Contributions Group 11

Pascal

- Worked on implementing the stock app (initial version)
- Set up the mongodb database for the order service (later converted to a sharded version)
- Set up helm values for mongodb, rabbitmq and postgresql in k8s (discarded)
- Spent a whole weekend debugging k8s scripts to get a minimum viable product with mongodb shards from Leon and Casper
- Helped with testing new implementations
- Created the k8s scripts for the postgres stock database
- Tried to add replicas to the services, but failed :(

Gustav

- spent most on his time on a now scrapped version of the stock app.
- implemented some sagas
- reviewed some pr's

Florian

- has worked with Gustav on the now scrapped version of the stock app
- implemented some sagas

Leon

- Implemented the initial mongoDB version of the payment service
- and later on the sharded version of the order service.
- Spent a significant amount of time on converting our working docker version into all the yamls required for the k8s version.
- Created the bash scripts that automatically execute the init scripts for the MongoDB configuration.
- Created the bash script that deploys the entire stack and executes the scripts mentioned before.

Casper

- Create initial Redis implementation of payment service
- Implemented the initial PostgreSQL version for the stock service
- Sharded the MongoDB database for the payment service with docker
- Implement idempotency keys

Issues that arose

The now scrapped version of the stock app had a consumer implemented to consume messages in batch, that the services put on rabbitmq, and then it returned answers to the stock app using rpc queues. It used a redis cluster that ended up being very inconsistent

because the lock manager library didn't work, so that was scrapped for a stock app using postgres. For the stock it was important to have transactions when it came to subtracting the stock, and that wasn't possible with the other approach. The RabbitMQ also does not deliver in order, but simply using "best effort", and that made the consistency tests fail a lot. The easier, better and safer implementation was doing postgres calls from the stock api, and using rabbitmq to add stock from failed checkouts.

Design Choices

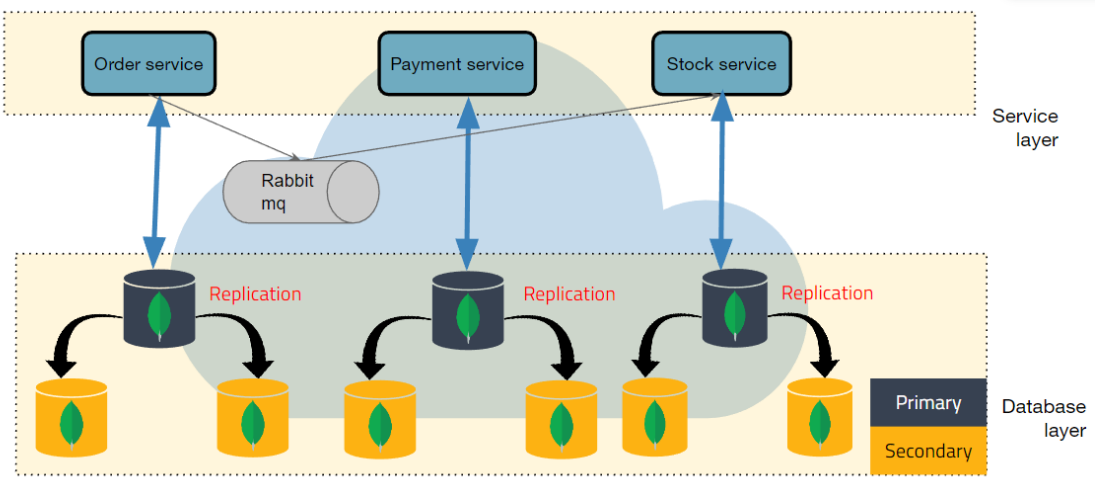
MongoDB sharding for payment and order service: We used mongoDB for the payment and order service since we made an assumption that these services (in real life) do not often have consistency issues. MongoDB is a lot faster than standard SQL databases such as PostgreSQL.

Additionally, we made the assumption that for the order service and payment service, it is unlikely that requests target the same document, but that the load of the documents is spread evenly. By (range) partitioning our database in three shards on order_id and user_id respectively we aimed to distribute the load over the different services evenly. By adding replica's into the mix, we increased our fault tolerance, such that the application is still consistent whenever pods shut down.

As an additional benefit, MongoDB offers easier storing and converting of complex objects. This was useful for the order service as it made it possible to store the items with their price in a dictionary format, which could be altered with native MongoDB queries.

PostgreSQL database for stock service: Since certain items of the stock database will likely be accessed a lot, we decided to use a database that is ACID compatible. We would have liked to also shard this database but there was no time left, so we lose a bit of scalability for some consistency. However it is very much possible to shard the database with kubernetes.

RabbitMQ for sagas:
We have chosen to implement sagas with RabbitMQ, as it allows for a communication channel between the different services, with a deliver once guarantee. With this option we removed most of the state out of our service layer, making it more effective in scaling horizontally. Additionally in case a service had to communicate to another service that



something has to be rolled back, it would only have to put a message on the correct channel and can then continue with the appropriate business logic.

We hereby declare on our honour that we have worked hard, struggled much and screamed at kubernetes many times.

Amen