Testing

Testing

1. Unit Testing
   - Unit testing focuses on verifying **individual components** or functions of the software in isolation. It ensures that each unit performs as expected independently. Typically, developers write and run these tests during the development phase
2. Integration Testing
   - Integration testing assesses the **interaction between different modules** or services within the application. It ensures that combined components function together correctly, identifying issues in interfaces and data flow between modules
3. Smoke Testing
   - Smoke testing involves a preliminary check to ensure the **basic functionalities** of the application work correctly. It's often referred to as a "sanity check" before proceeding to more rigorous testing.
4. Regression Testing
   - Regression testing ensures that recent code changes **haven't adversely affected existing functionalities**. It involves re-running previous test cases to confirm that the software continues to perform as expected.
5. Performance Testing
   - Performance testing evaluates the software's responsiveness, stability, and scalability under various load conditions. It includes:
     - **Load Testing**: Assesses the system’s behavior under expected user loads.
     - **Stress Testing**: Determines the system's robustness by testing beyond normal operational capacity.
     - **Endurance Testing**: Checks the system's performance over an extended period.

Load Testing

- load tests are a type of performance test where we check how a determined part of the system will handle many simultaneous users at the same time.

When should I use load tests?

Load tests will come in handy in many ways:

- When making **performance improvements**, we can check if the improvements are truly effective.
- When **making decisions** about a particular tech stack, load tests can be used to compare multiple approaches to the same problem and determine which works best.
- Estimate how many simultaneous users are supported by the given system.
- Find **bottlenecks**, that is, components of the system that will take too long to respond and affect the overall response time and user experience.

Load Testing – Locust 

- Locust is an open source performance/load testing tool for HTTP and other protocols. Its developer-friendly approach lets you define your tests in regular Python code
- The target of locust is load-testing websites and checking the number of concurrent users a system can handle. During a locust test, a swarm of locusts will attack the target i.e website. The behavior of each locust is configurable and the swarming process is monitored from a web UI in real time.

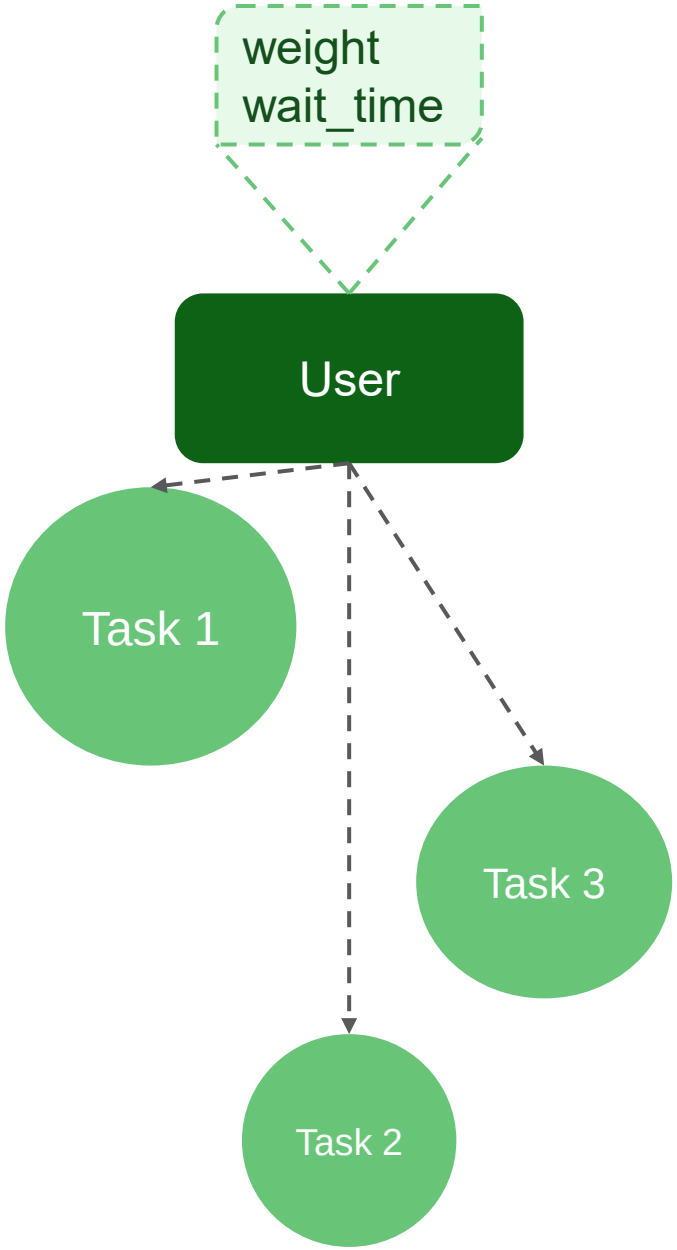
Specialty of locust:

- Test scenarios can be written in Python
- Distributed and scalable
- Web-based UI
- Any system can be tested using this tool
- Locust tests can be run from command line or using its web-based UI. Throughput, response times and errors can be viewed in real time and/or exported for later analysis.

Load Testing – Locust

Behavior 1

Behavior 2



Load Testing – Locust 

A user class represents one type of user/scenario for your system. When you do a test run you specify the number of concurrent users you want to simulate and Locust will create an instance per user.

User classes:

- HttpUser
  - gives each user a client attribute, which is an instance of **HttpSession**, that can be used to make HTTP requests to the target system that we want to load test.

- FastHttpUser
  - FastHttpUser provides the same API as HttpUser, but uses geventhttpclient instead of python-requests as its underlying client. It uses considerably less CPU on the load generator, and should work as a simple drop-in-replacement in most cases.

When a test starts, locust will create an instance of this class for every user that it simulates, and each of these users will start running within their own green gevent thread.

Load Testing – Locust 

The self.client attribute makes it possible to make HTTP calls that will be logged by Locust
It contains methods for all HTTP methods: **get**, **post**, **put**, …

```python
from locust import HttpUser, task, between

class MyUser(HttpUser):
    wait_time = between(5, 15)

    @task(4)
    def index(self):
        self.client.get("/")

    @task(1)
    def about(self):
        self.client.get("/about/")
```

Load Testing – Locust 

You can add any attributes you like to the user classes/instances, but there are some that have special meaning to Locust:

If you wish to simulate more users of a certain type than another you can set a weight attribute on those classes. If more than one user class exists in the file, and no user classes are specified on the command line, Locust will spawn an equal number of each of the user classes.

```
class AdminUser(User):
    wait_time = constant(600)
    fixed_count = 1

    @task
    def restart_app(self):
        ...

class WebUser(User):
    ...
```

Our class defines a wait_time that will make the simulated users wait between 1 and 5 seconds after each task (see below) is executed. For more info see wait_time attribute. If no *wait_time* is specified, the next task will be executed as soon as one finishes.

you can set the **fixed_count** attribute. In this case, the weight attribute will be ignored and only that exact number users will be spawned. These users are spawned before any regular, weighted ones.

```
class WebUser(User):
    weight = 3
    ...

class MobileUser(User):
    weight = 1
    ...
```

Let’s Code IT

Load Testing – Locust 

Install:
pip install locust

Load Testing – Locust

Locust Test Report

During: 11/4/2021, 9:03:39 AM - 11/4/2021, 9:11:17 AM

Target Host: https://django-todo-list-grjv6.ondigitalocean.app

Script: locustfile.py

Download the Report

Request Statistics

Method	Name	# Requests	# Fails	Average (ms)	Min (ms)	Max (ms)	Average size (bytes)	RPS	Failures/s
GET	/tasks/	21512	3	1980	36	13291	929	47.0	0.0
POST	/tasks/	4360	1	2002	14	13276	79	9.5	0.0
	Aggregated	**25872**	**4**	**1984**	**14**	**13291**	**786**	**56.5**	**0.0**

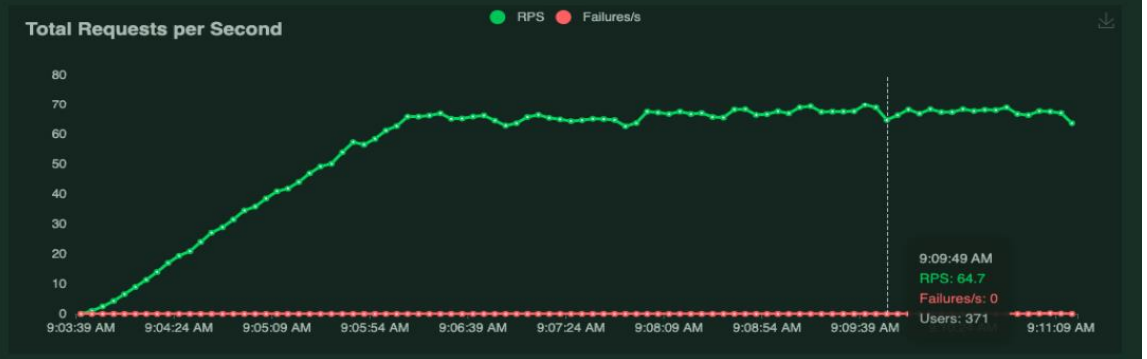
Response Time Statistics

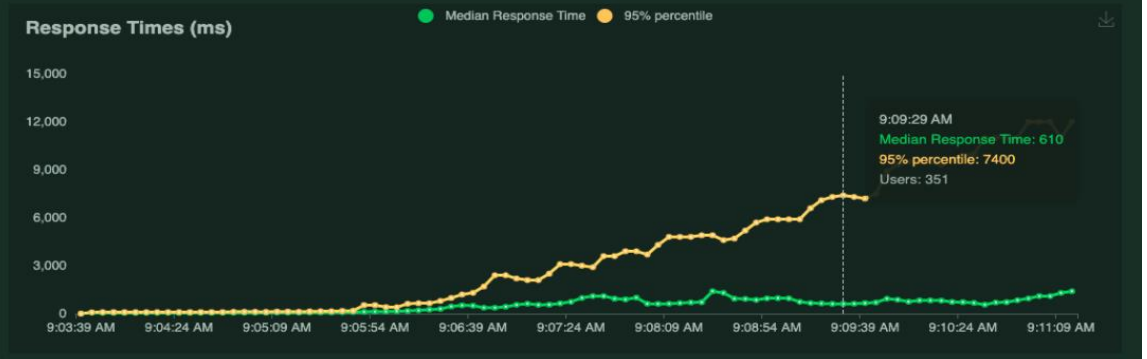
Method	Name	50%ile (ms)	60%ile (ms)	70%ile (ms)	80%ile (ms)	90%ile (ms)	95%ile (ms)	99%ile (ms)	100%ile (ms)
GET	/tasks/	550	740	1200	3600	7100	9800	12000	13000
POST	/tasks/	540	740	1200	3700	7100	9700	12000	13000
	Aggregated	**550**	**740**	**1200**	**3600**	**7100**	**9800**	**12000**	**13000**

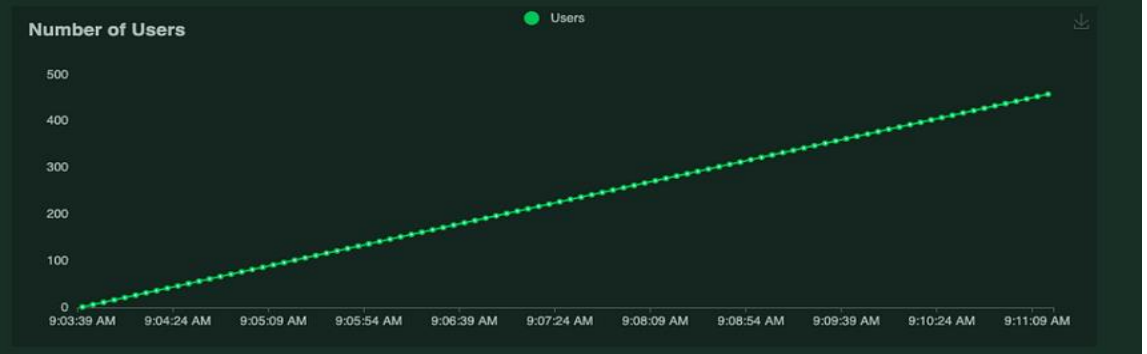
Failures Statistics

Method	Name	Error	Occurrences
GET	/tasks/	504 Server Error: Gateway Time-out for url: https://django-todo-list-grjv6.ondigitalocean.app/tasks/	3
POST	/tasks/	504 Server Error: Gateway Time-out for url: https://django-todo-list-grjv6.ondigitalocean.app/tasks/	1

Charts







Tasks

Ratio per User class

- 100.0% WebsiteUser
  - 83.3% index
  - 16.7% create

Total ratio

- 100.0% WebsiteUser
  - 83.3% index
  - 16.7% create

Load Testing – Locust 

Results and conclusions

Find below the complete report provided by **Locust.io.** Here are some observations:

- Until around 160 simultaneous users, we were having around 60 requests per second and the response time was very acceptable for this context (95% of requests finished in less than 400ms)
- The definition of what is an acceptable response time will vary depending on the context. Usually, we seek to have a response time lower than 300ms for APIs in general, but keep in mind this might change depending on the context of each system.
- After passing 160 simultaneous users, note the number of requests per second remains the same (60rps), but the response time increased a lot. This way, each user started having slower responses.
- In summary, with the available computing resources, we could serve around 160 simultaneous users with 60rps and a response time lower than 400ms. To increase these numbers, we would need more computing resources or make performance adjustments in the code.

Load Testing – Locust

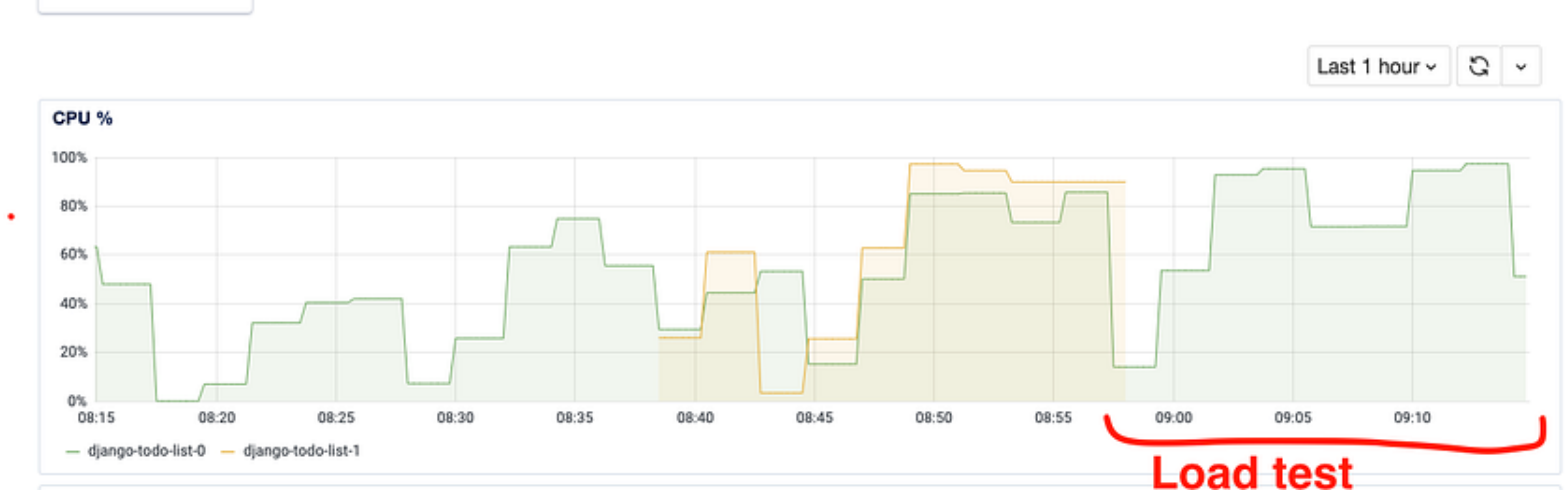
Commands to use: Htop & nvidia-smi

Understanding CPU and memory usage

Application layer

In our application layer, we see the CPU usage growing until it reached 100%. In a way, it's a good sign that CPU usage is ramping up. This means our application server is well configured and it is not leaving idle resources.

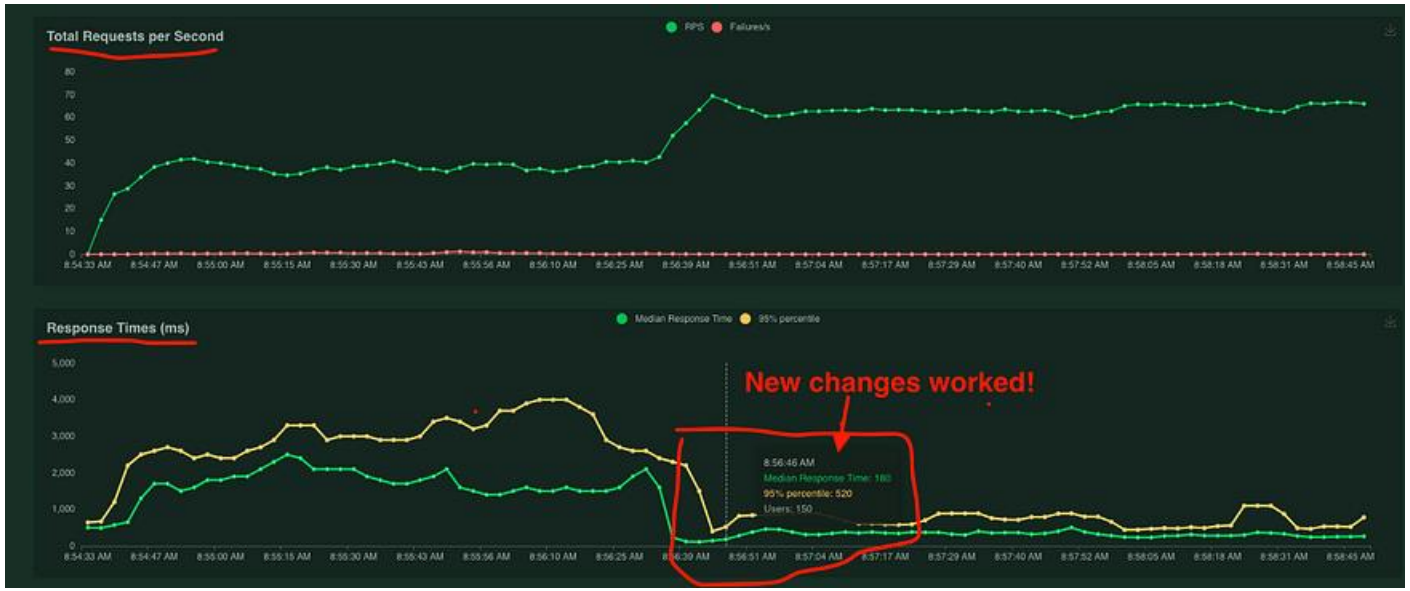
- On the other hand, since our CPU usage is getting to 100%, this also means we need more computing resources.



Load Testing – Locust

One step at a time

- When performing load tests and improving performance, it is important to make small changes, one at a time. This way we can verify how much a given change affected the results. In the screenshot below, we can see the exact point where I changed a configuration and the response time improved a lot:
- Combining small changes with monitoring (results and computing resources), we can understand how our system behaves under stress and adjust accordingly.



LAB

Load Testing – Lab

- What you'll do is to load test the deployed churn model of the previous lab:
  - Test your deployment through locust-python script
  - Observe the charts and experiment with different parameter values e.g. number of users
  - Submit your answer to those questions:
    - What's the peak RPS?
    - What other insights did you observe from the graph?