School of Computing

CA400 Year

Unit/Integration Testing Documentation

Genesis Uwumangbe 20459666 & Gideon Amaechi 20364806

Below is a high-level overview of how we carried out unit tests of our application. We will mention how we used Jest along with unit test packages and pytesttoextensivelytest our whole application

Unit Tests

These tests form a crucial part of the development process for Expo+. It ensures that individual components function correctly in isolation.

Frontend Testing

We used Jest as the front-end unit test framework for our Next.js application. Given Jest's strong compatibility with React and its seamless integration with Next.js, it emerged as the optimal choice for ensuring the reliability and stability of our application's user interface components. By harnessing Jest's intuitive API, powerful assertion library, and built-in mocking capabilities, we were able to construct comprehensive test suites tailored to our Next.js environment. This approach empowered us to write

concise and expressive tests that effectively isolated and verified individual components, thus bolstering the maintainability and quality of our front-end codebase. Ultimately, Jest played a pivotal role in streamlining our testing process and instilling confidence in the robustness of our Next.js application.Bellow is what each test file does

Authentication

- checks whether the user is correctly redirected to the preferences page if they do not have existing preferences.
- Test for Handling Invalid Credentials
- Test for Successful Sign-In and Redirect
- Test Successful Registration
- Test Session and Token Management

```
test('handles successful registration', async () => {
    axios.post.mockResolvedValue({ data: { message: 'User registered successfully' } });

    const { getByLabelText, getByRole } = render(<SignUp />);

    fireEvent.change(getByLabelText(/username/i), { target: { value: 'newuser' } });
    fireEvent.change(getByLabelText(/password/i), { target: { value: 'newpassword' } });
    fireEvent.submit(getByRole('button', { name: /sign up/i }));

    await waitFor(() => expect(axios.post).toHaveBeenCalled());
    await waitFor(() => expect(mockedRouter.push).toHaveBeenCalledWith('/login'));
  });

  test('displays error message when registration fails', async () => {
    axios.post.mockRejectedValue({
      response: {
        data: {
          message: 'Username is already taken.'
        }
      }
    });
```

```
 PASS  __test__/authentication.test.js
  Login/Auth
    √ allows the user to log in successfully and redirects to /recommended' (7 ms)
    √ displays an error message with invalid credentials (1 ms)
    √ redirects to preferences page if user has no existing preferences (2 ms)
    √ includes CSRF token in the form submission (1 ms)
    √ User registered successfully (2 ms)

Test Suites: 1 passed, 1 total
Tests:       5 passed, 5 total
Snapshots:   0 total
Time:        7.126 s
Ran all test suites.
(venv) PS C:\Users\desti\Downloads\FYP\2024-ca400-uwumang-2-amaechg-2\src\nextapp>
```

Maps/Svg Maps

- allows user to toggle visited state of a project
- should remove path ID from hoveredPaths on mouse leave
- should handle multiple paths correctly
- should not add the same path ID twice on re-entering
- fetches project data on mount if authenticated
- allows user to toggle visited state of a project

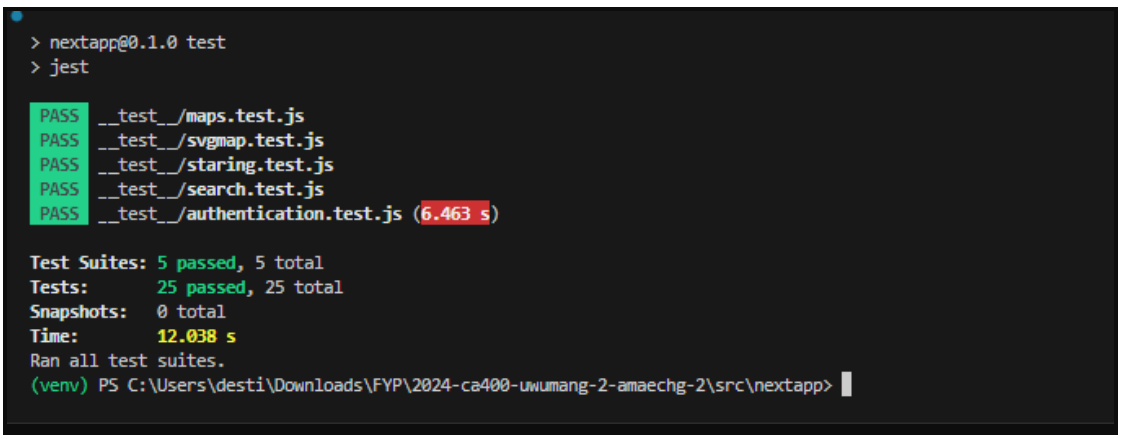
Rating projects

- renders default number of stars when count not provided
- sets temporary rating on mouse enter and resets on mouse leave
- handleClick makes a POST request and handles response
- updates rating on star click
- sets temporary rating on mouse enter and resets on mouse leave

Search

- renders Search component correctly
- allows user to enter search terms
- fetches data on form submit and updates state
- navigates to the correct page when a project ID is clicked
- handles no session and displays error on highlight attempt

Here is an image of all our tests passing.



```
jest.mock('next/router', () => ({
  useRouter: jest.fn(),
}));
jest.mock('next-auth/react');

// Mock for local storage
const mockLocalStorage = (function() {
  let store = {};
  return {
    getItem: function(key) {
      return store[key] || null;
    },
    setItem: function(key, value) {
      store[key] = value.toString();
    },
    clear: function() {
      store = {};
    },
  };
})();
```

When Jest is run our backend is not active to receive any endpoints so to combat this we created mock objects as shown in the image above.

```
describe('Search Component', () => {
  const mockPush = jest.fn();
  beforeEach(() => {
    useRouter.mockImplementation(() => ({
      push: mockPush,
    }));
    useSession.mockImplementation(() => ({ data: { user: { name: 'John Doe' } } }));
    fetch.mockClear();
    mockPush.mockClear();
  });

  test('renders Search component correctly', () => {
    render(<Search />);
    expect(screen.getByPlaceholderText('Enter search term...')).toBeInTheDocument();
    expect(screen.getByText('Search')).toBeInTheDocument();
  });

  test('allows user to enter search terms', () => {
    render(<Search />);
    const input = screen.getByPlaceholderText('Enter search term...');
    fireEvent.change(input, { target: { value: 'Python' } });
    expect(input.value).toBe('Python');
  });
```

Here is an example of us testing our search functionality. We created checks such as rendering the search component and allowing users to input search terms.
Our other tests are quite extensive and are all displayed inside the __test__ directory in the next app

Back end testing

For Our back end unit testing we used pytest, we installed it by adding it to our virtual environment \venv\Scripts\python.exe -m pip install pytest. Pytest is a powerful testing framework that simplifies the process of writing small tests, yet can scale to support complex functional testing for applications and libraries. It is used as a unit testing framework due to its easy-to-write syntax, auto-discovery of test modules and functions, and its ability to handle both simple unit tests as well as more complex functional tests. We will be showing a list of the tests we performed. All code will be placed inside the test directory in the flask app this directory consists of three files an __init__,. conftest.py (this helps with the configuration of the pytests for example feeding our connection strings.

```python
# conftest.py
import pytest
from flaskapp.app import app as flask_app
from pymongo import MongoClient

@pytest.fixture
def app():
    """Create and configure a new app instance for each test."""
    flask_app.config['TESTING'] = True
    yield flask_app

@pytest.fixture
def client(app):
    """A test client for the app."""
    return app.test_client()

@pytest.fixture
def db(request):
    # Setup: Connect to the MongoDB database
    client = MongoClient("", tls=True, tlsAllowInvalidCertificates=True)
    db = client['test_db']  # Use a separate database for testing
    yield db

@pytest.fixture
def actual_db(request):
    # Setup: Connect to the actual MongoDB database
    client = MongoClient("", tls=True, tlsAllowInvalidCertificates=True)
    actual_db = client['Exprojects']  # Use the actual database
    yield actual_db

```

We created a test client using the flask app configured for the tests. The app_context() is used to set up an application context to interact with the Flask app as if it is handling a request. To not interfere with the main database we also created a mock database to create fake data for testing

```python
import pytest
from flask import json
import sys

sys.path.append('..')

from flaskapp.app import app as flask_app
@pytest.fixture
def client():
    # Set the app to testing mode
    flask_app.config['TESTING'] = True

    # Create a test client using the Flask application configured for testing
    with flask_app.test_client() as testing_client:
        # Establish an application context
        with flask_app.app_context():
            yield testing_client
```

Here are the function names of the tests we carried out and the explanations.

test_registration_existing_user:
This test checks if the registration endpoint /register correctly handles the case where a user attempts to register with an existing username. It expects a status code of 409, indicating a conflict

.

test_user_login_success:
Verifies that the login endpoint /login works as expected by logging in a user with correct credentials and ensuring that a token is returned in the response.

test_user_login_failure: Tests the behaviour of the login endpoint when incorrect credentials are provided. It expects a status code of 401, indicating unauthorised access.

test_allprojects_pagination:

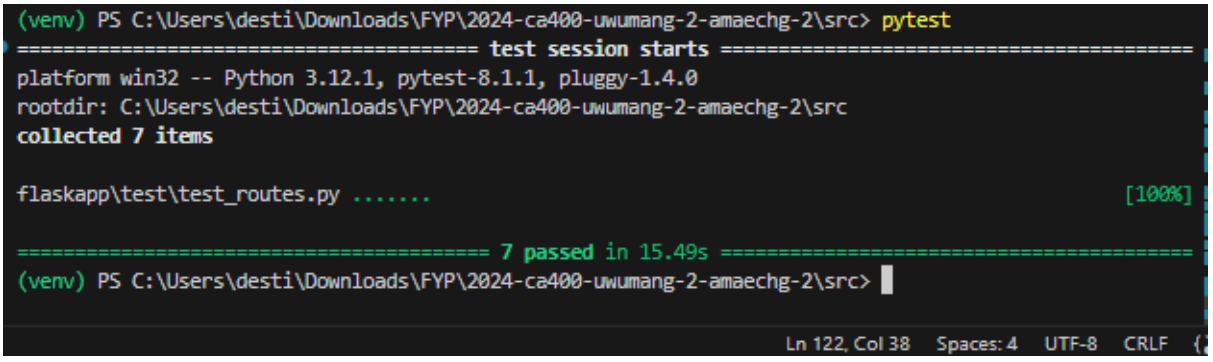
This test verifies the pagination functionality of the /allprojects endpoint. It inserts some test projects into the database, then sends a request with pagination parameters (page and limit). Finally, it checks if the response contains the expected number of projects based on the pagination parameters.

test_get_project_title:
Checks if the endpoint /api/allprojects/{project_id} returns the title of the project with the specified ID. It inserts a test project into the database, sends a request to retrieve the title of that project, and verifies if the response contains the expected project title.

test_check_preferences:
Tests the endpoint /check-preferences to see if a user's preferences exist in the database. It inserts a test user's preferences and then sends a request to check if preferences exist for that user. It expects a response indicating whether preferences exist or not.

test_recommend_projects:
Verifies the functionality of the /recommend endpoint by sending a request with a username and checking if it returns recommended projects based on the user's preferences. It inserts some test projects into the database and sends a request to recommend projects for a user. Finally, it checks if the response contains the expected recommended projects.

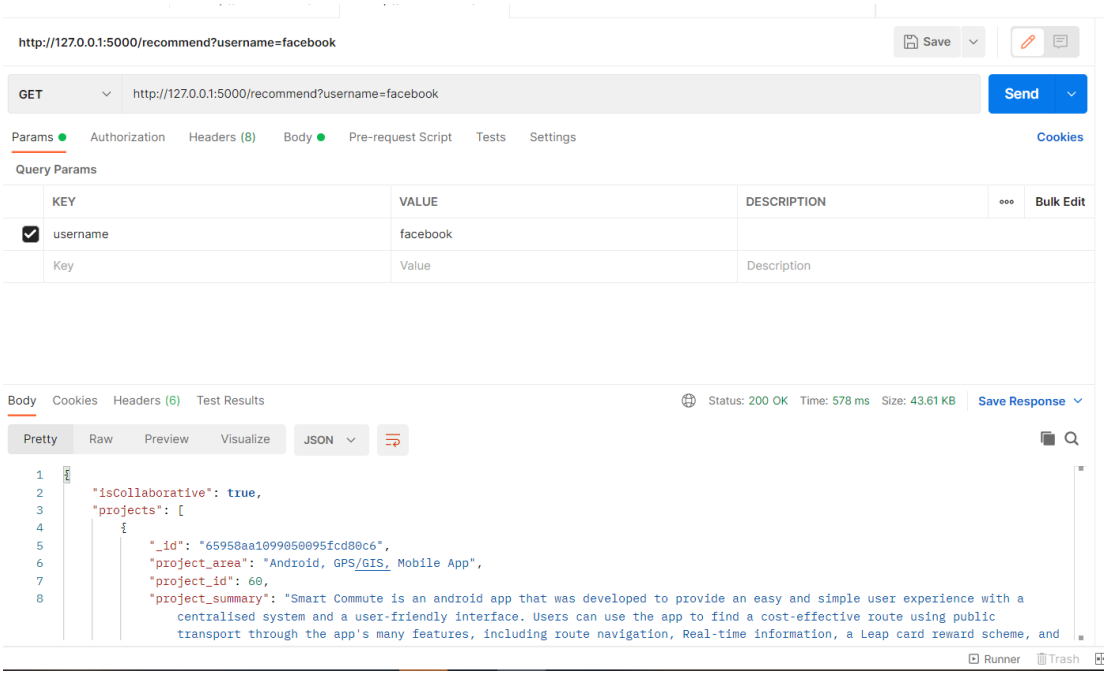
All of the tests passed using the command `pytest`



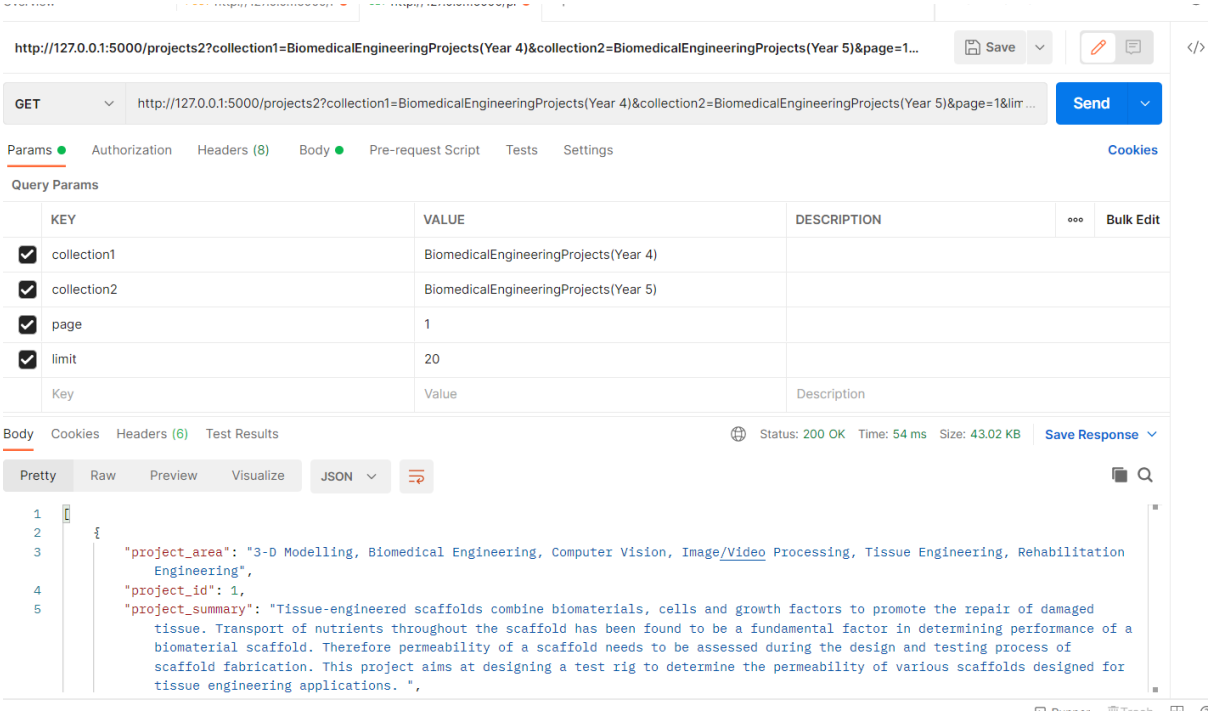
Integration testing using postman

Integration tests verify the interfaces between components against a software design. These tests are used to detect any discrepancies between how different modules or services used by the application interact with each other.
We chose Postman for integration testing and to test our API routes due to its comprehensive set of features that streamline the process of API testing. Postman is renowned for its user-friendly interface, which allows us to share, test, and document APIs efficiently. We particularly appreciated its ability to handle various HTTP requests and responses, facilitating thorough testing of our API endpoints under different scenarios.
Bellow is some pictures of us testing our api routes

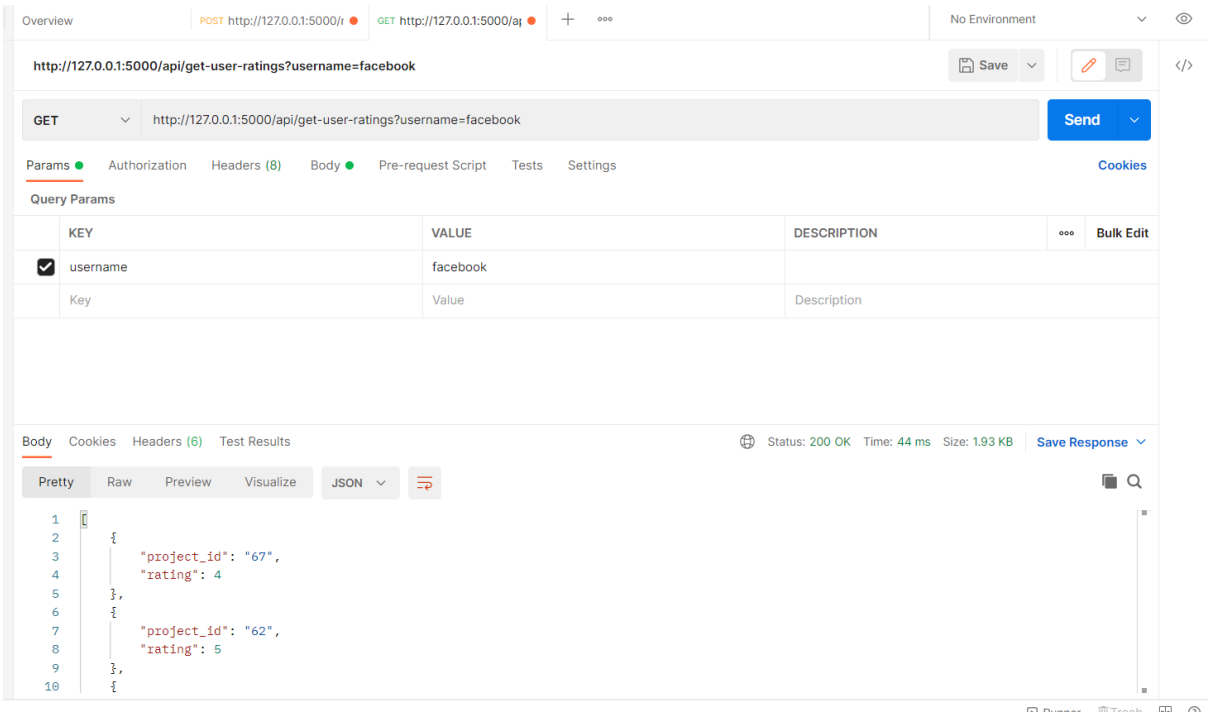
1. Recommendation



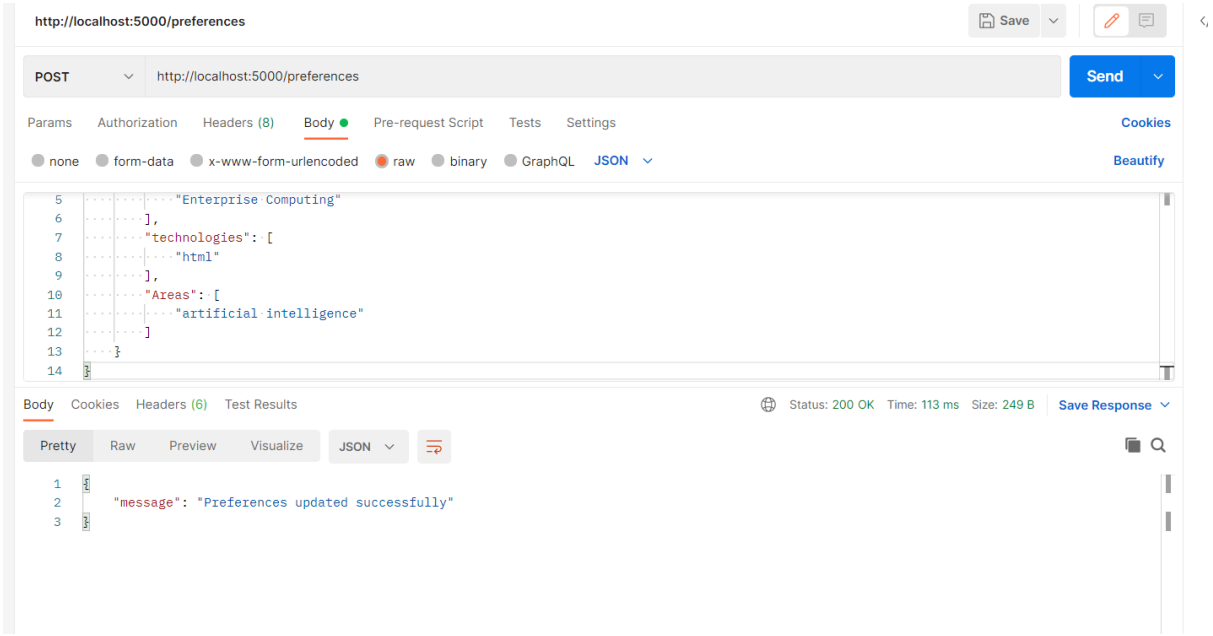
2. Retrieval of two collections and connecting them using pagination



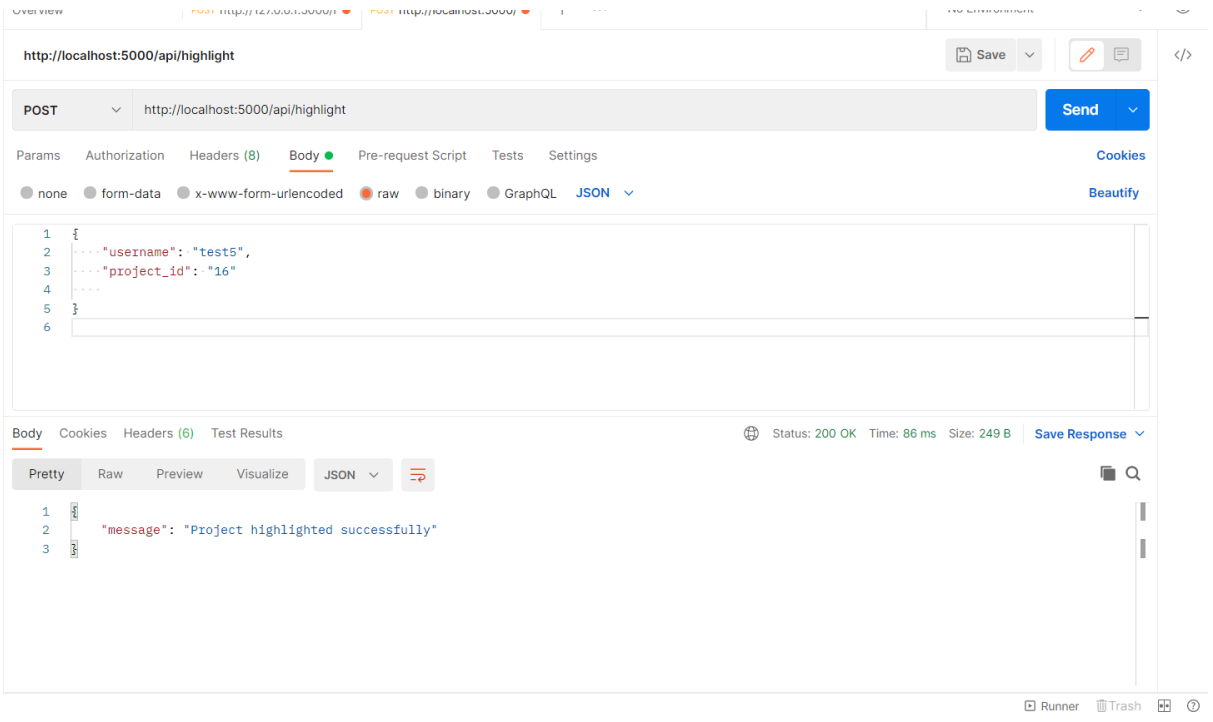
3. Fetching user ratings



4. Submitting preferences



5. Highlighting projects



As you can see from the images the expected body was produced after testing.