6 On Software Performance

Approaching the end of the project, we started on optimizing Dora to make it work faster and be more responsive to the user. There were a few potential bottlenecks we identified and proceeded to test them out. These are the potential bottlenecks and what we did to verify and rectify them:

1. Database Querying
2. Bad Data structures or Algorithms
3. Network Constraints
4. Memory Leaks

6.1 Techniques for Performance Improvement

The first step to improve the performance of the app is to first identify the real bottleneck. The approach we used was a top down approach where we logged the timings for each method and narrowed down the possibilities of where the bottleneck was.

The query we used to test the performance was “gender: M”, which would cause the Dora Server to return a JSON String with a little more than 5000 results. This large number of results allowed us to locate which part of our application is slowing it down. The table below shows in a glance the tests and the running time for each of them.

Test No.	**Description**	**Running Time/s**
1	Initial test on **query()**	50.82709
2	Test on query() after **removing Django templating** and **employing json library**	43.99224
3	Test on query() using **cjson library**	43.76312
4	Test on query() using **ujson library**	43.59803
5	Timing for **views.parse_request()**	00.00218
6	Timing for **get_query_result_set()**	00.00092
7	Timing for **create_json_response()**	43.96829
8	Timing for creating each query result python object	00.00841
9	Timing for creating all query result python objects	43.73904
10	Timing for converting from python object to JSON	00.03768

11	Timing when we removed unused attributes from response and removed indirection when dealing with “subject”	40.90047
12	Timing when locations filter if/else block is commented out	30.50749
13	Timing when we remove the assigning of query set attributes to python object and if/else block is commented out	30.38654
14	Timing when we added select_related when querying Encounters in **get_query_result_set()**	21.44122
15	Timing when we switched to deployment mode	19.66959

Table 6.1 - Tests made and timing results

From our initial tests, we found that the query method took 50 odd seconds and we tried optimizing it by removing the need to go through an extra layer of Django Templating. That reduced the timing to 43 seconds.

Thinking that the problem was in the conversion of the query result set into JSON Strings, we tried to use the python cjson library as we thought that C would be much faster than python. Afterwhich we tried the python ujson library, and concluded that the three libraries had minimal difference in performance improvement.

This was when we realized the problem was not in the conversion of the query but the generation of the query itself, and suspected the database data retrieval was the big problem. The next steps we took were to further analyse each method called (steps 5-7) and realized that the problem was in the `create_json_response()` method.

After further exploration (steps 10-16), we arrived at our optimal solution for this project, reducing our initial test timing by 61%. Here were the steps we took and our findings:

6.2 Reducing Unnecessary Indirections

One of the simplest optimizations was to store a local reference to an object that we had to use multiple times in our code. That is when we assigned queryset.subject (which was always called and we suspect, evaluated every time) to be an object “subject”. This meant that this object was only evaluated once and this simple step improved performance by 4 seconds.

6.3 Further Identifying Bottlenecks

The next thing we did was to comment out the if/else block in create_json_response() which gave us an insight as to which sub component of the method is giving us the problems. We realized that commenting out the if/else block reduced the timing by 25% to 30 seconds.

We tried to come up with a better way of querying the location to make this improvement permanent in our app. However, there was no visible improvement observed. This showed that the part that was slowing the system down was when we were accessing the attributes of each query in the query result set.

6.4 Reducing Unnecessary Database Accesses

After further research on the topic, we found out that it could be that the program was querying the database every time we accessed query_result.subject and when query_result.procedure and also query_result.observer. That resulted in a lot of overhead which slowed the program down.

We implemented .select_related() command when getting our initial set of Encounters as shown in the snippet below:

```
Encounter.objects.select_related('subject', 'observer', 'procedure').filter(query)
```

By doing this query, it tells Django to retrieve everything at one go when getting the information from the database. This means there is only one pulling of the information and there will not be the case of multiple database accesses per query_result.

This reduced the timing from 40 seconds down to 21 seconds as in Test 14 shown in Table 6.1. As a finale, we switched to deployment mode and found that the timing reduced a further 2 seconds, clocking in at 19 seconds.

6.5 Checking For Network Latency Issues

The next thing we did was to check for latency issues. The tool we used was Chrome's Developer Tools in-built to the application. It allows us to simulate a slow network. This gave us ballpark figures of the speed of sending the JSON String over from server to client.

The table below summarizes our findings:

Test	Description (with the query "Gender:M")	Time Taken/s

No.		(for data transfer)
1	Without any network throttling	00.580
2	Network capped at 30Mbps with minimum 2ms RTT	00.967
3	Network capped at 2Mbps with minimum 5ms RTT	06.782
4	Network capped at 750Kbps with minimum 100ms RTT	22.315
5	Network capped at 250Kbps with minimum 300ms RTT	50.755

Table 6.5 - Network constraints and timing results

We noted that the time taken to make a query was largely affected by the network latency when the network got below the 2Mbps mark, where it would start taking 40 seconds in total to make a query.

We considered Message Packer which was supposed to help compress the JSON file into a smaller format. However, there was no well supported Javascript Decoder for Message Packer, which wrote off this idea. The case was similar with BSON. The reason we did not go with the experimental builds was because we felt that getting results slowly was better than getting results wrongly.

We also evaluated that although the application was to be deployed in areas where technology might not be very developed, the usage of our branch of the application would be in an office where the managers will be able to analyse their resource allocation. In these offices, we expect the network speed to match 2Mbps which was benchmarked by Chrome as the speed of a DSL connection.

6.6 Checking For Memory Leaks

The last step we took was to check if there were any memory leaks in our web application. The tool we used to check this was also the Google Developer Tool built into the Chrome Browser. When we clicked on timeline, we were able to see how the usage of memory changes with time. Here are the steps that were taken to check for memory leaks:

Step 1: Start Recording
Step 2: Refresh
Step 3: Query gender:m
Step 4: Query diagnosis:fever
Step 5: Cluster On/Off
Step 6: Maximize Slider
Step 7: Click on patient in encounter list