

TECHNISCHE HOCHSCHULE INGOLSTADT

Fakultät Elektro- und Informationstechnik

Master's degree AI Engineering of Autonomous Systems

Software Engineering Team Project: Flash Card Web Application

SEMINAR PAPER

Bükülmez Gizem

Supervisor: Prof. Dr. Jürgen Bock

Date: June 30, 2024

Contents

1	Introduction	1
2	System Design	1
3	Development	2
3.1	Role	2
3.2	Flash App	2
3.3	What is Pytest?	2
3.4	Creating the Test File	2
3.4.1	Test Client and User Preparation	3
3.5	Test Functions	4
4	Measuring the Performance of a Flask Application with Load Testing	9
4.1	What is Locust?	9
4.2	Performance Testing of flask Application	9
4.2.1	Test Scenario	9
4.3	Installing the Required Libraries	9
4.4	Creating Locust Test File	10
5	Conclusion	11
6	References	12
7	Appendix	13

1 Introduction

Flash cards have long been used as an effective way to help people learn and remember information through active recall and spaced repetition techniques. The advancement of digital technologies has made it possible to use these flashcards as interactive and fun applications.

Software testing is one of the most critical components of the software development process. A proper testing strategy is necessary to verify that the software works as expected and to detect potential bugs at an early stage. In this paper, the pytest library and the process of testing a flask application will be discussed in detail. It will also explain step by step how a flask application is tested. In this framework, the importance of effective testing strategies to ensure the quality and reliability of flash card applications will be emphasized.

2 System Design

The Flash Card application is designed using a client-server architecture. In this architecture, server-side operations are managed by the Flask framework. Flask performs critical server-side operations such as user authentication, flash card management and game functionality. Data is stored in two JSON files, `user_data.json` containing user information and `data.json` containing flash card details. The client side is built using Bootstrap, HTML and JavaScript to provide a responsive and interactive experience for the user. This structure allows users to interact with flash cards easily and effectively.

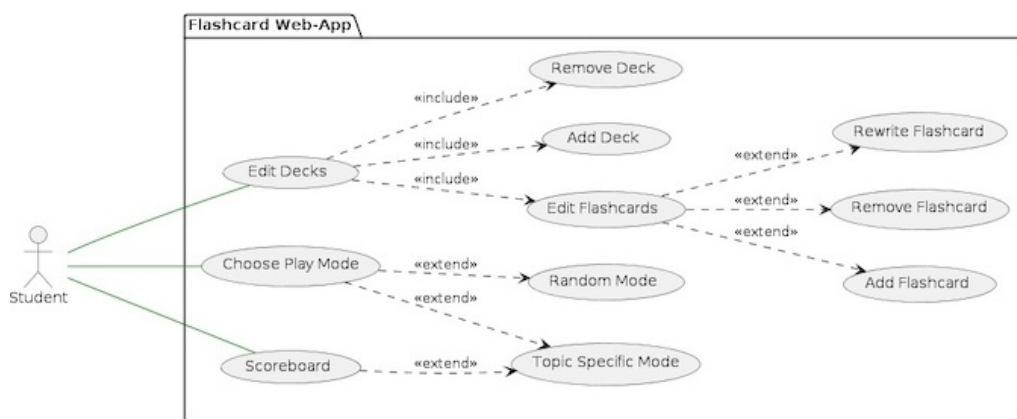


Figure 1: Flashcard Web Application Use Case Diagram

3 Development

3.1 Role

As a Tester, my task was to gain in-depth knowledge about the software architecture and application to ensure that the created software architecture works flawlessly. I created a test case by developing a new test architecture to match this architecture. However, during this process, I encountered some difficulties in the 'flask8' section and encountered pipeline errors. Fortunately, with the changes I made to the test architecture, I was able to overcome these challenges and make the application more reliable and robust.

3.2 Flash App

flask is a micro web framework written in Python. Thanks to its simple and flexible structure, it enables fast development of web applications. flask applications are usually defined in the `app.py` file and serve different endpoints through various routes.

3.3 What is Pytest?

Pytest is a testing framework written in Python programming language. It allows users to easily write tests with its simple, scalable and extensible structure. Pytest automatically discovers and runs your tests. It also allows you to easily manage setup and teardown processes.

3.4 Creating the Test File

In our test file, we will use pytest to test various functions and routes of our flask application. The file containing our tests is usually named `test_*.py` and is located in the application root directory or in a directory called `tests`.

Below you can find a detailed description of a `test_routes.py` file containing the tests of our flask application.

Required Imports First, we import the necessary libraries and modules:

```
1 import pytest
2 from app import app
3 from app.models import User, Card
```

3.4.1 Test Client and User Preparation

Test Client We create a test client to use in our tests. The test client allows us to run our application in test mode:

```
1 @pytest.fixture
2 def client():
3     app.config['TESTING'] = True
4     app.config['WTF_CSRF_ENABLED'] = False
5     with app.test_client() as client:
6         with app.app_context():
7             # setup database and other initializations
8             yield client
```

Here we set `app.config` to test mode and disable CSRF protection. We create a test client with `app.test_client()` and provide the application context with `app.app_context()`.

New User We create a new user to use in our tests:

```
1 @pytest.fixture
2 def new_user():
3     user = User(username='test_user', email='test@example.com')
4     user.set_password('password')
5     users = User.load_users()
6     users[user.username] = user
7     User.save_users(users)
8     return user
```

This function creates a user from the `User` model and loads and saves this user.

Logging in We define a fixture that performs the login process with the new user:

```
1 @pytest.fixture
2 def login(client, new_user):
3     response = client.post('/login', data=dict(
4         username=new_user.username,
5         password='password'
6     ), follow_redirects=True)
7     assert response.status_code == 200
8     return response
```

This fixture logs in with the newly created user and verifies that the login was successful.

3.5 Test Functions

User Registration Test We create a test function that tests the user registration function:

```
1 def test_register(client):
2     response = client.get('/register')
3     assert response.status_code == 200
4     response = client.post('/register', data=dict(
5         username='test_user2',
6         email='test2@example.com',
7         password='password',
8         password2='password'
9     ), follow_redirects=True)
10    assert response.status_code == 200
```

This function sends GET and POST requests to the `/register` route and checks the status code.

Login Test We create a test function that tests the login function:

```
1 def test_login(client, new_user):
2     response = client.get('/login')
3     assert response.status_code == 200
4     response = client.post('/login', data=dict(
5         username='test_user',
6         password='password'
7     ), follow_redirects=True)
8     assert response.status_code == 200
```

This function sends GET and POST requests to the `/login` route and checks if the login was successful.

Logout Test We create a test function that tests the output function after input:

```
1 def test_logout(client, login):
2     response = client.get('/logout', follow_redirects=True)
3     assert response.status_code == 200
```

This function sends a GET request to the `/logout` route and checks if the logout was successful.

All Cards Test We create a test function that tests the route where all cards are listed:

```
1 def test_all_cards(client, login):
2     response = client.get('/all_cards')
3     assert response.status_code == 200
```

This function sends a GET request to the `/all_cards` route and checks if the response is successful.

Home Page Test We create a test function that tests the home page route:

```
1 def test_index(client, login):
2     response = client.get('/')
3     assert response.status_code == 200
```

This function sends a GET request to the home page route and checks if the response is successful.

Post Login Test After logging in, we create a test function that tests the redirected page:

```
1 def test_post_login(client, login):
2     response = client.get('/post_login')
3     assert response.status_code == 200
```

This function sends a GET request to the `/post_login` route and checks if the response is successful.

Post Entry Action Test We create a test function that tests an action performed after logging in:

```
1 def test_post_login_action(client, login):
2     response = client.post('/post_login_action', data=dict(action="
3         start_game"), follow_redirects=True)
4     assert response.status_code == 200
```

This function sends a POST request to the `/post_login_action` route and checks if the response is successful.

Topic Based Initialization Test We create a test function that tests the initialization function based on a specific topic:

```
1 def test_start_by_topic(client, login):
2     response = client.get('/start_by_topic')
3     assert response.status_code == 200
```

This function sends a GET request to the `/start_by_topic` route and checks if the response is successful.

New Card Creation Test We create a test function that tests the function to create a new card:

```
1 def test_new_card(client, login):
2     response = client.get('/cards/new')
3     assert response.status_code == 200
4
5     response = client.post('/cards/new', data=dict(
6         topic='Test Topic',
7         question='Test Question',
8         hint='Test Hint',
9         answer='Test Answer'
10    ), follow_redirects=True)
11    assert response.status_code == 200
12
13    % Check if the card was actually added
14    cards = Card.load_cards()
15    assert any(card.topic == 'Test Topic' and card.question == 'Test
    Question' for card in cards)
```

This function sends GET and POST requests to a new card creation page and checks if the card was created successfully.

Showing Cards Test We create a test function that tests the page where all the cards are listed:

```
1 def test_show_cards(client, login):
2     response = client.get('/all_cards')
3     assert response.status_code == 200
```

This function sends a GET request to the `/all_cards` route and checks if the response is successful.

Game Launch Test by Topic We create a test function that tests the game initialization function based on a specific topic:

```
1 def test_start_game_by_topic(client, login):
2     % First add a card to ensure there's at least one card
3     client.post('/cards/new', data=dict(
4         topic='TestTopic',
5         question='TestQuestion',
6         hint='TestHint',
7         answer='TestAnswer'
8     ))
9     response = client.get('/start_game_by_topic/TestTopic')
10    assert response.status_code == 200
```

This function adds a card before starting a game based on a specific topic and then sends a GET request to the corresponding route.

Test Fetching Card Topic We create a test function that tests the function to fetch cards of a given topic:

```
1 def test_get_card_topic(client, login):
2     response = client.get('/cards/topic/test_topic')
3     assert response.status_code == 200
```

This function sends a GET request to the `/cards/topic/test_topic` route and checks if the response is successful.

Card Editing Test We create a test function that tests an existing card editing function:

```
1 def test_edit_card(client, login):
2     # First, add a card
3     client.post('/cards/new', data=dict(
4         topic='EditTopic',
5         question='EditQuestion',
6         hint='EditHint',
7         answer='EditAnswer'
8     ))
9     # Get the id of the newly added card
10    cards = Card.load_cards()
11    card_id = max(card.id for card in cards)
12    # Edit the card
```

```

13     response = client.post(f'/cards/{card_id}', data=dict(
14         topic='EditedTopic',
15         question='EditedQuestion'
16     ), follow_redirects=True)
17     assert response.status_code == 200
18     # Check if the card was actually edited
19     updated_cards = Card.load_cards()
20     edited_card = next((card for card in updated_cards if card.id ==
21         card_id), None)
22     assert edited_card is not None
23     assert edited_card.topic == 'EditedTopic'
24     assert edited_card.question == 'EditedQuestion'

```

This function edits an existing card and checks if the edit was successful.

Card Delete Test We create a test function that tests the function to delete an existing card:

```

1 def test_delete_card(client, login):
2     # First, add a card
3     client.post('/cards/new', data=dict(
4         topic='DeleteTopic',
5         question='DeleteQuestion',
6         hint='DeleteHint',
7         answer='DeleteAnswer'
8     ))
9     # Get the id of the newly added card
10    cards = Card.load_cards()
11    card_id = max(card.id for card in cards)
12    # Delete the card
13    response = client.post(f'/cards/{card_id}/delete', follow_redirects=
14        True)
15    assert response.status_code == 200
16    # Check if the card was actually deleted
17    updated_cards = Card.load_cards()
18    assert all(card.id != card_id for card in updated_cards)

```

This function deletes an existing card and checks if the deletion was successful.

Scoreboard Test We create a test function that tests the scoreboard sheet:

```
1 def test_scoreboard(client, login):  
2     response = client.get('/scoreboard')  
3     assert response.status_code == 200
```

This function sends a GET request to the /scoreboard route and checks if the response is successful.

4 Measuring the Performance of a Flask Application with Load Testing

Performance testing is an important type of testing used to analyze how an application behaves under certain loads. These tests are performed to determine how the application performs under high loads, response times and how resilient the system is. Locust is a popular performance testing tool used for this purpose. In this paper, we will cover how to performance test a flask application using Locust and create an example scenario step by step.

4.1 What is Locust?

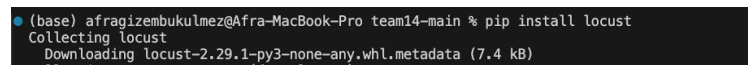
Locust is an open source performance testing tool used to perform load testing of a web application. By simulating the user's behavior, Locust puts a certain load on the application for a certain period of time and evaluates the performance of the application.

4.2 Performance Testing of flask Application

4.2.1 Test Scenario

In this performance test, we will simulate users registering, logging in and adding a new card. These steps will simulate the basic interactions of a user and evaluate the performance of the application under these actions.

4.3 Installing the Required Libraries



```
(base) afragizembukulmez@Afra-MacBook-Pro team14-main % pip install locust  
Collecting locust  
  Downloading locust-2.29.1-py3-none-any.whl.metadata (7.4 kB)
```

Figure 2: Installing locust

4.4 Creating Locust Test File

In order to test performance with Locust, we create a test file. In this file, we will write code to simulate user behavior. Here is a sample Locust test file:

```
1 from locust import HttpUser, task, between
2
3 class UserBehavior(HttpUser):
4     wait_time = between(5, 9) # time between requests
5     host = "http://127.0.0.1:5000" # base URL
6
7     @task
8     def register_login_add_card(self):
9         # Register a new user
10        response = self.client.post('/register', data=dict(
11            username='test_user3',
12            email='test3@example.com',
13            password='password',
14            password2='password'
15        ))
16
17        # check registration was successful by looking at response status
18        if response.status_code == 200:
19            # Log in with the new user credentials
20            login_response = self.client.post('/login', data=dict(
21                username='test_user3',
22                password='password'
23            ))
24
25            # if login was successful
26            if login_response.status_code == 200:
27                # add new card
28                self.client.post("/cards/new", data=dict(
29                    topic="Science",
30                    question="What is photosynthesis?",
31                    hint="Hint: It involves plants.",
32                    answer="The process by which plants convert light
33                    energy into chemical energy."
34                ))
```

Description of Test Scenario - `wait_time`: Sets the waiting time between requests. This is a random wait time between 5 and 9 seconds. - `host`: The base URL of the application to be tested. - `register_login_add_card method`: A task that simulates user behavior. This task first registers a new user, then logs in with that user and finally adds a new card.

Running the Test You can use the following command to run the Locust test:

```
locust -f locustfile.py
```

This command will launch the Locust web interface. You can access the Locust interface and start the test by going to `http://localhost:8089` in your web browser.

Evaluation of Test Results In the Locust interface, you can start the test by specifying the number of users and the test duration. During and after the test, Locust provides you with various metrics. These metrics include:

- Number of requests
- Number of successful requests
- Error rate
- Average response time
- Maximum response time
- Median response time

These metrics help you assess the performance of your application and identify potential performance bottlenecks.

5 Conclusion

In this article, we focused on testing Flask applications from both software testing and performance testing perspectives. First, we went through step-by-step how to test Flask applications using pytest. We created test files, defined the necessary fixtures, and wrote tests of various functions and routes. This process highlighted the use of pytest, a powerful tool that is critical for improving quality and reliability in the software development process.

Next, we learned how to perform performance tests of Flask applications using Locust. Performance tests are vital to understand how the application performs under high load and make necessary improvements. Locust's simple and flexible structure made it an ideal tool to run such tests effectively. By regularly running performance tests, we can ensure that the application always performs at its best.

6 References

- 1 Pytest Documentation: <https://docs.pytest.org/en/8.2.x/>
- 2 Locust Documentation: <https://docs.locust.io/en/stable/>
- 3 AI Tools - Chat-GPT, I ASK AI: <https://iask.ai/>

7 Appendix

Team Discussions

20.05.2024 - Overview

Basic structure of app.

Tools to be used.

Work flow and use cases.

29.05.2024 - Pre-Implementation Phase-1

Creation of task-based system and continuous monitoring in teams. Prioritization of tasks and sub-grouping to achieve a fast-paced environment. Tools to be used and how to proceed with these:

- Frontend - Angular
- Backend - Django
- Database - MYSQL
- CI/CD and Dockerization

Coding standards to be followed via documentation.

01.06.2024 - Implementation Phase-1

Change in use of tools due to complexity for such a simple app structure:

Frontend - HTML/CSS/JavaScript

Backend - Flask

Database - Text File Frontend - How Chat-GPT is very useful in creating bootstrap and playing with JS for creating forms and interaction with UI - we have a current structure of all forms by now.

Backend - Flask is less complex compared to Django and for such a small-scale application it's better to go with what is simple and easy to understand - implemented some basic routes such as login/home pages.

Database - Rather than using MYSQL we can go for simple text files, as we don't have complex multiple tables to deal with primary key/foreign key. It is easy to manipulate and the learning curve is reduced.

07.06.2024 - Implementation Phase-2

Change in use of tools:

- Database - JSON File

Database - Changed the use of Text file into JSON file which is easy to manipulate and has multiple key and value pairs which helps understand and parse easily in any component of the app - logic for two .json files for users and cards dataset.

Implemented a basic working app with all initial features.

Testing started parallelly on modules - unit testing.

Have a sample implemented overview on CI/CD pipelines for pytest in GitLab.

Parallelly working on Algorithm for display of cards.

12.06.2024 - Implementation Phase-3

Have a working prototype of a fully-fledged app.

Containerized and tested the app locally on Dockerhub.

Looking for Improvements and further proceedings.

19.06.2024 - Implementation Phase-4

Check for load testing for easy user experience.

Integrate the initial simple algorithm into the backend.

Working for further optimization and monitoring.

Checking for the deployment of the app on any open-source cloud platforms.

Checking for more complex algorithms to display the most prioritized card.

25.06.2024 - Implementation Phase-4

Stick to the basic algorithm - which works better than the complex one.

Tried to deploy the app on Heroku cloud - facing some difficulty in integration.

Looking for future works and improvements.