PG3402

Microservices

21.10.2023

Individuell mappeeksamen

Høyskolen Kristiania

Høst 2023

Individual reflection

Throughout the semester we have learned the theory of Microservices' design philosophy. During this semester, we learned the pros and cons of this pattern and the necessary technologies for making a functional application where Java is at the core.

Much of the time spent learning has been dedicated to integrating all the needed infrastructure for the services made in the end. Throughout this period, active development of my bank application was never a priority, but having a good enough grasp of the surrounding technology was vital for me.

After all the base servers I like to call “infrastructure” were done, Consul, RabbitMQ, Zipkin, PostgreSQL, Gateway, and Central Logging were implemented and working with two demo services before development of the bank could initiate. The reasoning behind finishing the infrastructure is quite simple. Like an actual building, you need a solid foundation before building. With all the infrastructure done, I felt confident that the development of the Bank app would have a high velocity.

By starting with my main user stories of making an account at the bank, there was much theory in play that I knew about but had yet to do much in practice. That made the general development speed slow, but learning and searching for best practices for how a server should handle RESTful API requests in practice in Java was a considerable learning curve.

Most of the time was spent enforcing HTTP standards and error handling. Some time into developing the first feature, testing became a field of interest. As before, a massive endeavor of learning essential tools and skills for testing Controller, Service, Repository, and Server integration tests was understood. This slowed down the development of features but saved me lots of time later when I discovered many potential bugs were fixed while writing tests.

With experience writing tests and a more robust understanding of best practices of RESTful API in Spring boot, the development continued. With a small skillset with frontend and no time to learn advanced JS frameworks like react, I landed on using Server-Side Rendering (SSR) frontend. Using the following technologies: HTMX, Thymeleaf, and Spring Boot. This allowed me to write basic HTML and CSS pages without learning the JS framework normally used. This gave me a significant advantage because the frontend server works as a security layer for validating incoming data.

Since everything from the frontend goes by the dedicated frontend server, I had robust validation tools integrated into Spring Boot. I had more decisive control over the communication between the front and backend.

Not everything has been going as intended and was left with what I would call “artifacts” of this learning process. While learning the frontend and best practices for the backend, I

started doing things one way and slowly adjusted how things were implemented. Some things got updated along the way, but sadly, not everything. As this is an exam with an extremely strict deadline, I have been limited in how much refactoring I can squeeze in before the deadline. I wish many frontend and backend things could be refactored for better readability and following best practices. Lastly, with the deadline, features were sped through to ensure we reached the MVP of a bank application. This led to some technical debt.

The stack used was equal to the one in the course, and the only deviation would have been the unique Server-Side Rendering frontend solution. Most details will be written in README.md inside the project, but most communications were done using regular HTTP messaging. At the same time, one place where asynchronous messaging was used was between the frontend and the transfer service. The reasoning is that a transaction would take some time to validate. Inside the transfer server, sleep mechanics are implemented to "simulate" the validation taking time. The frontend solves this by fetching information about the account values occasionally to give the user updated information.

When it comes to the Arbeidskrav design, I followed it quite closely. Spending time writing tests as much as possible and learning best practices gave me more knowledge as a developer than ruining the whole project from the beginning by crunching out features. That is why I managed to do only some things stated in the Arbeidskrav. README.md which includes the details of what was implemented.