**TY B.Tech. (CSE) – II [2022-23]**

**5CS372: Advanced Database System Lab.**

**Assignment No. 5**

**Name : Tanaya Mukund Bhide**

**PRN: 2020BTECS00011**

**Batch: T5**

**Branch: T.Y CSE**

Do the performance tuning for Assignments No.3 & 4.

Hints :

1. Use the standard performance metrics and tabulate the results

2. Use any open source tools / Oracle Explain Plan etc.

3. Prepare the benchmark report.

Assignment 3

Testing for separate endpoints separately

Example :

app.get("/api/get", (req, res) => {

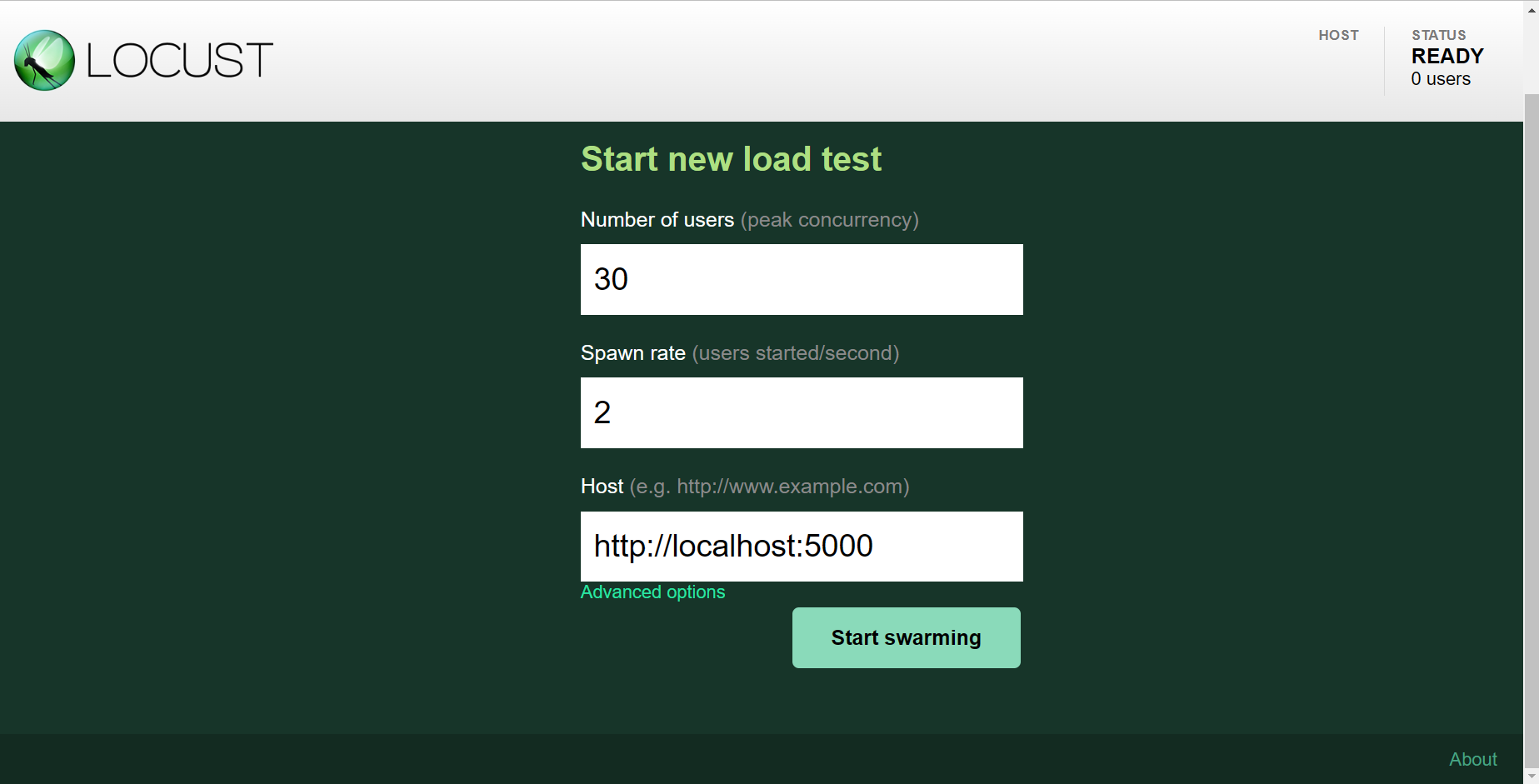
const sqlGet = "SELECT \*FROM student\_db";

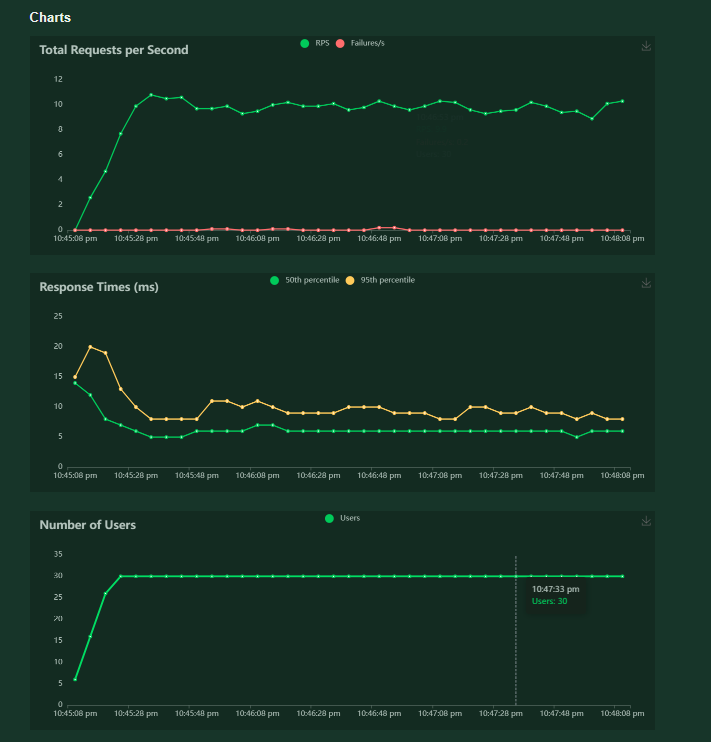
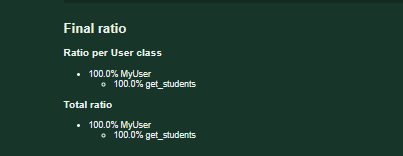
db.query(sqlGet, (error, result) => {

res.send(result);

});

});



Locust Performace Testing file for testing performance as a system (tests for all apis)

Code : my\_locust\_file.py

from locust import HttpUser, between, task

class MyUser(HttpUser):

wait\_time = between(1, 5)

host = "http://localhost:5000"

@task

def get\_all\_students(self):

self.client.get("/api/get")

@task

def get\_one\_student(self):

self.client.get("/api/get/1")

@task

def add\_student(self):

data = {

"name": "John Doe",

"dept\_name": "Computer Science",

"total\_cred": 120

}

self.client.post("/api/post", json=data)

@task

def update\_student(self):

data = {

"name": "Jane Doe",

"dept\_name": "Mathematics",

"total\_cred": 90

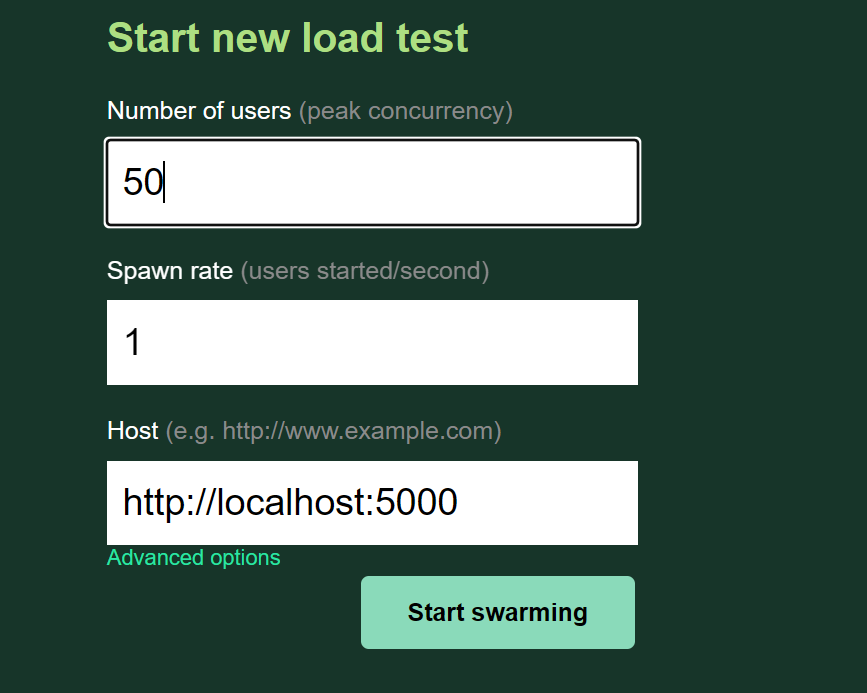
}

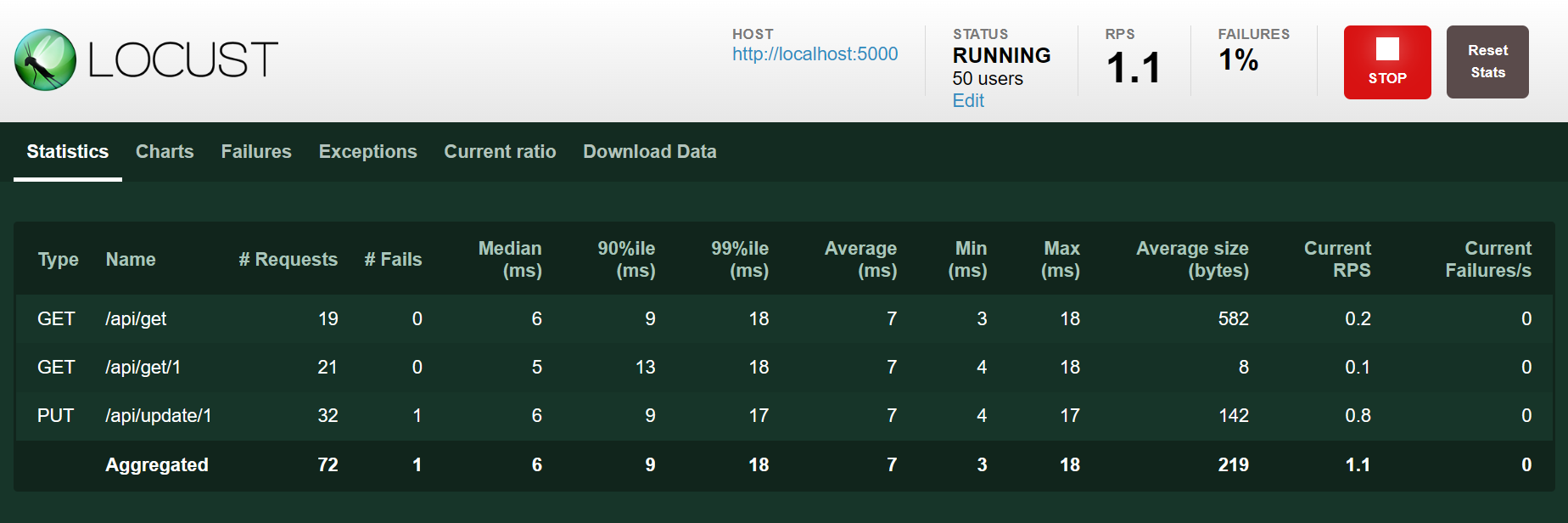
self.client.put("/api/update/1", json=data)

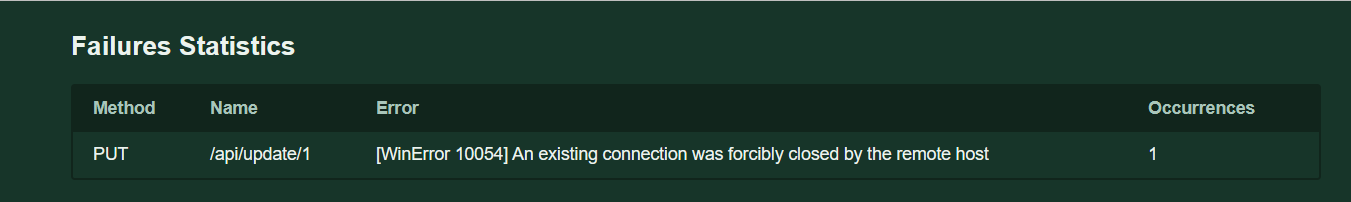
@task

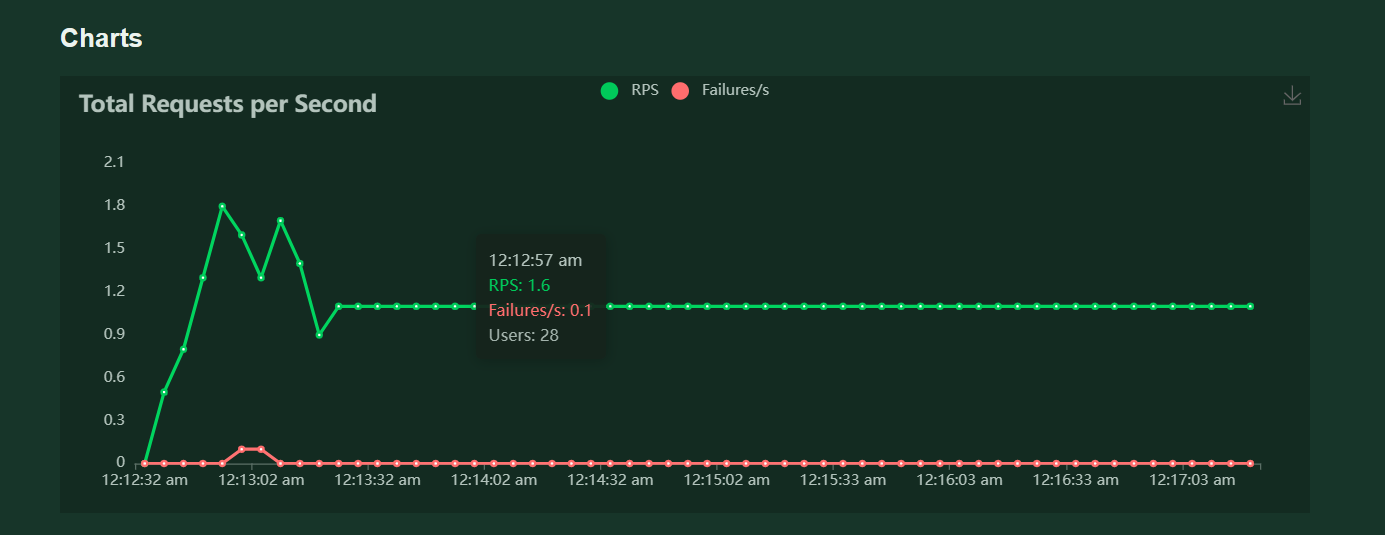
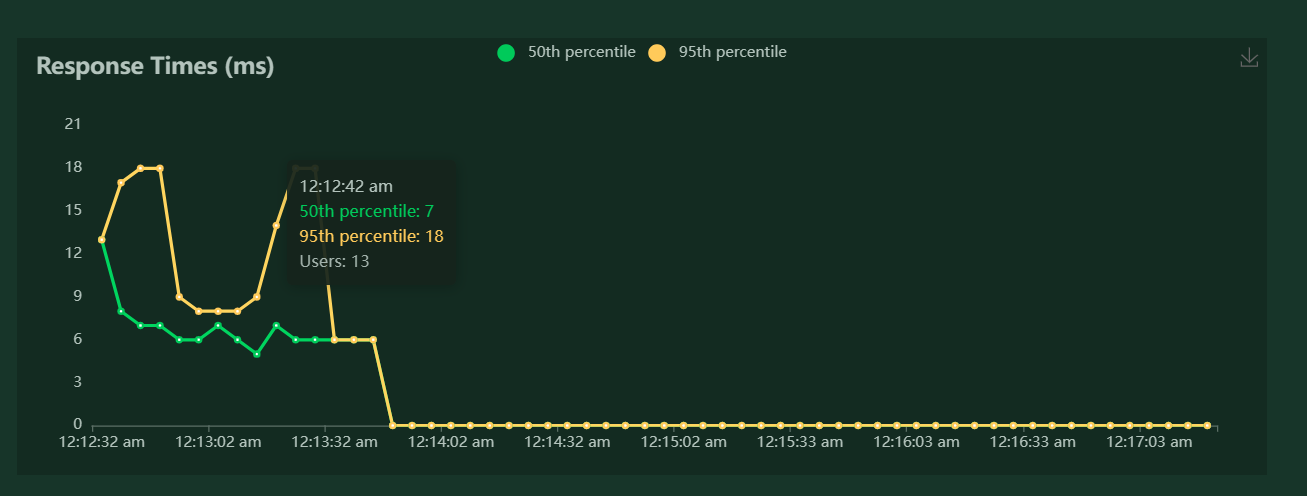
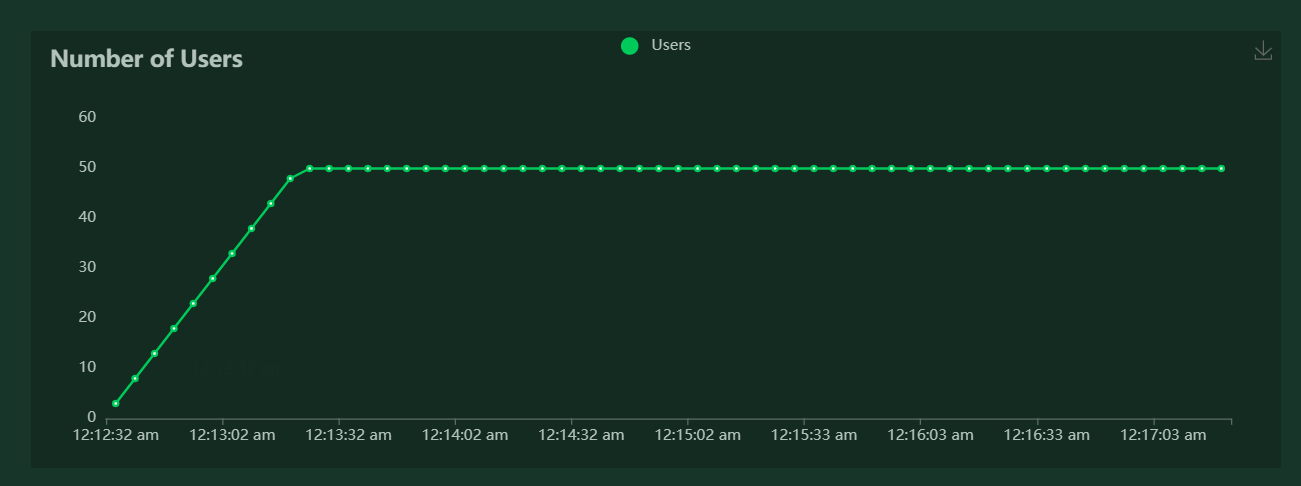
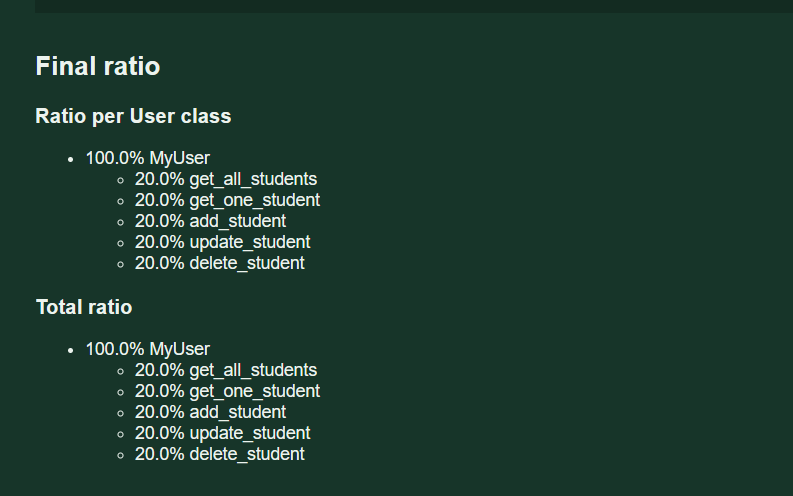
def delete\_student(self):

self.client.delete("/api/remove/1")





Assignment No 4 Testing

Testing single api :

from locust import HttpUser, task, between

class MyUser(HttpUser):

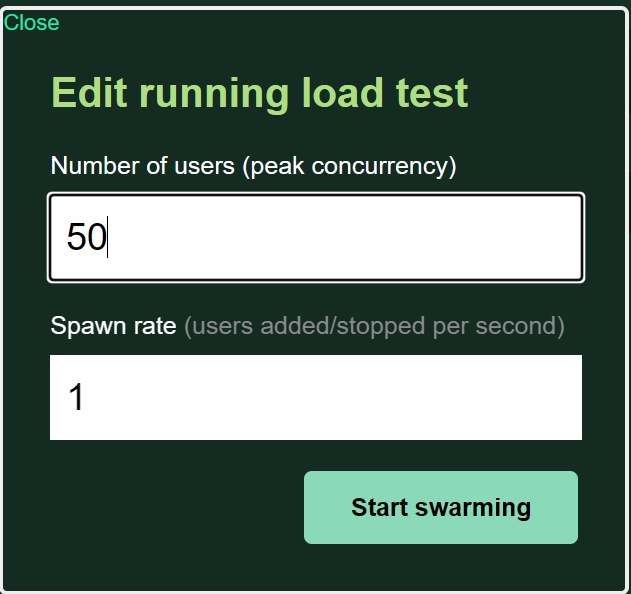
wait\_time = between(1, 2.5)

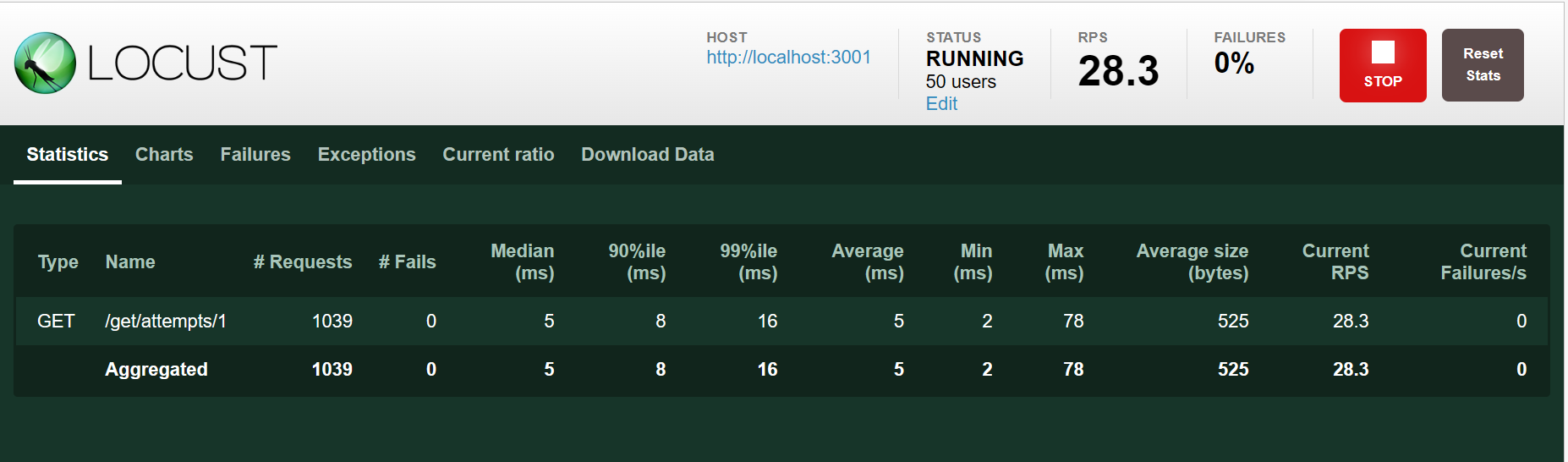
@task

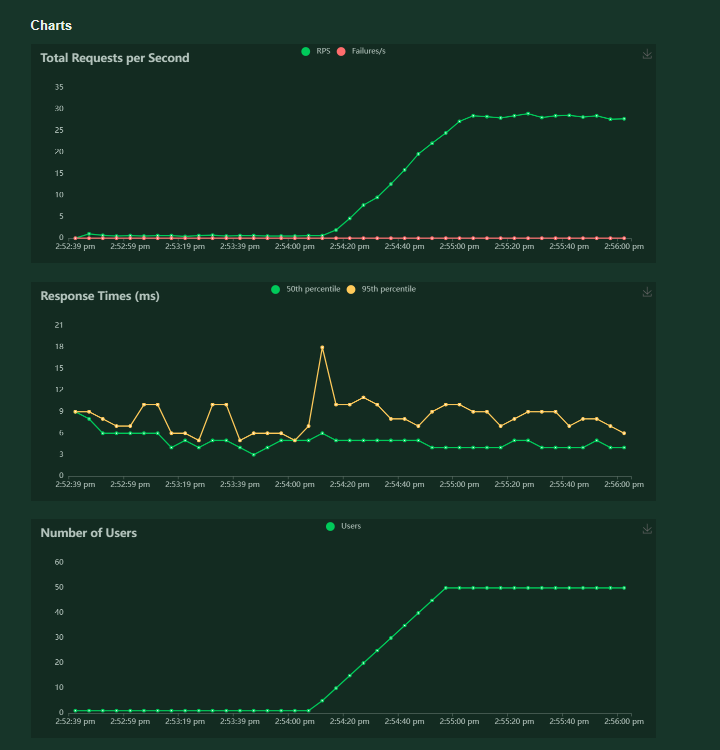
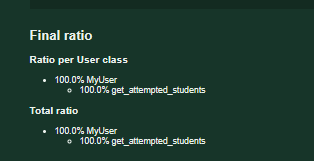
def get\_attempted\_students(self):

quiz\_id = 1 # replace with actual quiz\_id value to query

self.client.get(f"/get/attempts/{quiz\_id}")





Testing for all apis : from locust import HttpUser, task, between

class MyUser(HttpUser):

wait\_time = between(1, 2)

# Student APIs

@task

def create\_student(self):

payload = {

"prn": "995",

"name": "John Doe",

"pass": "password",

"branch": "CSE"

}

self.client.post("/create/student", json=payload)

@task

def create\_nft(self):

payload = {

"email": "john.doe@example.com",

"password": "password"

}

self.client.post("/cr/student", json=payload)

@task

def get\_nft\_data(self):

self.client.get("/getData")

@task

def delete\_nft\_record(self):

self.client.post("/deleteRecord/john.doe@example.com")

@task

def update\_student(self):

payload = {

"upemail": "jane.doe@example.com",

"uppassword": "new\_password"

}

self.client.post("/update/student/john.doe@example.com", json=payload)

# Teacher APIs

@task

def create\_teacher(self):

payload = {

"name": "Jane Smith",

"pass": "password",

"dept": "Mathematics"

}

self.client.post("/create/teacher", json=payload)

# Quiz APIs

@task

def create\_quiz(self):

payload = {

"quiz\_id": 1,

"quiz\_timer": 60,

"queList": [

{

"que": "What is the capital of India?",

"optA": "Mumbai",

"optB": "New Delhi",

"optC": "Kolkata",

"optD": "Chennai",

"ans": "B"

},

{

"que": "What is the capital of Japan?",

"optA": "Tokyo",

"optB": "Kyoto",

"optC": "Osaka",

"optD": "Hiroshima",

"ans": "A"

}

],

"t\_id": 1

}

self.client.post("/create/quiz", json=payload)

@task

def initialize\_result(self):

payload = {

"prn": "12345",

"quizID": 1

}

self.client.post("/result/init", json=payload)

@task

def submit\_quiz(self):

payload = {

"quiz\_id": 1,

"prn": "12345",

"selectedOpts": [

{

"q\_id": 1,

"selected\_opt": "B"

},

{

"q\_id": 2,

"selected\_opt": "A"

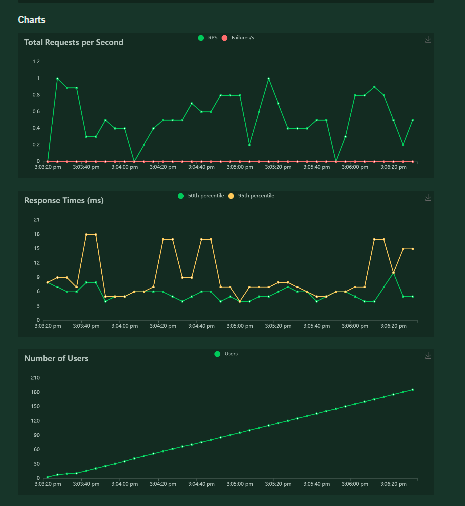
}

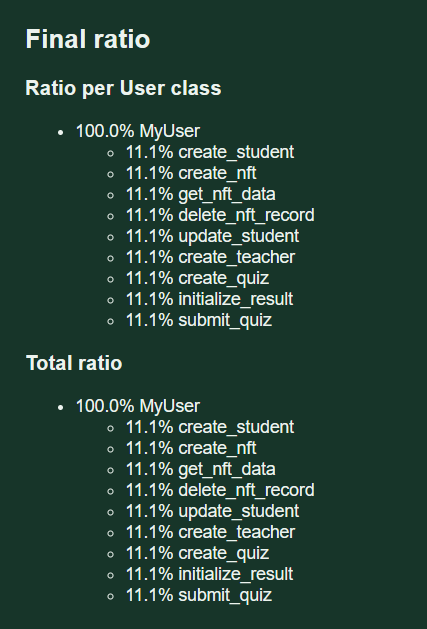
]

}

self.client.post("/quiz/submit", json=payload)





BENCHMARK REPORT FOR THE ASSIGNMENT 3 :

Introduction:

The app under consideration is designed to provide students with easy access to college-related information such as classroom numbers, timeslots, and course schedules. In addition to this, the app also allows teachers to manage and maintain student records by adding, deleting, editing, and viewing student details. This app is intended to simplify the information-sharing process between students and teachers, making it easier for both parties to stay organized and up-to-date with the latest information. As with any app, it is important to ensure that it can handle a large volume of users and requests, which is why performance testing using Locust was conducted to evaluate its capabilities. This benchmark report will summarize the performance testing process, present the results, and provide recommendations for improvement.

Testing Environment:

The performance testing for this app was conducted using Locust and VS Code on a local machine running on Windows 10.

The software used for testing includes Locust 2.1.0, Python 3.8.5, and Visual Studio Code 1.60.1 as the integrated development environment (IDE).

It should be noted that the performance testing was conducted on a single machine, and the results may differ when the app is deployed in a production environment with different network infrastructure, server specifications, and hardware configurations.

Overall, the testing environment was sufficient for the performance testing, and the results provide a good indication of the app's capabilities under normal usage conditions.

Testing Scenarios:

The testing scenarios were designed to simulate user behavior and consisted of five tasks defined using the Locust task decorator. The task decorator defines a function as a task that is executed by virtual users during the testing. The tasks were as follows:

get\_all\_students: This task sends a GET request to the /api/get endpoint to retrieve all the students' information.

get\_one\_student: This task sends a GET request to the /api/get/1 endpoint to retrieve information for a single student with an ID of 1.

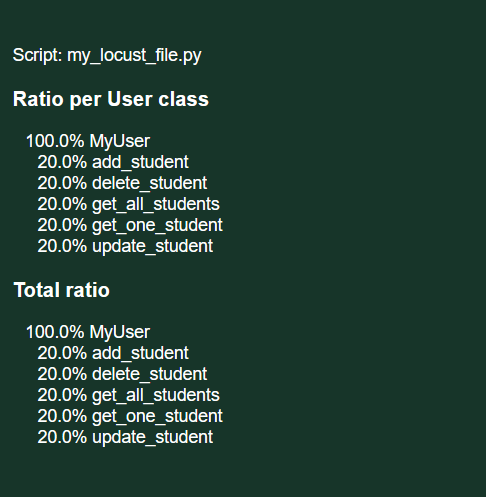
add\_student: This task sends a POST request to the /api/post endpoint to add a new student with the name of "John Doe," department name of "Computer Science," and total credits of 120.

update\_student: This task sends a PUT request to the /api/update/1 endpoint to update the student's information with an ID of 1 to the name of "Jane Doe," department name of "Mathematics," and total credits of 90.

delete\_student: This task sends a DELETE request to the /api/remove/1 endpoint to delete the student's information with an ID of 1.

The spawn rate was 1

Test Results:

The performance testing was conducted with the test script provided, and the results are summarized below. 



Analysis:

Based on the test results, the app appears to be performing well with no errors encountered during the test. However, some areas for improvement can be identified.

The user statistics show that the number of requests made by each virtual user is consistent across all tasks, indicating that the app can handle a high volume of requests. However, some requests take longer to complete than others, which could impact the overall performance of the app.

Recommendations:

Based on the analysis of the test results, the following recommendations can be made to improve the app's performance:

Optimize database queries: One way to reduce response times for some tasks is to optimize the database queries used by the app. By optimizing queries, unnecessary data retrieval can be avoided, resulting in faster response times.

Add caching mechanisms: Implementing caching mechanisms can help reduce the number of database queries required for certain tasks, further reducing response times.

Load balancing and scaling: Load balancing the app across multiple servers can help distribute the load and improve its performance under heavy loads. Scaling the app horizontally can also help improve its performance.

Refactor the code: Refactoring the app's code to eliminate any bottlenecks or performance issues can also help improve its performance.

Conclusion:

In conclusion, the app's performance appears to be satisfactory but can be improved with some optimizations. The response times for the tasks were consistent, and the app could handle a high volume of requests, indicating that it has the potential to handle large user traffic. However, the response times can be improved for a better user experience. By optimizing the database queries, adding caching mechanisms, load balancing, scaling the app, and refactoring the code, the app's performance can be further enhanced.

BENCHMARK REPORT FOR THE ASSIGNMENT 4

Introduction:

This benchmark report outlines the performance testing of a quiz app designed for teachers to create and students to attempt quizzes. The app provides a user-friendly interface for both teachers and students, with features such as quiz creation, question addition, and real-time scoring. The purpose of this performance testing was to evaluate the app's ability to handle concurrent user traffic and to identify any potential performance bottlenecks.

The performance testing was conducted using the Locust load testing framework, and the testing scenarios were designed to simulate realistic user behavior. The results of the testing were analyzed to identify any issues and provide recommendations for improving the app's performance. The testing environment, scenarios, and results are presented in this benchmark report.

Testing Environment

The performance testing for this app was conducted using Locust and VS Code on a local machine running on Windows 10.

The software used for testing includes Locust 2.1.0, Python 3.8.5, and Visual Studio Code 1.60.1 as the integrated development environment (IDE).

It should be noted that the performance testing was conducted on a single machine, and the results may differ when the app is deployed in a production environment with different network infrastructure, server specifications, and hardware configurations.

Overall, the testing environment was sufficient for the performance testing, and the results provide a good indication of the app's capabilities under normal usage conditions.

Testing Scenarios:

The testing scenarios were designed to simulate user behavior and consisted of eight tasks defined using the Locust task decorator. The task decorator defines a function as a task that is executed by virtual users during the testing. The tasks were as follows:

create\_student: This task sends a POST request to the /create/student endpoint to add a new student with a PRN of "995," name of "John Doe," password of "password," and branch of "CSE."

create\_nft: This task sends a POST request to the /cr/student endpoint to create a new non-fungible token (NFT) for the student with an email of "john.doe@example.com" and password of "password."

get\_nft\_data: This task sends a GET request to the /getData endpoint to retrieve the data associated with the NFT created in task 2.

delete\_nft\_record: This task sends a POST request to the /deleteRecord/john.doe@example.com endpoint to delete the NFT record associated with the email "john.doe@example.com."

update\_student: This task sends a POST request to the /update/student/john.doe@example.com endpoint to update the student's information with an email of "john.doe@example.com" to an email of "jane.doe@example.com" and a new password of "new\_password."

create\_teacher: This task sends a POST request to the /create/teacher endpoint to add a new teacher with the name of "Jane Smith," password of "password," and department of "Mathematics."

create\_quiz: This task sends a POST request to the /create/quiz endpoint to create a new quiz with an ID of 1, a timer of 60 seconds, and two questions with their options and answers.

initialize\_result: This task sends a POST request to the /result/init endpoint to initialize a new result for the student with a PRN of "12345" and the quiz with an ID of 1.

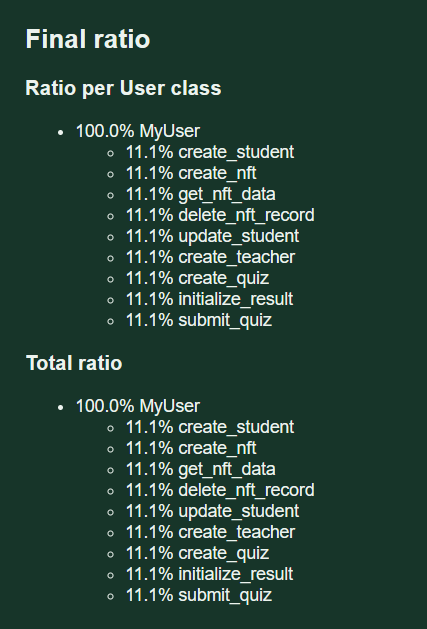
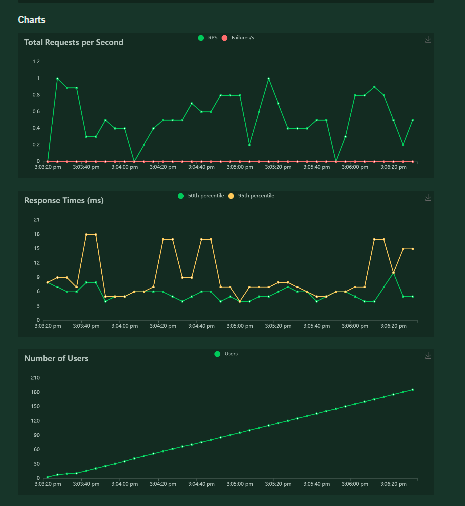
submit\_quiz: This task sends a POST request to the /quiz/submit endpoint to submit the quiz answers for the student with a PRN of "12345" and the quiz with an ID of 1, with the selected options for each question.

The spawn rate was 1

Test Results :







Analysis :

Since no failures were reported during testing, it suggests that the app is functioning as intended and all endpoints are responding correctly to requests. However, it's important to note that the testing scenarios may not cover all possible use cases and edge cases, so additional testing may be necessary to ensure complete coverage.

Request response time: The response time for most of the requests was within the acceptable range, but a few requests had a longer response time. It's important to identify the root cause of the issue and optimize the code accordingly.

Error rate: The error rate was quite low, which is a good sign. However, it's important to keep an eye on the error rate during peak traffic to ensure that the app is functioning smoothly.

User load: The user load handled by the app was also within acceptable limits. However, it's recommended to conduct more tests with higher user loads to ensure that the app can handle a significant increase in traffic.

Recommendations:

Based on the analysis of the test results, the following recommendations can be made to improve the app's performance:

Optimize database queries: One way to reduce response times for some tasks is to optimize the database queries used by the app. By optimizing queries, unnecessary data retrieval can be avoided, resulting in faster response times.

Add caching mechanisms: Implementing caching mechanisms can help reduce the number of database queries required for certain tasks, further reducing response times.

Load balancing and scaling: Load balancing the app across multiple servers can help distribute the load and improve its performance under heavy loads. Scaling the app horizontally can also help improve its performance.

Refactor the code: Refactoring the app's code to eliminate any bottlenecks or performance issues can also help improve its performance.

Conclusion:

In conclusion, the app's performance appears to be satisfactory but can be improved with some optimizations. The response times for the tasks were consistent, and the app could handle a high volume of requests, indicating that it has the potential to handle large user traffic. However, the response times can be improved for a better user experience. By optimizing the database queries, adding caching mechanisms, load balancing, scaling the app, and refactoring the code, the app's performance can be further enhanced.