



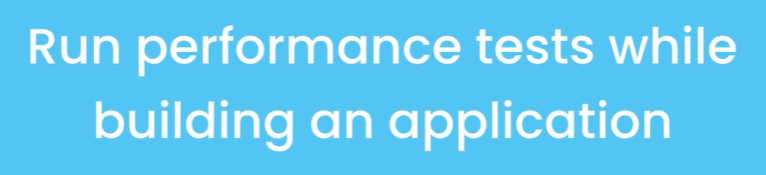
In this section we will learn about performance testing where we are going to throw a few hundred users into our application and see how it performs.

**Why Performance Testing**:

Just like automated testing, performance testing is something we need to practice while building an application, because once we go to production, we don’t want to get surprises.

Surprises may include some minor or major hidden performance problems that we tried to ignore will eventually come up.

So remember,



This is especially true if we are building a mission critical system where a slow, unresponsive application can be very costly or disastrous to the business.

For example, in an air traffic control system, downtime is simply not an option, *the system should always be up and running and response time should be in an acceptable range*.

So with performance testing we can,



Before they end up being so costly or disastrous.

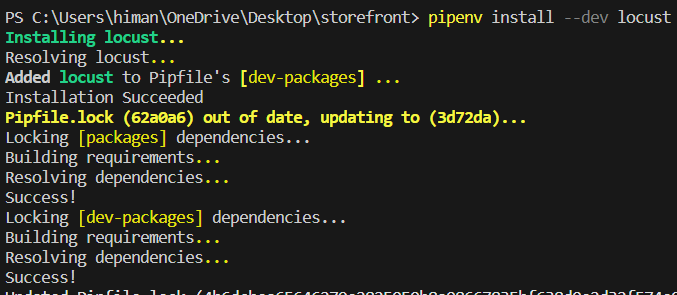
In this section, we will learn the basics or performance testing and learn a few tools that we can use to easily identify most of the performance problems in our application.

**Installing Locust**:

There are so many performance testing tools out there but in this section we will learn about *Locust* since *its really simple to use, has a comprehensive UI and we can write our performance tests using python*.

To install locust as a dev dependency:

pipenv install --dev locust



Next we will see how to write a test script using python.

**Creating a Test Script**:

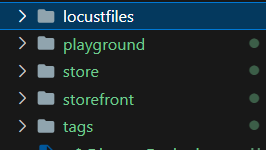
As part of performance testing, we need to identify the core use cases or the functions that are essential to our business.

In our ecommerce application, core use cases include,



*After identifying the use cases we create a test script for each use case which specifies the actions that user will take*.

In our project directory, we create a new folder called *locustfiles* (*name does not matter*).



In this folder we will create a new file called *browse\_products.py* which is the use case we are going to test.



Now in this module first we import *HttpUser* class from locust which will be used to extend our very own defined class.

from locust import HttpUser

class WebsiteUser(HttpUser):

*When we run a performance test, locust is going to create an instance of this class for each user and execute the tasks that we define in this class*.

These tasks may include *viewing products, viewing particular product details and adding a product to cart*. So for each task we can define a separate method.

Each method should be decorated with @*task* decorator which we need to import from locust.

from locust import HttpUser, task

and then method itself will be defined in our class like this,

class WebsiteUser(HttpUser):

    @task

    def view\_products(self):

        self.client.get("/store/products/")

But In a real world situation, user is going to browse from one collection to another, so here we should generate a random value for the collection-id. So import *randint* function from *random* module.

from random import randint

Then we call it and give it a range.

class WebsiteUser(HttpUser):

    @task

    def view\_products(self):

        collection\_id = randint(2, 4)

        self.client.get(f"/store/products?collection\_id={collection\_id}")

We pass the collection\_id to our URL as a query parameter.

When we run this test, for each url that is generated (*from /store/products?collection\_id=randint(2,4)*) will show a separate row in the report which is going to make our report too verbose.

*To solve this problem, we can add all the generated URLs to a particular group in order to simplify our report* ,so we set the *name* attribute to an endpoint.

class WebsiteUser(HttpUser):

    @task

    def view\_products(self):

        collection\_id = randint(2, 4)

        self.client.get(

            f"/store/products?collection\_id={collection\_id}", name="/store/products" *🡪 Here*

        )

Next we will create a task for viewing a particular product.

 @task

    def view\_product(self):

        product\_id = randint(1, 1000*) 🡪 random product id*

        self.client.get(f"/store/products/{product\_id}", name="/store/prducts/:id") *🡪 group all urls under this name*

Now a task to add a product to a cart.

Here we will limit product\_id range from 1, 10 because we want to see if adding duplicate products in shopping cart which results in increasing product quantity has any performance issues.

    @task

    def add\_to\_cart(self):

        product\_id = randint(1,10)

        self.client.post(f'/store/carts’) 🡪 ? *where is cart\_id?*

To send a post request to the /store/carts/ endpoint *we need to have a cart\_id and this cart\_id should be automatically generated for the user* when they start browsing our application.

Using ***on\_start*** hook:

For this we have a special method called *on\_start*. This is not a task, it’s a lifecycle hook which gets called every time a new user starts browsing our app.

  def on\_start(self):

        response = self.client.post('/store/carts/')

        result = response.json() 🡪 convert the result into json

        self.cart\_id = result['id'] 🡪 get the id from dictionary

Now we can reference this cart\_id in our *add\_to\_cart* task like this,

@task

    def add\_to\_cart(self):

        product\_id = randint(1,10)

        self.client.post(f'/store/carts/{self.cart\_id}/items/', name = '/store/carts/items')

Since this is a post request to items endpoint where we need to send product\_id and item quantity to the server as well, so we can use json attribute in our post request for this purpose.

 @task

    def add\_to\_cart(self):

        product\_id = randint(1, 10)

        self.client.post(

            f"/store/carts/{self.cart\_id}/items/",

            name="/store/carts/items",

            json={"product\_id": product\_id, "quantity": 1}, 🡪 *here*

        )

Assigning *weight / priority* to each task:

In a real world situation, its more likely that a user views different products than adding products to the shopping cart.

class WebsiteUser(HttpUser):

    @task(2)

    def view\_products(self):

        collection\_id = randint(2, 4)

        self.client.get(

            f"/store/products?collection\_id={collection\_id}", name="/store/products"

        )

    @task(4)

    def view\_product(self):

        product\_id = randint(1, 1000)

        self.client.get(f"/store/products/{product\_id}", name="/store/prducts/:id")

    @task(1)

    def add\_to\_cart(self):

        product\_id = randint(1, 10)

        self.client.post(

            f"/store/carts/{self.cart\_id}/items/",

            name="/store/carts/items",

            json={"product\_id": product\_id, "quantity": 1},

        )

    def on\_start(self):

        response = self.client.post("/store/carts/")

        result = response.json()

        self.cart\_id = result["id"]

Here for viewing product task weight is 4 while viewing product in a collection is task weight is 2.

*Because user is twice more likely to execute this task than viewing products in a collection*.

For adding item in cart, task weight is 1 because it is less likely to happen then the other two tasks.

Adding *wait\_time*:

We don’t want these tasks to happen right after one another because in real world, it takes a user a few seconds or a minute to execute each task.

So we set a new attribute called wait\_time and use the *between* function from locust to set a time value in seconds.

from locust import HttpUser, task, between

from random import randint

class WebsiteUser(HttpUser):

    wait\_time = between(1, 5)

    @task(2)

    def view\_products(self):

        collection\_id = randint(2, 4)

        self.client.get(

            f"/store/products?collection\_id={collection\_id}", name="/store/products"

        )

    @task(4)

    def view\_product(self):

        product\_id = randint(1, 1000)

        self.client.get(f"/store/products/{product\_id}", name="/store/prducts/:id")

    @task(1)

    def add\_to\_cart(self):

        product\_id = randint(1, 10)

        self.client.post(

            f"/store/carts/{self.cart\_id}/items/",

            name="/store/carts/items",

            json={"product\_id": product\_id, "quantity": 1},

        )

    def on\_start(self):

        response = self.client.post("/store/carts/")

        result = response.json()

        self.cart\_id = result["id"]

Locust is going to randomly wait between 1 to 5 seconds in between each task.

So *when we run this test, for each user that we simulate locust is going to create an instance of WebsiteUser class and it will repeatedly call these tasks and apply a wait time after each task*.

Note: The beauty of this approach is that our test script is a part of our codebase, so we are going to commit it to our repository and every time we change our code, we can rerun our performance tests to see if we have any issues or not.

**Running a Test script**:

In this lesson we will put a print statement inside each task in order to understand how locust execute each script.

    @task(2)

    def view\_products(self):

        print("view products") *🡪 Here*

        collection\_id = randint(2, 4)

        self.client.get(

            f"/store/products?collection\_id={collection\_id}", name="/store/products"

        )

    @task(4)

    def view\_product(self):

        print("view product details") *🡪 Here*

        product\_id = randint(1, 1000)

        self.client.get(f"/store/products/{product\_id}", name="/store/prducts/:id")

    @task(1)

    def add\_to\_cart(self):

        print("Add to cart") *🡪 Here*

        product\_id = randint(1, 10)

        self.client.post(

            f"/store/carts/{self.cart\_id}/items/",

            name="/store/carts/items",

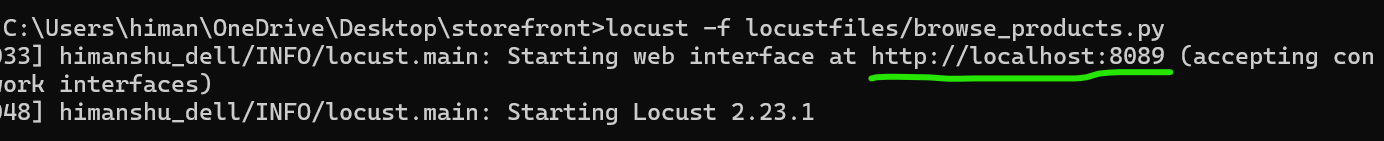
            json={"product\_id": product\_id, "quantity": 1},

        )

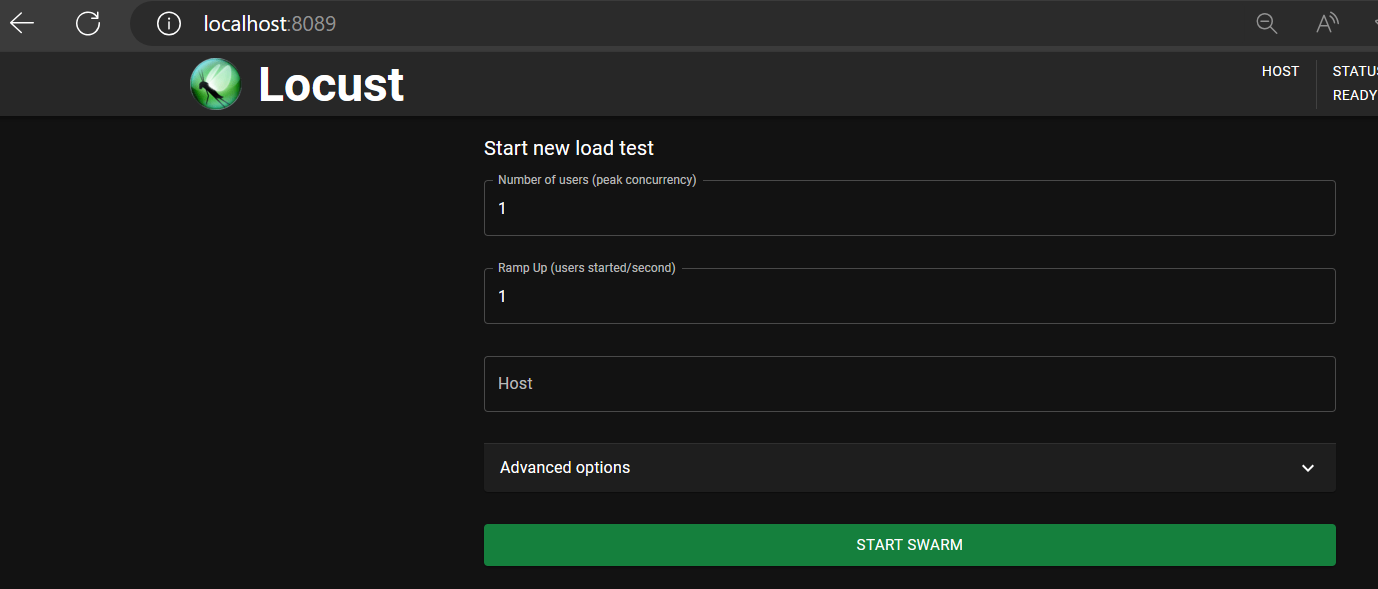
And make sure our development server is running.

In other terminal window, we run,

locust -f locustfiles/browse\_products.py

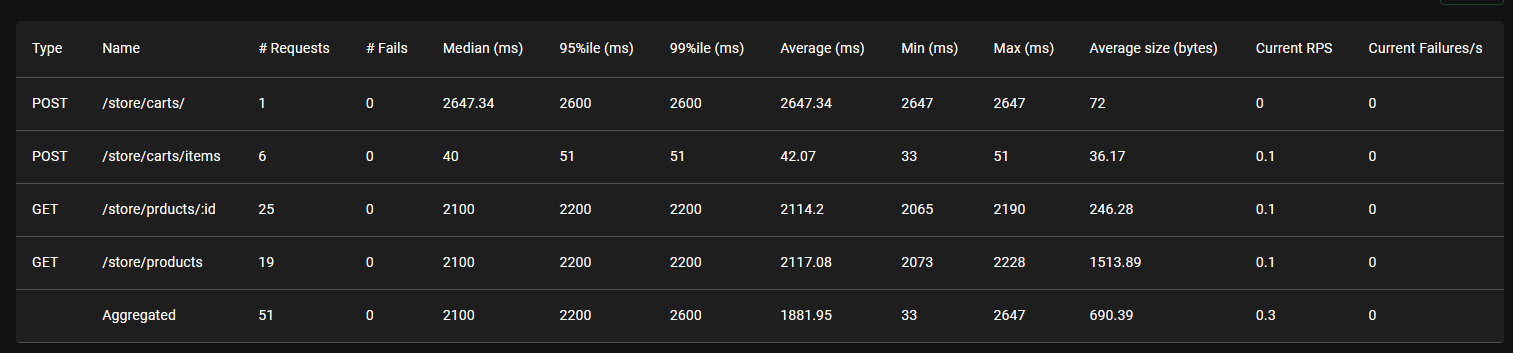


Now, locust is running at <http://localhost:8089>.

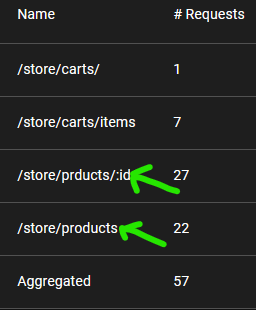


On this page we can specify number of users, spawn rate(number of users spawned per second) and for host we can specify <http://localhost:8000>. (*make sure don’t add* ***/*** *at the end because our URL endpoints in test script already have* ***/*** *prefixed*).

As soon as we start swarming, on the statistics page we can see requests.

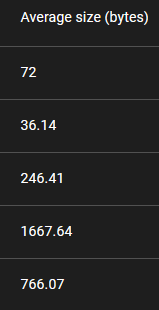
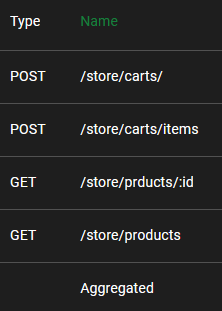


Notice the grouping that we specified for endpoints

 If we did not apply grouping we would end up with tons of rows in this table.

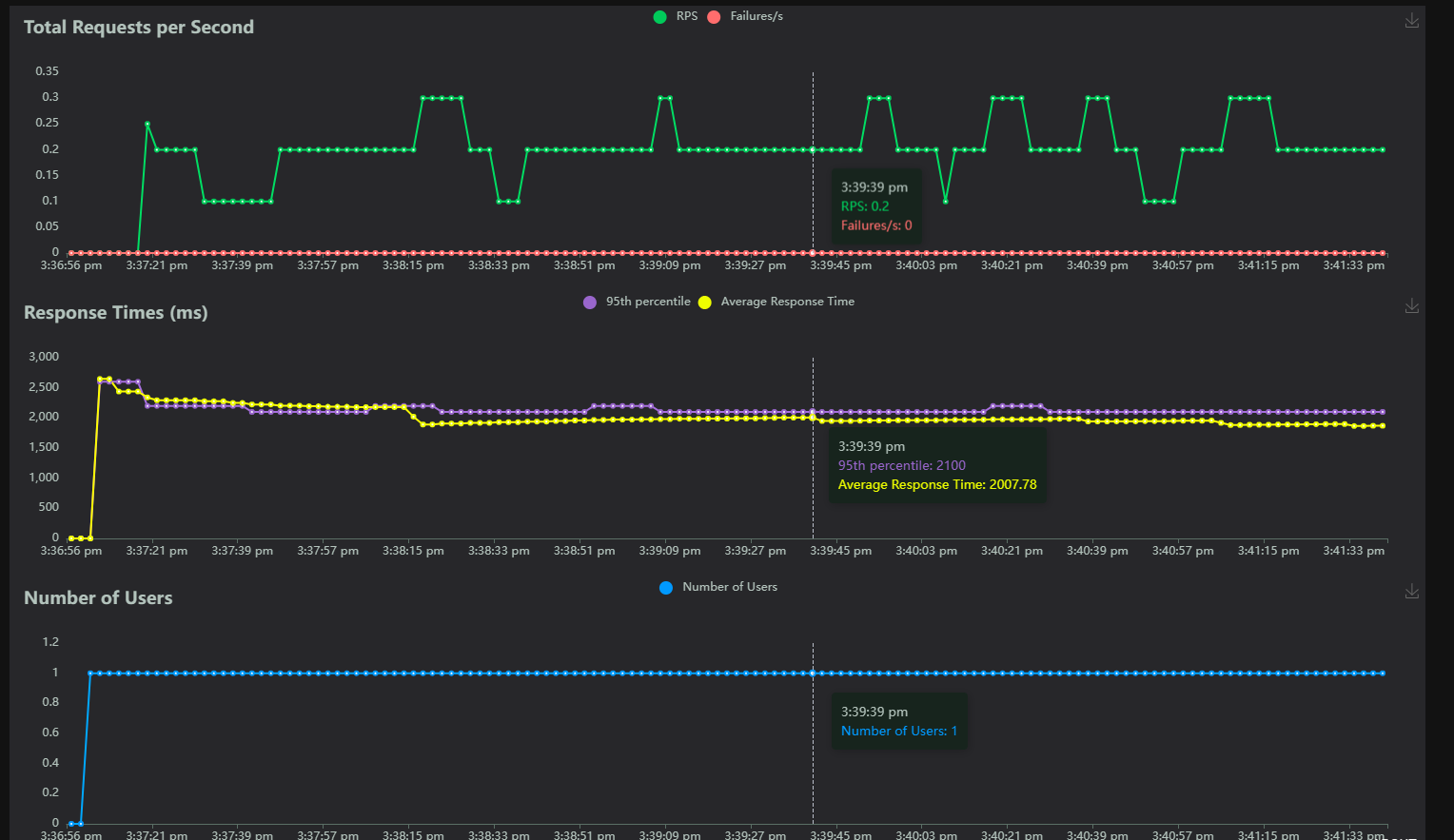
We can also see we have more requests sent to /store/products/:id endpoint because of more weight over this task.

99%ile(ms) 🡪 *indicates maximum response time that only 1% of requests exceed. For example if 99%ile is 100ms that means 99% of requests completed in 100ms or less*. Same goes with 95%ile as well for 95% of the requests.



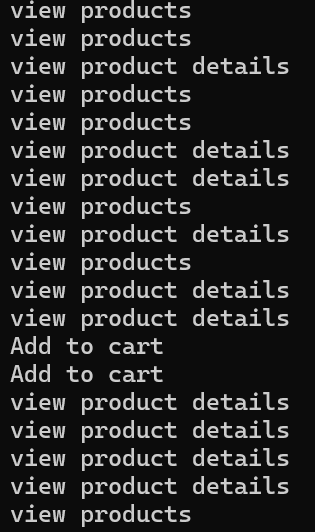
Notice the average size in bytes, for products endpoint it is quite high as compared to other endpoints but its not because of any issue, it is rather by design.

In the charts tab,

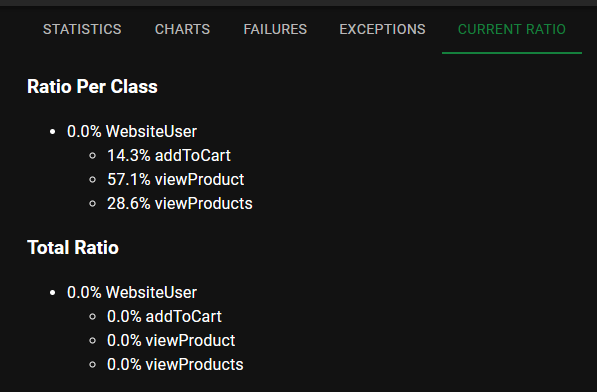


In this graph we can see Locust has started one user and that user is continuously browsing our site with tasks that we defined.

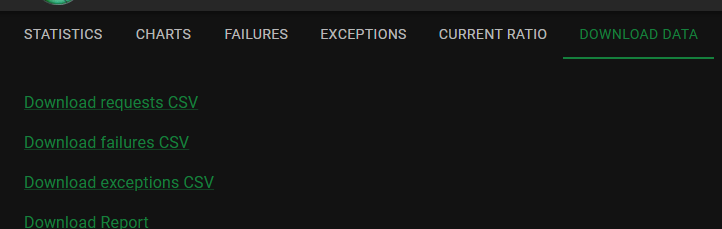
In the terminal window, we can see what the user is doing from our print statements.



In the current ratio tab,



We can also download all this data in csv from download data tab,



In next lesson we will do a proper performance test to identify our slow endpoints.

**Running performance test**:

In this lesson we will deliberately create a performance problem.

Note: In this lesson a prefetch related method from queryset is removed from the ProductViewSet to see the effect of this change on performance.

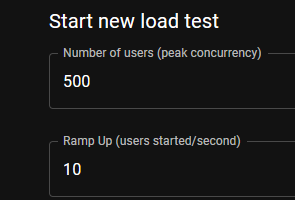
class ProductViewSet(ModelViewSet):

    queryset = Product.objects.all() 🡪 *prefetch\_related(‘images’) is removed*

Note: Comment out django debug toolbar from the list of middleware as it add some overhead in the performance.

Note: To start a clean test, stop the locust server using ctrl + C from terminal in order to have no data from previous test.

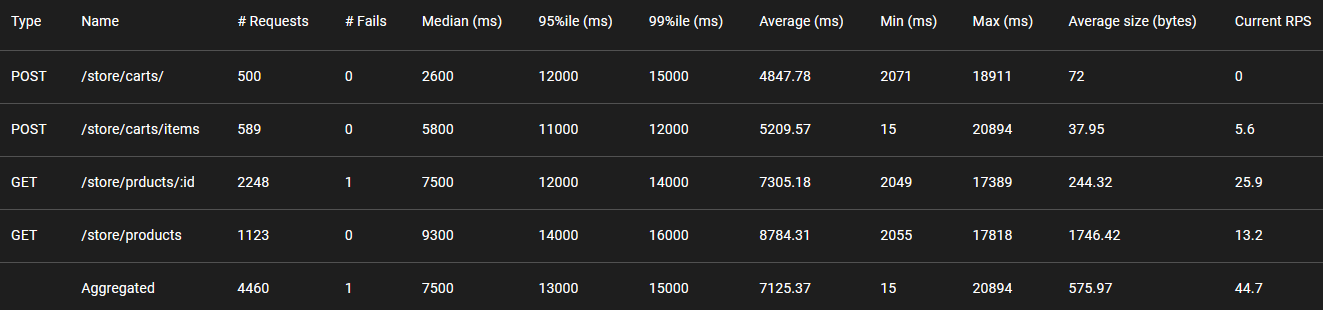
Let’s keep these settings for testing performance,



Every second locust is going to spawn 10 users until we get to 500 users and those 500 users will continuously browse our website.

Note: The webserver that comes with django is meant for development it does not have performance of a real production ready webserver, so if we run these tests on production environment we are going to see different results.

So don’t treat any of the values like median / 99%ile / 95%ile as absolute but rather relative values.



**Performance Optimization Techniques**:

So by running a performance test we can identify our slow end points. Now let’s talk about few optimization techniques. 90% of the time the issue is either in the query or the database.

***Step 1***: *optimize the python code*:

To resolve this first step we can do is optimize the python code, since we are using Django ORM to execute queries, so we should make sure that our python code does not translate to costly queries.

We can preload related objects using *select\_related*, *prefetch\_related*.

Product.objects.select\_related('...')

Product.objects.prefetch\_related('...')

We can also load what we need,

Product.objects.only('title')

Product.objects.defer('description')

Using the *only* method we can *load only the fields that we are interested in*. We also have *defer* which is the opposite of only, so *if we have a field with a lot of data that we are not going to show to the user, then we can defer it* by calling *defer* method.

We can also use *values* and *values\_list* method.

Product.objects.values()

Product.objects.values\_list()

*With values we get a dictionary with values\_list we get a list and we know that it’s cheaper than initializing a Django model*. So if don’t need any behavior in django model like creating, updating or deleting then we can optimize by using one of these methods.

Another thing we need to watch for is counting objects. So if we want to count products, the right way is to use *count* method instead of pulling all product objects in memory and then call the *len* function to count them.

Product.objects.count()

len(Product.objects.all())  # BAD

Last thing is *bulk updating or bulk creating*. If we need to create or update multiple objects, its more efficient to use *bulk\_update* or *bulk\_create* method instead of creating or updating a bunch of objects in a loop.

Product.objects.bulk\_create()

Product.objects.bulk\_update()

It’s because with bulk\_create command we send one instruction to database to create multiple records.

What to do if our queries are still slow?

***Step 2***: Rewrite the query from scratch, because the query that django ORM generate may not be optimal. So if you know SQL well this is where you may want to rewrite the query from scratch in an optimized manner.

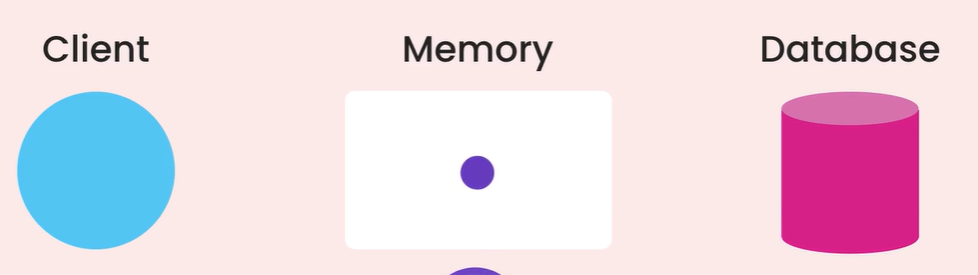
If queries are still slow?

***Step 3***: Tune the database by redesigning our tables, adding indexes and so on.

If queries are still slow?

***Step 4***: Then *we will store the results in memory with a technique called caching*.

First time the query gets executed, it is going to be slow. But then we will store the results in memory and all subsequent requests will read the data from memory which is faster than reading it from a database or network.

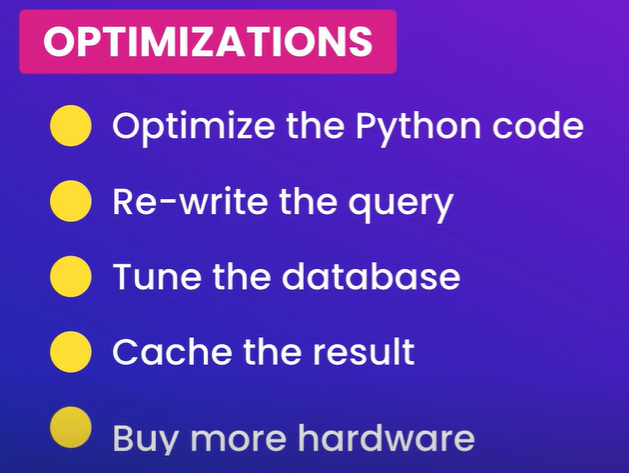


Note: Caching is not always a good strategy because sometimes executing a simple query is faster than reading the result from cache, especially in production environment where we have a separate cache server. So reading a data from cache incurs a network call.

If after doing everything, our performance tests show that after a certain point let’s say 1000 concurrent users, our application fails, so some requests never get a response or they take way too long.

***Step 5***: Buy new hardware, if you can afford. So we can upgrade our server to a faster CPU and more RAM.

***Summary***:



Just remember…



**Profiling with Silk:**

With locust we can identify our slow end points, as we know that most of the time issue lies in the query or in the database. This is where we use another amazing tool to identify source of the issue called *django*-*silk*.

This django-silk is what we call a *profiling tool means we can use it to identify the execution profile of our application*. So we can see 🡺 how each function gets executed

🡺 what queries are sent to the database

🡺 what time is spent on those queries and so on…

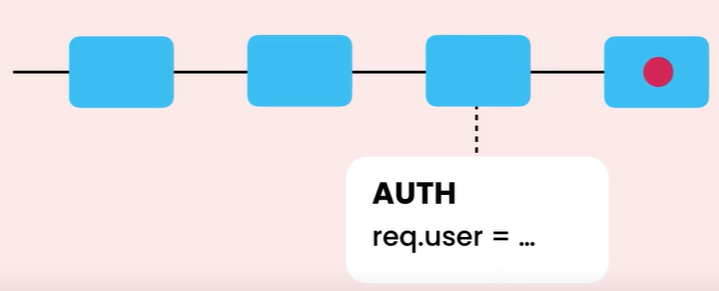
Installation instructions

<https://github.com/jazzband/django-silk>



Placement of this middleware is important.

*Side Note about how middleware process requests*: Each middleware function takes an incoming request, add something to it and then passes it to the next middleware function in this list. For example AUTHENTICATION middle ware adds the current user to the request.



Now a middleware function can choose to process the request, if that happens, the request will not be passed to the next middleware function and that will be the end of requests lifecycle.

*Since SILK is going to intercept our request, we need to make sure that request gets passed to this middleware. So the request should not gets processed by a middleware function before SILK’s middleware*.

So let’s put it at the end,

   "django.contrib.auth.middleware.AuthenticationMiddleware",

    "django.contrib.messages.middleware.MessageMiddleware",

    "django.middleware.clickjacking.XFrameOptionsMiddleware",

    "silk.middleware.SilkyMiddleware",

]

And put it in the list of installed apps,

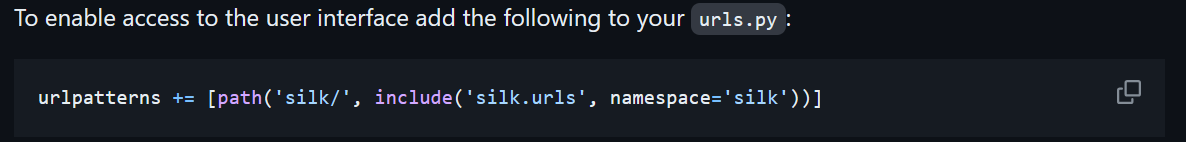
    "rest\_framework",

    "djoser",

    "silk",

    "playground",

Next we register a URL pattern,



   path("auth/", include("djoser.urls.jwt")),

    path("\_\_debug\_\_/", include("debug\_toolbar.urls")),

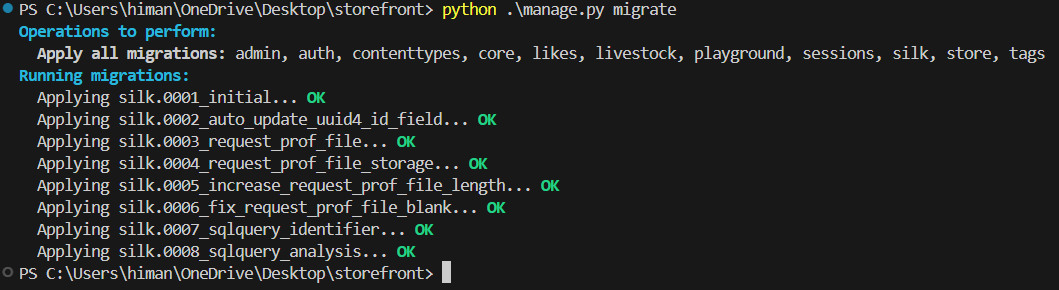
    path("silk/", include("silk.urls", namespace="silk")),

]

*Note*: Only use silk in debug mode and never in production, because it adds significant overhead on requests.

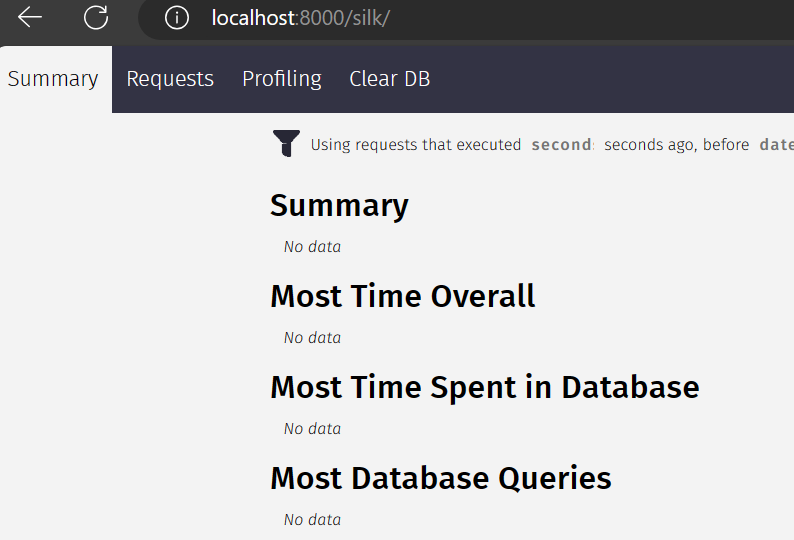
And run migrations to create database tables, optionally we can also collect static files. Silk create a new table for storing information about each request.

python manage.py migrate



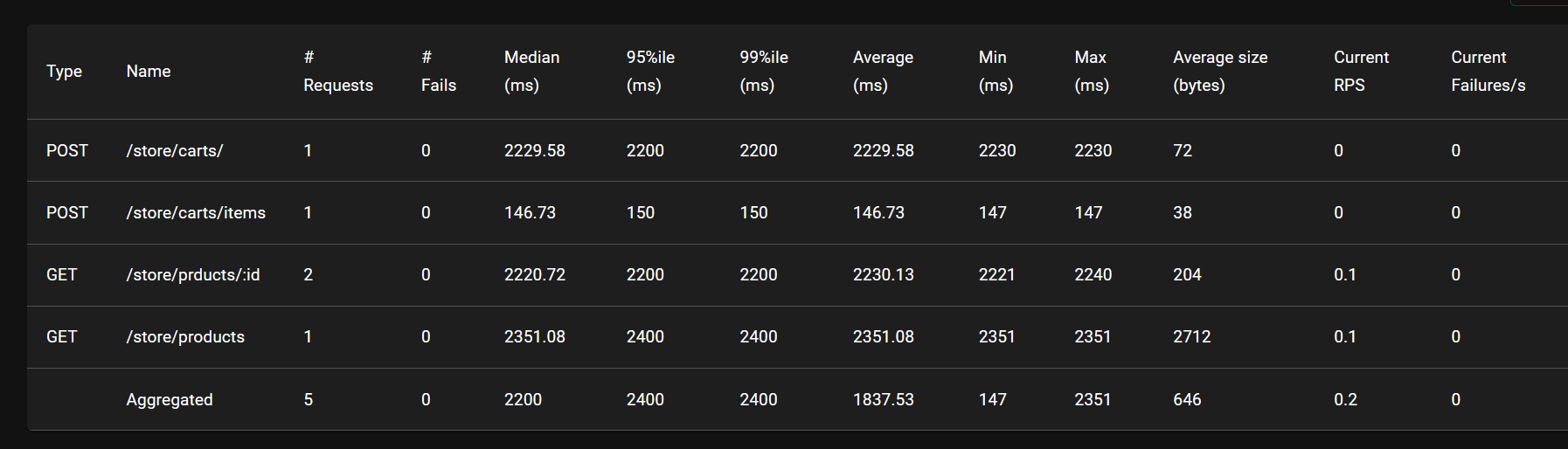
Now we can access silk on this endpoint

<http://localhost:8000/silk/>

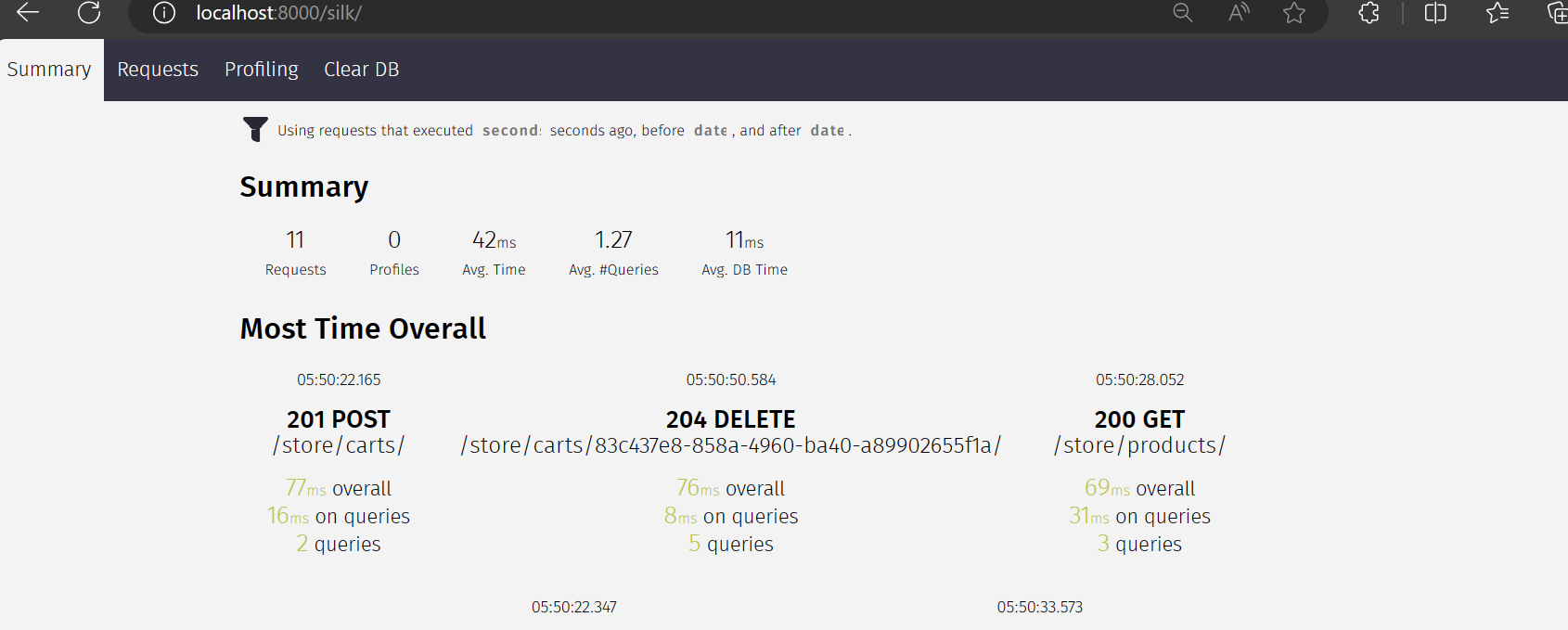


This is the silk dashboard, because we currently don’t have any data. But now if we start browsing our application, silk is going to intercept every request and collect some information for us here.

But If we don’t want to manually browse our application, we can do an automatic performance test.



Now if we refresh our silk page dashboard,



*Continue later*…