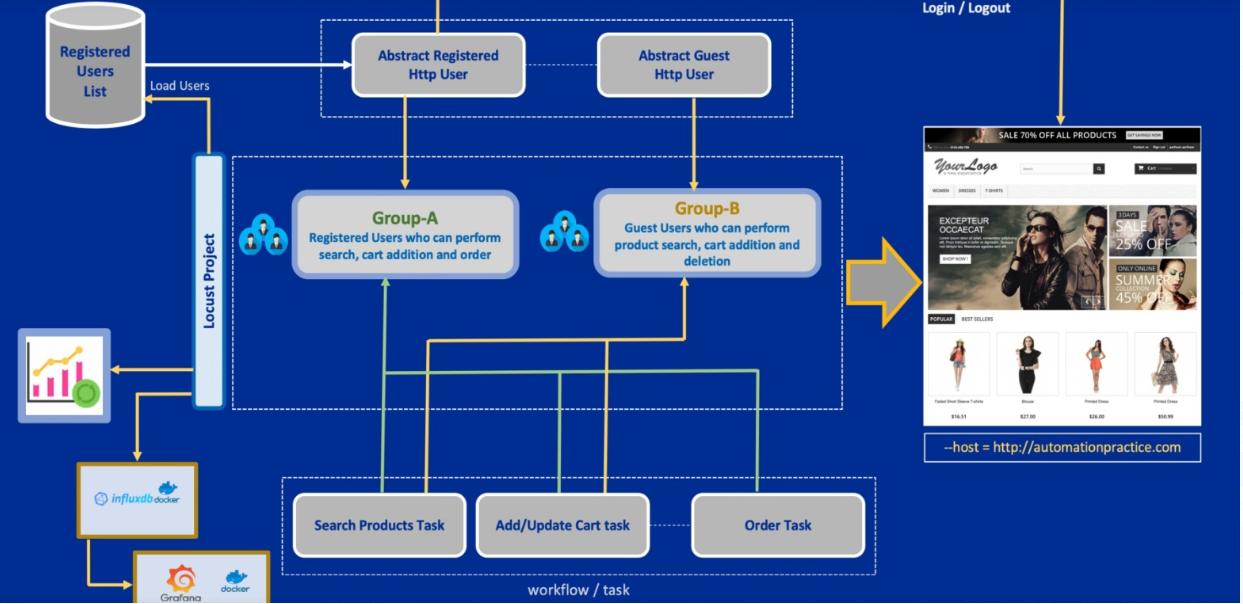


**Locust Framework & Project architecture**:

1. We will divide our users into two groups.

🡪 Group-A (Registered Users)

🡪 Group-B (Guest Users)

🡪 These users will inherit properties from an Abstract user class in order to hide common functionalities from the main locust file.

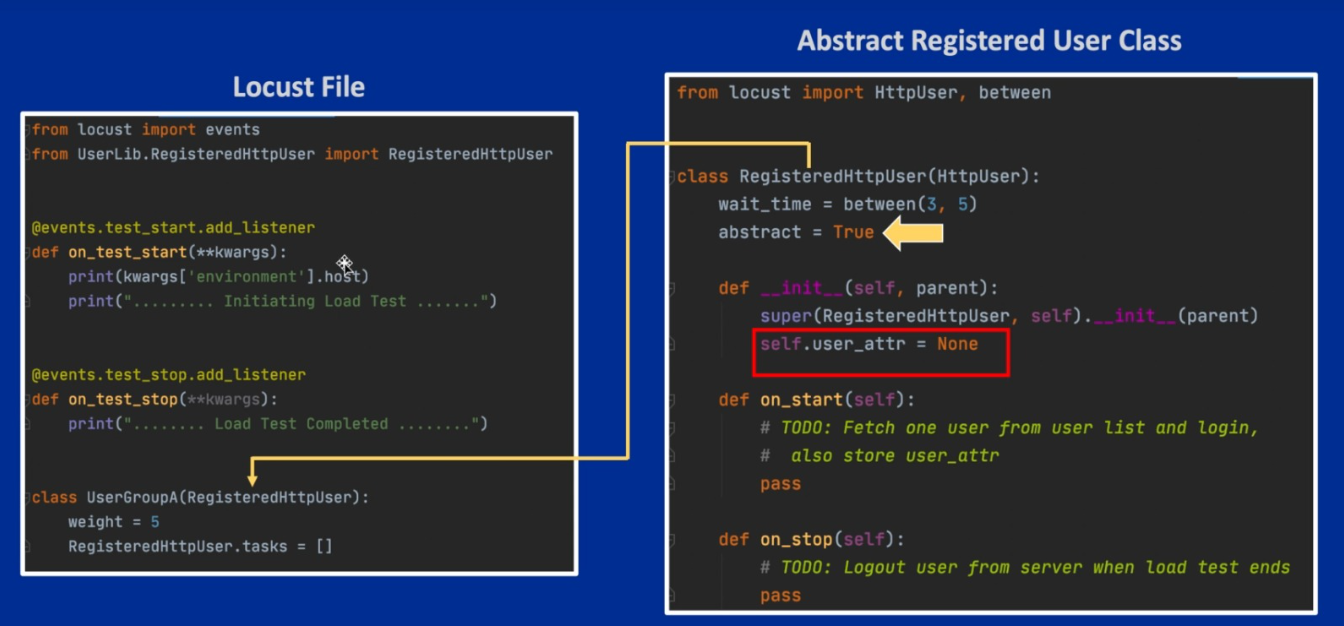
🡪 will load users from csv file.

1. All the tasks will be managed in a similar way by creating different TaskSet classes in individual modules.
2. Once we have created our specific User classes and TaskSet classes we will stitch them together to create a workflow.
3. In order to create better reports we will use influxdb with docker and then integrate Grafana with influxdb.
4. As for project structure we will use *Python package* system and create modules inside it , so that they can be imported into other modules.

**Abstract HTTPUser class and usage**:

To declare a class as an abstract class, make sure to set

abstract = TrueThis abstract property is coming from HTTPUser class.



*This is just an example not the actual inheritance structure in use*

on\_start in abstract user class

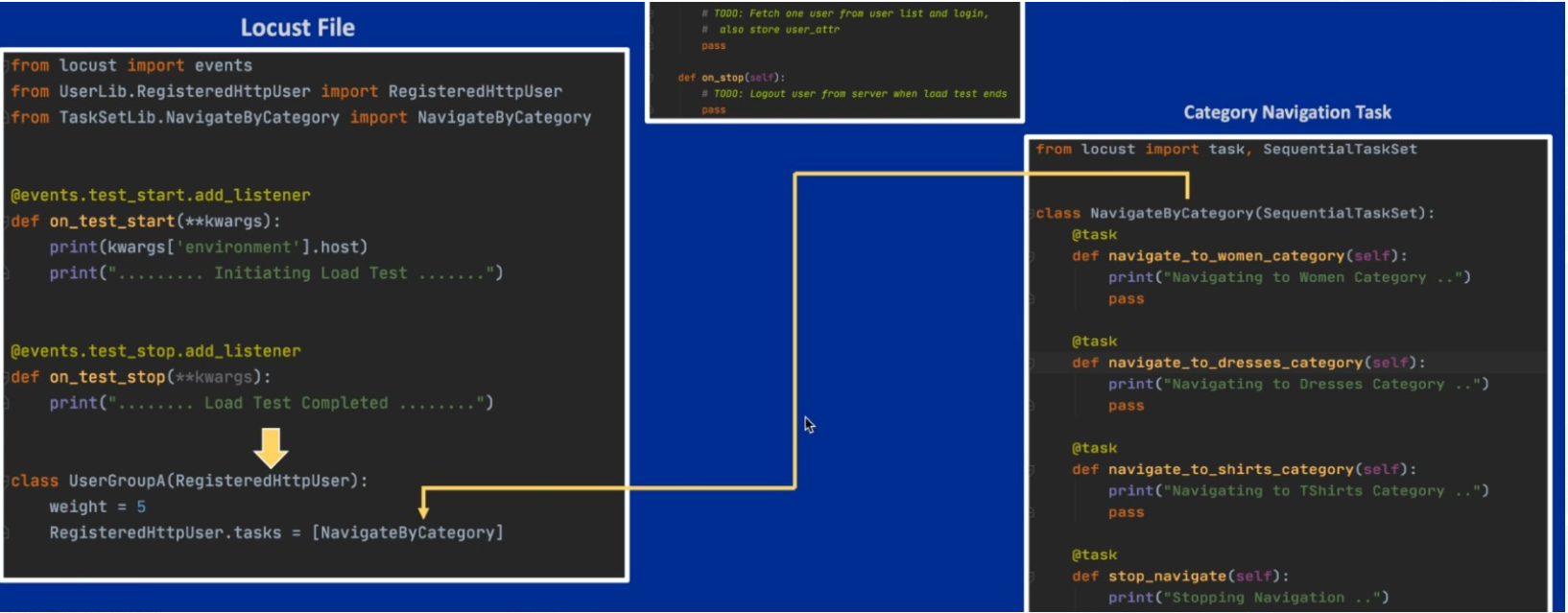
🡪 will be used to fetch user from user list and then login.

🡪 and also store user\_attr(*will be used to store user id or password, username etc.*).

🡪 on\_stop will be used for logout.

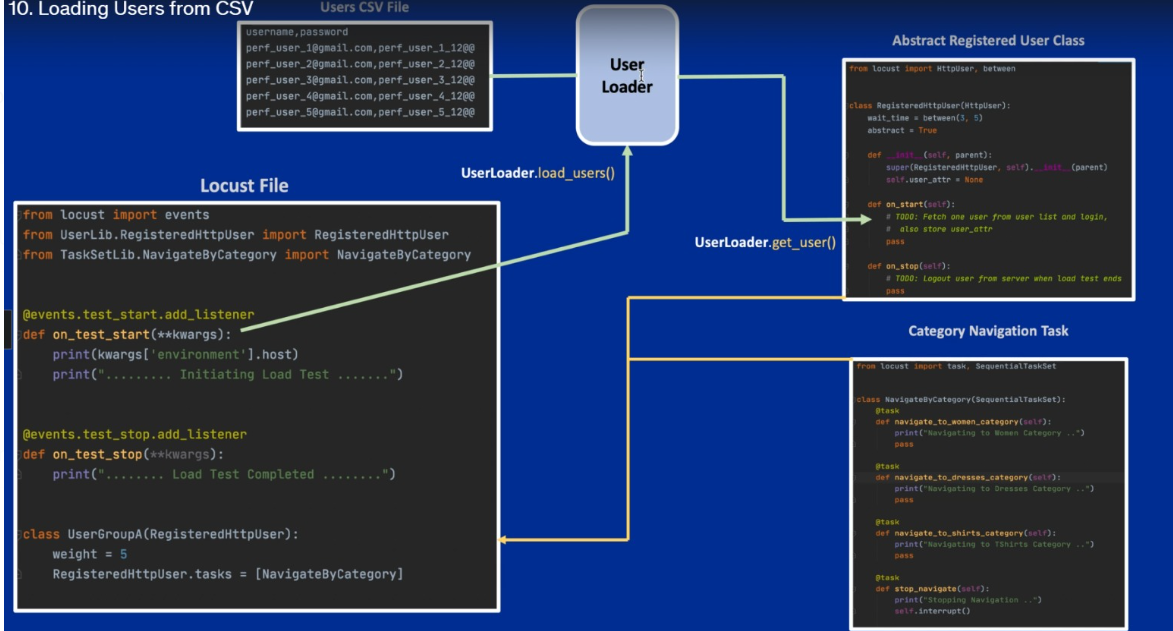
**TaskSet modules and User integration**:

We will just import the task modules in our Locust file and reference it in our users as *tasks*.



**Loading Users from CSV**:

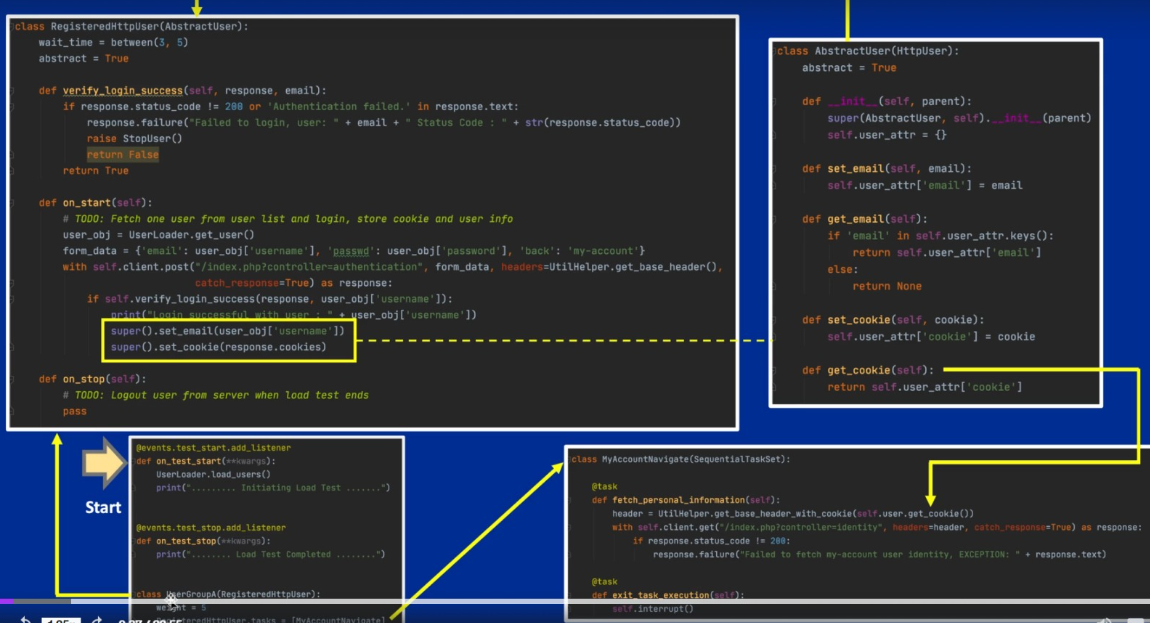
🡪 In Users CSV file, we have kept the dump of all user information.



🡪

**User Login & Session Management**:

It is time to see where exactly we are doing Login for the registered users and how we are going to save the session or the cookies to use for later REST calls and TaskSet classes.



We are now in RegisteredUser class,

**raise StopUser()**

is used in case registered User Login is failed, because we do not want it to continue on rest of the tasks after its initial failure to login and keep the pipeline busy.

from locust.exception import StopUser

If we do not use it we will see unnecessary failure in our logs.

If login is successful,

super().set\_email(user\_obj[‘username’])

super().set\_cookies(response.cookies)

Here, *super*() function is used to give access to methods and properties of a parent or sibling class.

Our parent class here is *AbstractUser* class which define these two methods

*Who will use this cookies information*?

This information will be needed by our TaskSet class. But there is a way to call this information.

self.user.get\_cookie()

The moment we use self.user it point to User where this task is being performed. This is done by Locust in background.

Now we know that get\_cookie is a function we created in our abstract class and we will get information from there.

*Why need cookie information*?

To map specific information with its respective user.

user1 🡪 cookie1 || user2 🡪 cookie2

another example is If we want to see order history of only customer#1 out of 100 customers then cookie is the only way we can map order history to that particular customer.

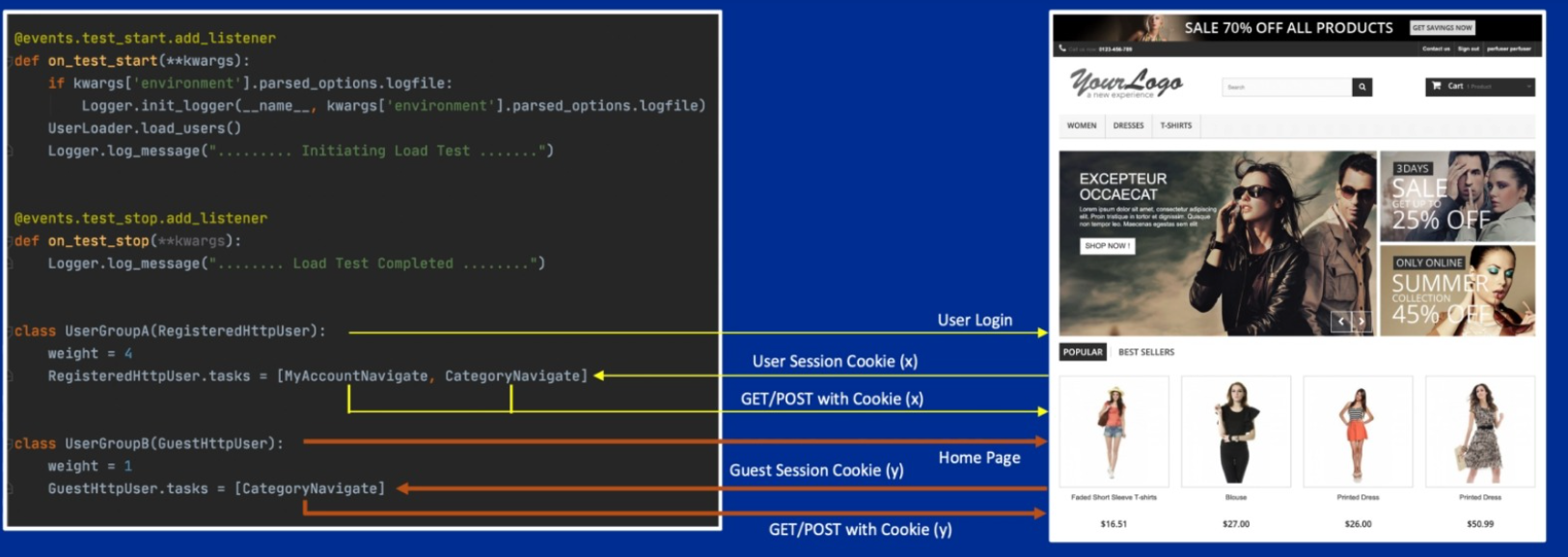
*Where we will send cookie*?

In the UtilHelper.py we created a method called

get\_base\_header\_with\_cookie() which is taking cookie information as a string first and then adding this information in our base\_header.

**Multiple User Types and Distribution**:

Here we will talk about different types of users and their distribution when we create the load.



We have two user groups here A and B. The weight attribute for A is set to 4 and for B its set to 1.

This means for every 5 users, 4 will belong to user group A and 1 will be in group B.

🡪 User group A or registered users will first login, save cookie as user session information and then do REST api GET/POST calls using those cookies.

🡪 User group B or guest user will skip Login and go to Homepage directly, get guest session cookies and do API calls with that.

Note: If there is an instant peak in the Response Times graph, that means we need to check the server logs around that time about what happened.

🡪 **Median** and **Average** in the statistics must be closer to each other.

Know about all the tables in Locust statistics report.

*What will happen if one of the user I create does not work*?

If we stop a specific user, say by using raise StopUser() intentionally or by accident, those users will not be spawned by Locust automatically. You can verify it by seeing total number of users hatched by Locust. (*suppose you hatch 10 users but you will see less than 10 users spawned*)

**Registered & Guest User cookie**:



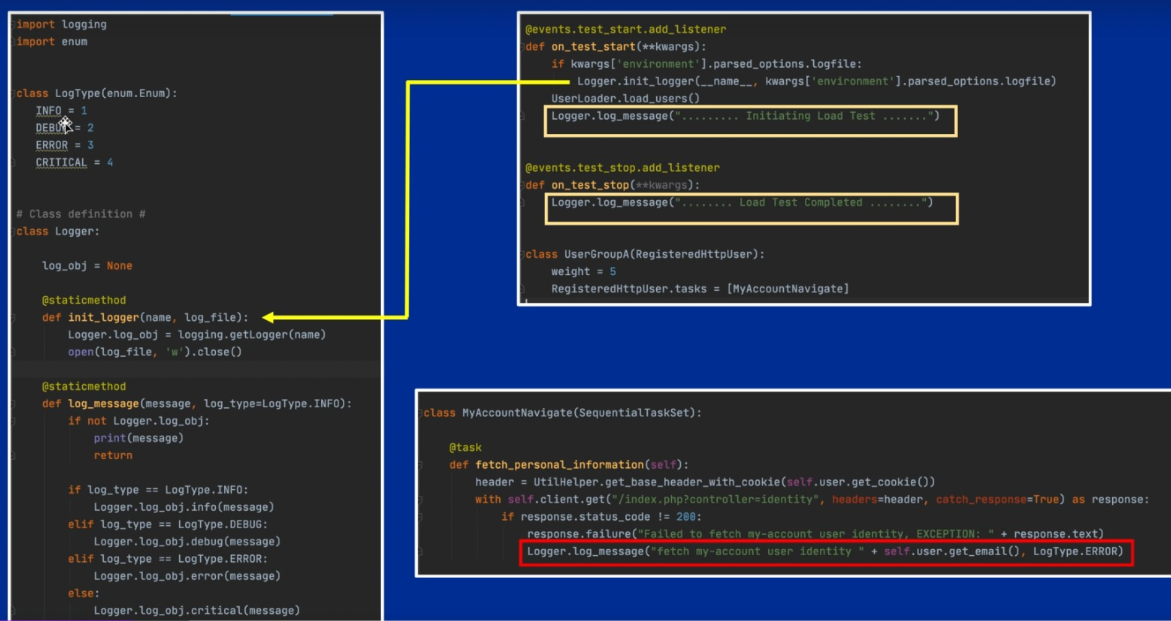
**Custom logger and logging module**:

Add This extra argument in the command line at the end

--logfile run.log

It will create a log file by the name of *run.log* inside our project.

When you run the command and start swarming, view this file.



You will not see print statements that we used in our project, only the information provided by the locust library(*/INFO/locust.runners:*) about current test execution.

So if we want to include these too, we need to create a custom logger with python logging module.

Inside our custom Logger.py file,

1. In *init\_logger* we are opening file in ‘w’ write mode, so that when we run the command again, it clears all the previous content and then write from fresh.

Log initialization will start from suite test\_start.

if kwargs[‘environment’].parsed\_options.logfile:

here *parsed\_options* are command line options we are passing

if the condition is true,

Logger.init\_logger(\_\_name\_\_, kwargs[‘environment’].parsed\_options.logfile)

We will pass these two as arguments to init\_logger function.

Where \_\_name\_\_ is the name of the locust file.

1. Now we will not use print messages in our file instead we will use

Logger.*log\_message*(“….initializing Load test…”)

1. This log\_message() takes two arguments.

log\_message(message, log\_type = LogType.INFO)

first argument is the message string , second argument is type of log (by default we have set of type INFO)

1. Then we set conditions

        if log\_type == LogType.INFO:

            Logger.log\_obj.info(message)

        elif log\_type == LogType.DEBUG:

            Logger.log\_obj.debug(message)

        elif log\_type == LogType.ERROR:

            Logger.log\_obj.error(message)

        else:

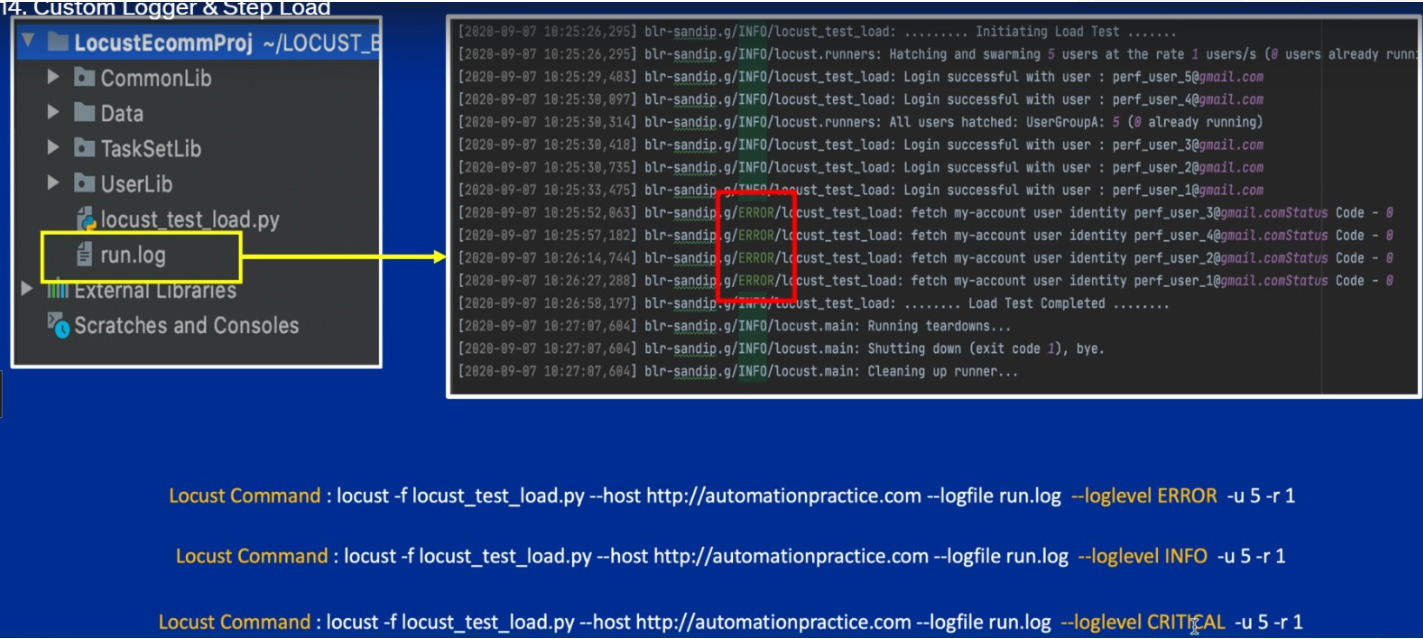
            Logger.log\_obj.critical(message)

1. If a certain case fails we will log the error message like this in a file by using second argument as LogType.ERROR,

Logger.log\_message("Failed to fetch my-account user address " + self.user.get\_email(), LogType.ERROR)

Note: In case there are thousands of users swarming and we only want to see only ERRORS or CRITICAL, we can directly filter them by passing command line parameters as

--loglevel ERROR or –loglevel CRITICAL



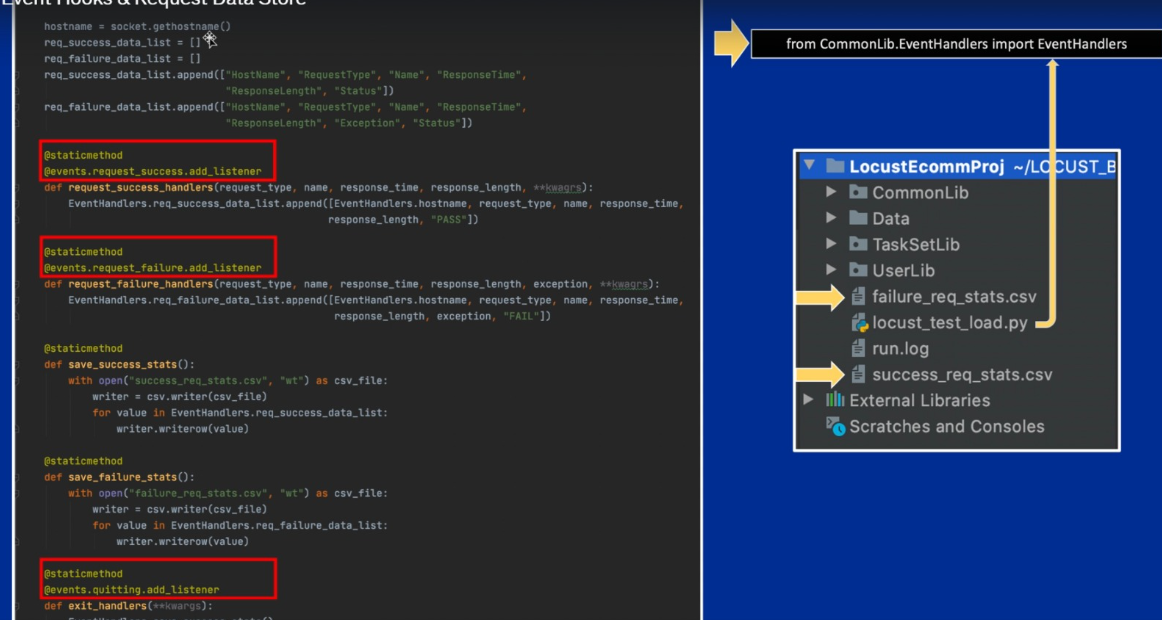
*What is step load*?

Suppose we are simulating 100 users in Locust and suddenly in the middle of testing we increase it to 1000(*to simulate start of an important event at a specific time*).

There is a step increase in load on your system which you can see in your graph as a step ladder.

**Event Hooks and Request Data Store**:

In this, we will learn about locust event hooks and how can we store request – response data or the detailed information of every request.



1. Create a separate file called EventHandlers.py

🡪 from locust import events

events class provide locust event handlers. For example, quitting is an event hook,

    quitting: EventHook

    """

    Fired when the locust process is exiting.

    Event arguments:

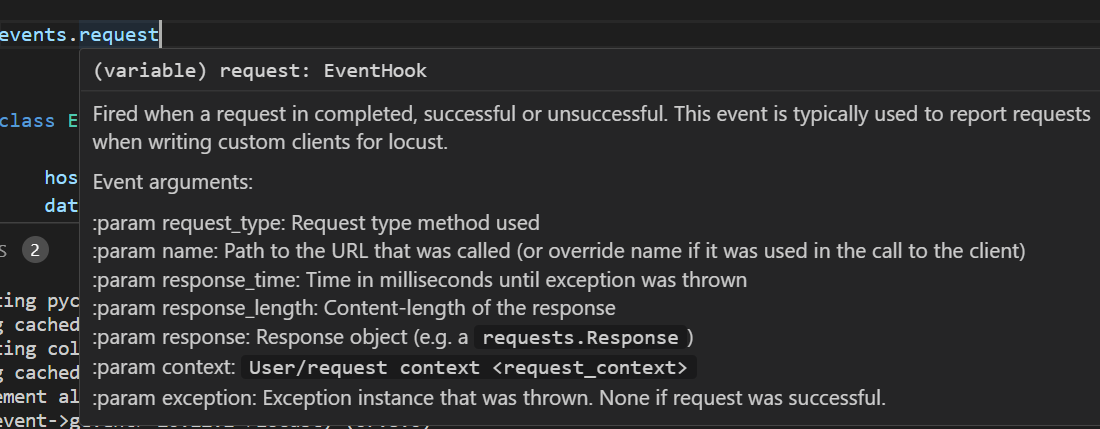
Events will be fired under certain scenarios, it depends on *whether we are listening to those events* or not.

If we are listening to those events, we can go inside that event and execute our own code.

We are already listening to test\_start and test\_stop events in locust file.

@events.test\_start.add\_listener

def on\_test\_start(\*\*kwargs): 🡪 executing this code inside test\_start event



If the request fails or be successful events.request will give us all information about that request.

An Example:

Here we are listening to quitting event handler,

    @staticmethod

    @events.quitting.add\_listener

    def exit\_handlers(\*\*kwargs):

        EventHandlers.save\_success\_stats()

        EventHandlers.save\_failure\_stats()

We created a function inside it *exit\_handlers* which will be called at the time of quitting.

This exit\_handlers function further calls two more methods from inside on of them is save\_success\_stats

    @staticmethod

    def save\_success\_stats():

        with open("success\_req\_stats.csv", "wt") as csv\_file:

            writer = csv.writer(csv\_file)

            for value in EventHandlers.req\_success\_data\_list:

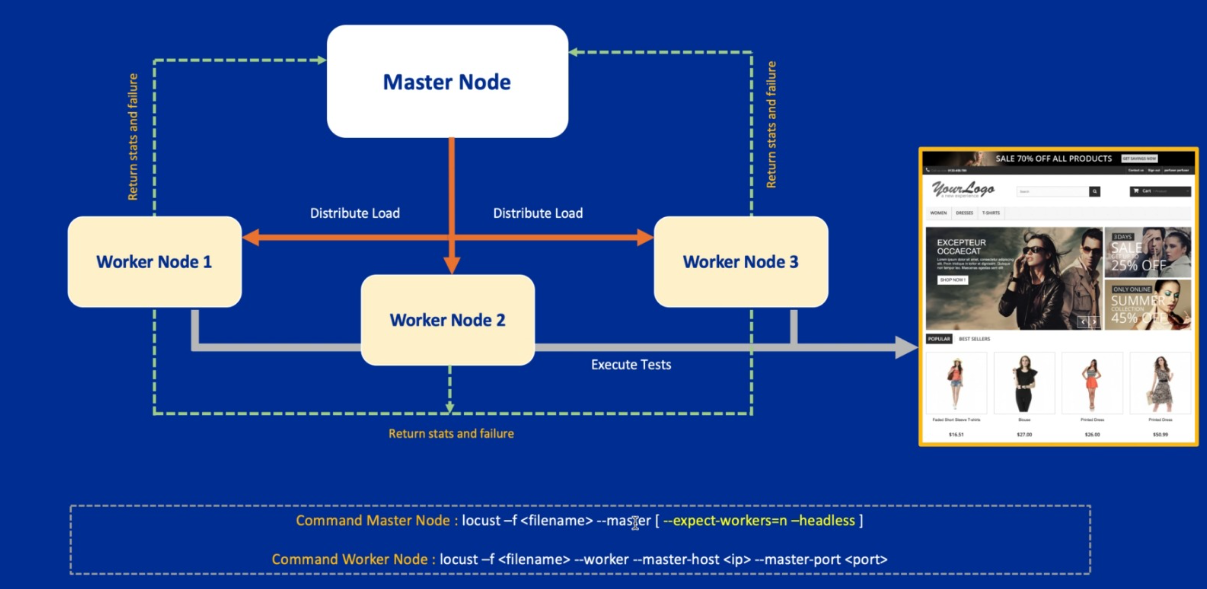
                writer.writerow(value)

This function creates a file and write data in to it. So this file will be indirectly created at the time of quitting.

**Request Data Store in InfluxDB**:

pip3 install influxdb

**Locust master- Slave Architecture Model**:



**Things need to revise in Python**:

1. Dictionary manipulation like setting new keys dynamically.

user\_attr = {}

def set\_email(email):

    user\_attr["email"] = email

set\_email("himanshu@gmail.com")

print(user\_attr) //{'email': 'himanshu@gmail.com'}

1. Getting keys

def get\_email():

    if "email" in user\_attr:

        return user\_attr["email"]

    return None

1. Using super() method in class to use inheritance.
2. Python packages and modules.
3. Write/Read csv file in python.

Read csv file as a dictionary

import csv

with open("user.csv", mode="r") as file:

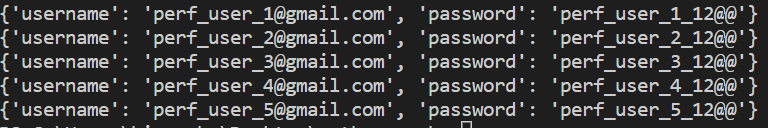
    # reading the CSV file

    csvFile = csv.DictReader(file)

    # displaying the contents of the CSV file

    for lines in csvFile:

        print(lines)



1. Packing and unpacking \*\*kwargs
2. Learn about this HTTPUser constructor

class AbstractUser(HttpUser):

    abstract = True

    def \_\_init\_\_(self, parent):

        super(AbstractUser, self).\_\_init\_\_(parent)

        self.user\_attr = {}

1. Learn about @staticmethod decorator

@staticmethod

<https://www.tutorialsteacher.com/python/staticmethod-decorator>

**Can be used for** :

🡪 Unit testing

🡪 Integration testing

🡪 Acceptance

how many concurrent users can we support on current infrastructure

**Why load test**?

🡪 To find breaking points (like 500 internal server error/ running out of memory)

🡪 Throughput (How many requests can get through application)

🡪 Latency (How long are requests taking?)

**Bottlenecks**:

Maybe a particular service on which we are relying for resources is slower, maybe some IO calls,

Monitor the system under test.

**Action**:

Fix the code, scale up hardware resources (throw some new AWS instances)

**Advanced topic**: running locust from multiple machines using master slave architecture