Install / Run the code

1. Cd to downloaded folder, Pip3 install -r requirements.txt
   Start Servers; python3 ResearcherServer.py = port 5002, python3 FundingAgencyServer.py == port 5001, python3 UniversityServer.py = port 5000 – pip install "cloud-sql-python-connector [pymysql]" for centralised database file (altough wont work as database instance has been stopped due to money)

Video Link ; https://drive.google.com/file/d/1TLbzeoFAoHafID6WFnetHn-NB-GJZLmd/view?usp=share_link

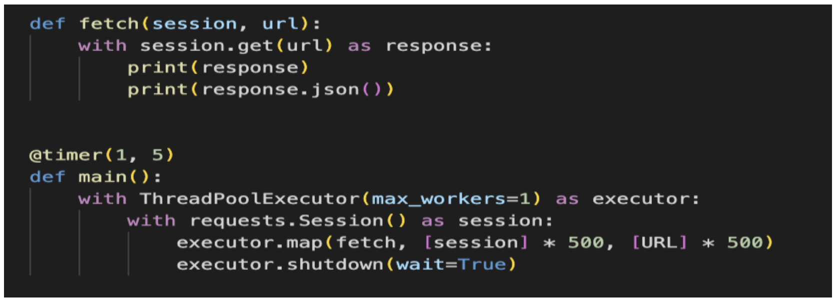
There is another folder in the zipped folder showing the centralised database code.

Evaluating/ Improving performance

Originally, I had implemented a sqllite3 database but this would later be moved to an online database, there are limitations to testing for the online database that will be discussed later so for now I will just use the results for the SQLite database as an indicator. If I send multiple requests to the database, I will be charged money so I will just use SQLite for testing portions.

To evaluate the performance of the flask server I used http requests to calculate the time it would take for the server to process x processes, where each process would represent a client

on the server. There would be a threadpoolexecutor created with a set number of workers. It uses the request session object to get the response from the flask webserver. It uses the map function to create 500 sessions using the same URL 500 times to simulate.

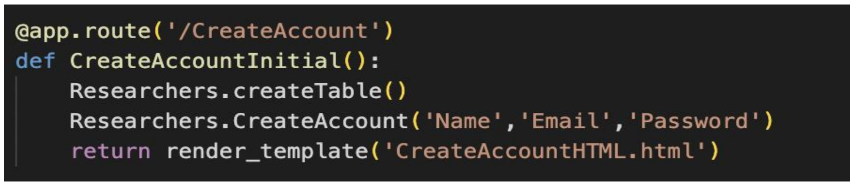


```
def fetch(session, url):
    with session.get(url) as response:
        print(response)
        print(response.json())


@timer(1, 5)
def main():
    with ThreadPoolExecutor(max_workers=1) as executor:
        with requests.Session() as session:
            executor.map(fetch, [session] * 500, [URL] * 500)
            executor.shutdown(wait=True)
```

As most of the functionality of the server came from html requests through flasks socket Io to fully test the functionality and evaluation of the server, I created a test page where all the users had to do was go on to the page for the page to execute what it needed without the input of the user. For example, I used the create account functionality to create another test page for the user to create a test account with parameters already defined.

Originally, I left it as default like so:



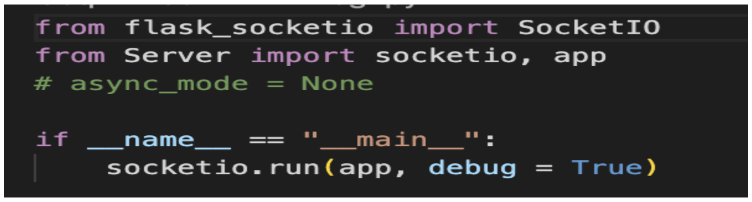
```
@app.route('/CreateAccount')
def CreateAccountInitial():
    Researchers.createTable()
    Researchers.CreateAccount('Name','Email','Password')
    return render_template('CreateAccountHTML.html')
```

Applying the first script to this url, I got an original elapsed time of 0.8829609334.

After 10 runs of the script, I got an average elapsed time of 0.8758437708199999.

My original concept was to create a Gunicorn web server.

I created a wsgi file as follows;



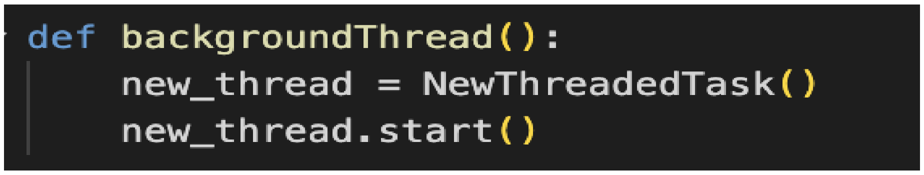
```
from flask_socketio import SocketIO
from Server import socketio, app
# async_mode = None

if __name__ == "__main__":
    socketio.run(app, debug = True)
```

And ran the gunicorn server as follows; 'gunicorn --worker-class eventlet -w 1 wsgi:app'

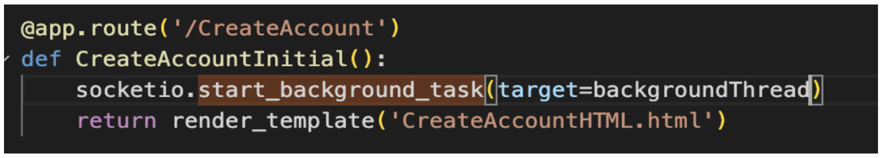
Gunicorn will have a threaded worker running to help the CPU and hopefully increase the efficiency of the server. After running the script I got an elapsed time of 0.945056875, with an average elapsed time of 0.9323414900200001. The performance decrease meant that I moved on to other options available.

Socket IO has a start_background_task method which will treat its target function as a thread and run it in the background of the server. To create this, I first had to wrap the logic inside its own thread like so:



```
def backgroundThread():
    new_thread = NewThreadedTask()
    new_thread.start()
```

And then create the background task of the background thread target function and wait for it to run in the background while the program continues.



```
@app.route('/CreateAccount')
def CreateAccountInitial():
    socketio.start_background_task(target=backgroundThread)
    return render_template('CreateAccountHTML.html')
```

Initially I hoped this would increase performance but again the first elapsed time was 0.9668322829999999 with an average elapsed time of 0.9780918283. Again, showing a decrease in performance for the server.

Another Idea I had was to create a redis server and use gentlet monkey patch to help improve the efficiency of the server. I added this part of the code to the top of the server file:

```
import eventlet
eventlet.monkey_patch()
```

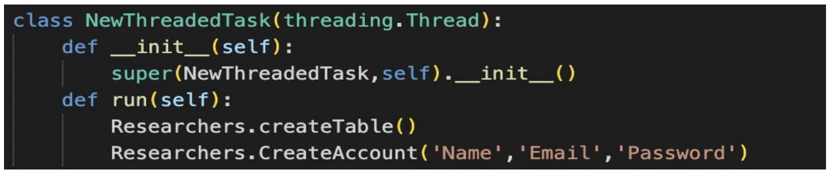
Then ran socketio with the following code:

socketio = SocketIO(app, message_queue='redis://').

While running redis with redis-server.

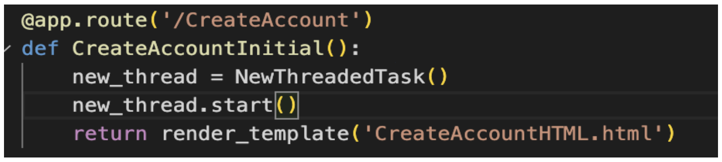
This still results in slower efficiency as the first elapsed time was 0.9118181167999999 with an average elapsed time of 0.91757737326.

Finally, the final version I settled with that got the best results was to create a new thread class with the functionality inside it like so :



```python
class NewThreadedTask(threading.Thread):
    def __init__(self):
        super(NewThreadedTask,self).__init__()
    def run(self):
        Researchers.createTable()
        Researchers.CreateAccount('Name','Email','Password')
```

I then created the thread and started it.



```python
@app.route('/CreateAccount')
def CreateAccountInitial():
    new_thread = NewThreadedTask()
    new_thread.start()
    return render_template('CreateAccountHTML.html')
```

The first elapsed time was 0.5637092916, with an average elapsed time of 0.5638141290400001 showcasing a significant improvement compared to the original implementation. On average for each client request it will save about 35% of request time for each client. This will be initiali

Database Synchronization

Using pyunit I used threads to test multiple clients trying to access and write the database inside the file called test_database. Before running the script, I first deleted the database before running the script. Upon originally running this file with ‘pytest’ it fails as there are multiple threads trying to access the database and it creates a ‘UNIQUE constraint’ exception for the ID column. To ensure that this exception does not happen when there are multiple clients trying to access the database at the same time, I used synchronization to ensure that only one client can connect to the database at one time. Calling @synchronization before inserting into the database implemented synchronization for the database. Now when I run the test with synchronization enabled the tests now pass. This helped achieve strong consistency as we are ensuring another thread or client does not affect our current client.

After converting to an online database to enable consistency across all the servers I ran into the same unique constraint ID, so synchronization was still implemented.

Websockets

To achieve concurrency and ensure in flask that what a client does on its webpage does not affect another client's webpage flask socket Io was used. Here web sockets enable clients to affect only their webpage. When a client joins a server, he is automatically assigned a random ID. I then utilized socketio’s join_room functionality to join a room based on that ID, now anything that is passed through any socket will be specific to the current room the client is in.

Consistency

To ensure consistency between both the offline server and the online server an online database was created. For this I used google cloud sql. I created a database that I could connect to in python using the cloud-sql-python-connector class. All database interactions came from the one centralized database online to ensure that every server/host had access to the most up to date information possible. Originally, I used sqllite3 but found it hard to implement for both the online/offline server as there would be two different databases correct and not a shared resource, leading to inconsistency for each server with each server having different information. Unfortunately, I found some time constraints for connecting to the online database when sending multiple synchronous requests, however this is a limitation of cost more than anything else, it could easily be scaled improving its performance if I payed money to get the best possible database from google cloud. Also, if I sent thousands of requests in a short period of time, I would get too many requests errors. If however, you were a large-scale company that had to deal with these limitations you could pay for the proper database to accommodate such requests.

Failure

To deal with failure in the event of the server failing on the local machine I used an online server using google cloud to host a backup server. Now if for whatever reason the flask server fails then it will be rerouted to the online server as a backup. In the event of the server crashing completely the user can go directly to the backup server instead. I added an exception handler in the code so if a http exception occurs as a result of a bug in the code e.g wrong template name then the user will be redirected to the online server instead.

To run the server, I used google clouds flexible engine. I deployed the app using gcloud app deploy. Which will run the app.yaml file and host the server online. Now all the functionality for the offline local app now has a backup webserver hosted online.

Self Reflection

In this assignment I learned how distributed systems work. I learned that consistency means that each process connected to the distributed system has access to the most up to date version of the data so you must ensure that the data is being constantly updated. It relates a lot to the course material as we have covered distributed systems in depth and the importance of achieving concurrency to ensure that multiple users can use distributed systems in paralel with each other. I also learnt that failure deals with what happens when a distributed system “breaks” and to ensure that it is constantly running. If a big company has significant downtime for their system, they could potentially lose millions, to deal with this they use failure to ensure that their users constantly have access to their services.

If I was to do things differently I maybe would have chosen a different online server option than google, AWS provides a free tier that maybe I could have made better use of, however when I went to create a instance in AWS it said my account was blocked and I would have to contact customer support, researching online it said it might take a while to get my account back, so due to the time constraints of the project I elected to go with google cloud instead.