Pelle Hald Vedsmand, Lasse Hansen, Nicolai Rosendahl - EK 10.9/2025

Reflection on Coverage and Performance:

Integration- and unit-testing

We believe we reached a good balance of unit and integration testing by having a test-driven approach to development. This was a natural way of doing it, since it would allow us to split development tasks up between us. One developer was focusing on data persistence and retrieval. A natural way of making sure everything worked as intended without having a runnable application, was by running integration- and unit-tests under development. This both ensured the working condition of a part of the application for other developers to integrate with, and ensured a regression test-suite for further development.
Code coverage could then be verified by code coverage reports by Jacoco, and missed branching in the code could be covered by additional unit-tests, until desired code coverage was reached.
As more functionality was implemented, additional higher level testing was added as system tests, where we tested all layers of the system by automated and manual HTTP-requests, and response validation.

The code coverage is an important metric to get an idea if our code has been tested thoroughly enough. However it does not tell us about the strength of each test being performed, only about which parts of the code have been tested.

Load test:

The load testing is done “unrealistic”, which means that we spawned 100 virtual users, that each send 100 requests as fast as possible. It would make more sense to perform these tests with a gaussian random timer, that random sends requests from each user, to get a more realistic load on the servers.
All of the 10000 requests were successful, with all of them giving the correct response.
The median response time was 22 ms, which is very respectable, but the average (including the 99th outliers) was 1063 ms, which is not great.
We think this is due to some users in the back of the queue having very long response times, which shows the 99th percentile being extremely slow.
The max response time was 69000 ms.

The load testing was done to look at the response time and concurrency. The results from the load tests can give us an idea of how many users are able to use the system at a given time, and how the amount of requests affect the overall performance of the application.
With these results we have an idea of how many expected users the system would be able to handle, but this can depend on the computer system running the application.
We think that Javalin has problems with this many threads, but it could also be due to Java, which uses virtual threads, which dynamically are allocated, but can be limited by the processing power of the system running the application. It would therefore be a better test environment, to run this application on a production-like test environment, to really get an idea of the amount of concurrent requests it could be able to handle.

Another thing we should think about is the reduced processing power of both the Javalin server, but also Jmeter, by containerizing each with its own set of cpu and ram on the same computer.