Rabab Khan Rongon

Testing Approach for NoteIt Application

1. Objective

To ensure the NoteIt API functions as intended, is secure, reliable, and performs well under different scenarios. The goal is to validate both happy paths and edge/failure cases through automated and manual testing.

2. Testing Approach

A. Test Types

- Functional Testing: Verify that each API endpoint behaves as expected
- Negative Testing: Ensure the system gracefully handles invalid input or bad behavior
- Integration Testing: Validate interaction between authentication and note management APIs
- Regression Testing: Ensure existing features work after changes

B. Tools Used

- Postman – Manual API test creation and execution
- Newman – Automated command-line test execution
- Node.js – Runtime for running Newman
- Environment Variables – For dynamic values like tokens, user IDs, etc.

C. Test Structure

- Authentication Flow
- Register new users
- Log in and store token
- Get current user data
- Note Management
- Create, read, update, delete (CRUD)
- Validate ownership rules
- Verify timestamps and structure
- Sharing & Permissions
- Share notes with other users
- View notes shared with me
- Prevent unauthorized access to others' notes
- Bookmarking
- Bookmark and unbookmark notes

- Get bookmarked notes
- Search Functionality
- Search by title (case-insensitive)
- Handle empty and non-matching queries

D. Dynamic Test Data

Emails are dynamically generated using an incrementing counter (EMAIL_COUNTER) to avoid duplicate registration errors.

Sharing targets are calculated by referencing users created in previous steps (e.g., login_email - 2).

E. Validation & Assertions

Each test includes assertions like:

- Status codes (200 OK, 201 Created, 400 Bad Request, etc.)
- Response body structure (e.g., has note.id, user.email)
- Ownership and permission checks
- Sorting and filtering validations (e.g., notes sorted by created_at)

G. Error Handling Scenarios Covered

- Duplicate registration
- Invalid login credentials
- Sharing with nonexistent or self accounts
- Accessing notes not owned/shared
- Empty required fields

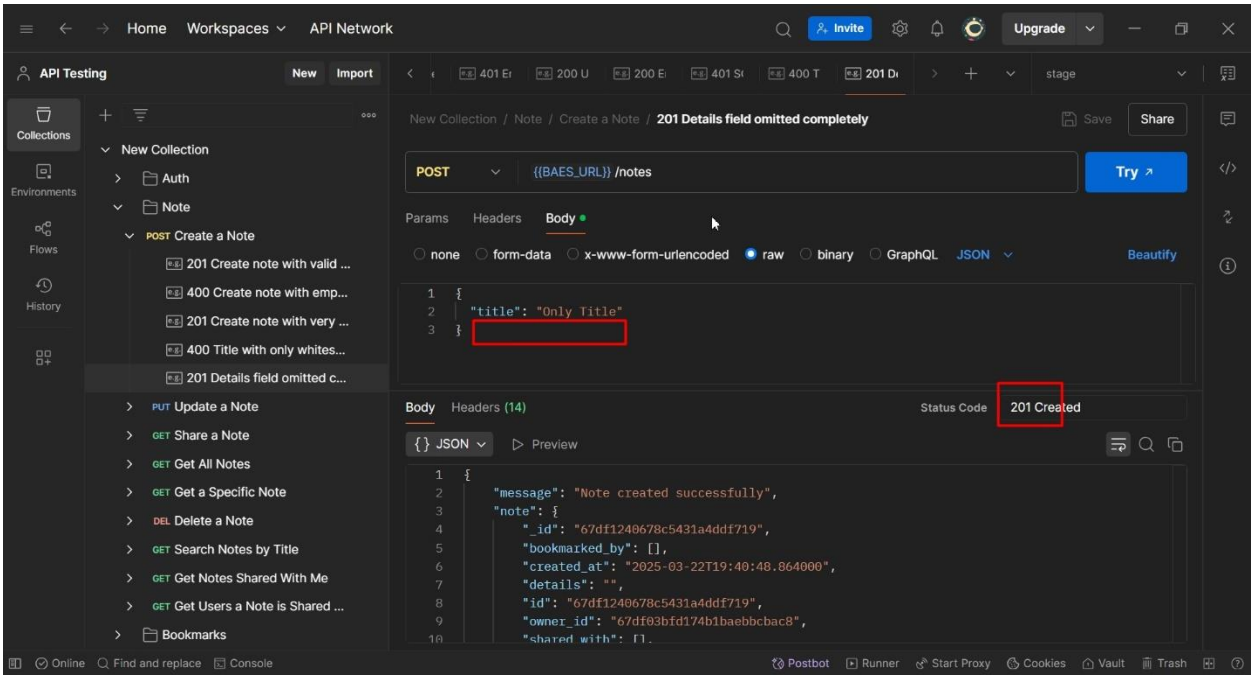
Test Cases:

All cases from Documentation also added some cases in a example portion of postman.

Rabab Khan Rongon

Bug Reporting:

1. Title: Note Details field omitted completely in Note Creation



Steps to Reproduce:

1. Attempt to create a new Note via API (POST request).
2. Check the form fields or request body schema.
3. Submit the note with valid values, including a Note Details field (description/body).
4. Check the saved result in the database or UI.

Expected Result:

- There should be a field to enter or pass Note Details during note creation.
- The content should be saved and visible in the created note.

Actual Result:

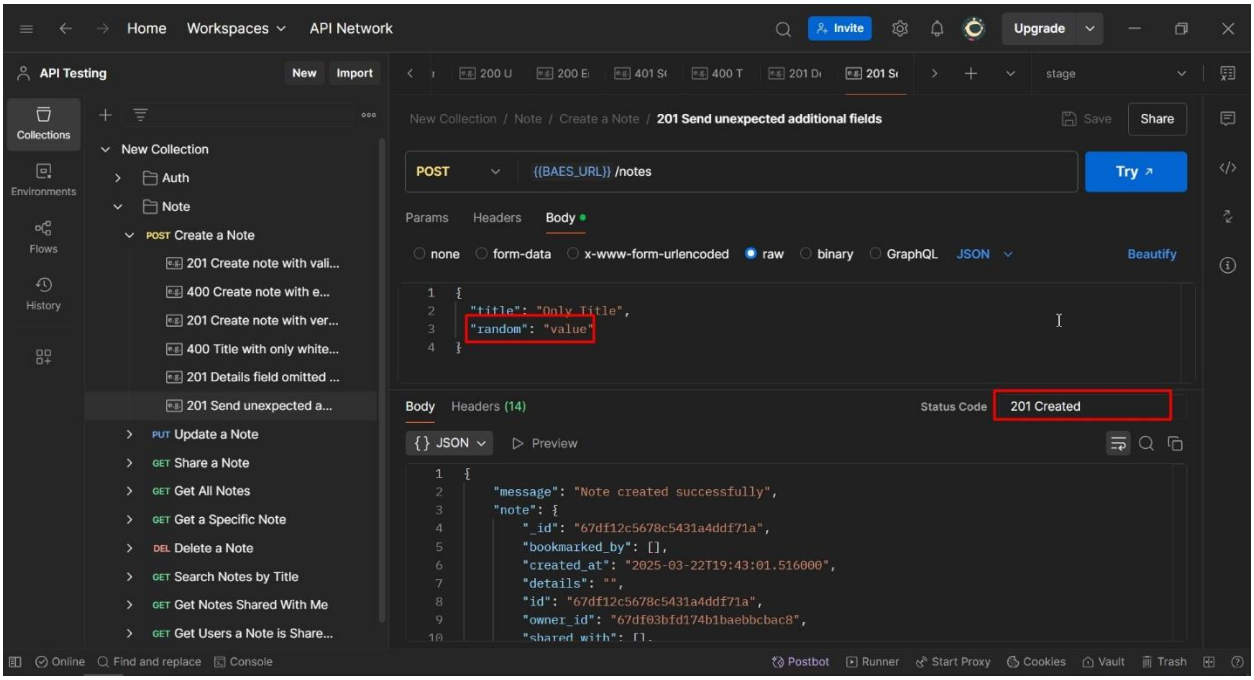
- The Note Details field is completely missing in both the form and the API structure.
- Notes are created without any descriptive content.
- No validation error shown.

Additional Information:

- Issue discovered while performing API testing.
- This could be due to a missing field definition, form rendering issue, or API schema mismatch.

Rabab Khan Rongon

2. Title: API Accepts and Stores Unexpected Additional Fields in Notes Creation



Steps to Reproduce:

1. Perform a **POST request** to the **Notes Creation** endpoint.
2. In the request body, include all valid required fields (e.g., title, linked ID, note details).
3. Add **additional, non-documented or random fields** (e.g., testKey, unauthorizedField, randomText).
4. Submit the request.
5. Review the response and saved data in the system.

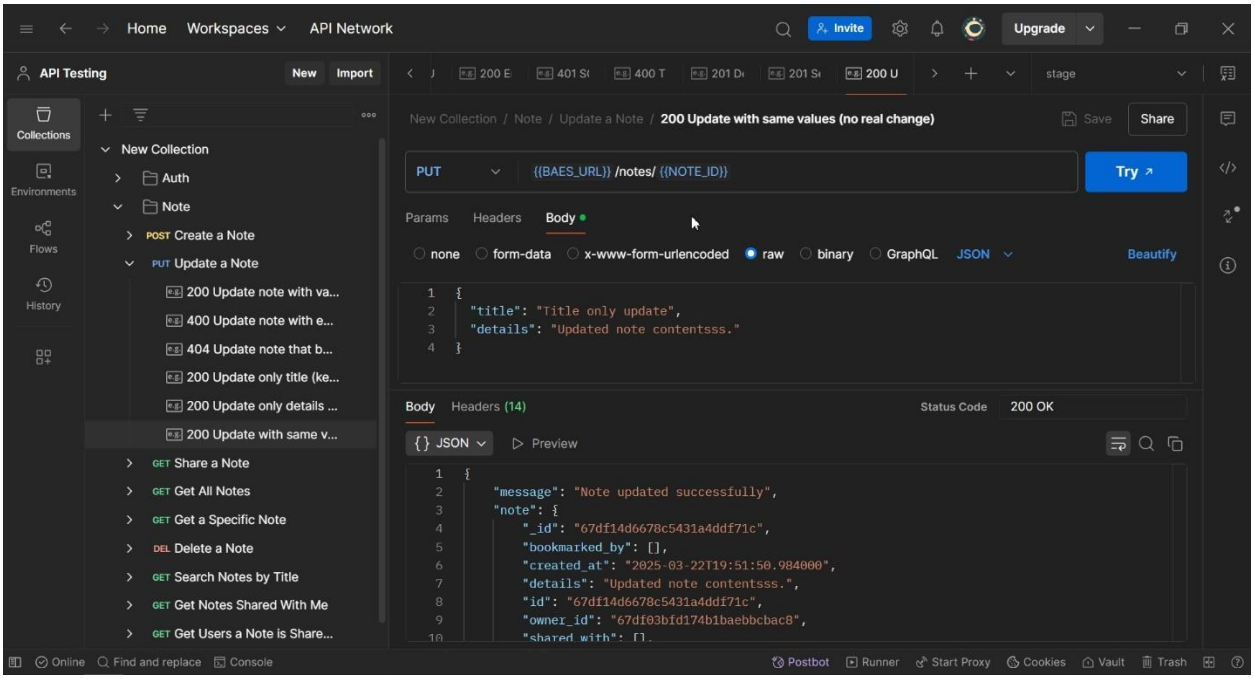
Expected Result:

- The API should **reject any unexpected fields** with a 400 Bad Request or **ignore them entirely**.
- Only **defined and valid fields** should be accepted and stored.

Actual Result:

- The API **accepts and stores unexpected fields** silently.
- These fields are **not documented or intended**, which may lead to **data integrity issues** or **security risks**.

3. Title: Notes Update API Processes Requests Even When No Actual Changes Are Made



Steps to Reproduce:

1. Retrieve an existing note using the **GET Note API**.
2. Send a **PUT/PATCH request** to update the note using **the exact same values** (no modification to title, details, or any other field).
3. Observe the API response and backend behavior.

Expected Result:

- The system should detect that **no real changes were made**, and either:
  - **Return a 304 Not Modified**, or
  - Respond with a message stating **"No changes detected"** without updating timestamps or creating logs.

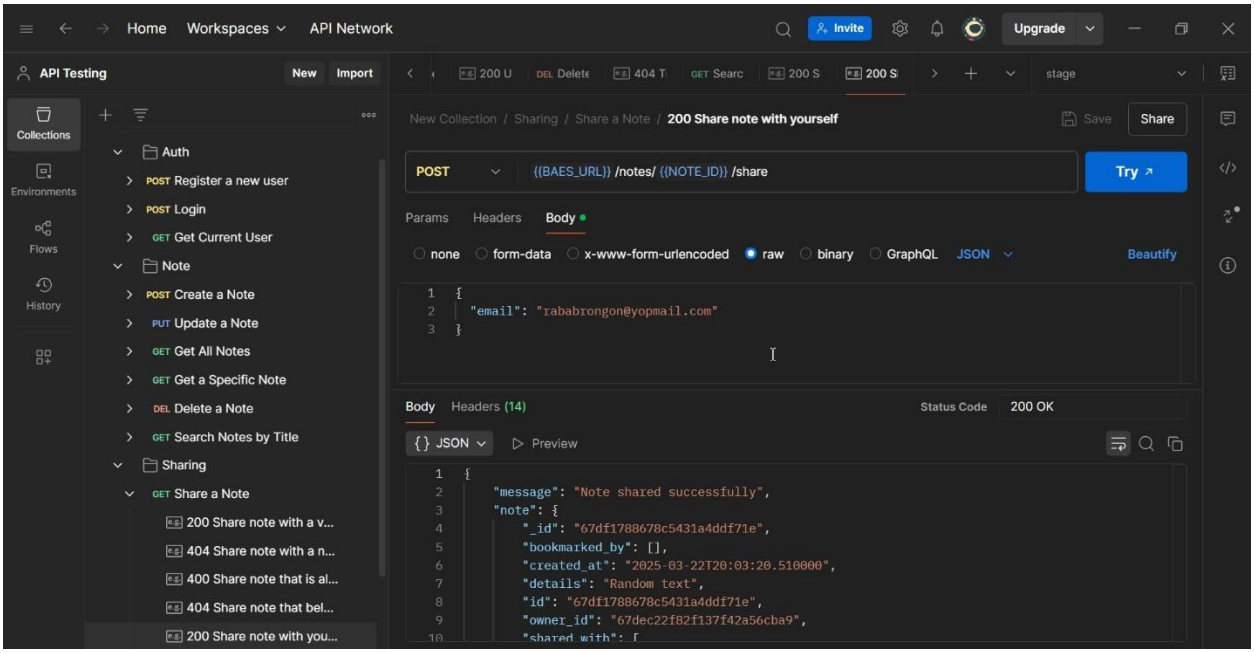
Actual Result:

- The API **accepts and processes the update**, even though the data is **identical**.
- This may result in **unnecessary database writes**, **log noise**, or **confusing activity trails**.

Additional Information:

- No validation is present to **check for actual data changes** before updating.
- Could lead to **performance overhead** and misleading update logs.

4. Title: User Can Share a Note with Themselves – Should Be Prevented



Steps to Reproduce:

1. Create or open an existing Note.
2. Click on the "Share" option.
3. In the user selector, choose your own user profile.
4. Save or confirm the sharing action.

Expected Result:

- The system should prevent users from sharing notes with themselves, as it's unnecessary and redundant.
- A message should appear such as: "You cannot share a note with yourself."

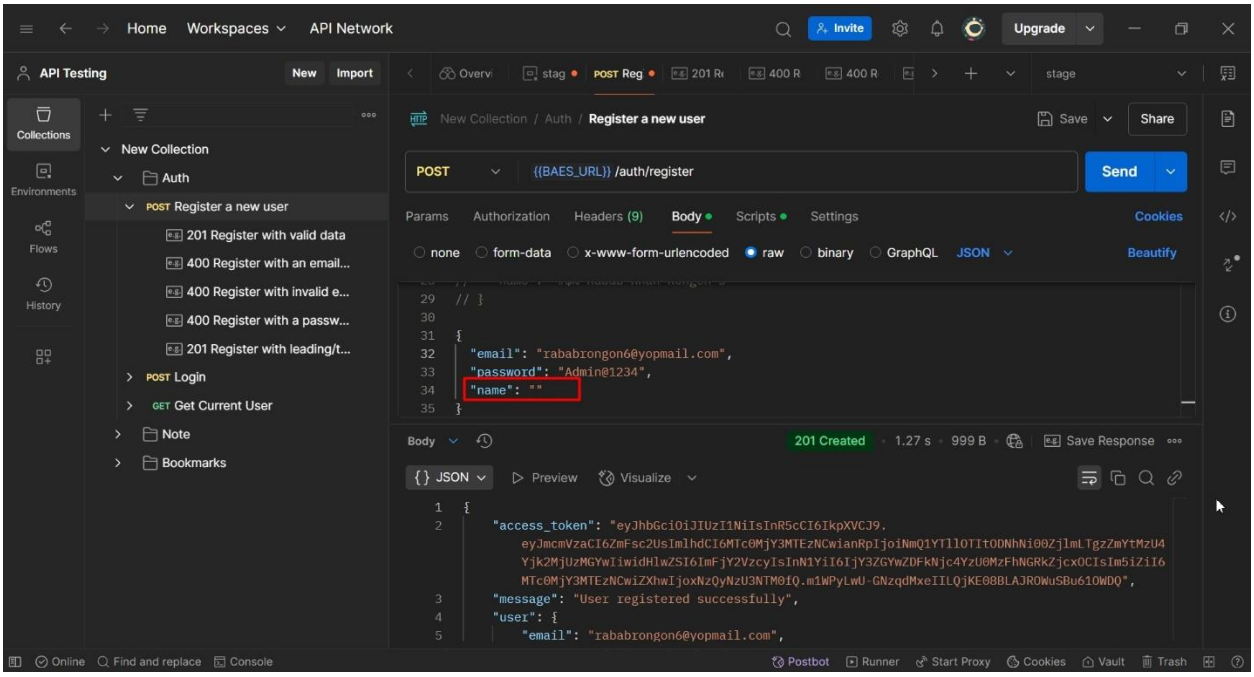
Actual Result:

- The user is able to share the note with themselves.
- No error or warning is displayed, and the action is treated as valid.

Additional Information:

- This leads to confusion in sharing logs and permissions list, and adds no functional value.
- Could clutter the shared user list with self-entries.

5. Title: User Can Be Created Without Providing a Name



Steps to Reproduce:

1. Send a POST request to the user creation endpoint without a name field or with an empty string.
2. Observe the response.

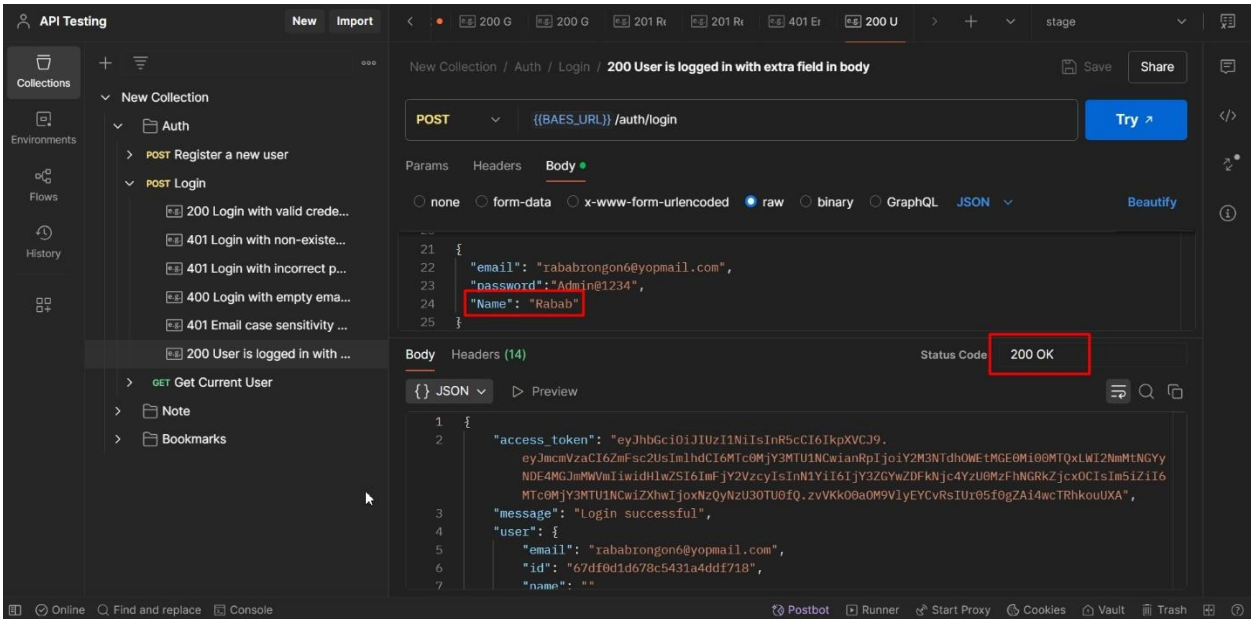
Expected Result:

- The system should validate and require the Name field before creating a user.
- An error should be shown: "Name is required."

Actual Result:

- User is created without a name, both in UI and via API.
- Results in blank names in user lists, which affects identification, permissions, and communication.

6. Title: Login API Accepts Extra Fields in Request Body and Still Authenticates User



Steps to Reproduce:

1. Send a POST request to the Login API endpoint (e.g., /auth/login).
2. Include valid fields like:

```
{
  "email": "user@example.com",
  "password": "validpassword"
}
```

3. Now add extra, unexpected fields to the request body:

```
{
  "email": "user@example.com",
  "password": "validpassword",
  "extraField": "test123",
  "random_data": true
}
```

4. Submit the request and observe the response.

Expected Result:

- The API should strictly validate input and reject any request with unexpected fields, returning a 400 error or warning.

Actual Result:

- The API authenticates the user successfully, even with additional, unvalidated fields.
- No warnings or validation messages are returned.

API improvement suggestions:

1. Enforce Strict Schema Validation

- Why: Prevents unexpected or malicious data from being processed.
- How:
  - Reject any unexpected fields with a 400 Bad Request.
  - Use schema validation libraries (e.g., Joi, Zod, Yup) on all endpoints

2. Implement Field-Level Validation and Error Messaging

- Why: Helps developers understand exactly what's wrong.
- How:
  - Return detailed, structured errors:

```
{
  "error": "Validation Failed",
  "fields": {
    "name": "This field is required",
    "email": "Invalid email format"
  }
}
```