CC_Lab3

Name: Vidhu Vagale

SRN: PES1UG22AM189

Monolithic Applications: The Great Indian Paratha

Imagine walking into a dhaba, and the cook serves you a massive paratha stuffed with aloo, paneer, gobhi, and even some leftover dal from last night. That's your monolithic application—one big, all-inclusive piece of software. It has everything: the dough (UI), the stuffing (business logic), and the ghee on top (data layer).

At first glance, it looks great—one wholesome package, easy to eat, and no extra plates. But when it's time to break it apart or make changes (like adding some green chilies), you realize things aren't as simple as they seemed.

Advantages of Monolithic Applications: Why We Love the Paratha

1. **Simple and Straightforward**

   Building a monolith is like rolling out one large paratha. All the ingredients go into a single dough (codebase), and it's cooked together. Easy to prepare, serve, and consume without fuss. Great for beginners and small teams!

2. **Efficient Communication**

   In a paratha, the flavors of aloo and masalas blend beautifully because everything is together. Similarly, in a monolithic app, all parts (UI, business logic, and data) are tightly integrated, allowing them to work efficiently without the overhead of connecting through APIs or network calls.

3. **Easier Debugging**

   If your paratha tastes off, you don't have to guess—everything is in one place. The same goes for monoliths. Troubleshooting is simpler since all components live within the same codebase.

4. **Perfect for Small Kitchens (Teams)**

   A single cook (developer or small team) can handle a monolith. It's ideal when resources are limited, and you want to deliver something quickly.

Disadvantages of Monolithic Applications: When the Paratha Gets Too Big

1. **Scaling Struggles**

Imagine trying to roll out a paratha that's too big for the tawa. A monolithic app grows the same way—when the application gets larger, adding or updating features becomes increasingly difficult. Changing one thing might affect everything else.

2. **Deployment Drama**

   Adding a pinch of ajwain (a new feature) to your paratha? You'll need to knead the dough and cook the entire thing all over again. Similarly, even small updates in a monolith require redeploying the entire application.

3. **Not Flexible with "Custom Orders"**

   Let's say someone only wants plain roti or just paneer stuffing. Too bad—a monolith can't separate components. It's all or nothing.

4. **Maintenance Challenges**

   Imagine 20 cooks working on the same paratha, each with their own ideas about how much masala to add. Chaos! With a growing team and codebase, managing a monolith becomes messy and error-prone.

5. **Single Point of Failure**

   Burn the paratha on one side, and the entire thing is ruined. Similarly, if one part of a monolith fails, the whole application can go down with it.

The Verdict: A Paratha or a Thali?

Monolithic applications are like a hearty paratha—simple, satisfying, and perfect for small setups. But as your needs grow, and your customers (users) demand variety, you may want to switch to a thali (microservices)—where each dish (service) is prepared and served separately. This way, scaling, maintenance, and customization become easier.

So, the choice is yours: stick to the comfort of a single paratha or upgrade to a thali to meet the demands of a bigger crowd!

Well, That Was Theory—Let's Do a Practical!

To better understand a monolithic application, let's dive into a small practical example. We'll provide a small Flask app that uses HTML for front-end and SQLite3 as the database.

Why Is This a Monolithic Application?

In this setup:

- All services (UI, business logic, and database) run within a single process.

- The entire application is tightly coupled and works as a single cohesive unit.

The code for this practical will be provided as part of the lab.

(PS: The code probably has lots of bugs and UI is bare minimum, so bare with it please, or you could improve it and help us :))

1. First create a directory with your SRN. Unzip the file provided to you in that folder.

First create a directory with your SRN. Unzip the file provided to you in that folder. mkdir PES1UG2XCSXXX Once that's done the pwd output should look something like

```
PS C:\Users\admin\downloads\PES1UG22AM189> cd CC_Monolith
```

2. Now we will run the code a. So we will setup python virtual environment so that there is no collision between module versions

Windows

```
PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith> python -m venv .venv
PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith> .\.venv\Scripts\activate
(.venv) PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith>
```

Install module

```
(.venv) PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith> pip install -r requirements.txt
```

Start server

```
(.venv) PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith> python main.py
```
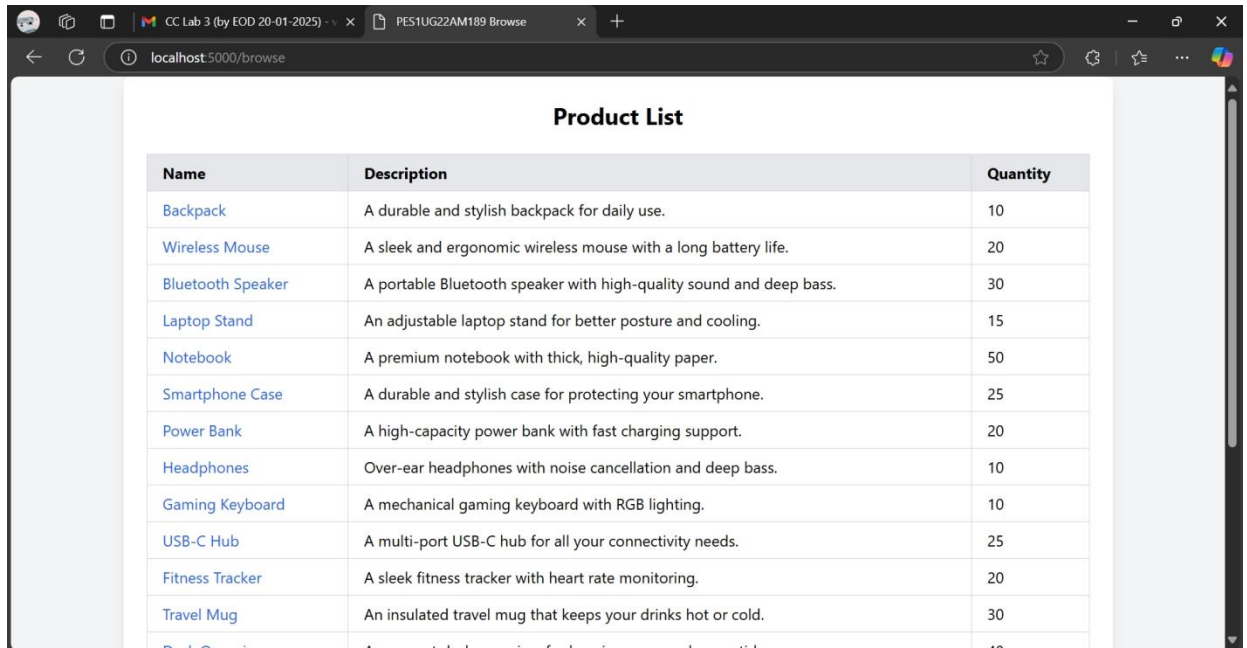
change main.py

```python
import jwt
from flask import import render_template, request, redirect, url_for
import products
from auth import do_login, sign_up
from products import list_products
from cart import add_to_cart as ac, get_cart, remove_from_cart, delete_cart
from checkout import checkout as chk, complete_checkout
import os

app = flask.Flask(__name__)
app.template_folder = 'templates'
SRN = "PES1UG22AM189"

if(SRN[-3:]=="XXX"):
    print("Please update your SRN on line 15")
```

6. Visit the site (note if localhost didn't work try with 127.0.0.1) a. http://localhost:5000/register to register a user b. http://localhost:5000/login to login c. http://localhost:5000/browse to browse for stuff, click on the product name to view the details about the product, and if you like the product go ahead and add the product to your cart. On successful setup and run the following screen should appear. This will be the First Screenshot(SS1). Make sure your SRN is clearly visible.


d. When you are done adding to the cart, you can checkout. http://localhost:5000/cart

6. Time to checkout, you can click on checkout http://localhost:5000/cart here or you can visit http://localhost:5000/checkout . You will see this page , Take a Screenshot(SS2).
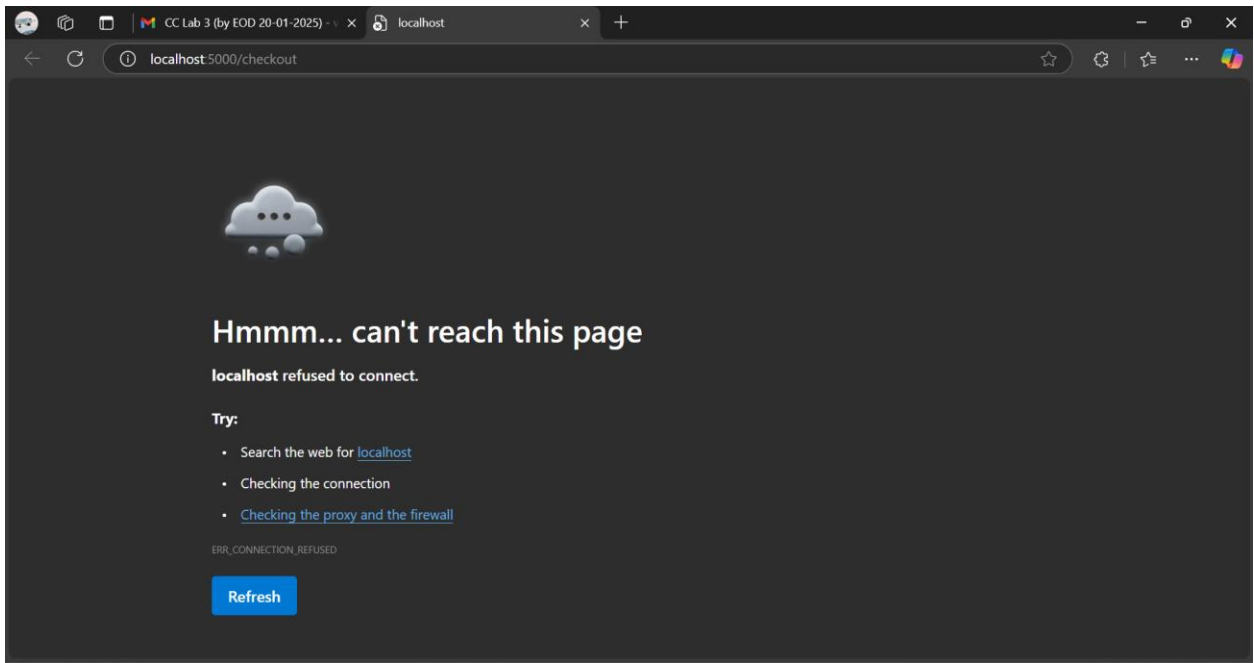
a. Well what happened, did the server crash, oops looks like we left a bug in there that brought the entire server down. This is one of the main disadvantage of the monolithic apps, even though other service can work because of one error whole thing came down, to avoid this we have microservices, which we will look into upcoming labs b. Let's fix the bug, go to /checkout/__init__.py, comment out line 15. c. Rerun the server d. And visit http://localhost:5000/checkout . This will be Screenshot 3 (SS3). Make sure your SRN is clearly visible.
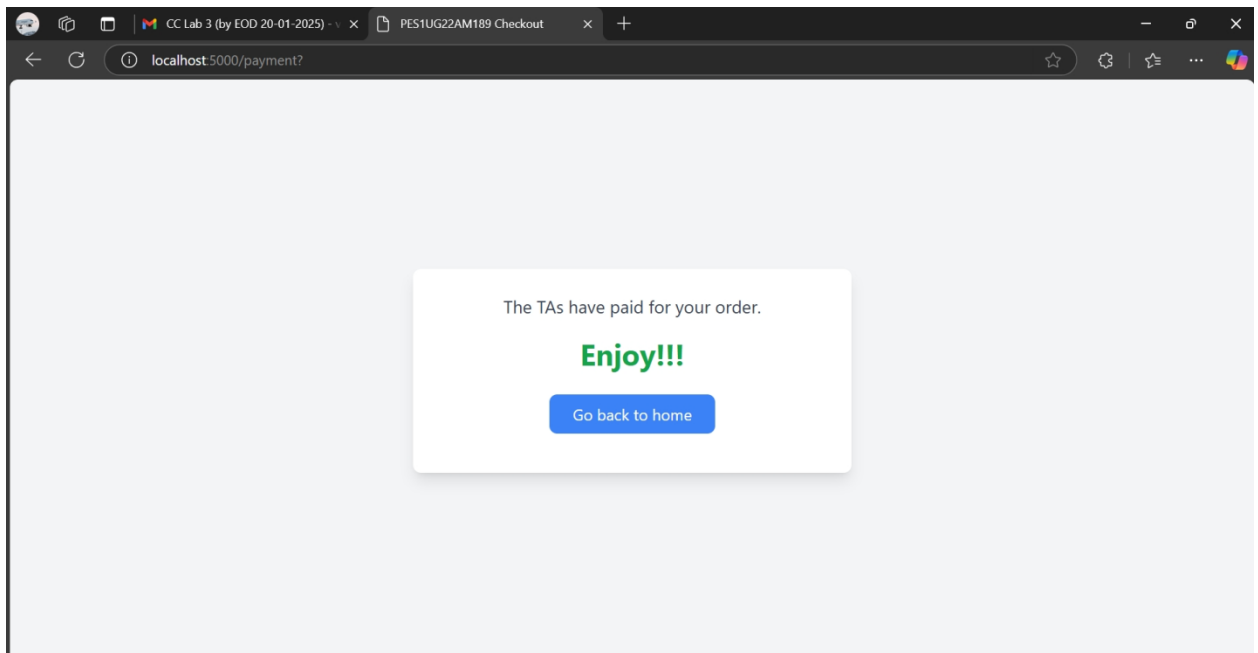
Great, we have bug-free code (hopefully). You might think having no bugs is good, but that is not true, especially in monolithic apps where everything is merged together. We need to optimize the code. How do we know if the code is doing any good or not? There are many tools for that, but we will look into one of them. Locust - A load-testing tool, where you can define a test and run it on your endpoints. Before you start you need to set the database, run insert_product.py

Windows:

```
(.venv) PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith> python insert_product.py
```

Running locust:

```
(.venv) PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith> cd locust
(.venv) PS C:\Users\admin\downloads\PES1UG22AM189\CC_Monolith\locust> locust -f checkout-locustfile.py
```

You can visit the site 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), Rampup(1) and measure the performance for 30 seconds and

terminate locust running on your terminal. The Screenshot (SS4) expected

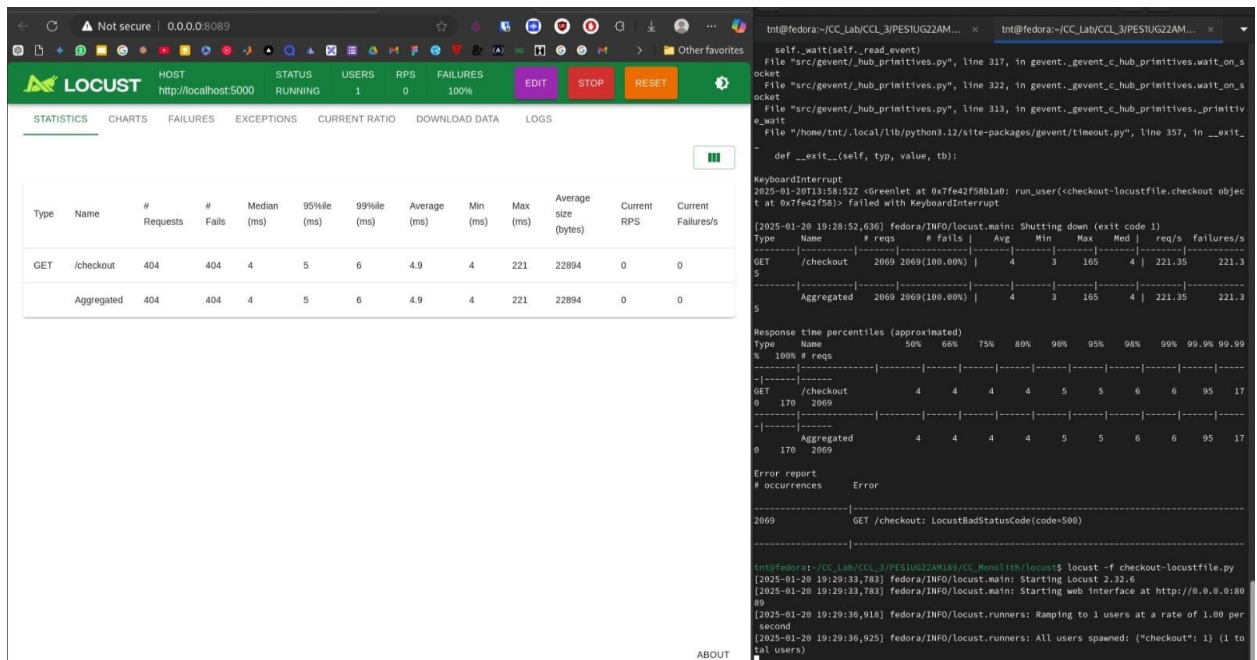here is a split screen of the terminal and the locust dashboard. Make sureyour SRN is visible in the terminal
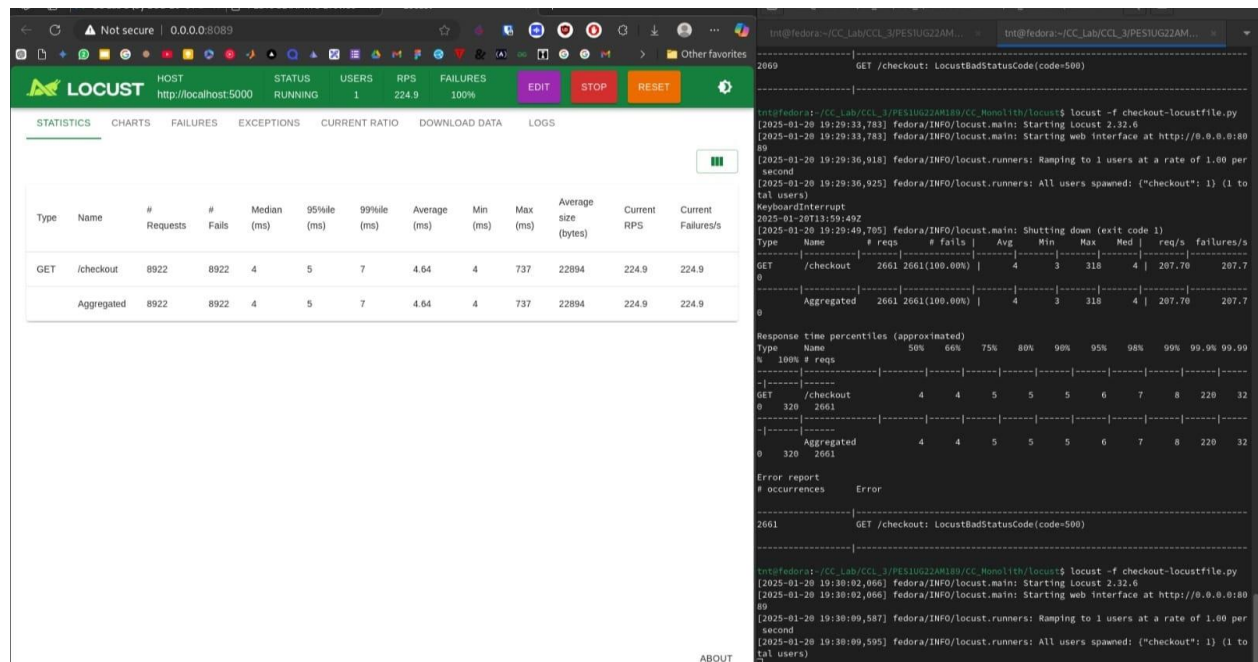
c. We could probably optimise this.

2. Optimise the route /checkout

a. So if we look at the /checkout handler on the main.py file at line no. 127, we

see that the handler is fairly optimised, so there is not a lot we can do here.

b. Next we will go through user defined functions, and under this only chk

function exists which is an alias of checkout function imported from checkout

module.

c. Now if we go to /checkout/__init__.py, and look at the code we see that the

code is bad.

After changing code

```python
def checkout(username):
    cart = get_cart(username)
    total = 0
    for item in cart:
        total += item.cost

    #Here the exit can happen when a illegal memory is accessed
    # or when a error is not handled properly
    #os._exit(1)
    return total
```

e. Now rerun the locust and notice how it has significantly improved. Notice how

the number of requests processed has doubled and the average time to handle them has

decreased. The Screenshot (SS5) expected here is a split screen of the terminal and the

locust dashboard. Make sure your SRN is visible in the terminal.
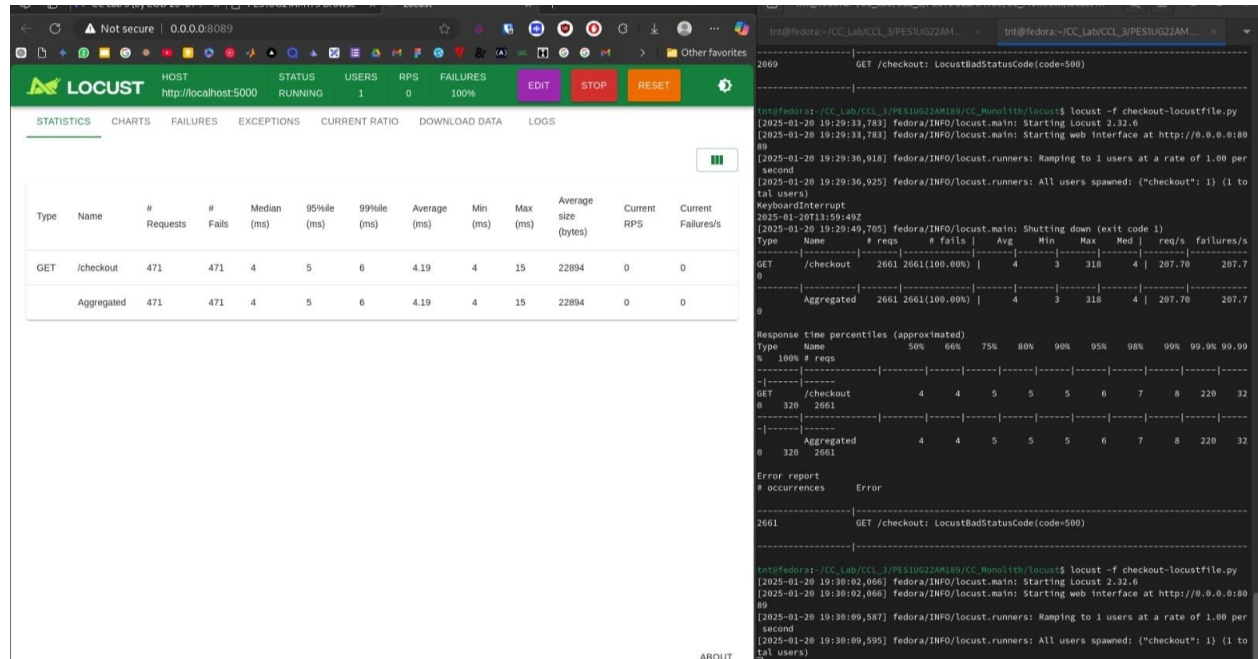
2. Until this point, we have guided you through this lab, now we would like you to optimize the
    next 2 routes /cart and /browse, we have provided you with 2 more locust files(get-cart-
    locustfile.py and browse-locustfile.py) preconfigured with the test. Optimize the code. Similar to
    the optimization done in /checkout route, we expect you to take the Screenshots (SS6 and SS7)
    of before and after optimizing /cart . Similarly Screenshots (SS8 and SS9) of before and after
    optimizing /browse. Also we expect you to provide why and how you have optimised the code.

3. BEFORE /CART :

4. After

Before browse

After browse

Git hub repository and modified code :