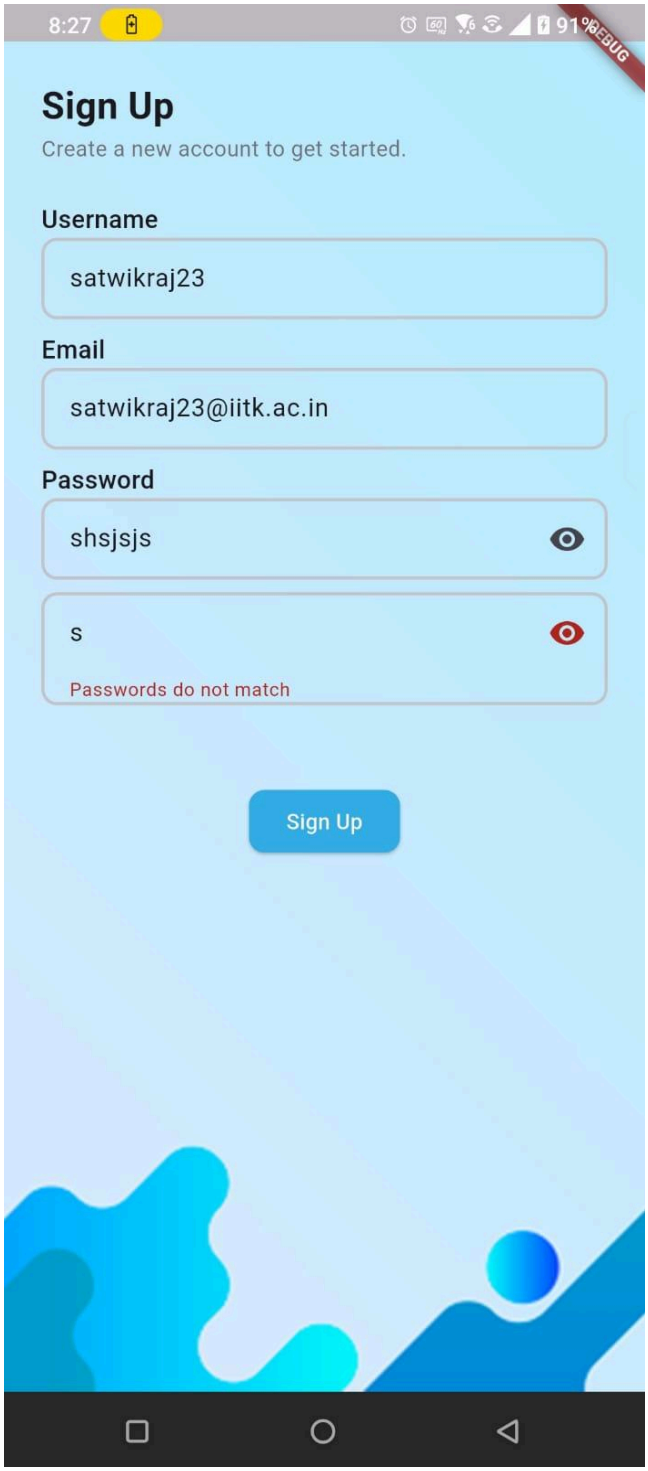
TEST #3
Details: Registering with an invalid email on the sign up page.



Result: Error showing “Please enter a valid email address”. Sign up button won’t work

TEST #4

Details: Confirm Password not matching with the password on the sign up page.



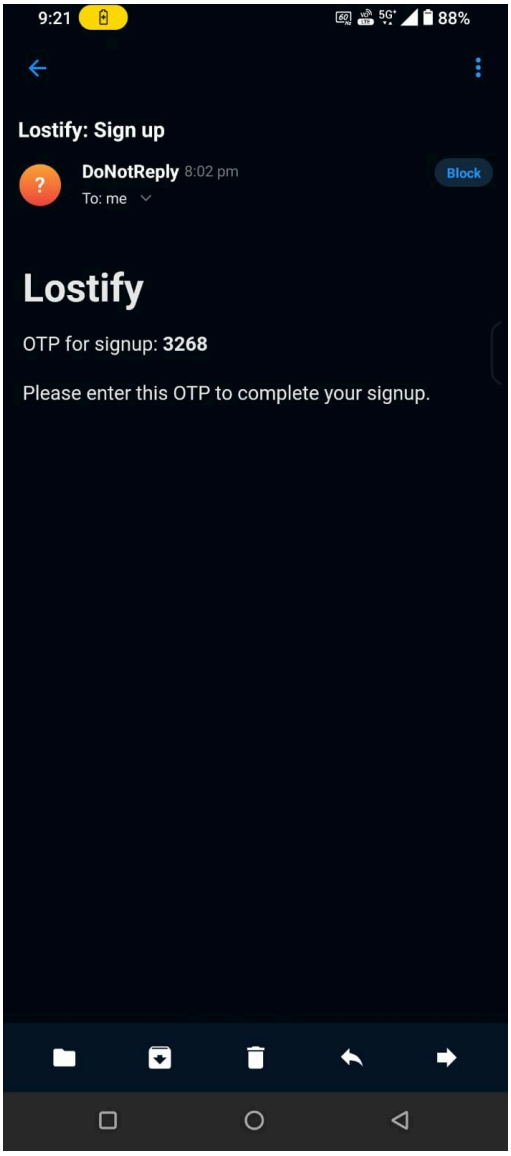
Result: Error showing “Passwords don’t match”. Sign up button won’t work

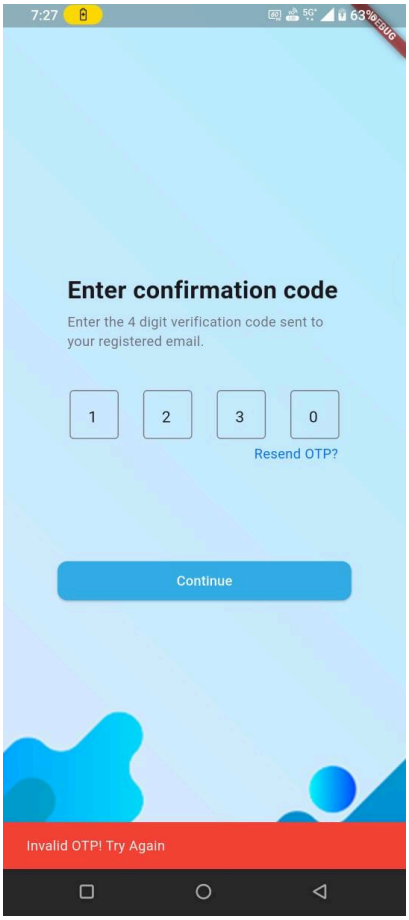
TEST #5
Details: OTP verification and registration of a new user.
Flow:

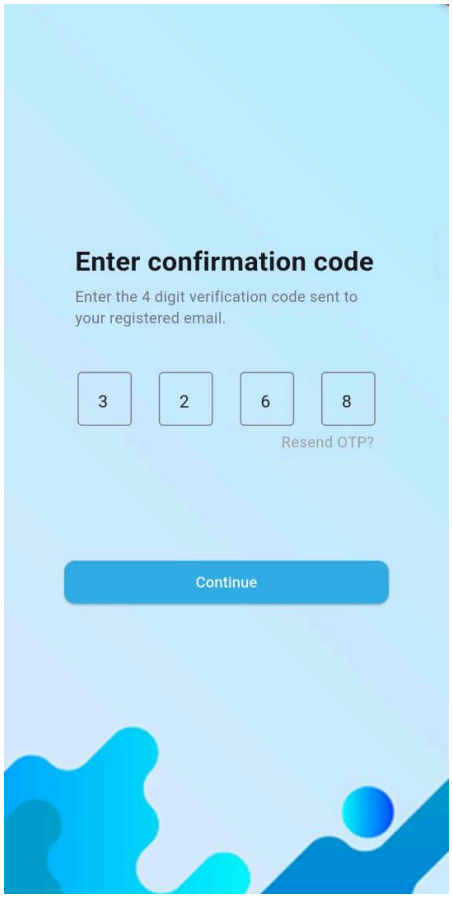
- Email entered in Register Page
- OTP mailed to the email
- User directed to Verify OTP page
- On correct OTP, directed to Fill Details page
- User submits details and is redirected to Login page
- User data is updated in the SQLite database

Mail Sent for OTP Verification:

- When entered OTP does not match the sent OTP:



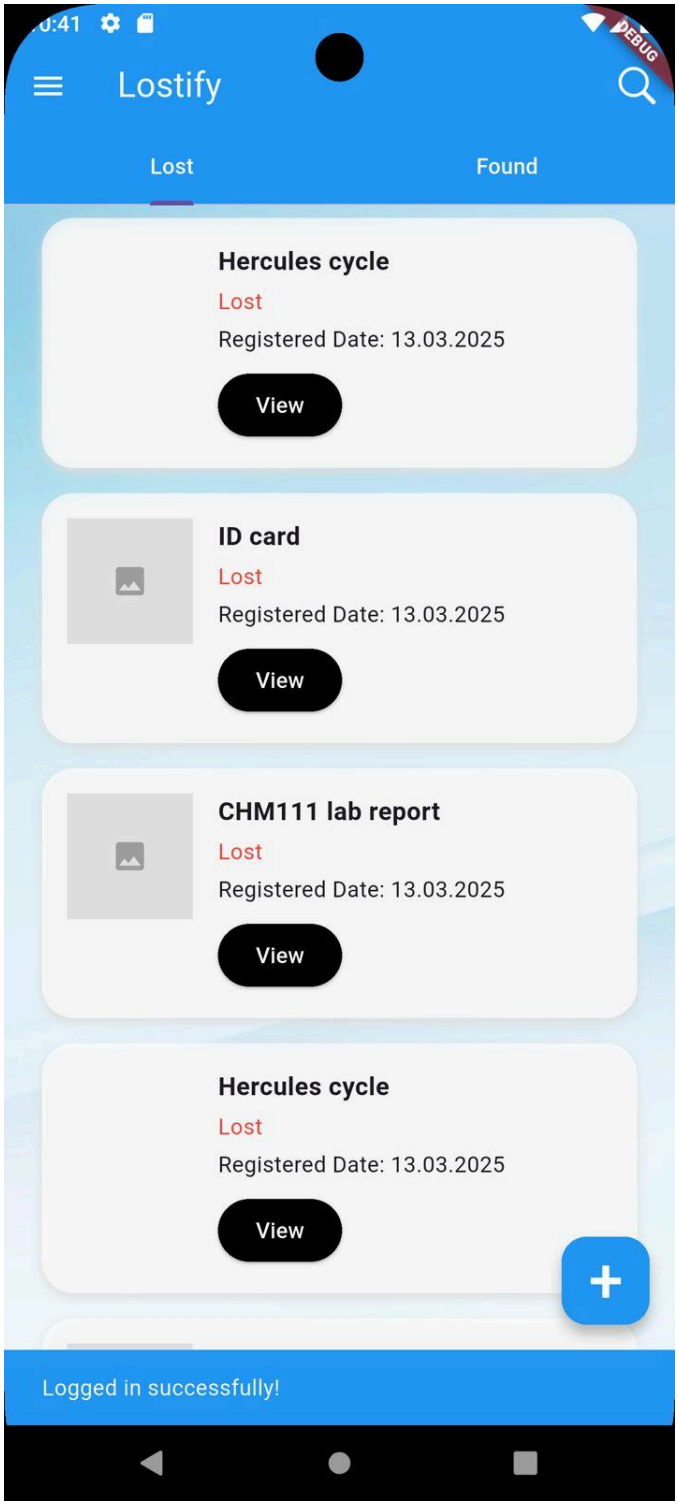




- When resend OTP is pressed; **OTP is resent to the registered email.**

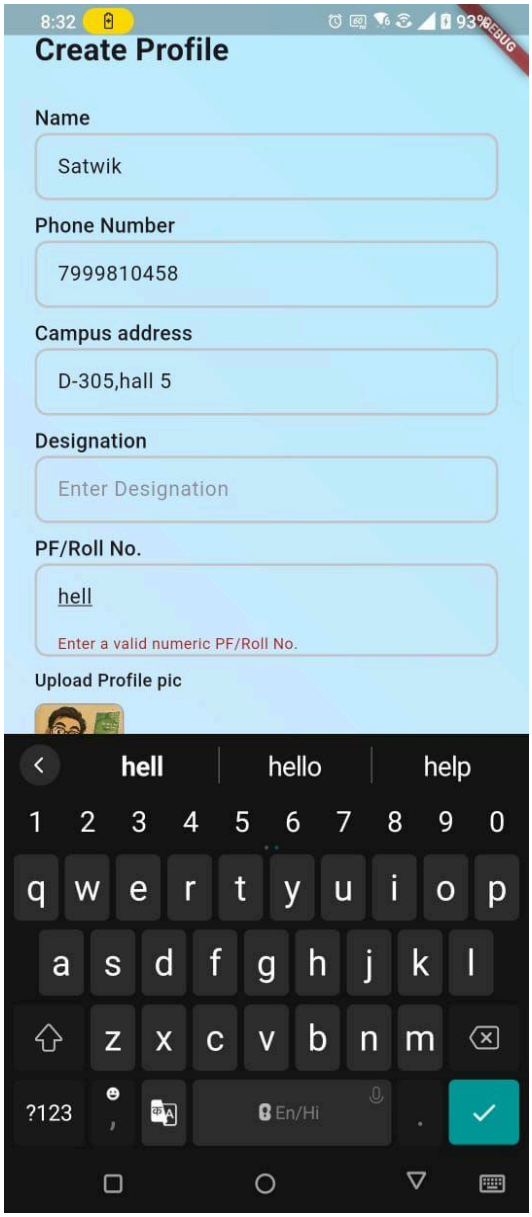
Result:
When a new user enters all details correctly and verifies with the right OTP, they are successfully registered and can proceed to login.

TEST #6
Details: Login with correct credentials.
Frontend Console (Request for Login):
After login, the user is directed to the Dashboard.



TEST #7

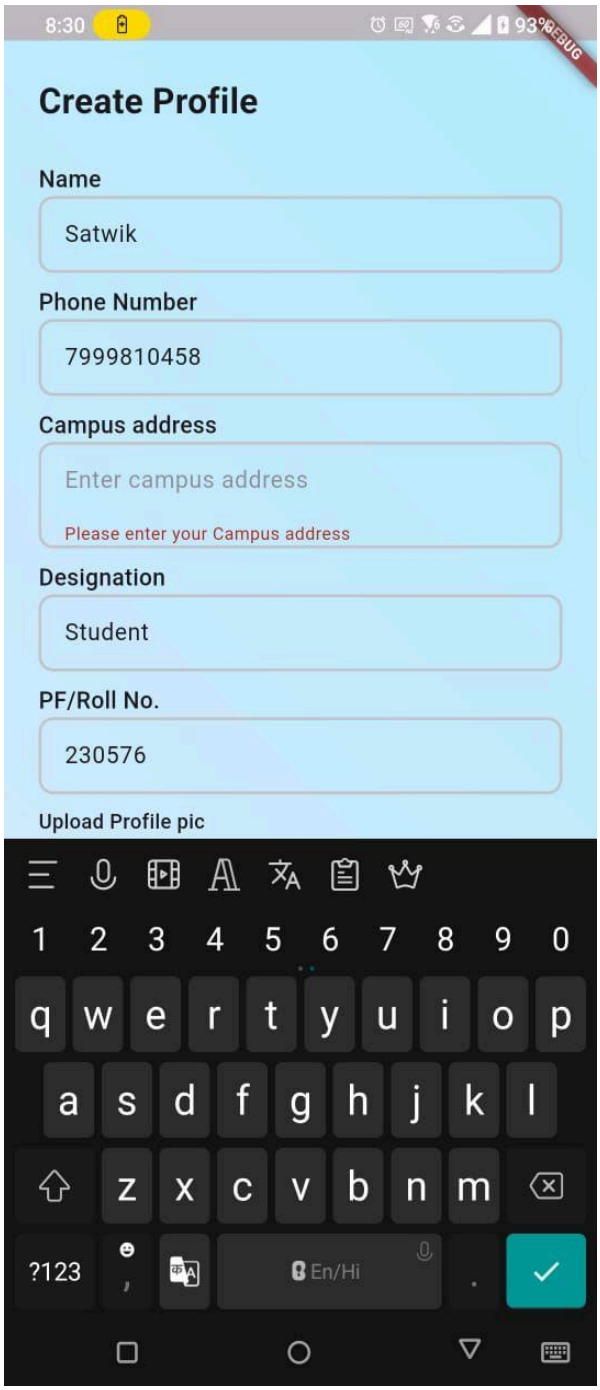
Details: Invalid PF/Roll Number entered while creating a profile.



Result: Error message displayed "Enter a valid numeric PF/Roll No." Profile creation button doesn't work.

TEST #8

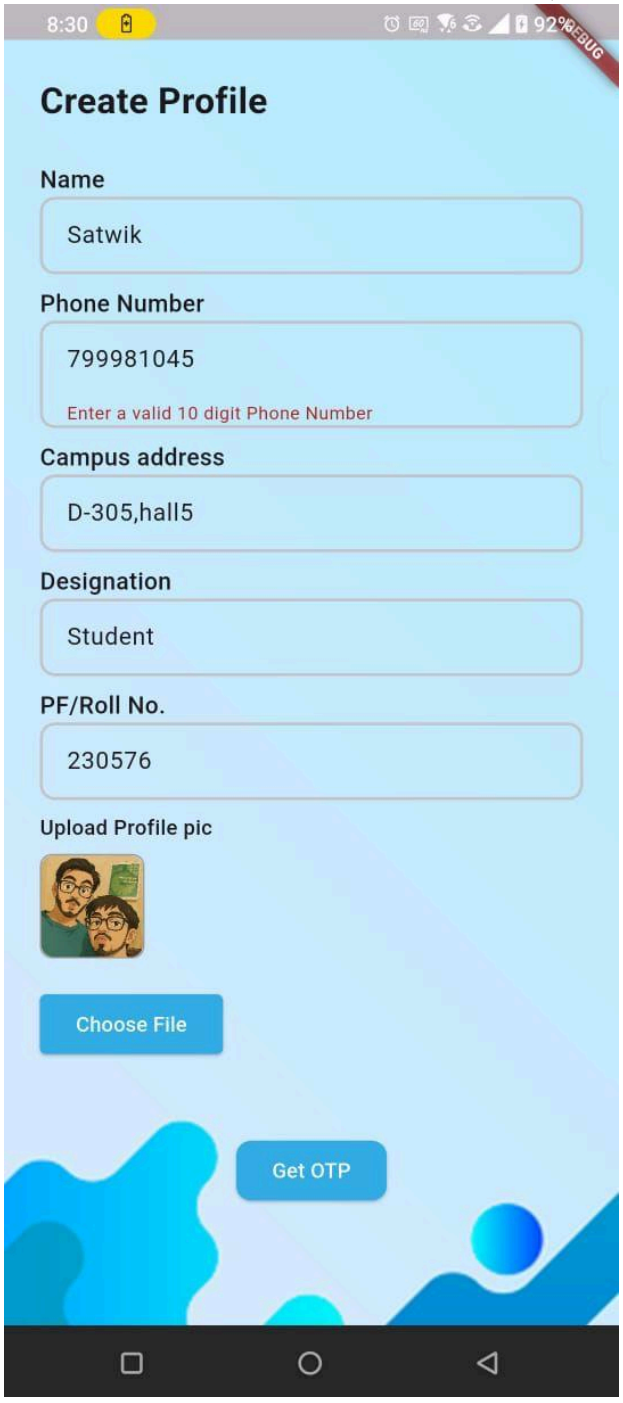
Details: Missing Campus Address while creating a profile.



Result: Error message displayed "Please enter your Campus address." Profile creation button doesn't work.

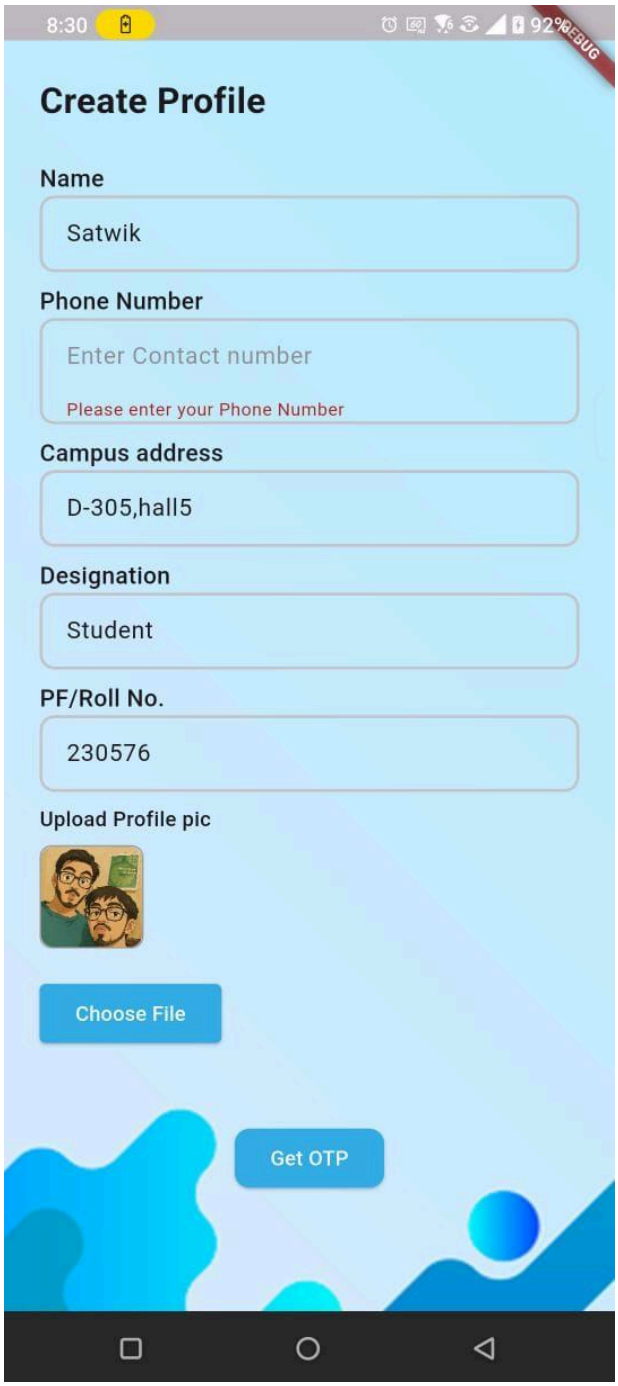
TEST #9

Details: Invalid Phone Number format (less than 10 digits) entered while creating a profile.



Result: Error message displayed "Enter a valid 10 digit Phone Number." Profile creation button doesn't work.

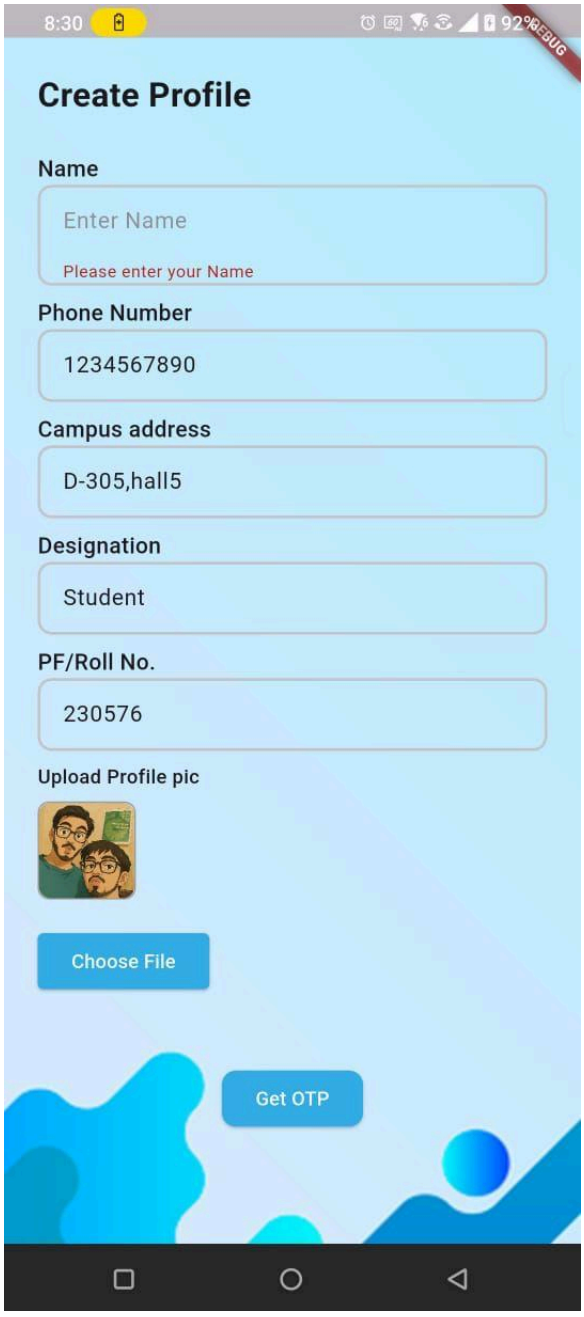
TEST #10
Details: Missing Phone Number while creating a profile.



Result: Error message displayed "Please enter your Phone Number." Profile creation button doesn't work.

TEST #11

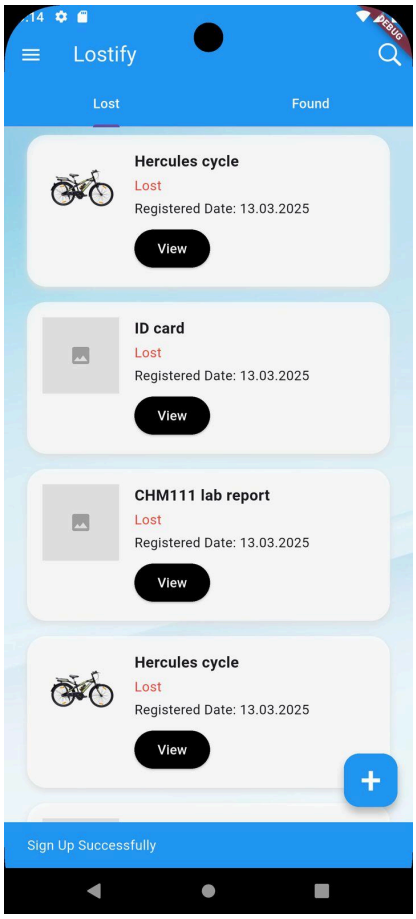
Details: Missing Name field while creating a profile.



Result: Error message displayed "Please enter your Name" as shown in the image. Get OTP button doesn't work until the name is provided.

TEST #12

Details: Successful profile creation with all valid fields.



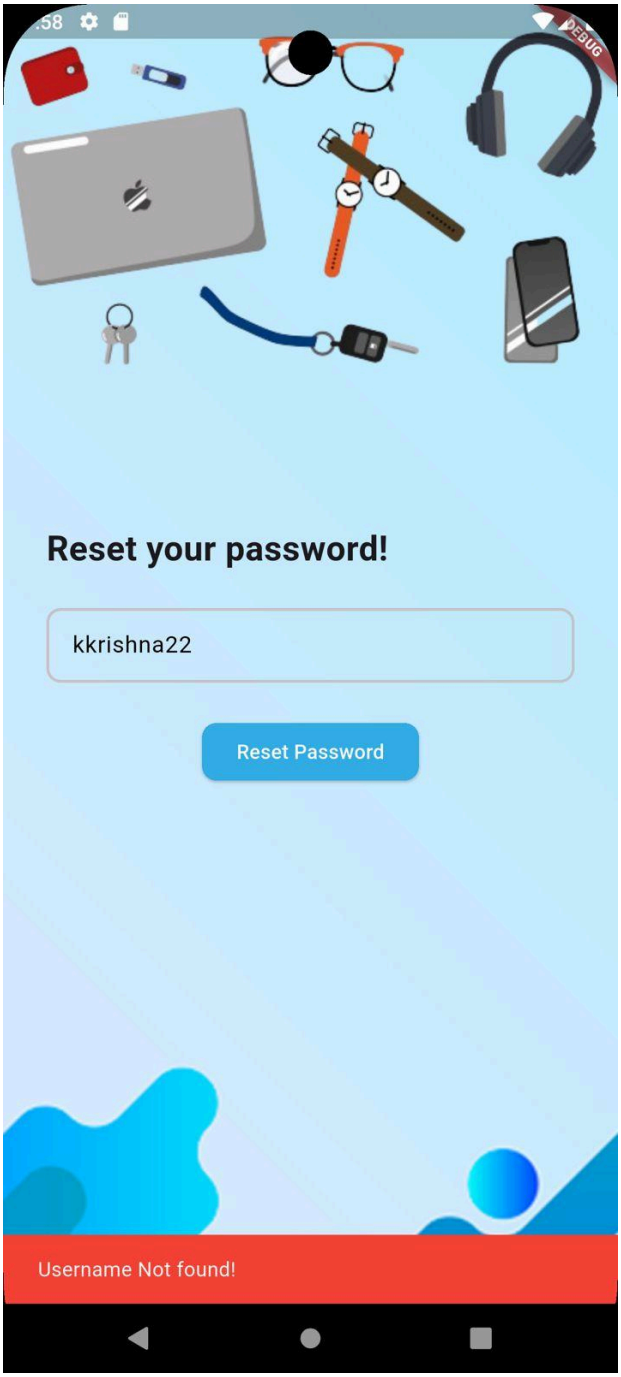
Flow:

- User enters valid Name (Satwik)
- User enters valid Phone Number (1234567890)
- User enters valid Campus Address (D-305,hall5)
- User selects valid Designation (Student)
- User enters valid PF/Roll No. (230576)
- User uploads a Profile Picture
- User clicks on "Get OTP" button
- OTP is sent to user's registered email/phone
- User verifies OTP
- User profile is created successfully

Result: Profile is successfully created and stored in the database. User is redirected to the Dashboard.

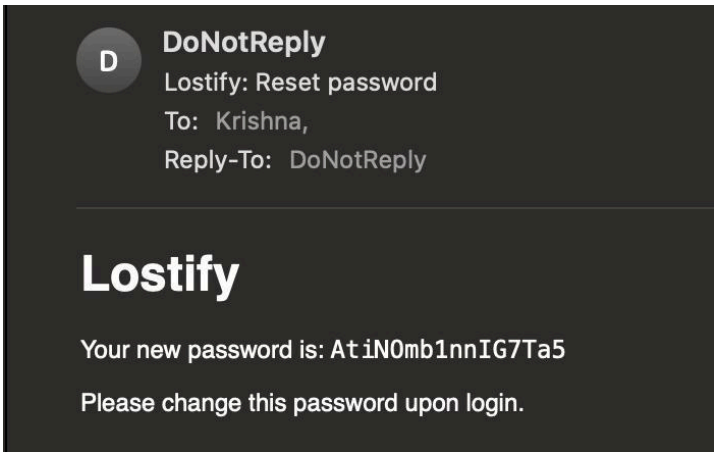
TEST #13
Details: To verify the Forgot Password feature and check if the password is updated correctly.

-
- **Case 1: Unregistered Email**



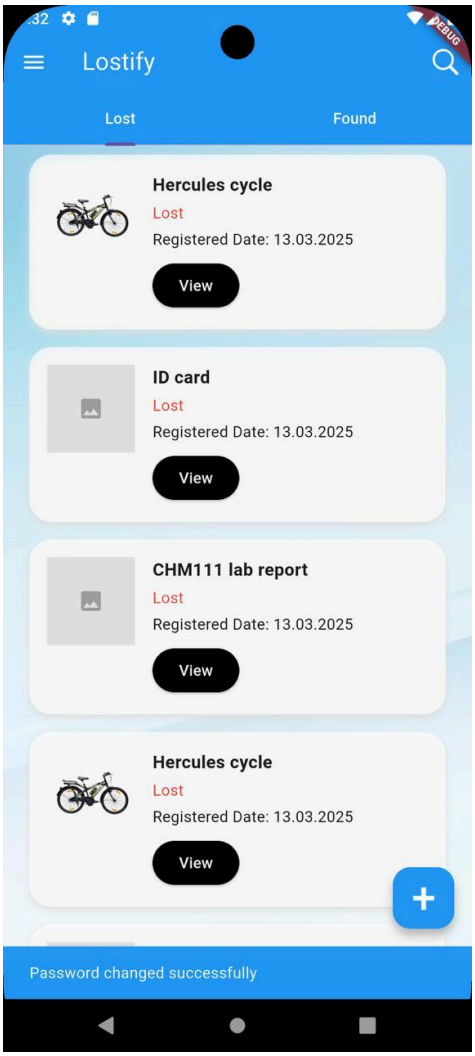
Result: Popup box displays **“Email ID is not registered.”**

- **Case 2: Registered Email**
    - On entering a registered email ID, A new password is sent to the registered email.



    - **Result:** The new password is received, entered by the user while .

- **Change Your Password Page:** Displayed to the user.



  - **Result:** After entering a new password, a popup box appears with the message **“Password changed successfully.”**
  - On clicking **"OK"**, the user is redirected back to the Login page.
  - **Final Result:** The user is now able to log in using the new password.

3.2 Lost Item upload

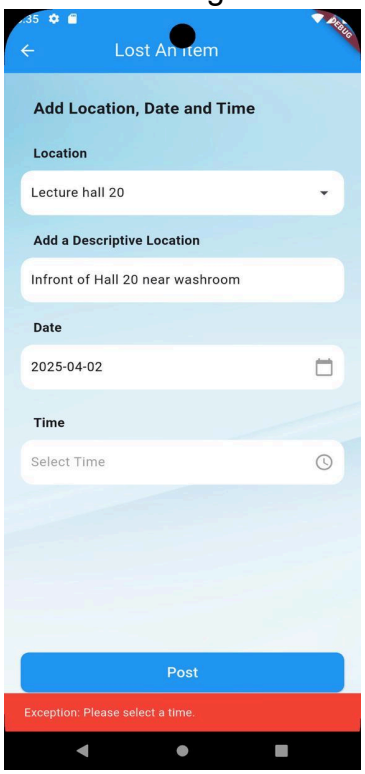
Module Details:
This module encompasses testing procedures for uploading a lost item by entering details of the item, including Title, Description, Image, Location, Date and Time. Additionally, it includes validating the redirection to respective pages upon completion of these actions.

Test Date: 04/04/2025

Test Results:
In this test, we validated the successful integration of backend and frontend components of Sign Up, Login, Email Verification, and Forgot Password features. We also validated that the database is updated upon registration of a new user, and the password is stored in encrypted format to maintain user security.
The items API is tested here.

Throughout the testing process, we also scrutinized the redirection of users to the correct pages upon completion of these actions.

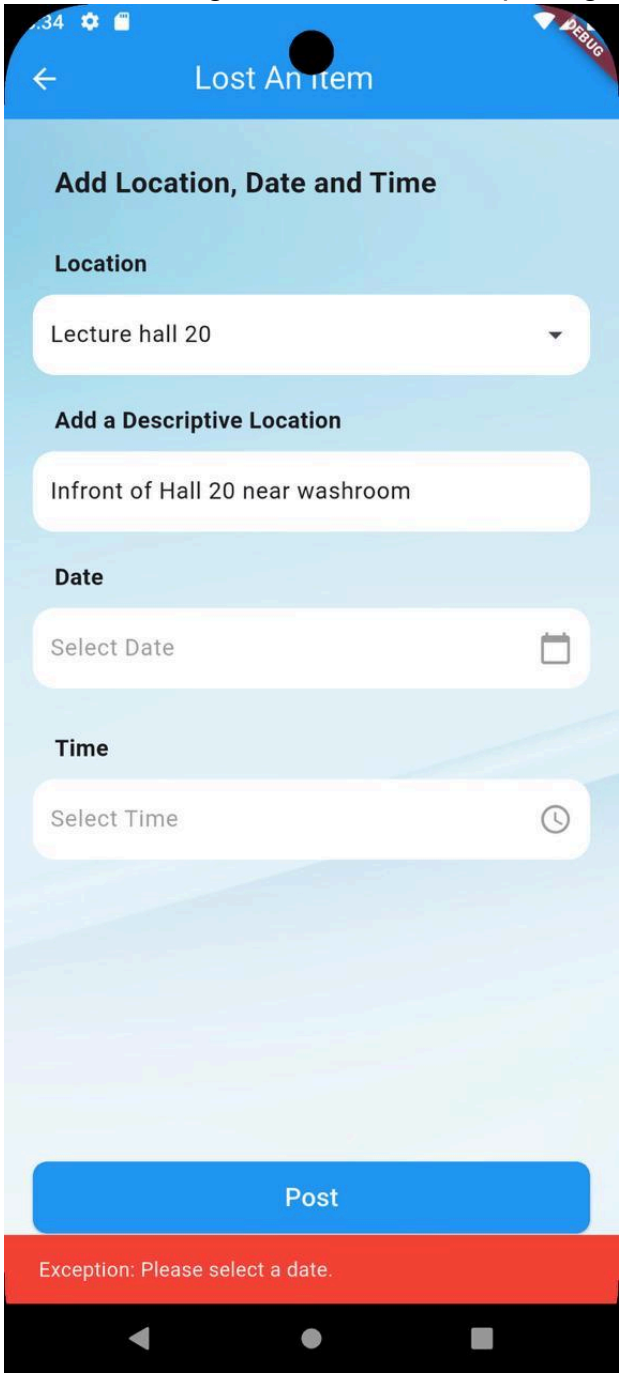
TEST #1
Details: Missing Time Field when reporting a lost item.



Result: Error message displayed **"Exception: Please select a time."** Post button doesn't complete the upload process until a time is selected.

TEST #2:

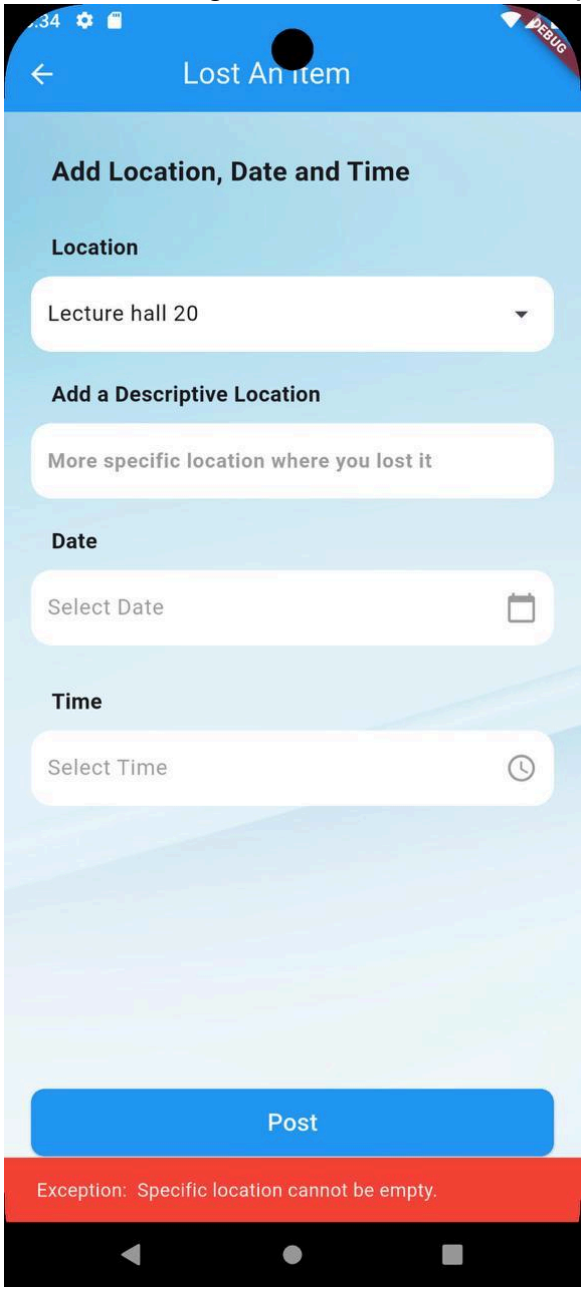
Details: Missing Date Field when reporting a lost item.



Result: Error message displayed **"Exception: Please select a date."** Post button doesn't complete the upload process until a date is selected.

TEST #3:

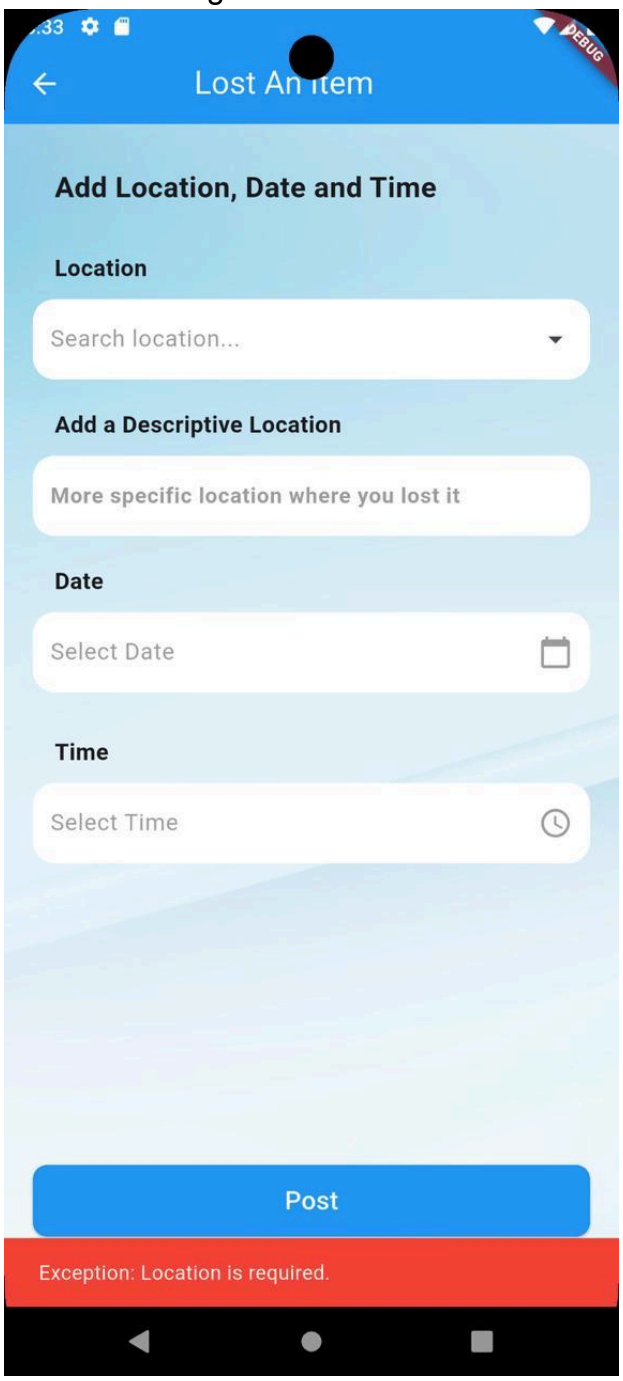
Details: Missing Detailed Location description when reporting a lost item.



Result: Error message displayed **"Exception: Specific location cannot be empty."** Post button doesn't complete the upload process until a detailed location is provided.

TEST #4:

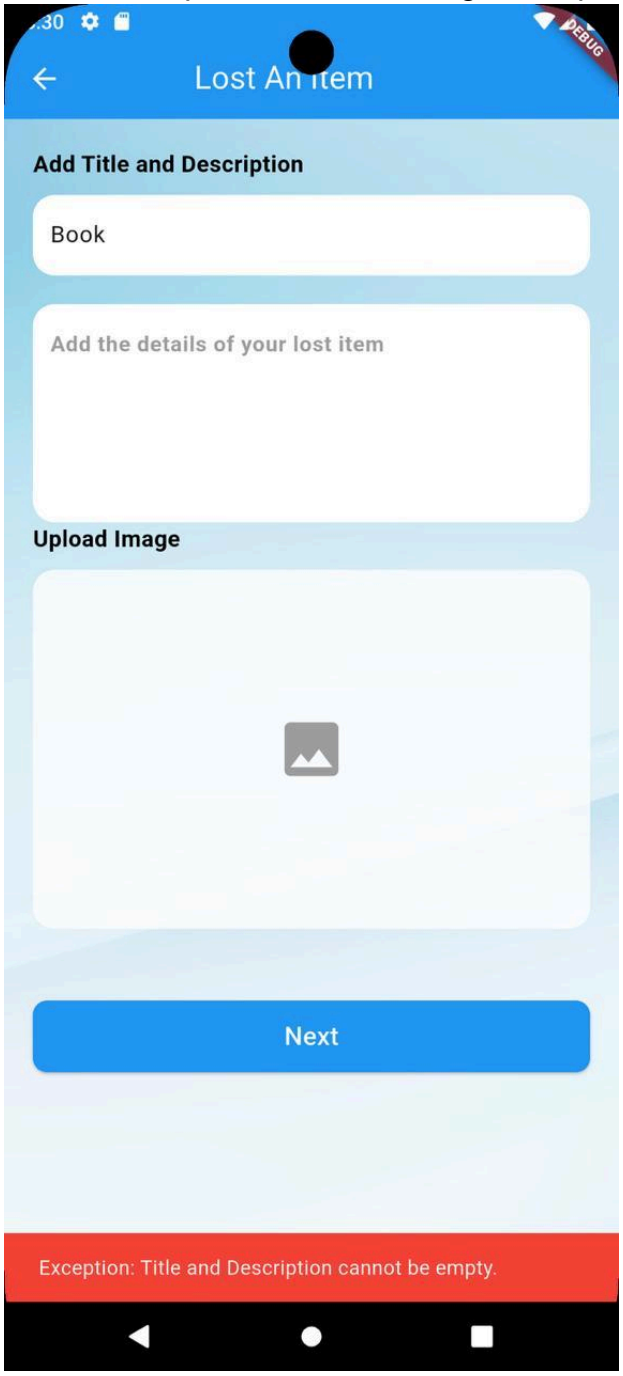
Details: Missing General Location selection when reporting a lost item.



Result: Error message displayed **"Exception: Location is required."** Post button doesn't complete the upload process until a location is selected from the dropdown.

TEST #5:

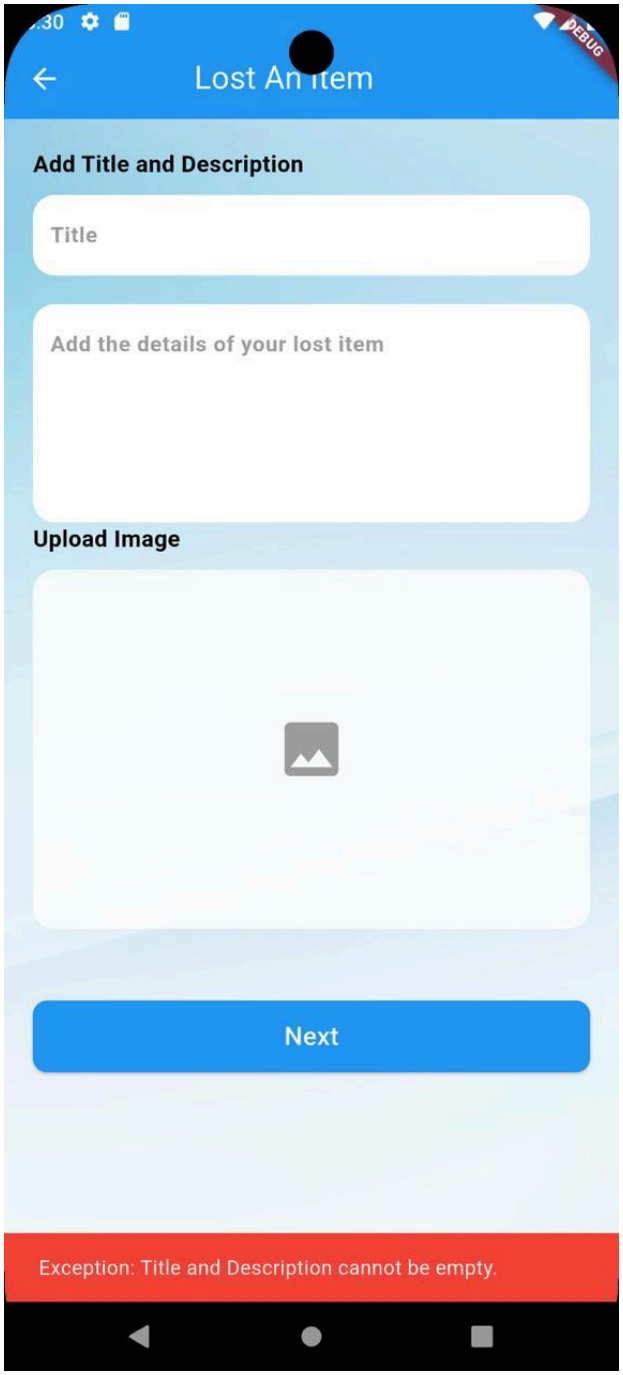
Details: Title provided but Missing Description Field when reporting a lost item.



Result: Error message displayed **"Exception: Title and Description cannot be empty."**
Next button doesn't proceed to the following screen until both fields are completed.

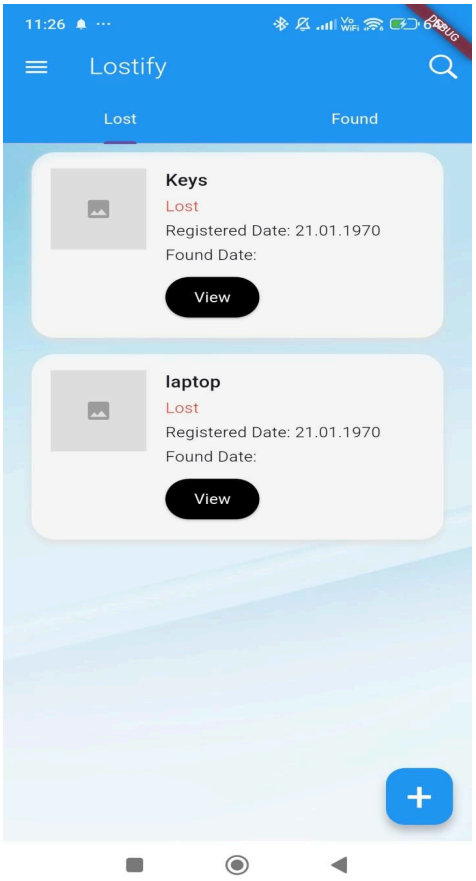
TEST #6:

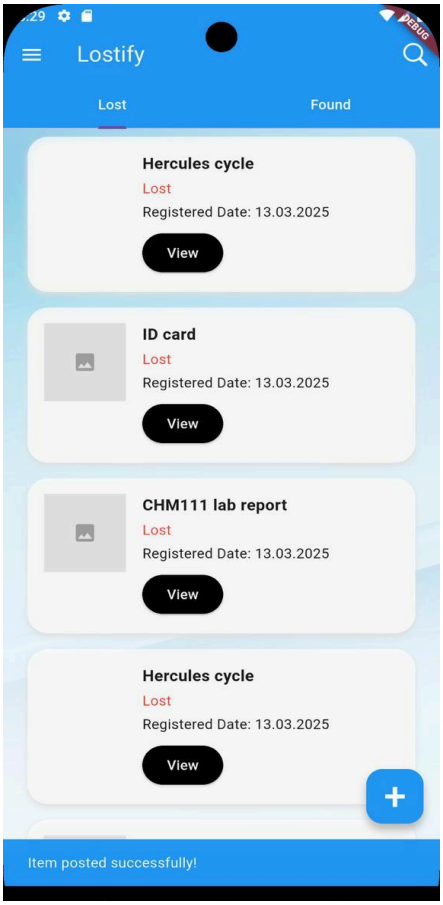
Details: Empty Title and Description Fields when reporting a lost item.



Result: Error message displayed "**Exception: Title and Description cannot be empty.**" Next button doesn't proceed to the following screen until both fields are completed.

TEST #7:
Details: Successful lost item upload with all valid fields.





Flow:

- User navigates to "**Lost An Item**" screen
- User enters valid Title and Description
- User uploads Image of the lost item
- User clicks "**Next**" button to proceed
- User selects general location from dropdown
- User enters detailed location
- User selects date and time of loss
- User submits the form
- System displays "**Item posted successfully!**" notification
- Item appears in the Lost items list in the dashboard with correct registered date

Result: Lost item is successfully uploaded and stored in the database. User is redirected to the dashboard with a success message "Item posted successfully!"

3.3 Found Item upload

Module Details:

This module encompasses testing procedures for uploading a found item by entering details of the item, including Title, Description, Image, Location, Date and Time. Additionally, it includes validating the redirection to respective pages upon completion of these actions.

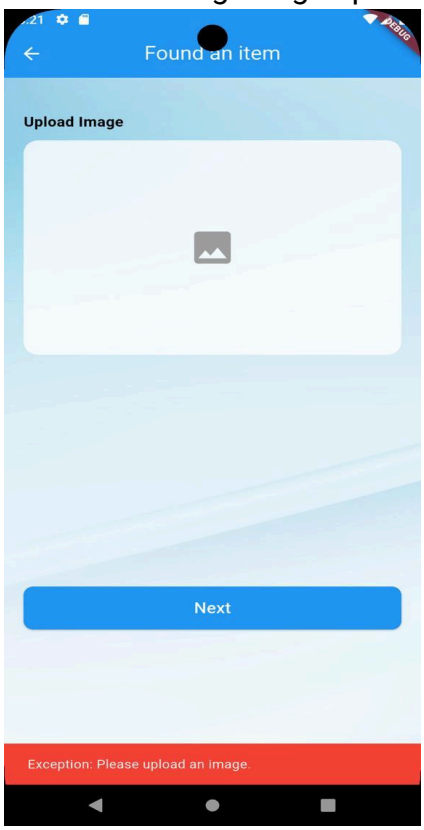
Test Date: 04/04/2025

Test Results:

In this test, we validated the successful integration of backend and frontend components of the Found Item Upload feature. We also validated that the database is updated upon uploading a found item.

The items API is tested here.

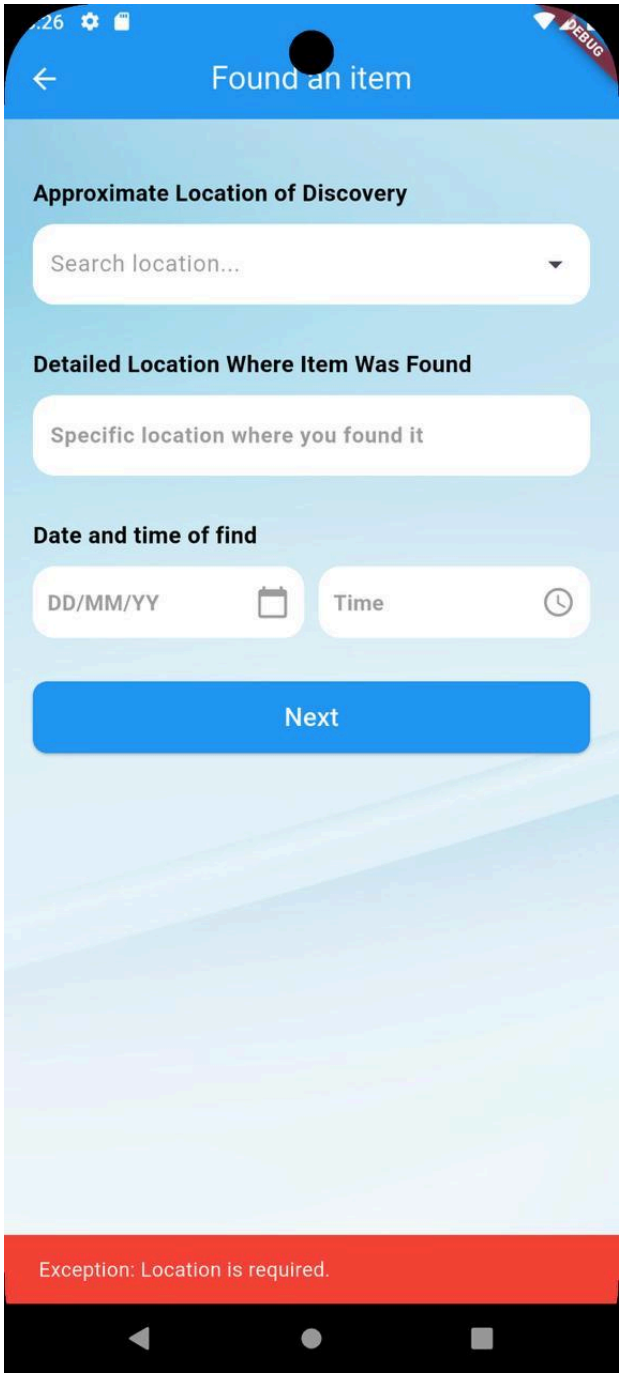
TEST #1
Details: Missing Image upload while reporting a found item.



Result: Error message displayed "**Exception: Please upload an image.**" Next button doesn't proceed to the following screen until an image is uploaded.

TEST #2:

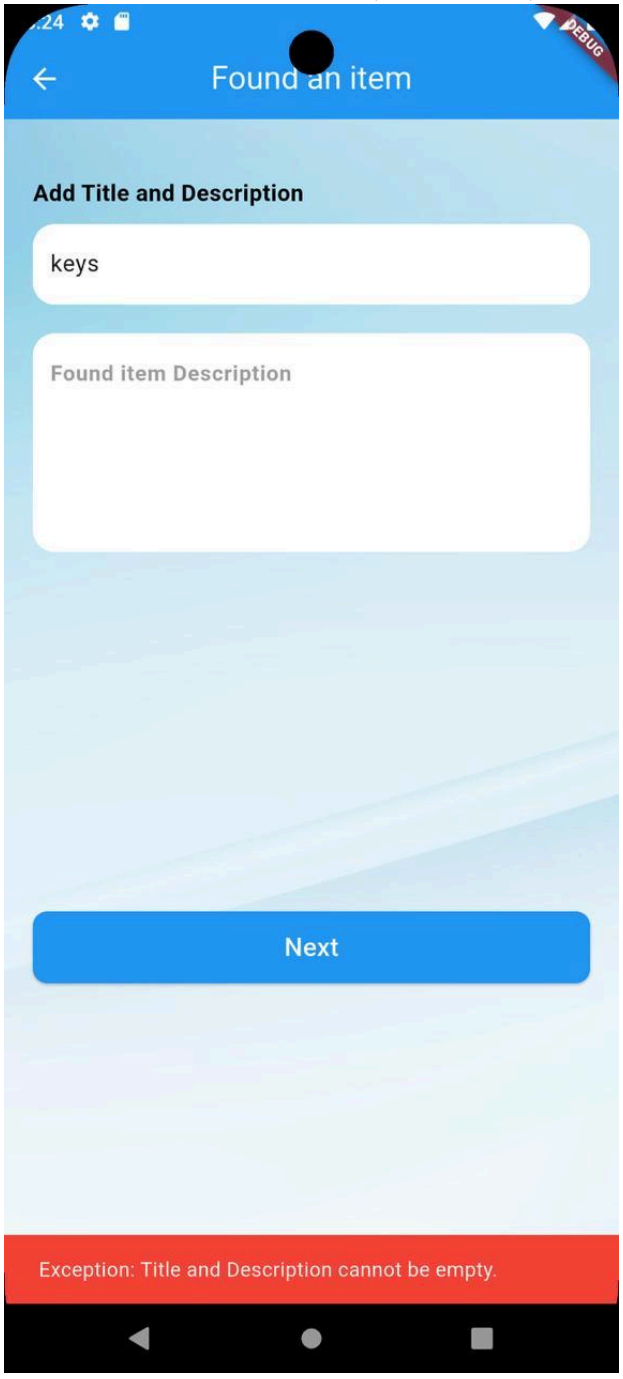
Details: Missing Location information while reporting a found item.



Result: Error message displayed "**Exception: Location is required.**" Next button doesn't proceed to the following screen until location is provided.

TEST #3

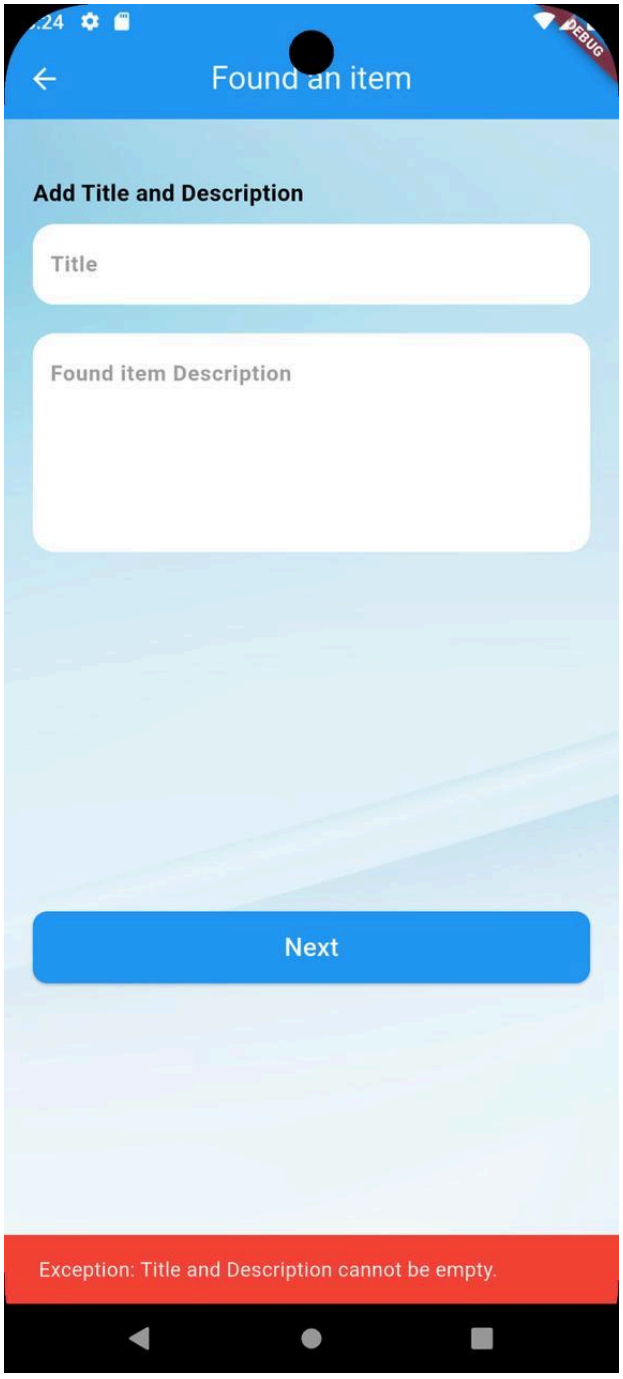
Details: Partial Title entry (only "keys") but missing Description for a found item.



Result: Error message displayed "**Exception: Title and Description cannot be empty.**" Next button doesn't proceed to the following screen until both fields are completed.

TEST #4

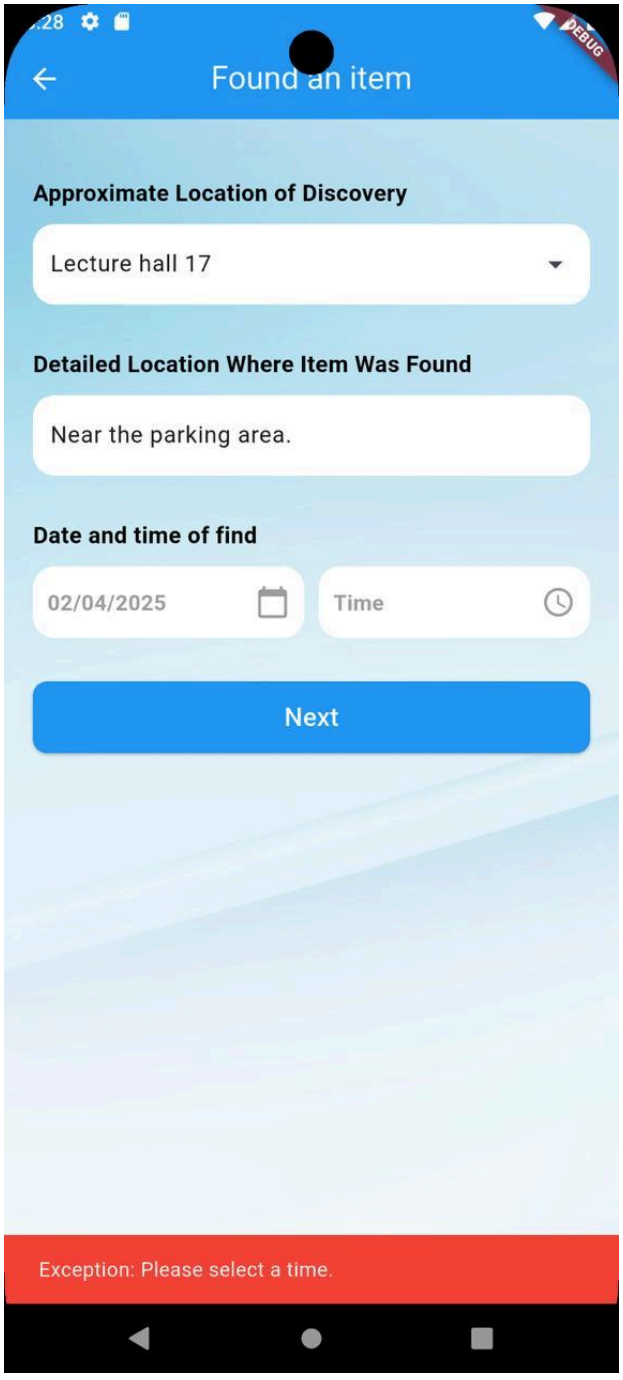
Details: Empty Title and Description fields while reporting a found item.



Result: Error message displayed "**Exception: Title and Description cannot be empty.**" Next button doesn't proceed to the following screen until both fields are completed.

TEST #5

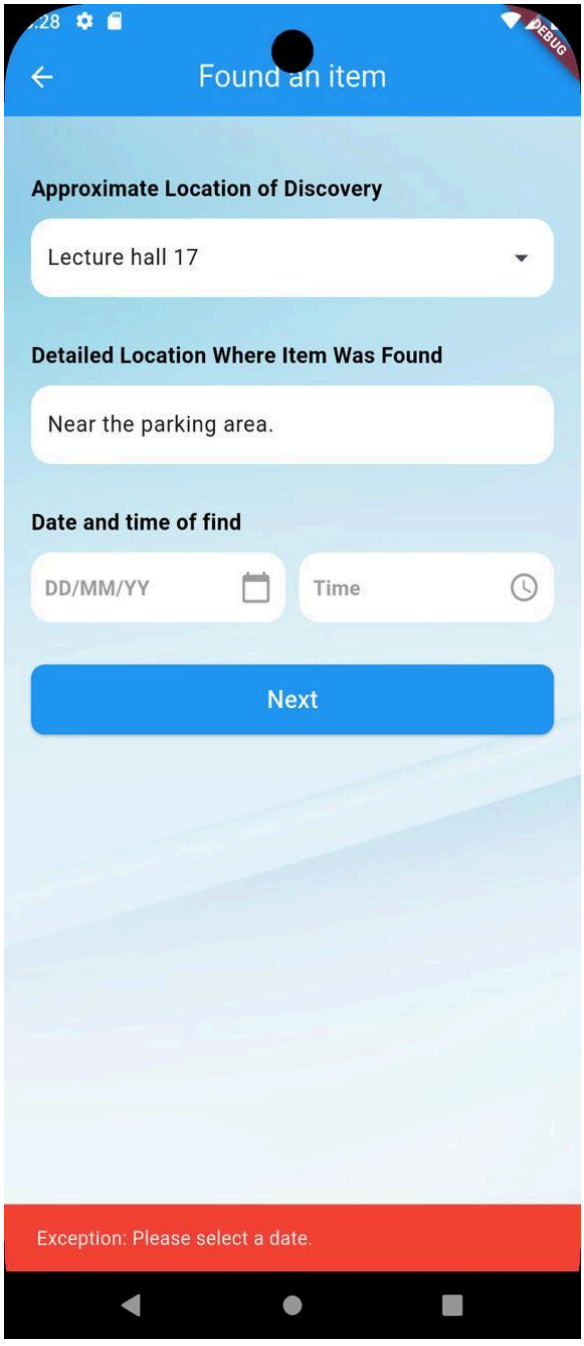
Details: Missing Time Field when reporting a found item.



Result: Error message displayed **"Exception: Please select a time."** Next button doesn't proceed until a time is selected.

TEST #6

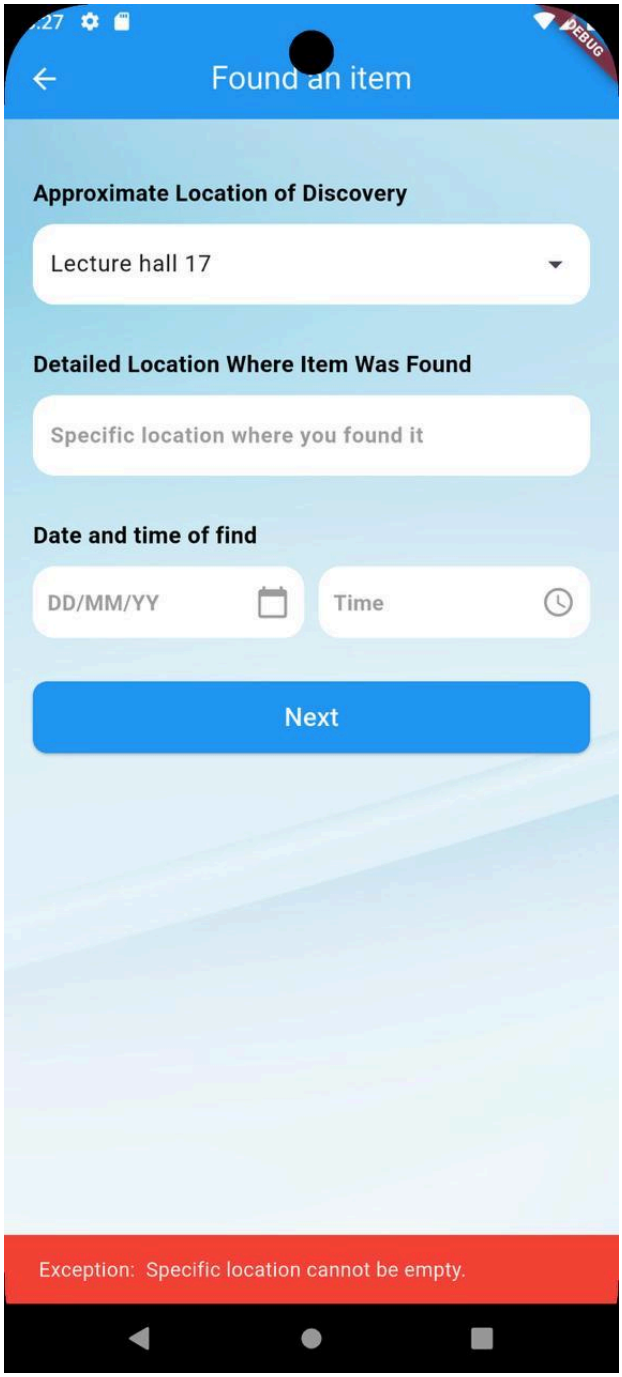
Details: Missing Date Field when reporting a found item.



Result: Error message displayed **"Exception: Please select a date."** Next button doesn't proceed until a date is selected.

TEST #7

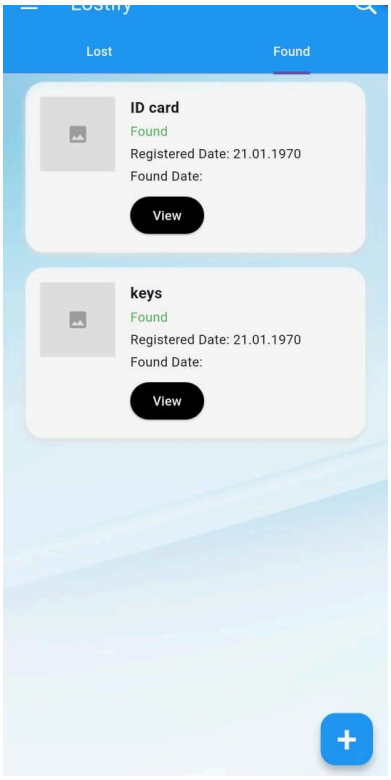
Details: Missing Detailed Location description when reporting a found item.

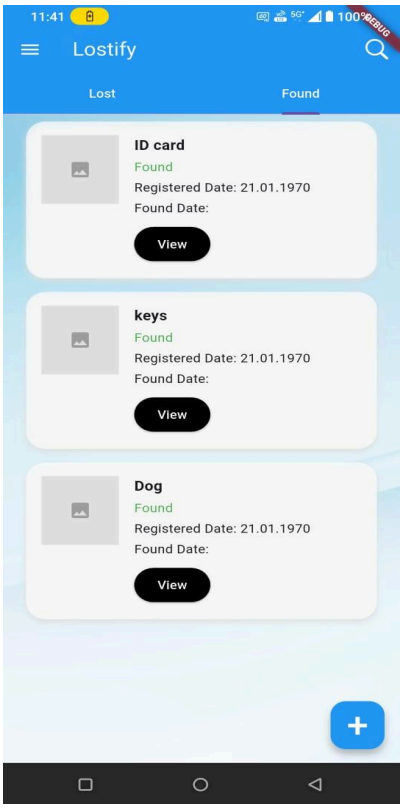


Result: Error message displayed **"Exception: Specific location cannot be empty."** Next button doesn't proceed until a detailed location is provided.

TEST #8

Details: Successful found item upload with all valid fields.





Flow:

- User navigates to **"Found an item"** screen
- User selects approximate location from dropdown (e.g., "Lecture hall 17")
- User enters detailed location (e.g., "Near the parking area")
- User selects date (e.g., "02/04/2025") and time of find
- User clicks **"Next"** button to proceed
- User enters item title and description
- User uploads image of the found item
- User submits the form
- System displays **"Item posted successfully!"** notification
- Item appears in the Found items list in the dashboard

Result: Found item is successfully uploaded and stored in the database. User is redirected to the dashboard with a success message **"Item posted successfully!"**

3.4 Edit Profile

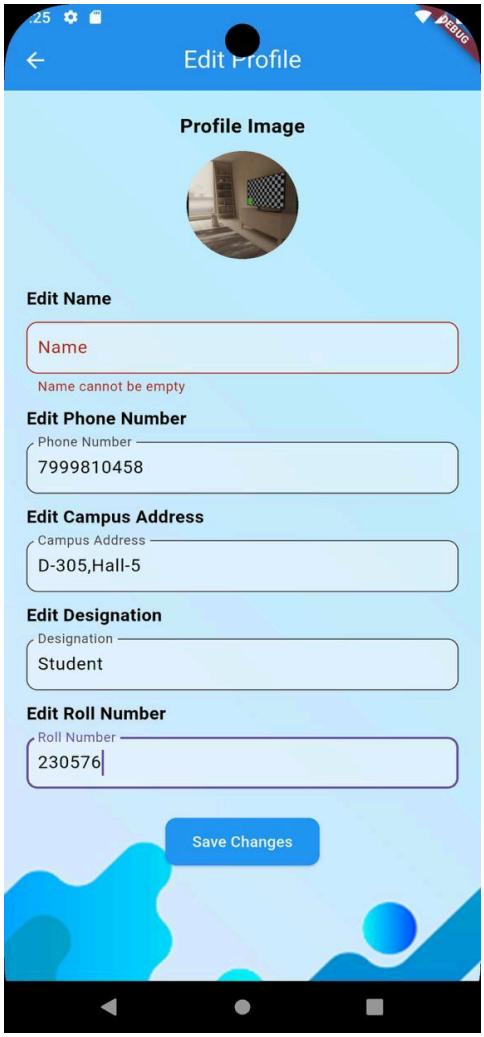
Module Details:

This module encompasses testing procedures for editing a user's profile information, including Name, Phone Number, Campus Address, Designation, PF/Roll Number, and Profile Picture. Additionally, it includes validating the redirection to respective pages upon completion of these actions.

Test Owner: Group #6

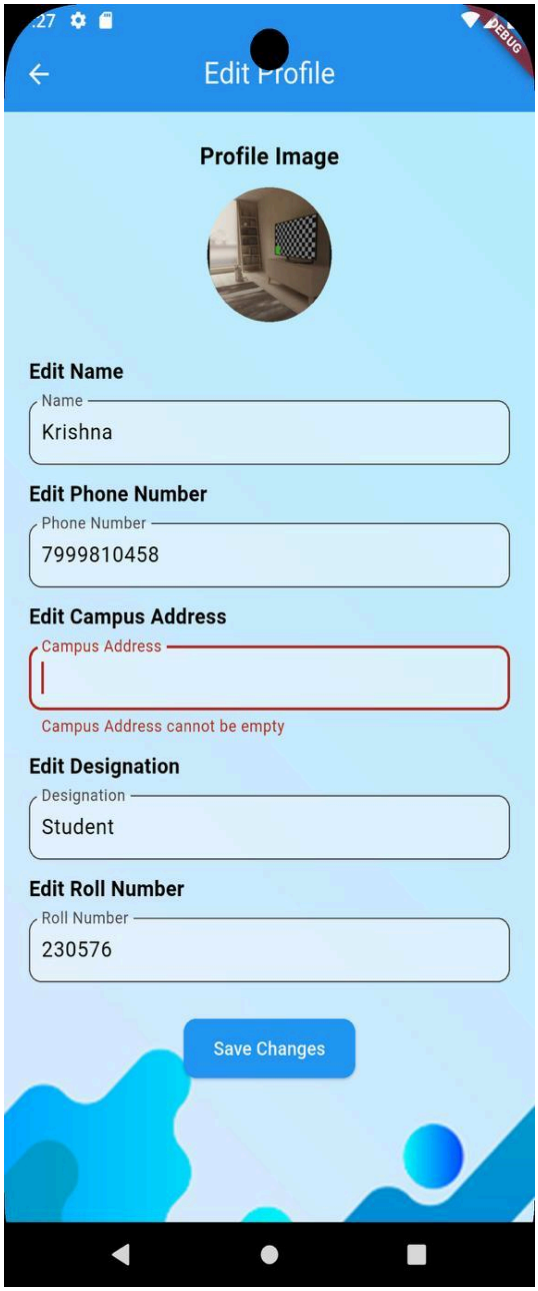
Test Date: 06/04/2025

TEST #1:
Details: Empty Name field when editing profile.



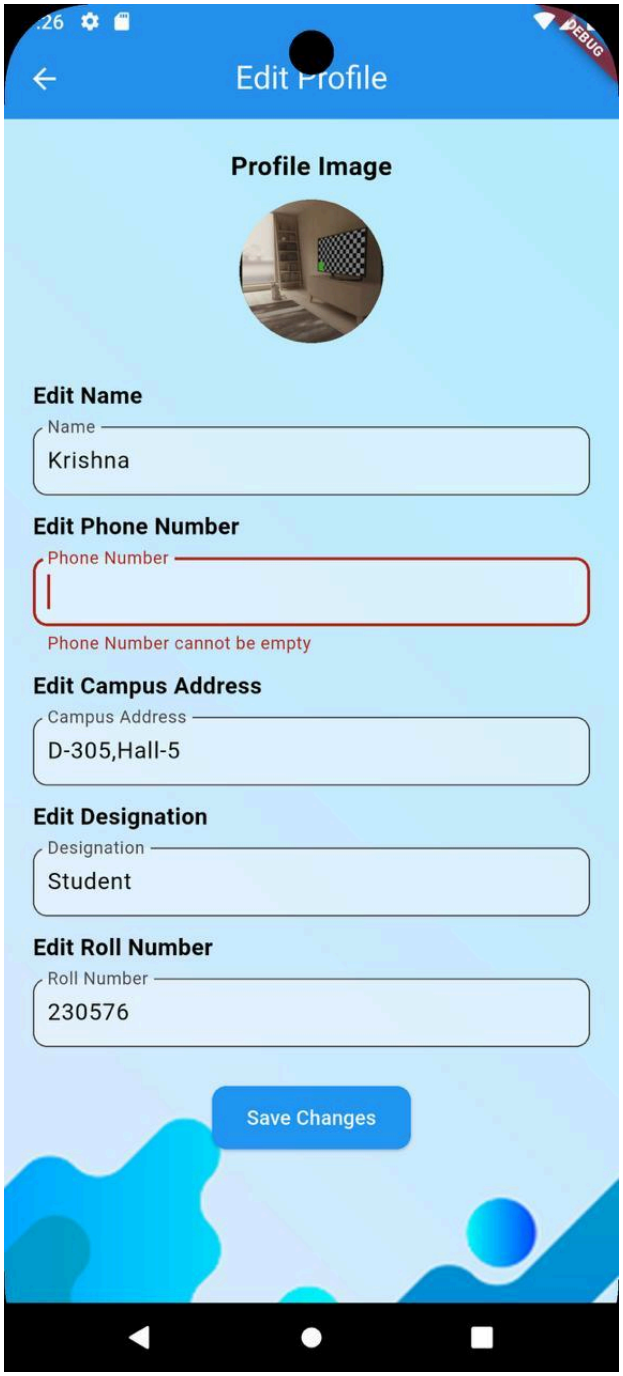
Result: Error message displayed **"Name cannot be empty"** as shown in the image. Save Changes button doesn't complete the edit process until a name is provided.

TEST #2:
Details: Empty Campus Address field when editing profile.



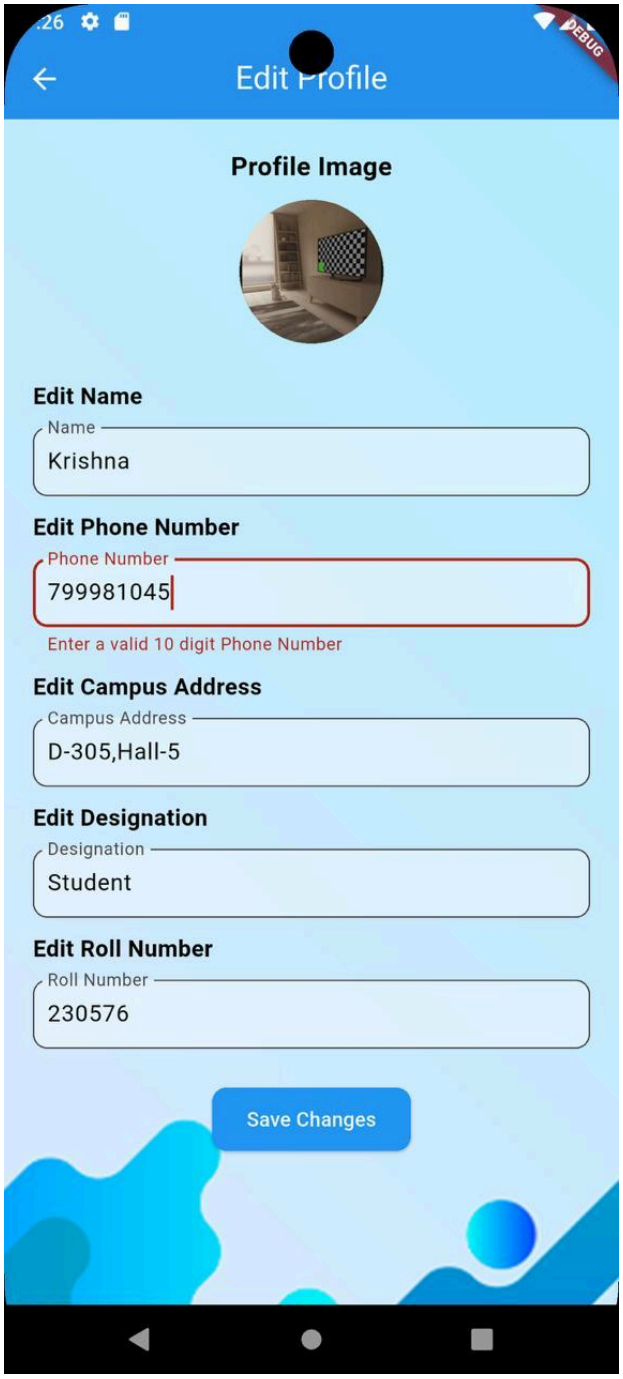
Result: Error message displayed "**Campus Address cannot be empty**" as shown in the image. The Save Changes button doesn't complete the edit process until the campus address is provided.

TEST #3:
Details: Empty Phone Number field when editing profile.



Result: Error message displayed "**Phone Number cannot be empty**" as shown in the image. The Save Changes button doesn't complete the edit process until the phone number is provided.

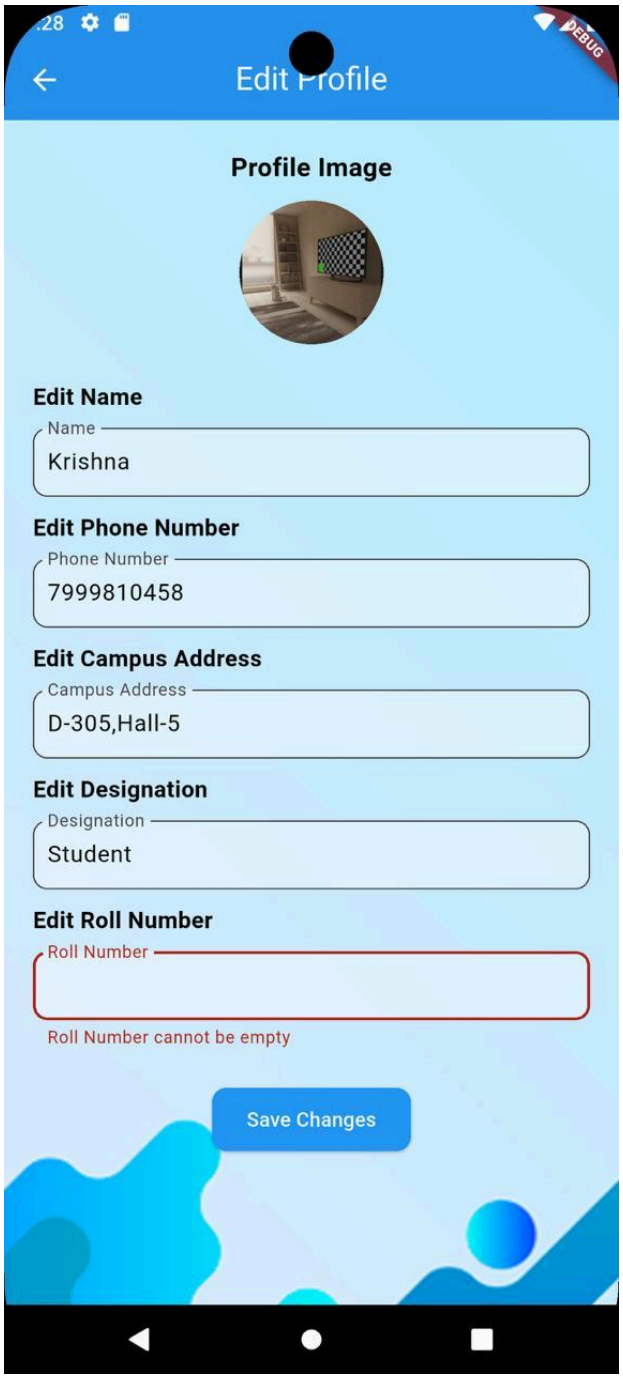
TEST #4:
Details: Invalid Phone Number format (less than 10 digits) when editing profile.



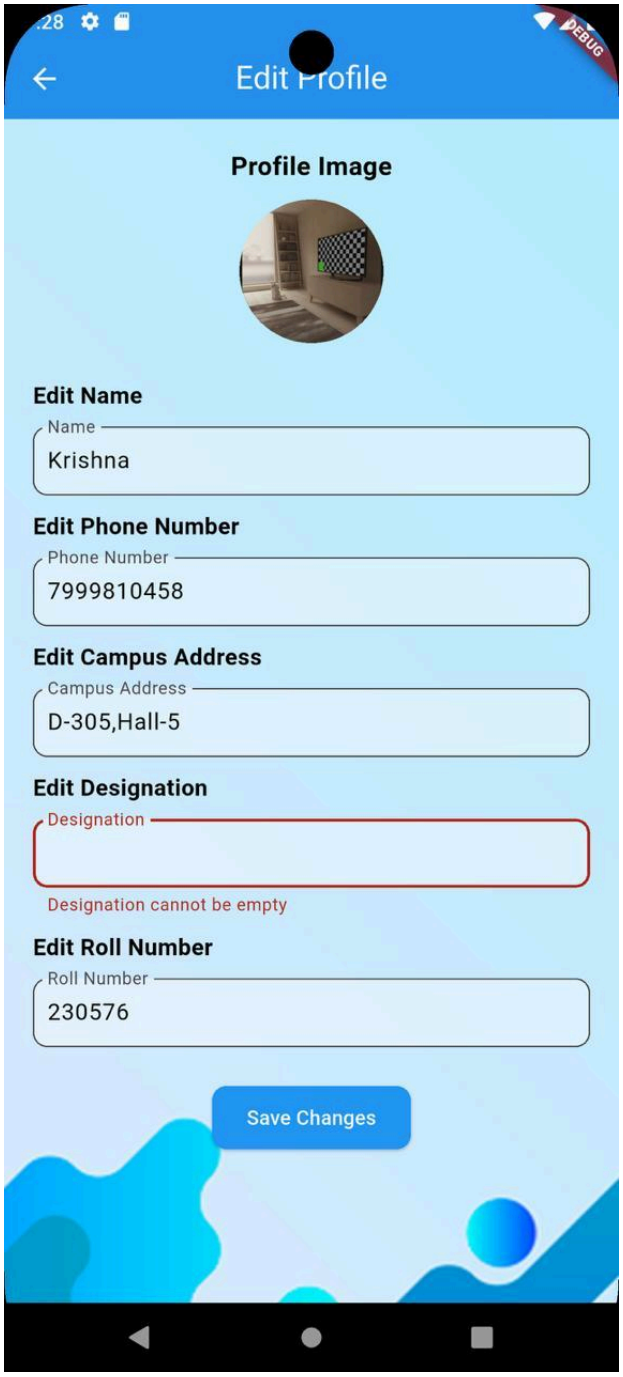
Result: Error message displayed "**Enter a valid 10 digit Phone Number**" as shown in the image. The Save Changes button doesn't complete the edit process until a valid 10-digit phone number is provided.

TEST #5:
Details: Empty Roll Number field when editing profile.

Result: Error message displayed "**Roll Number cannot be empty**" as shown in the image. The Save Changes button doesn't complete the edit process until roll number is provided.

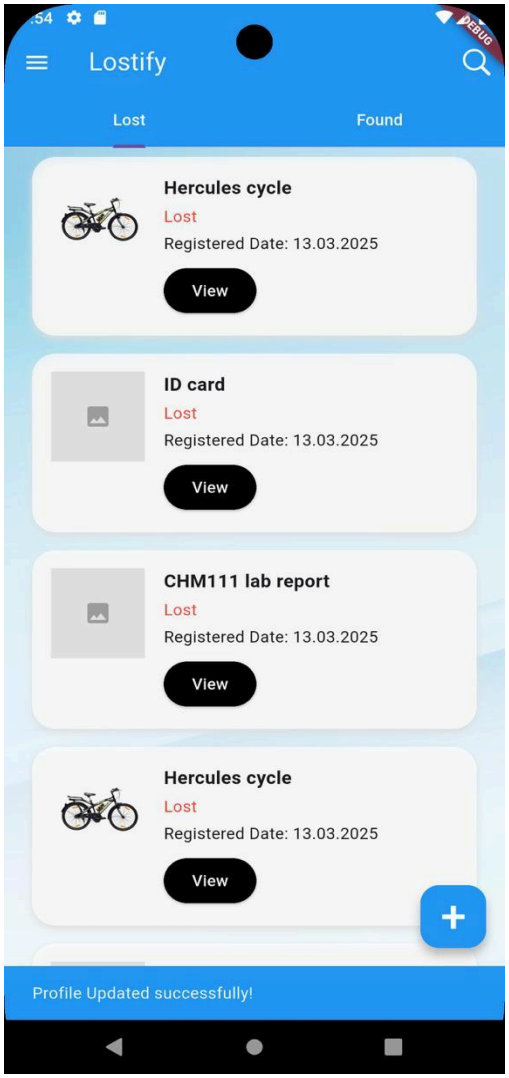


TEST #6:
Details: Empty Designation field when editing profile.



Result: Error message displayed "**Designation cannot be empty**" as shown in the image. The Save Changes button doesn't complete the edit process until designation is provided.

TEST #7:
Details: Successful profile edit with all valid fields.



Flow:

- User navigates to Dashboard using hamburger icon
- User selects **"Edit Profile"** option
- User edits necessary fields (Name: "Krishna", Phone Number: "7999810458", Campus Address: "D-305,Hall-5", Designation: "Student", Roll Number: "230576")
- User clicks **"Save Changes"** button
- System processes the changes and updates the database with new information
- User is redirected to profile page or dashboard
- System displays confirmation message indicating successful profile update

Result: Profile information is successfully updated in the database and the changes are reflected in the user's profile view.

3.5 Messages

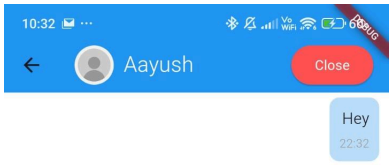
Module Details:

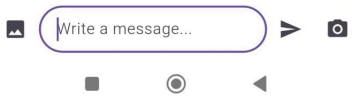
The Messages module provides a streamlined one-to-one chat interface where users can send text and images from their gallery or by taking a new photo. The UI features a header with the recipient's name, back control and close control (**to permanently delete the saved chats**), timestamps on each message, auto-scroll to the latest entry. The send button remains disabled until valid input is detected, preventing empty or whitespace-only messages. Image uploads are automatically rejected with clear error messages if they exceed size limits or are in unsupported formats, and users are prompted to grant camera or storage permissions when needed. In case of network or server failures, concise alerts appear with a retry option, and any unsent drafts are preserved if the user navigates away.

Test Owner: Group #6

Test Date: 06/04/2025

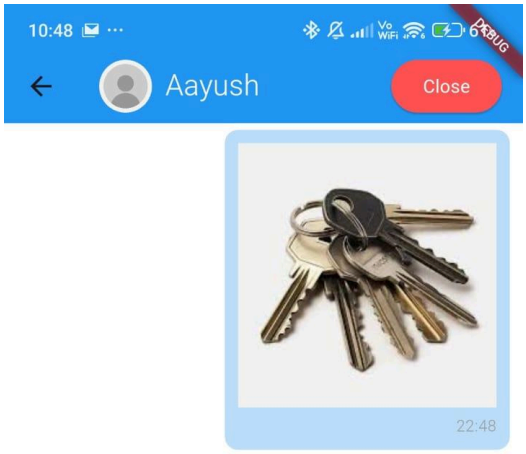
TEST #1:
Details: Successful sending of text:

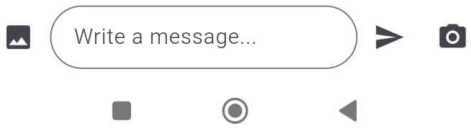




Result: Text is sent successfully.

TEST #2:
Details: Successful sending of image:

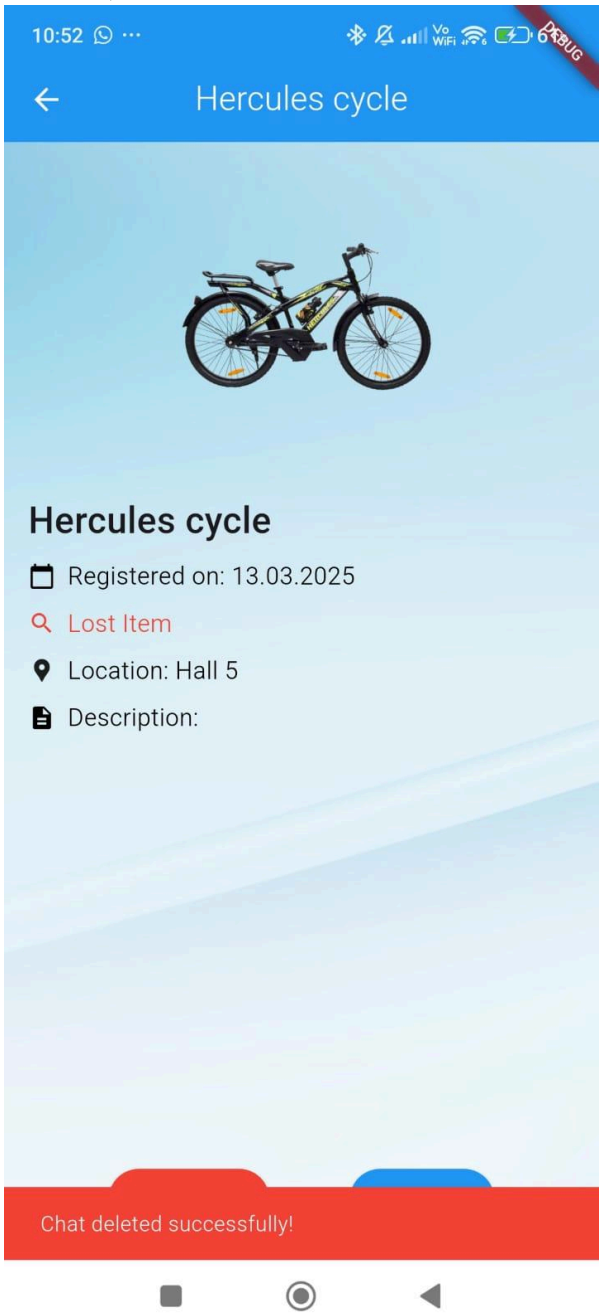




Result: Image is sent successfully with correct timestamp.

TEST #3:

Details: Closing of chat (i.e. permanently deleting saved chat by clicking on closed buttons) :



Result: Chat is deleted successfully.

3.6 Report Item

This module encompasses testing procedures for reporting inappropriate or fake items in the system. When users view items, they have the option to report them, which increases the report frequency count by 1. Additionally, it includes validating the admin-only interface for viewing reported items and the ability to take action (delete or ignore) on reported items.

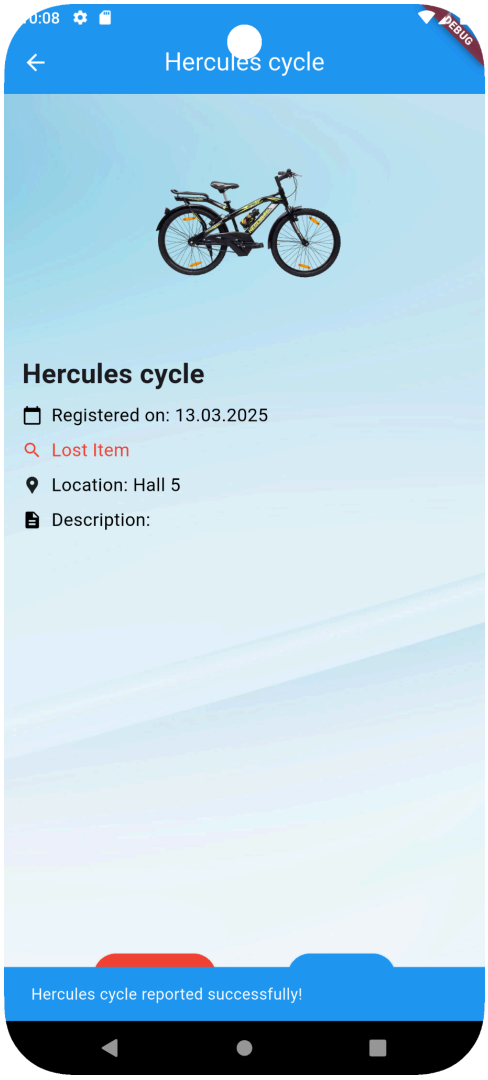
Test Owner: Group #6

Test Date: 06/04/2025

Test Results:
In this test, we validated the successful integration of backend and frontend components of the Report Item feature. We also validated that the database is updated when an item is reported, and that the admin interface correctly displays reported items with their report frequency.

The report API is tested here.

TEST #1
Details: User reporting an item as inappropriate.



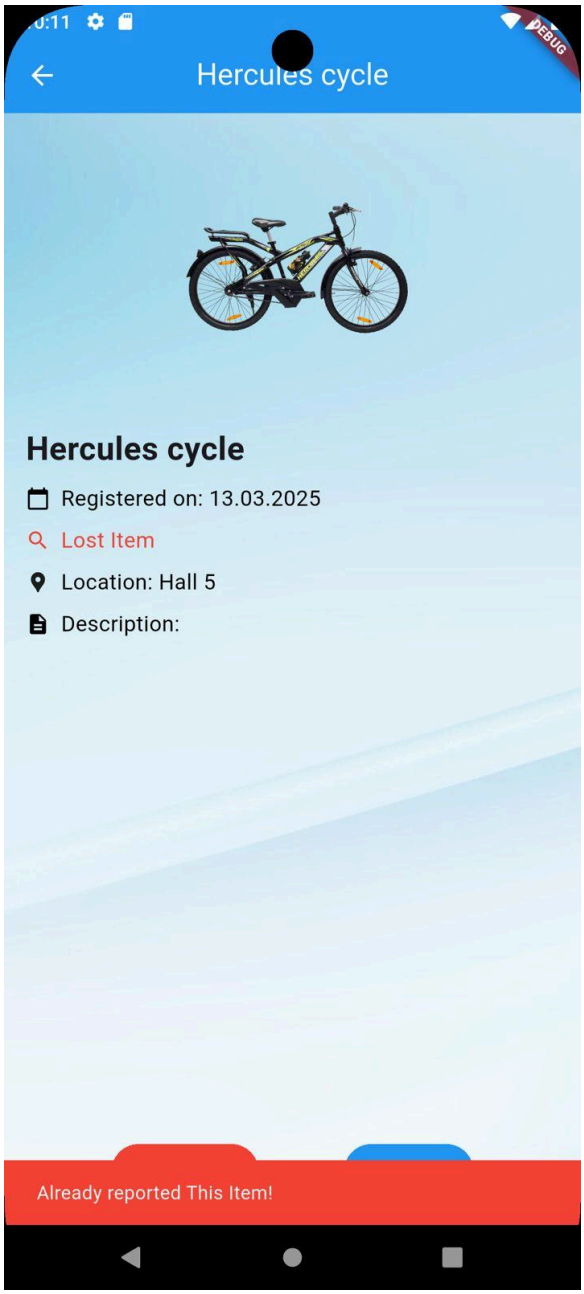
Flow:

- User views an item in the lost/found items list
- User clicks on the "Report" button or icon
- System asks for confirmation "Are you sure you want to report this item?"
- User confirms the report
- System updates the report frequency count for that item in the database
- System displays confirmation message "Item reported successfully"

Result: Item's report frequency is incremented by 1 in the database, and a confirmation message is shown to the user.

TEST #2

Details: User attempting to report the same item multiple times.



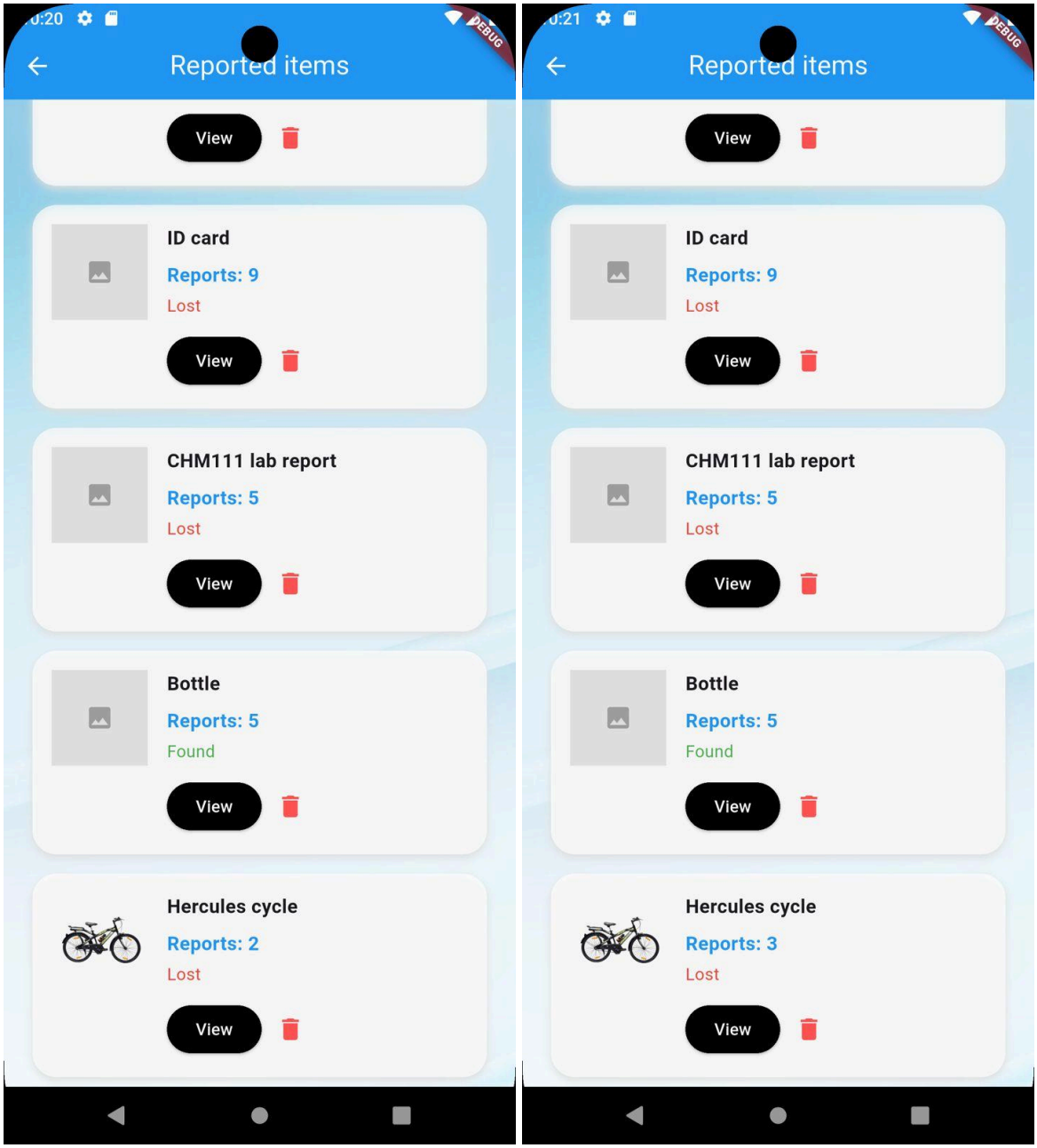
Flow:

- User views an item they have already reported
- User clicks on the "**Report**" button again
- System identifies that the user has already reported this item
- System displays message "You have already reported this item"

Result: System prevents duplicate reports from the same user, maintaining integrity of the report frequency count.

TEST #3

Details: Admin viewing the reported items page.



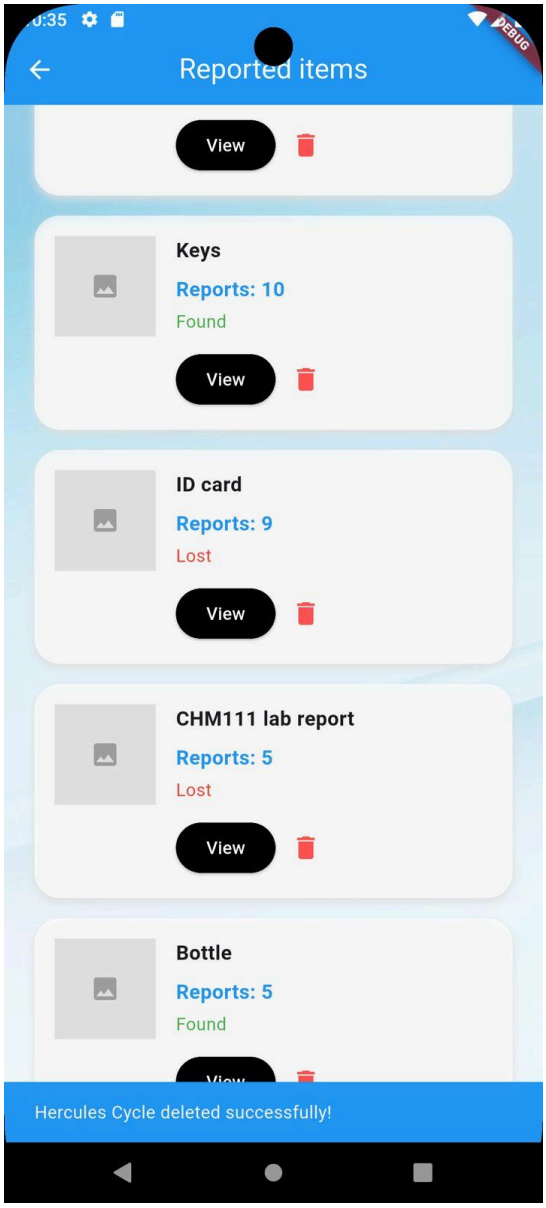
Flow:

- Admin logs into the system
- Admin navigates to the admin dashboard
- Admin selects "**Reported Items**" option
- System displays a list of all reported items sorted by report frequency
- For each item, the system shows: item details, image, report frequency, and action buttons

Result: Admin can successfully view all reported items with their report frequencies.

TEST #4

Details: Admin deleting a reported item.



Flow:

- Admin views the reported items list
- Admin selects an item with high report frequency
- Admin clicks "Delete Item" button
- System asks for confirmation "Are you sure you want to delete this item?"
- Admin confirms deletion
- System removes item from the database
- System displays confirmation message "Item deleted successfully"

Result: Reported item is successfully removed from the database and no longer appears in any item listings.

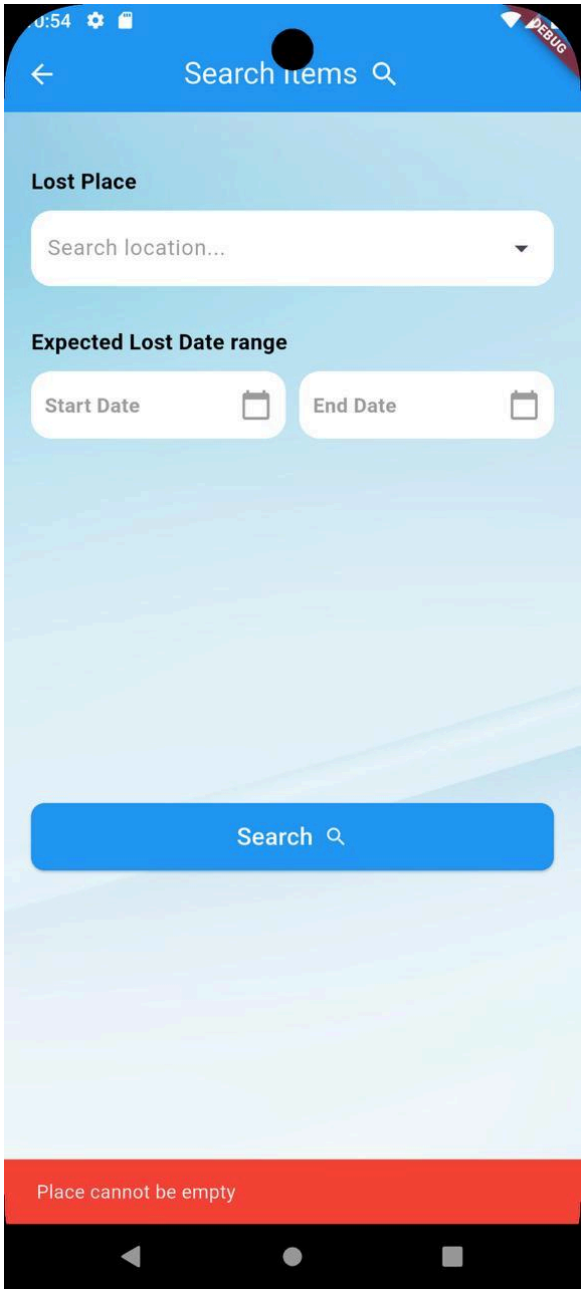
3.7 Search Items

Module Details:
This module encompasses testing procedures for the search functionality that allows users to find lost/found items by location and date range. Users can select a location from a dropdown list (with typeahead filtering) and specify a date range to narrow down their search results. Additionally, it includes validating that the system correctly displays items matching the search criteria.

Test Owner: Group #6
Test Date: 06/04/2025
Test Results:
In this test, we validated the successful integration of backend and frontend components of the Search feature. We also validated that the search algorithm correctly filters items based on location and date range parameters.
The search API is tested here.

TEST #1

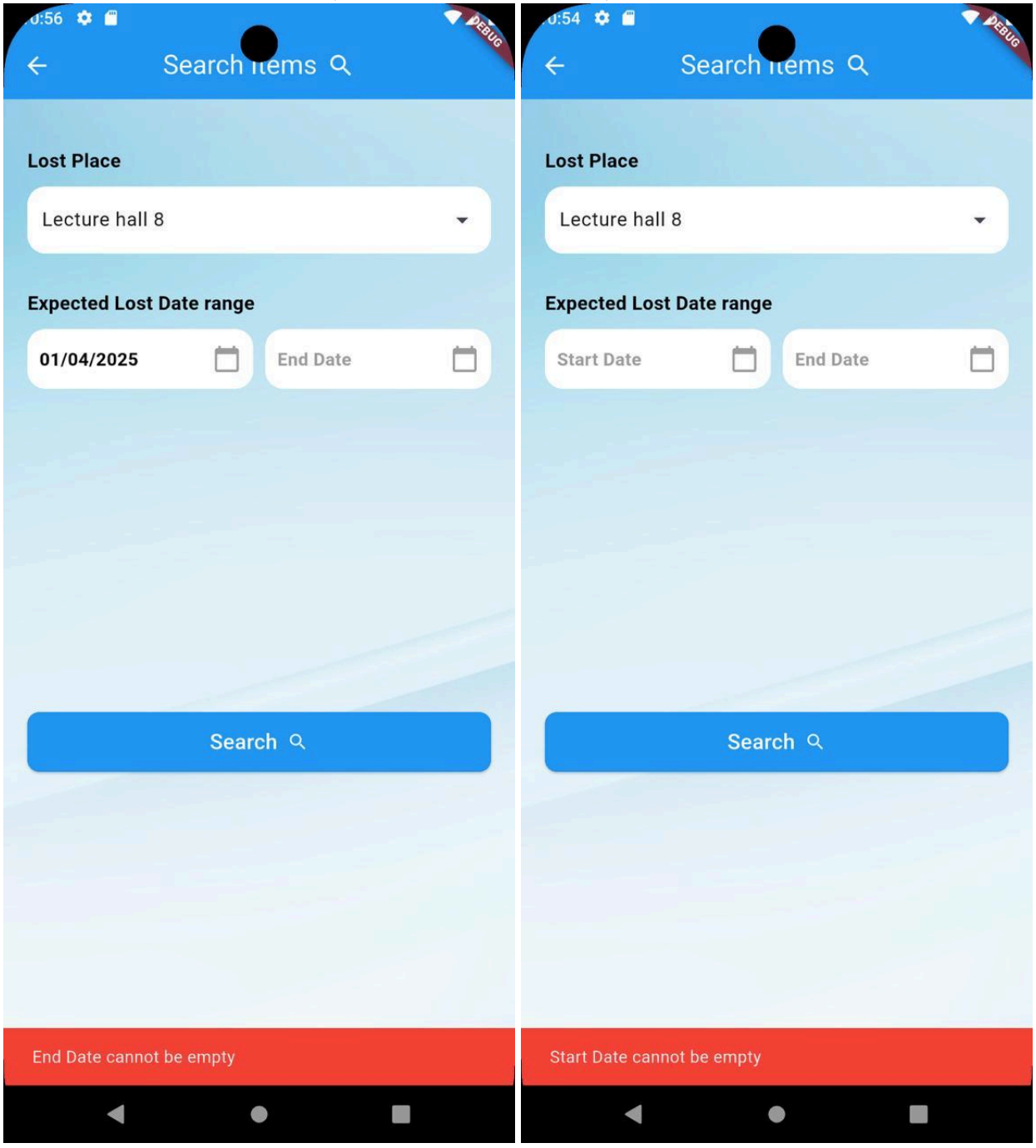
Details: Empty location field when performing a search.



Result: Error message displayed **"Please select a location"** as shown in the image. Search button doesn't execute the search until a location is selected from the dropdown.

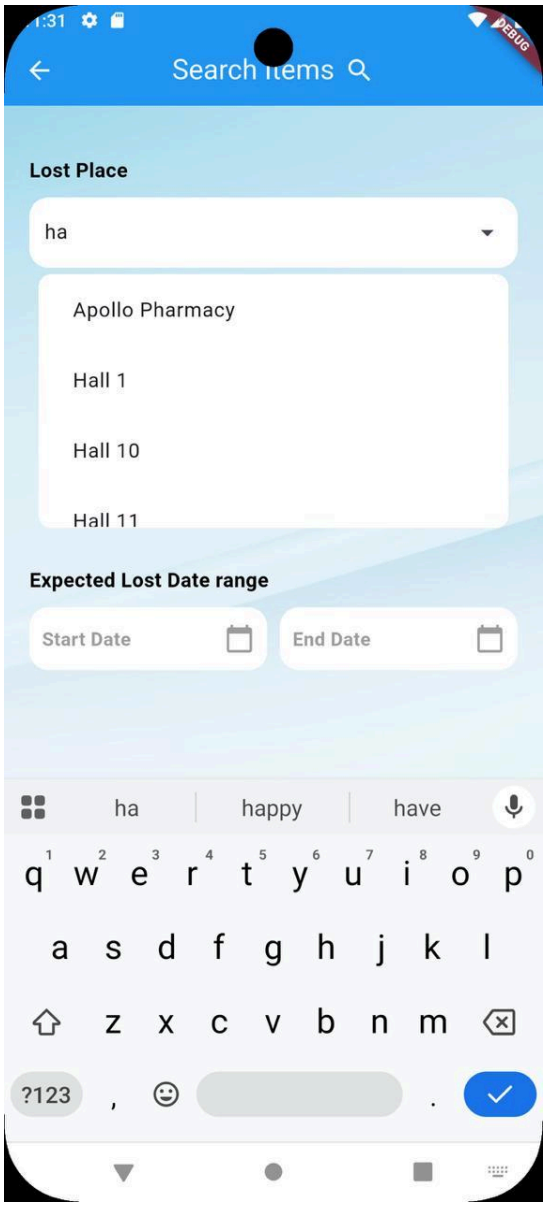
TEST #2

Details: Missing date range (either start or end date) when performing a search.



Result: Error message displayed **"Start Date cannot be empty"** or **"End Date cannot be empty"** as shown in the image. Search button doesn't execute the search until both dates are provided.

TEST #5
Details: Typeahead functionality of location dropdown.



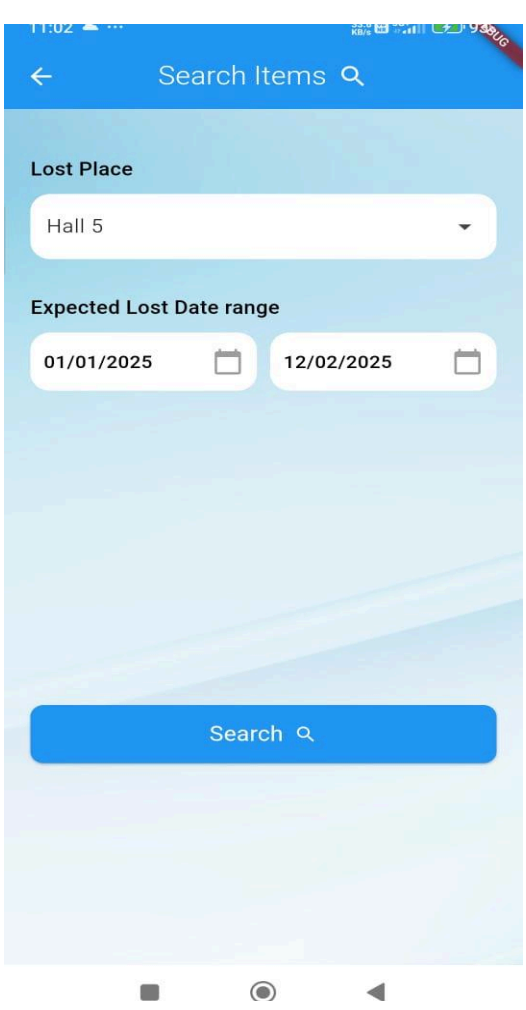
Flow:

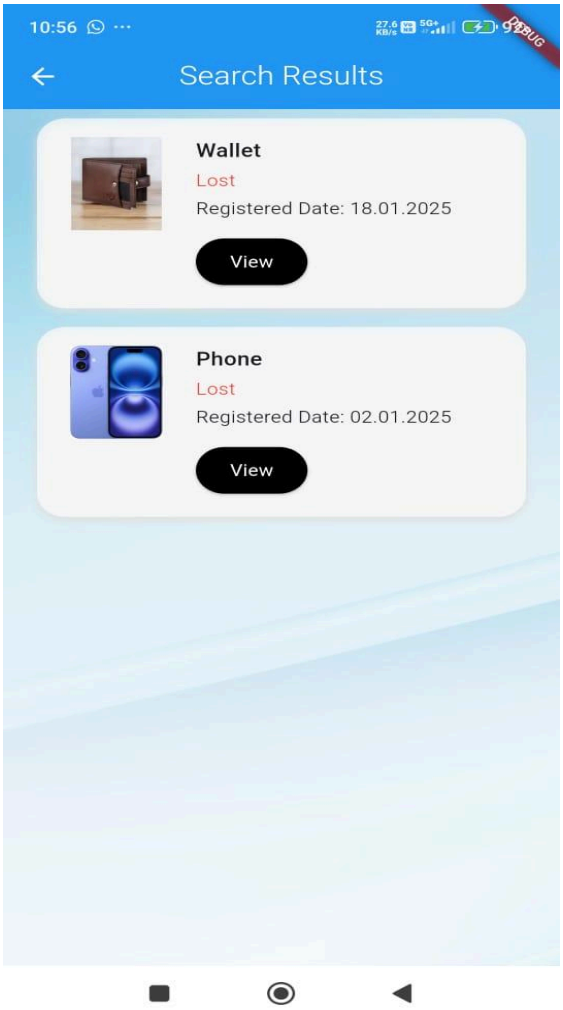
- User navigates to Search page from homepage
- User starts typing **"lec"** in the location field
- Dropdown narrows down to show only locations containing **"he"** (e.g., "Hall 1")
- User selects a filtered location
- Location is correctly populated in the search field

Result: Location dropdown correctly filters options as user types, improving search efficiency.

TEST #6

Details: Search with valid location and date range with matching items.





Flow:

- User navigates to Search page from homepage
- User selects location "**Hall 5**" from dropdown
- User selects date range (e.g., "01/01/2025" to "12/02/2025")
- User clicks "**Search**" button
- System processes the search query
- System displays list of items matching the criteria
- Each item shows title, description, image, and date found/lost

Result: System correctly displays all items that match both the location and date range criteria.

4 SYSTEM TESTING

4.1.1 Account set-up and authentication

1) Requirement: Only users associated with IIT Kanpur can register with and use the app.

Test Owner: *Ayush Patel*

Test Date: *04/04/25*

Test Results: The login credentials use users' IIT Kanpur CC username to authenticate. This ensures that only registered IIT Kanpur affiliated users can sign up. Identity verification is based on users' IIT Kanpur email address.

Additional Comments: NA

2) Requirement: The user should be able to login using a password-based mechanism.

Test Owner: *Ayush Patel*

Test Date: *04/04/25*

Test Results: The user can use an arbitrary-length password to secure his/her account with account recovery options available. The user may change his/her password anytime from the 'change password' option available in the dashboard.

Additional Comments: NA

3) Requirement: The user should be able to set and edit his/her profile.

Test Owner: *Ayush Patel*

Test Date: *04/04/25*
Test Results: The user is asked to set up his/her profile, including details such as name, phone number, and place of residence, during signup. Subsequently, users may edit the details through the edit profile page that can be navigated to through the dashboard.

Additional Comments: NA

4) Requirement: The user should be able to recover his/her account in case he/she forgets his/her credentials or his/her credentials are hijacked.

Test Owner: *Ayush Patel*

Test Date:*04/04/25*

Test Results: The user is provided an option to reset his/her password at login. On tapping, a random 16-character password is generated and mailed to the user on his/her registered (IIT Kanpur) email. The user may then log in using this new password and subsequently set a secure password.

Additional Comments: NA

5) Requirement: Users can log out.

Test Owner: *Ayush Patel*

Test Date:*04/04/25*

Test Results: Upon logging out, the user is redirected to the welcome page from where he may login again.

Additional Comments: NA

4.1.2 Item Posting and Categorisation

6) Requirement: Users can create posts about lost/found articles.

Test Owner: *Ayush Patel*

Test Date: *04/04/25*

Test Results: Users can open forms to create new posts for lost or found actions through the floating action button on the homepage. Multiple fields are provided and mandatory fields are enforced.

Additional Comments: NA

7) Requirement: Users can delete their own posts if the item is found or recovered.

Test Owner: *Ayush Patel*

Test Date: *04/04/25*

Test Results: Users are given an option to delete their posts manually after expanding them through a dedicated button.

Additional Comments: NA

4.1.3 Search and Filter

8) Requirement: Users can search for lost or found items using parameters.

Test Owner: *Ayush Patel*

Test Date: *04/04/25*

Test Results: Users can filter search results using the location of loss/find of an article and a date range within which an article was lost or could have been found.

Additional Comments: NA

4.1.4 Interactive Chat

9) Requirement: Users can initiate private chats with post creators.

Test Owner: *Ayush Patel*

Test Date: *04/04/25*

Test Results: Users are given the facility to open a chat with a poster from his/her post. The chat may contain text and images.

Additional Comments: NA

4.1.5 Claim System

10) Requirements: Ensure that lost items are correctly claimed by the rightful owners in a secure manner.

Test Owner: *Shaurya Johari*

Test Date:*04/04/25*

Test Results: Users can request to claim a found item. The item's poster will review the claim and approve or deny it based on the verification process.

Additional Comments: NA

4.1.6 Reporting and Moderation

11) Requirements: Ensure that the system provide facilities common to all users as well as special permissions to admins for the same.

Test Owner: *Shaurya Johari*

Test Date: *04/04/25*

Test Results: Admins can take action on Reported items, such as removing posts to maintain the integrity of the platform.

Additional Comments: NA

4.1.7 Profile Management

12) Requirements: Ensure that the system provide its users the facilities to edit its profile details such as name, contact, designation, campus address, PF/Roll No and profile picture

Test Owner: *Shaurya Johari*

Test Date: *04/04/25*

Test Results: The user is able to successfully change his/her account details. The edited data gets stored in the backend and successfully reflects on profile.

Additional Comments: NA

4.1.8 Password Change

13) Requirements: Ensure that the system allows the user to change password through dashboard

Test Owner: *Shaurya Johari*

Test Date: *04/04/25*

Test Results: The user is able to successfully update the profile's password. Upon logging in after logging out, the new password gets accepted for logging into profile successfully.

Additional Comments: NA

14) Requirements: Ensure that the system enables users who had forgotten their old password to log in through 16 digit password that works once

Test Owner: *Shaurya Johari*

Test Date: *04/04/25*

Test Results: The user can log in through the unique password sent once and change password successfully. The one time password fails to log in in the subsequent attempts.

Additional Comments: NA

4.1.8 Edit Item Post

15) Requirements: Ensure that its possible for the user to edit details related to the lost items posted

Test Owner: *Ayush Patel, Shaurya Johari*

Test Date: *04/04/25*

Test Results: The edited lost post description gets successfully stored inside the database and the change gets reflected on other accounts as well.

Additional Comments: NA

16) Requirements: Ensure that its possible for the user to edit details related to the found items posted

Test Owner: *Ayush Patel, Shaurya Johari*

Test Date: *04/04/25*

Test Results: The edited found post description gets successfully stored inside the database and the change gets reflected on other accounts as well.

Additional Comments: NA

4.1.9 Dashboard Functionality

17) Requirements: The hamburger button on the top left must toggle the dashboard on and off

Test Owner: *Ayush Patel*

Test Date: *05/04/25*

Test Results: The edited lost post description gets successfully stored inside the database and the change gets reflected on other accounts as well.

Additional Comments: NA

18) Requirements: Ensure that the option “Your Lost Items” on dashboard successfully link to different relevant pages of the app

Test Owner: *Shaurya Johari*

Test Date: *05/04/25*

Test Results: The user gets redirected to the “Your Lost Item” page, where the user can see all items he had reported as “Lost”

Additional Comments: NA

19) Requirements: Ensure that the option “Your Found Items” on dashboard successfully link to different relevant pages of the app

Test Owner: *Shaurya Johari*

Test Date: *05/04/25*

Test Results: The user gets redirected to the “Your Found Item” page, where the user can see all items he had reported as “Found”

Additional Comments: NA

20) Requirements: Ensure that the option “Messages” on dashboard successfully link to the Messages page

Test Owner: *Shaurya Johari*

Test Date: *05/04/25*

Test Results: The user gets redirected to the “Messages” page, where the user can see various chats he had with different users of Lostify.

Additional Comments: NA

21) Requirements: Ensure that the option “Edit Profile” on dashboard successfully link to the relevant page for profile update

Test Owner: *Shaurya Johari*

Test Date: *05/04/25*

Test Results: The user gets redirected to the “Edit profile” page, where the user can edit existing user data with current/new data about the user .

Additional Comments: NA

22) Requirements: Ensure that the option “Change Password” on dashboard successfully link to different relevant pages of the app

Test Owner: *Shaurya Johari*

Test Date: *05/04/25*

Test Results: The user gets redirected to the “Change Password” page, where the user can enter his cc username to get the password change process started .

Additional Comments: NA

5 CONCLUSION

- **How Effective and exhaustive was the testing?**

Testing was done almost exhaustively. Each API underwent testing to ensure that they were modifying the database and returning the desired information when needed. We have taken care of all cases, thus ensuring complete branch coverage. The frontend was also tested and bugs were resolved. Validity of each input has been checked in all forms. Appropriate alerts have been shown in such cases. Edge cases were also tested and were confirmed to work as expected. The testing was quite effective. Developers tested components not developed by him/her.

- **Which components have not been tested adequately?**

Our test coverage for non-functional requirements outlined in the SRS document could be further expanded. Perform tests adequately for non-functional requirements like reliability, maintainability and ease of learning could not be performed. These can be tested only when the application is beta-tested or receives a significant amount of users/traffic.

- **What difficulties have you faced during testing?**

The main difficulty faced during testing was ensuring non-functional requirements are satisfied, in system testing.

- **How could the testing process be improved?**

The testing could have been improved by doing the unit testing for the frontend or integration testing as automated rather than manual. The developer might subconsciously assume a few crucial things which may be not so obvious to an end-user or someone who was not involved in the development of the software.

APPENDIX A- GROUP LOG

Date	Timings	Duration	Venue	Minutes
29 Mar	14:00 – 18:00	4 hrs	1st floor, Rajeev Motwani Building	• Debugged some Errors and Issues related to Integrating Front End and Back End
30 Mar	14:00 – 18:00	4 hrs	1st floor, Rajeev Motwani Building	• Searched for Resources about Flutter app testing and discussed further the stages for unit testing • Distributed and Divided the work for Unit Testing • Discussed User Manual Work, decided the work and distributed work
31 Mar	14:00 – 20:00	6 hrs	1st floor, Rajeev Motwani Building	• Completed App Unit Testing • Started work for user manual documentation for Lostify.
2 Apr	22:00 – 02:00	4 hrs	2nd floor, KD Building	• Completed the main contents of the User Manual • Work on Testing Document
3 Apr	22:00 – 04:00	6 hrs	2nd floor, KD Building	• Began system testing • Some Integration Testing
4 Apr	21:00 - 2:00	5 hrs	2nd floor, KD Building	• Completed Integration Testing
5 Apr	21:00 - 02:00	5 hrs	2nd floor, KD Building	• Completed majority of system testing
6 Apr	17:00 - 00:00	7 hrs	2nd floor, KD Building	• Finalized Testing Document and reviewed all heads