My answers to the challenge questions are at the end of this file. First, I would like to walk you through my changes.

Side note: It was my personal understanding that an order could only have one order detail. By the time I've realized that wasn't the case it was already too late for me to make the changes. That's why you will see something like order["order_details"][0]["product_id"] in my code. With that in mind, I hope this doesn't take away from my solution to the proposed challenge.

1st commit
This commit is pretty straight forward. I am adding a delete product rpc call. In the gateway file, I add the endpoint for deletion and I return 204 as a response code. Additionally, in the dependency file, I try to delete a product from redis and if it is not found I raise a NotFound error.

2nd commit
In this commit I am adding tests for my delete product rpc call. The test_can_delete_product is checking if the response is null and with code 204 (deletion). The test_delete uses a mock storage to test if the deletion is working. Finally, test_delete_product creates a product and tests its deletion right after.

3rd commit
Here I am adding the order list rpc call in the gateway service and in the order service so they can communicate.

* In the gateway file you'll see that for every order (in PostgreSQL) I have to fetch a product (in Redis).

4th commit
This commit goes back to my side note in the beginning of this document.

5th commit
Adding the tests for list orders. The test_can_list_orders gets the list of orders and makes sure that every order detail has a product associated to it
test_list_orders_empty tests the list orders with empty response.
test_list_orders tests to see if a list with more than one element is returned, check its content as well

6th commit
Wire both delete product and list orders into smoketest.sh.

7th commit
Wire both delete product and list orders into perf test. It is worth noting that the performance tests will have errors because we are introducing the product deletion as part of the perf test. This will result in problems when listing the orders, since the listing of orders fetches a product for each order, and some products might have been deleted by the flow of the execution. This is not a problem in our overall analysis, but it is worth mentioning.

8th commit

In this commit I tackle one side of the performance issue. In the gateway service, for every _get_order the code would fetch all the products from redis so it could check against the order details. Part of the enhancement is to get the product directly from the database instead of listing all of them and saving into memory.

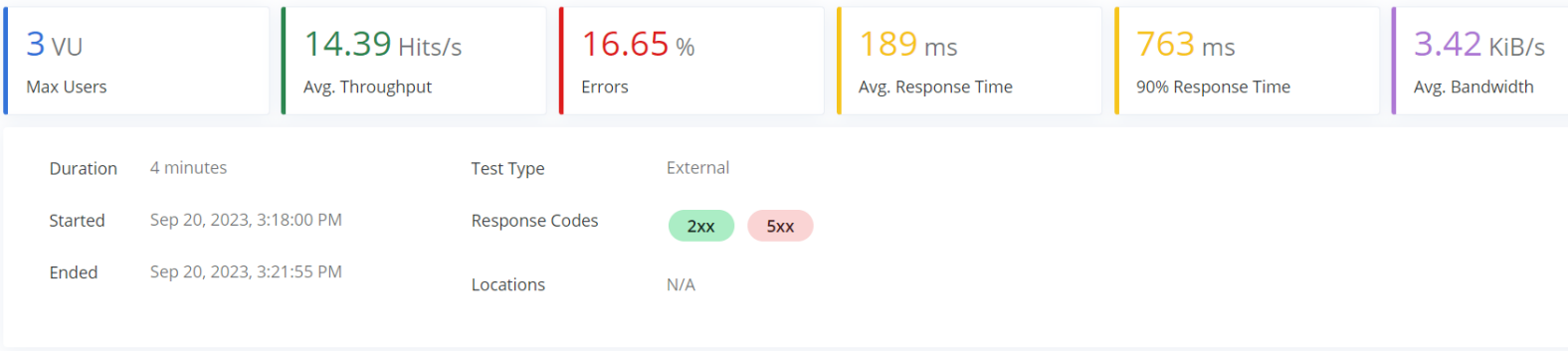
In the _create_order we find the same case. Line 221 of this commit will raise an error 500 (not ideal) if it can't find a product. The reason behind this is that you cannot create an order of a product that doesn't exist.

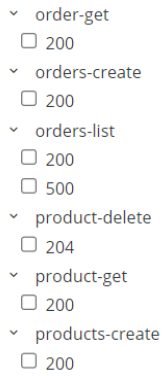
9th commit

Because I changed the GET order and POST order in the previous commit, I had to make adjustments in the tests of those functionalities.

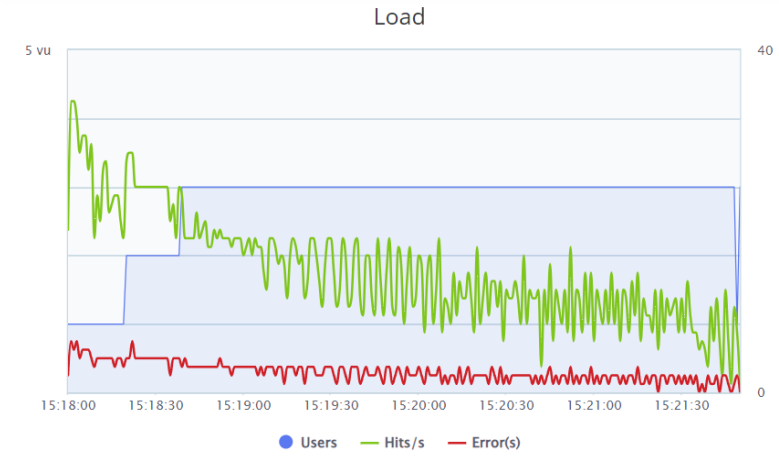
Perf test

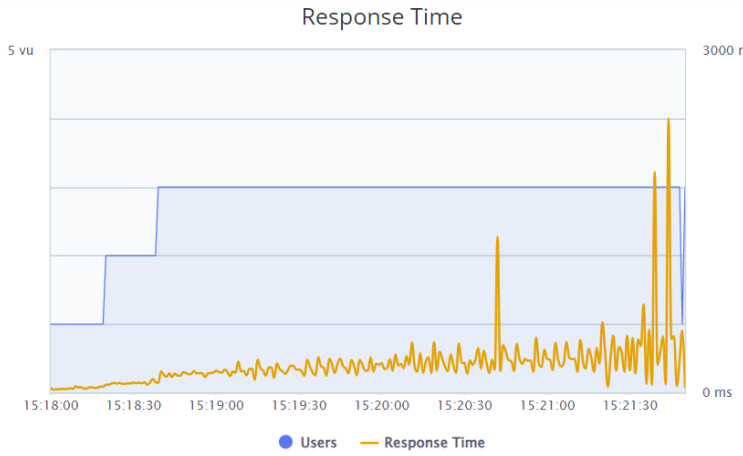
Now I would like to show the performance test results and explain a bit about them.



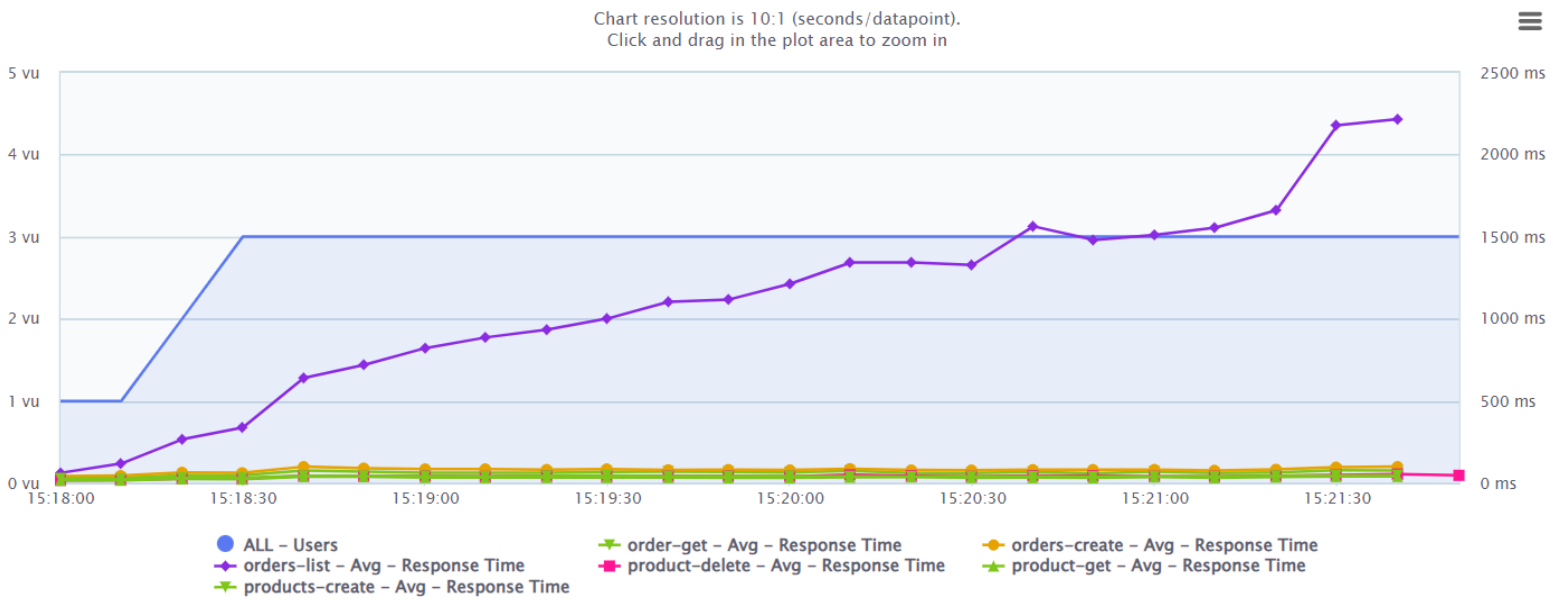


Like I said in commit 7, we will have errors introduced in the perf test, but this doesn't invalidate them. Check commit 7 to understand why ***orders-list*** have codes 500.





These green spikes are the hits, and they are expected to decrease with time, especially because of our main bottleneck here, the ***orders-list***, which I will talk about more below. The spikes in the response time are also from the ***orders-list*** as we will see in the next images.



This shows the average response time for each endpoint. Before my enhancement (commit 8), ***orders-create*** and ***orders-get*** were much higher in the graph, so the change in the code seems good. Now, the bottleneck still continues, ***orders-list*** keeps increasing in time because products keep getting created as part of the perf test.

Average Throughput
14.4
Hits/s

Avg. Response Time
189
Milliseconds

90% Response Time
763
Milliseconds

Error Rate
16.65
%

TOP 5 SLOWEST RESPONSES (BY AVG. RESPONSE TIME) 5

Request	# Samples	Avg Time	90% Time	Max Time
orders-list	552	893 ms	1556 ms	2964 ms
orders-create	552	75 ms	101 ms	177 ms
order-get	552	58 ms	78 ms	132 ms
products-create	552	38 ms	48 ms	179 ms
product-delete	550	37 ms	49 ms	102 ms

* Even with my code enhancement it takes much longer than the other requests because for each order in the list we need to fetch something from the database. In this case the context switch (from PostgreSQL to Redis) slows it down a bit. A possible solution for this would be to make the product_id a foreign key to order_details, and thus bring them into one database only. This could only improve so much and I am aware that separate services may use separate databases, so another solution would be to create indexes in the postgreSQL tables as well as in the Redis hashes.

I will summarize the * points in this document into an answer.

Why is performance degrading as the test runs longer?

With the introduction of ***orders-list*** it becomes more visible. But even before I added, the performance would get worse as time progressed.

1-) As the products database grew in size, there wasn't an efficient way to fetch a certain product. For every order creation and order get operation the code would have to get all of the products in the product database. And when I introduced ***orders-list*** the code has to get a product for every order before it prints them all.
2-) Read and write on two separate databases. Switching the context will add some time between operations.
3-) The Products model is not related to the Order model or the OrderDetail model.

How do you fix it?

1-) Make the product_id a foreign key to order_details, and thus bring them into one database only.
2-) Create indexes in the postgreSQL tables as well as in the Redis hashes.
3-) Use caching in postgreSQL.