LO3 Supporting document

3.1 Range of techniques

We have been using various range of techniques for our requirements. Here is a reminder of our two chosen requirements:

- R1: The system should allow users to get view their orders stored in the database
- R2: The system should be able to handle a high number of requests and process them in a timely manner.

We have been using systematic testing for both our requirements as it follows an organised approach (identifying the requirements, outline testing goals, etc..) to ensure that all aspects of the system are throughly tested. Because systematic testing is repeatable, test cases can be executed multiple times, allowing devs to identify any new issues that may have been introduced in later iterations. Furthermore, because systematic testing is done in a controlled environment, the testing process is done under specific conditions and with known inputs, allowing the team to reproduce and isolate any issues that may occur.

We have also performed structural testing with different levels of test, such as unit tests and integration tests (and system test for performance testing-R2). These are great to test R1 as it allows to verify that the system is implemented correctly and that the different components of the system work together as intended. This helps to identify any issues or defects that need to be fixed and ultimately improves the quality and reliability of the system.

For performance testing, we have done load testing and simulated various scenarios to put the system under heavy load and test the system as a whole. Note that the Test Plan from LO2 refers to the scaffolding and code instrumentation that was required for these techniques for our chosen requirements.

3.2 Evaluation criteria for the adequacy of the testing:

For unit tests, we can use code coverage as a metric. A higher code coverage would indicate that more of the code is being tested, which helps to build confidence that the system is working correctly. We could also use speed: faster tests mean that it is

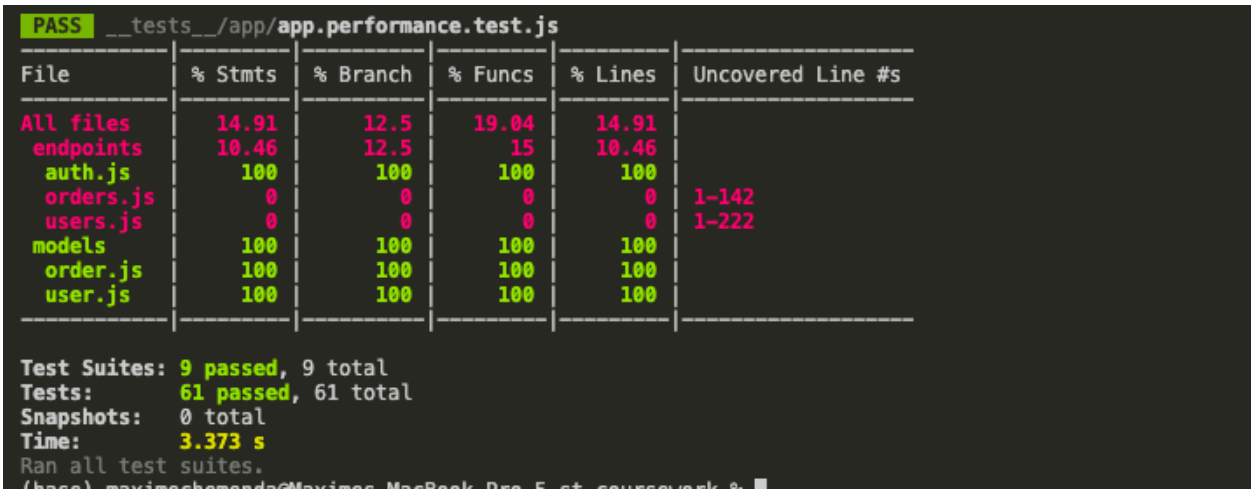
possible to run the tests more frequently and make changes to the codebase with more confidence.

Regarding integration tests, we can use functionality as an evaluation criteria, which gives confidence that the different components of the system are working together as expected. The unit tests provide confidence that the individual units of code work as expected, and the integration tests provide confidence that the system functions correctly as a whole.

Finally, for performance testing, we could use, as an evaluation criteria, the response time (how long it takes for the system to respond to a user request). Lower response times would ensure that users are experiencing a good user experience. We could also measure throughput; higher throughput would give confidence that the system can handle a large amount of traffic. We could also use other metrics, as specified in the test plan, such as memory usage, network usage or even CPU usage. By using these evaluation criteria, developers can build confidence that the software meets performance requirements and that the system is performing well under different loads and conditions. The performance tests using Artillery helps to identify and troubleshoot any performance issues early on in the development process, which can be addressed before the system is deployed to production.

3.3 Results of testing:

Unit and integration tests:



Here are the logs of performing unit and integration tests. We see that we have in total 61 tests, with 9 test suites in total, which all passed, within 3.4 seconds. This can give us great confidence in confirming that requirement R1 is satisfied. Furthermore, we see that we get excellent code coverage for both user and order models, and for the User Auth module (100%). However, I didn't have enough resources to get the code coverage for the remaining files from the endpoints folder (orders.js), but this is because Jest is not able to track the code coverage for the lines of code that are executed during the axios requests because they are not being run within the JavaScript runtime that Jest is monitoring. One way to solve this problem is to use a library like nock which would allow to mock http requests made using axios. This way, tests can make requests to the mocked endpoints, and Jest will be able to track the code coverage for those lines of code.

Load testing:

For the various tests we performed, we created virtual users and simulating various types of requests (registering users, placing orders etc…) using Artillery. We depict here the results when changing the arrival rate of requests, as these are the ones with the most significant change, and we omit results from other tests we performed, such as changing the amount of users created, and the tests duration.

From the table below, we see that the percentage of responses with a status code of 409 (meaning that a conflict occured) is around 35-40% for all arrival rates. This is not a great number, and it means that it is not the increase of load that causes this error, as this value is already big at the lowest arrival rate. It is important to identify the part of the

system that causes this and improve it. Furthermore, when looking at the median response time, we see that there is a sudden spike from 10 requests per second to 100 requests per second. This is clearly indicating that load pressure is influencing these results. However, despite the increase, it is important to consider whether the result for 100 requests per second is acceptable. It seems that 19ms is acceptable, especially for a MVP. Indeed, considering that for the moment, it is a MVP, we shouldn't expect to have more than 10 requests per second. However, it could be more concerning as soon as marketing starts being done, and that a lot of users are expected to arrive on the platform. Again, we would need to identify if a response time of 19ms is a problem or not.

Tests performed for a duration of 10 seconds with a varying arrival rate of requests:

Arrival rate of requests	1/second	10/second	100/second
409 code	40%	37%	35%
vusers failed	0%	0%	0%
median response time	2ms	2ms	19ms

3.4 Evaluation of the results

Recall that for unit and integration testing, we are using functionality as an evaluation criteria. Based on our results, we see that functionality works as expected for R1, meaning that we are satisfied with this part of the evaluation. Code coverage (our other criteria) furthermore confirms that we have tested various parts of the codebase related to R1, in addition to the fact that we have tried various cases of error handling, and tried different kinds of inputs.

Regarding performance testing, we saw we could use various metrics, such as the response time, or throughput. We also could take as a metric the percentage of requests made that have failed. In our case, we saw that for the lowest arrival rates (1 request/second and 10 requests/second) had extremely optimal response times. However, all tests showed limitations in terms of conflict handling. Given the limited time

and resources, it wasn’t possible to calculate other metrics specified in the planning, such as memory usage, CPU usage or network usage.