LAB 2 - MONOLITHIC ARCHITECTURE

Monolith Analogy: The Great Indian Fest Combo

Imagine a college fest where:

1. Registration desk
2. Event listing
3. Checkout/payment
4. My events dashboard

are all handled by **one giant counter** (one team). If one part of the counter breaks, everything goes down.

That's a **Monolithic Application.**

Advantages of Monolithic Applications: Why We Love the Fest "All-in-One Counter"

1. Simple and Straightforward

Building a monolith is like running a college fest with **one big main counter**. Registration, event listing, "my events", and checkout are all handled from the same place. It's easy to set up, easy to understand, and perfect when you want something working quickly without too much complexity. Great for beginners and small teams!

2. Everything Works Together Smoothly

In a fest, things feel faster when the registration desk and the event desk are right next to each other. Similarly, in a monolithic app, all modules – UI, business logic, and database; are tightly integrated and communicate directly inside the same process, which makes development simpler and avoids the extra overhead of network calls or APIs.

3. Easier Debugging and Testing

If something goes wrong at the fest, you know where to look – everything is in one place. The same with monoliths: since all the code is in one project, debugging is usually easier, logs are centralised, and testing a feature doesn't require setting up multiple services.

4. Faster Development for Small Teams

One team can build the entire fest system end-to-end, from login to checkout. Monoliths allow teams to ship faster in the early stages because there are fewer moving parts and less infrastructure to manage.

5. Simple Deployment

Deploying a monolith is like setting up one stall for the fest; just start it and it runs. You don't need separate deployments, separate environments, or service discovery. One server process and you're good to go.

Disadvantages of Monolithic Applications: When the Fest System Becomes a Nightmare

1. Scaling Struggles

Imagine your fest becomes famous and suddenly 5000 students show up. If your entire fest system is one counter, it becomes overwhelmed. Similarly, as a monolithic app grows, scaling becomes harder because you often need to scale the entire application even if only one feature is under heavy load.

2. Deployment Drama

Want to change just the event fee display or tweak the checkout calculation? In a monolith, you can't redeploy only that part – you must redeploy the **entire application**. It's like shutting down the whole fest just to replace one signboard.

3. Hard to Customize and Split Features

What if the fest wants to upgrade only "checkout" or add a new "payment gateway"? In a monolith, the components are not cleanly separated, so changes in one part can affect others. It's an "all-or-nothing" setup.

4. Maintenance Gets Messy Over Time

At the beginning, it's manageable. But as more developers add features, everything becomes intertwined. Imagine 15 organisers updating the same fest counter simultaneously. In monoliths, merging code, managing dependencies, and maintaining stability become harder as the codebase grows.

5. Single Point of Failure

This is the biggest issue and the main focus of this lab. If the fest counter crashes, *everything stops*. Similarly, if one part of a monolith fails, the entire application can go down because all modules share the same runtime and deployment unit.

The Verdict: One Counter or Multiple Stalls?

Monolithic applications are like running a college fest with **one big all-in-one counter**; simple, quick, and perfect for small crowds. But as the fest grows and the crowd increases, the better solution is to use multiple stalls (microservices), where registration, events, and checkout can run independently. That way, scaling, maintenance, and reliability improve dramatically.

Why Is Our Fest App a Monolithic Application?

In this setup:

- All features (login, events, registrations, my-events, checkout) run in a **single FastAPI application**
- UI, business logic, and database access exist in the same codebase
- The application is tightly coupled and deployed as **one unit**
- A bug in one feature can affect the **entire system**, which we will demonstrate using intentional crashes and load testing

PART 1: Setup & Run

1. Create folder with your SRN

```
mkdir PESXUGXXCSXXX
```

2. Enter your folder using

```
> cd PESXUGXXCSXXX
\PESXUGXXCSXXX>
```

3. Unzip the provided lab into this folder, your path should look like: **PESXUGXXCSXXX/CC_LAB2/..**

4. Create virtual environment

Mac/Linux

python3 -m venv .venv
source .venv/bin/activate

Windows

python -m venv .venv
.\.venv\Scripts\activate

```
\PES1UG23CSXXX>python -m venv .venv
```

```
\PES1UG23CSXXX>.\.venv\Scripts\activate
```

Install the required dependencies

Command: *pip install -r requirements.txt*

```
pip install -r requirements.txt
```

Initialize database

```
python insert_events.py
```

```
>>
✅ Events inserted successfully!
```

Running insert_events.py populates fest.db with sample event records so the app has data to display on the Events page.

Run the server

Command: *uvicorn main:app –reload*

```
uvicorn main:app --reload
```

```
INFO:     Uvicorn running on http://127.0.0.1:8000 (Press CTRL+C to quit)
INFO:     Started reloader process [32100] using StatReload
INFO:     Started server process [39480]
INFO:     Waiting for application startup.
INFO:     Application startup complete.
```

If uvicorn is not recognised on Windows, use this instead:

Command: *python -m uvicorn main:app –reload*

```
python -m uvicorn main:app --reload
```

PART 2: Use the Application

In main.py you have to add your SRN on line number 8.

Please make sure your SRNs are visible in all screenshots.

Visit these pages (you have to add the endpoints manually in your browser, just clicking the url that shows up in your browser will lead to “Page not found”):

1. Register – create an account for yourself

   Let the username and password be your SRN (PESXUGXXCSXXX)

   http://localhost:8000/register

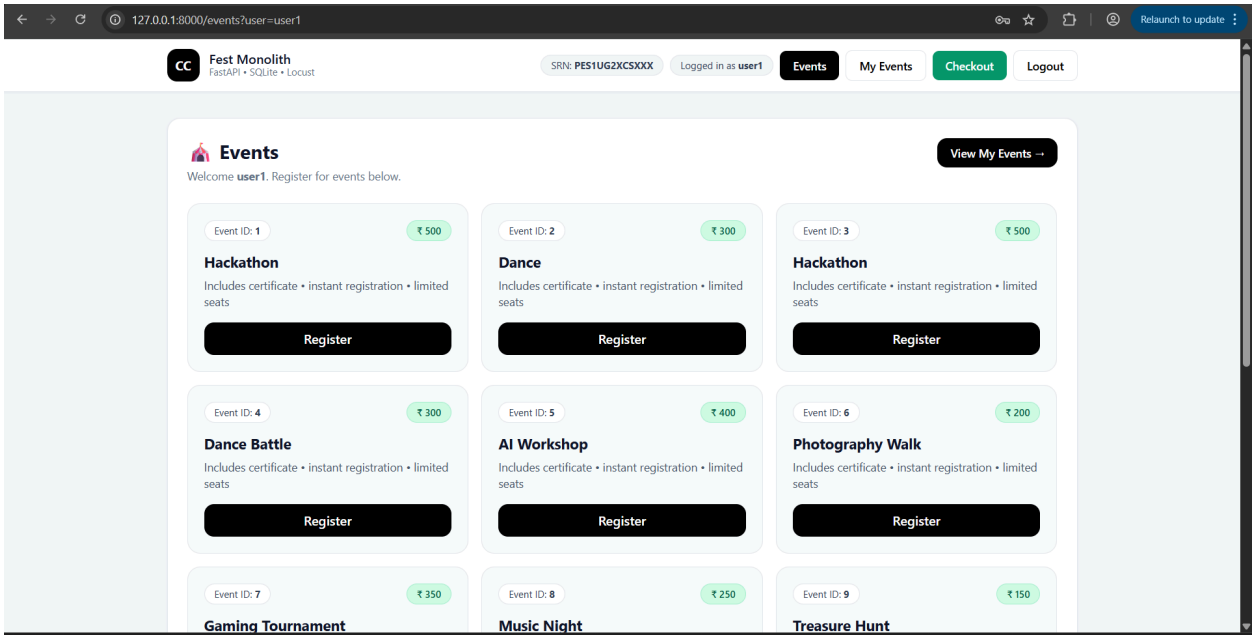
2. Login – log in with the same credentials you’ve just created

   http://localhost:8000/login

3. Events

   http://localhost:8000/events

   (You can access the Events page only after logging in. 😂)

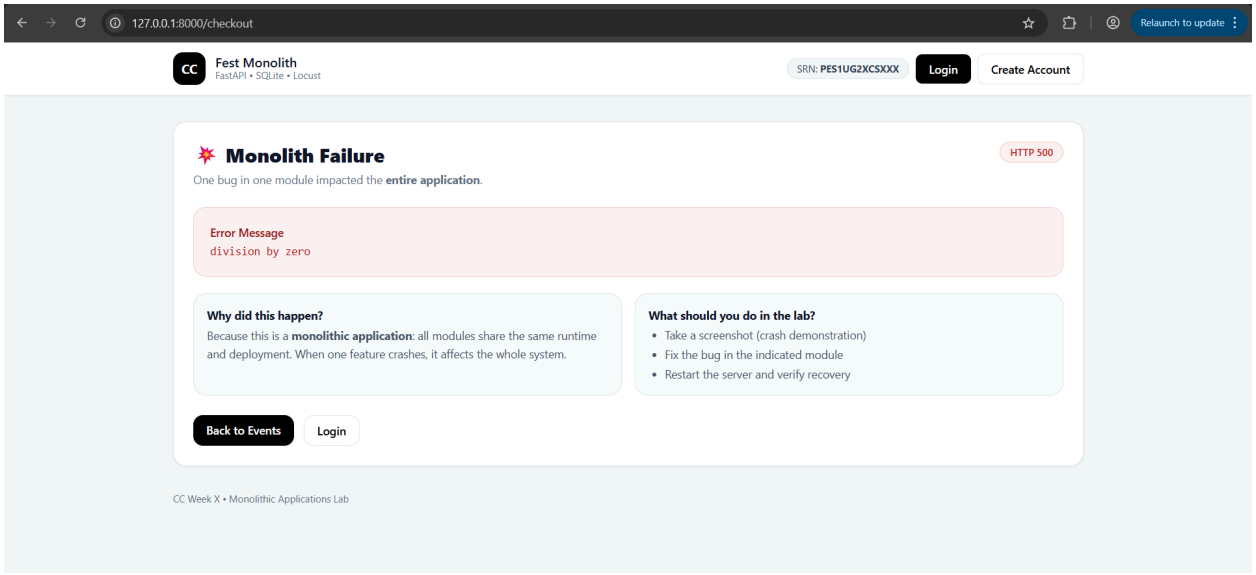


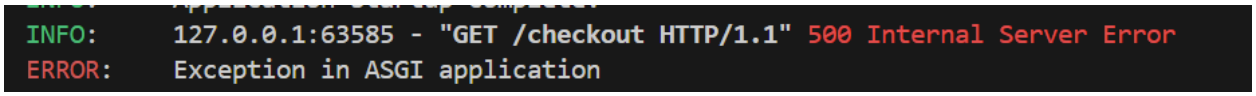
On successful setup and run, take your first **screenshot, SS1** (Events page loaded)

PART 3: Observe Monolithic Failure (Crash)

1. **Go to Checkout**
   http://localhost:8000/checkout

This will be your **screenshot, SS2.**





Well, what happened? Did the server crash? Oops, looks like we left a bug in there that brought the entire server down. This is one of the main disadvantages of the monolithic apps; even though other services can work, because of one error, the whole thing came down. To avoid this, we have microservices.

PART 4: Fix the Bug

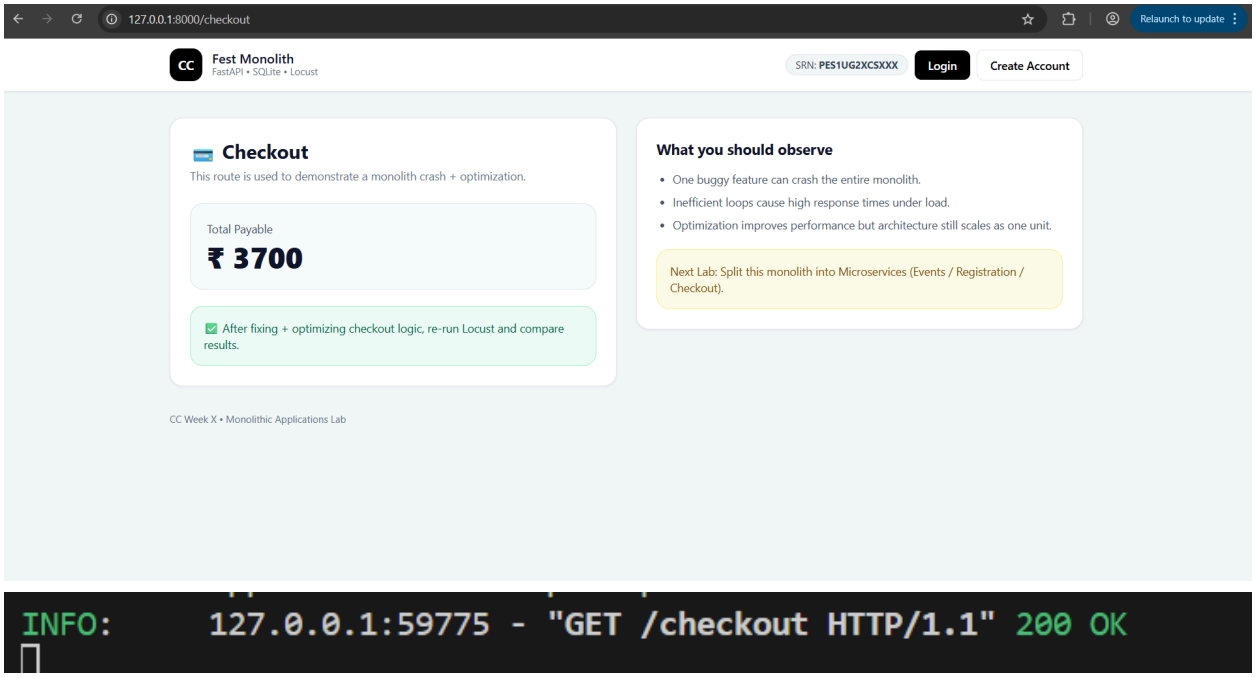
Let's fix the bug.

Go to /checkout/__init__.py and comment out line 10.

Restart the server and now revisit:

http://localhost:8000/checkout

This will be your Screenshot **SS3**



Why are we using FastAPI in this lab?

Even though this lab is about **monolithic architecture,** the framework we use should reflect what modern cloud apps use today.

We are using **FastAPI** because it:

- is **modern and high performance**
- supports **asynchronous programming (async/await)** out of the box

- provides **automatic request validation**
- is widely used for building **REST APIs in cloud deployments**
- has excellent integration with tools like Docker/Kubernetes later

Important Note:
Even though FastAPI supports async, our application is still a monolith because the architecture is based on one codebase + one deployment + a shared database.

PART 5: Load Testing using Locust

Locust is a load testing tool where we simulate multiple users hitting an endpoint.

Run Locust for Checkout

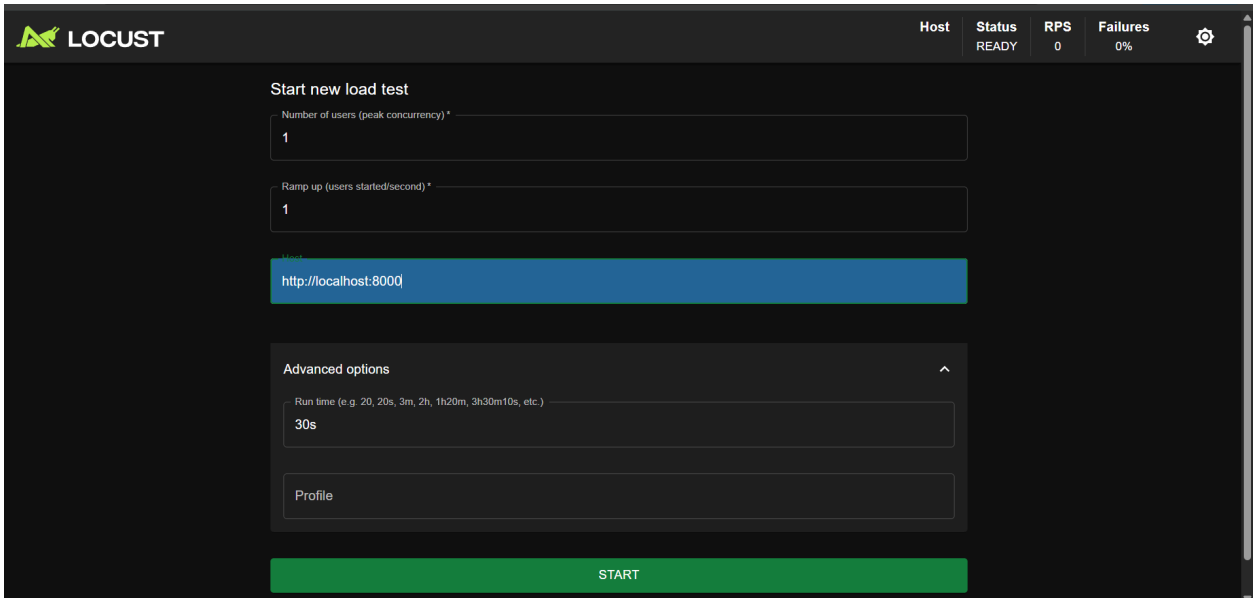
Keep your FastAPI server running in the background. Add a new terminal, activate your virtual environment and run the following commands.

From the project root:

```
locust -f locust/checkout_locustfile.py
```

Open UI:

http://localhost:8089

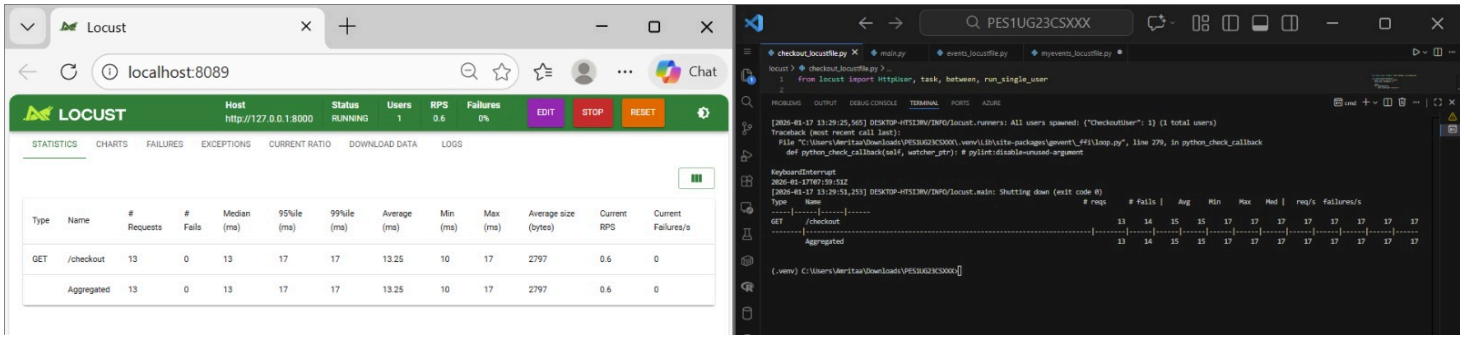


On the UI you can specify the number of users you want it to simulate. 'Ramp up' is the number of users that will be started per second. Under 'Advanced' you can set the time until the test runs.

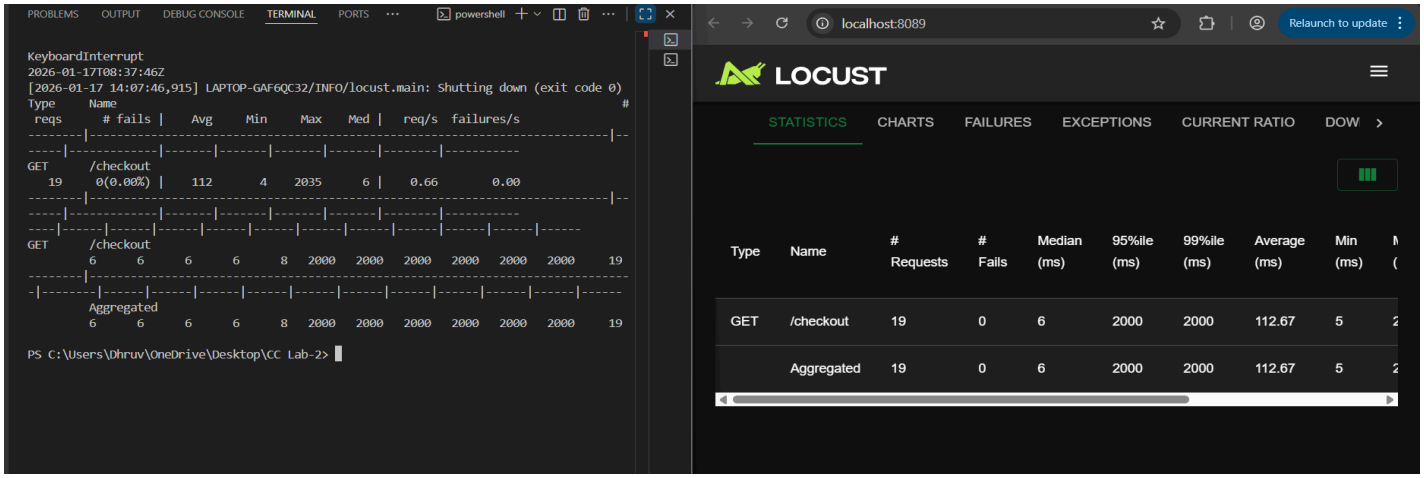
1. Here we will try to optimise the /checkout route

a. The locust directory has the locust files which are already configured with the load tests.

b. Add user(1) and rampup(1), measure the performance for 30 seconds and terminate locust running on your terminal. The **screenshot (SS4)** expected here is a split screen of the terminal and the Locust dashboard. **Make sure your SRN is visible in the terminal.**



Locust performance numbers (like response time and requests/sec) will vary from computer to computer depending on system speed and background load, so focus on the before vs after improvement trend, not exact matching values.



PART 6: Optimize the Checkout Route

Open: checkout/__init__.py

You will find inefficient logic such as:

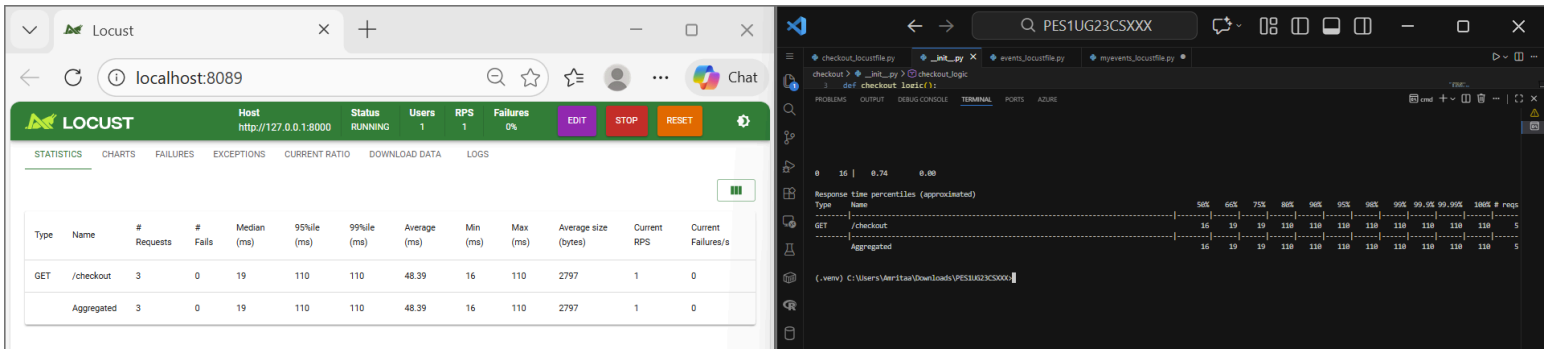
```
        while fee > 0:
            total += 1
            fee -= 1
```

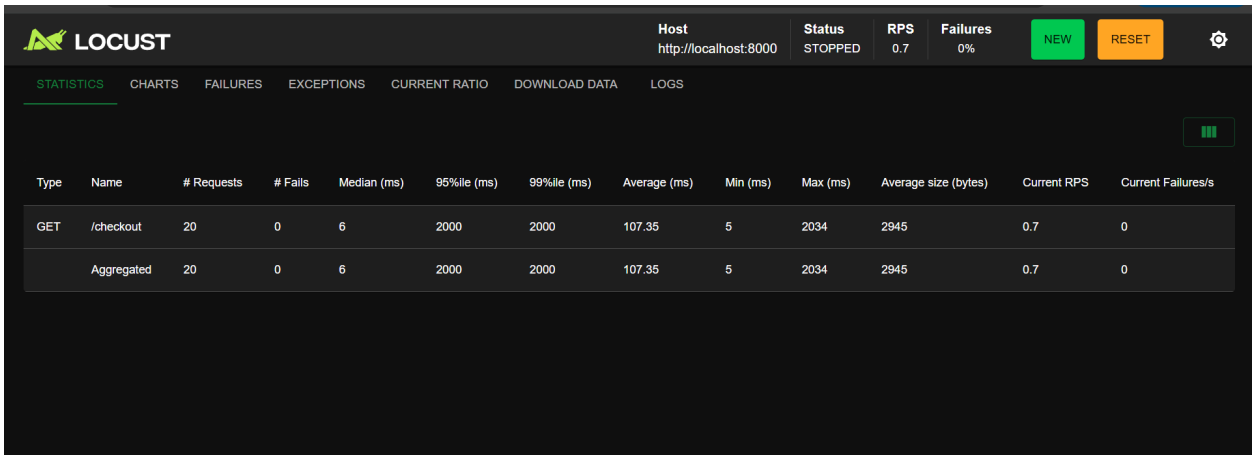
Replace with an optimised version:

```
    total = 0
    for e in events:
        total += e[0]

    return total
```

Restart the server and rerun Locust.





After optimization, as you can see in the SS, the average response time dropped from ~113 ms to ~107 ms, while requests/sec stayed almost the same (~0.7 RPS), meaning the API became slightly faster but could handle the same load.

This is your **Screenshot SS5.** The average times after optimisation and this one too has to be a split screen **with your terminal clearly showing your SRN number. (After optimization: compare requests/sec and avg response time)**

PART 7: Optimise *events* and *my_events*(DIY)

Now you will optimize **two more routes** independently:

Route 1: /events

Run:

locust -f locust/events_locustfile.py

Screenshot BEFORE optimization → **SS6**
Optimize code → Re-run locust
Screenshot AFTER optimization → **SS7**

Route 2: /my-events

Run:

locust -f locust/myevents_locustfile.py

Screenshot BEFORE optimization → **SS8**
Optimize code → Re-run locust
Screenshot AFTER optimization → **SS9**

What you must write (short explanation)

For both routes:

- What was the bottleneck?
- What change did you make?
- Why did the performance improve?

FINAL SUBMISSION

Submit:

1. Public GitHub repository link with your code.
2. Screenshots **SS1 to SS9**
3. Short explanation of optimizations