Distributed Systems Assignment 1

Lars De Leeuw s0205693

April 2024

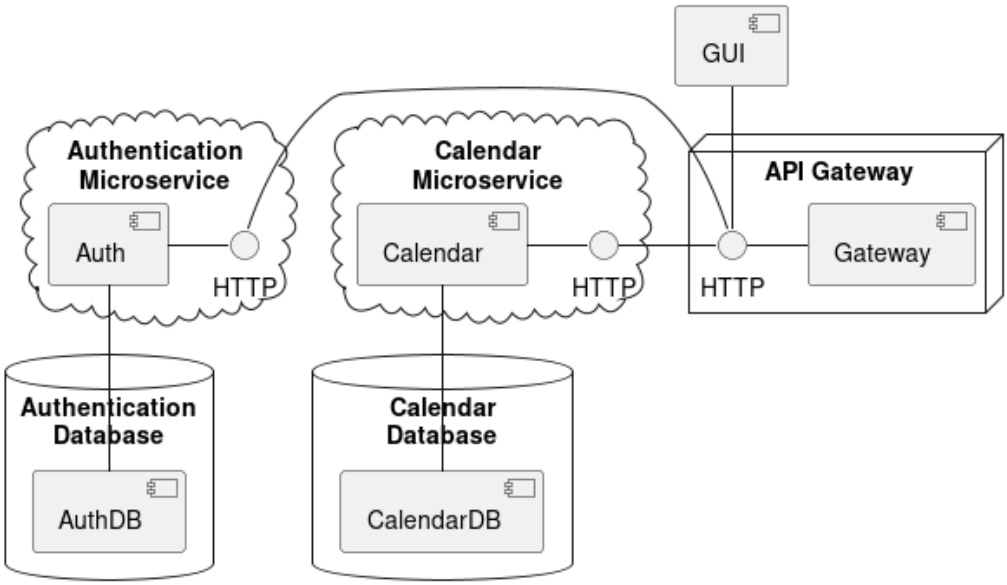

Figure 1: Architecture overview

1 Microservice decomposition

1.1 API Gateway

For my architecture I implemented an API Gateway service. This service orchestrates all communication with the other micro-services and allows the GUI-service to make all its API calls to a singular service. I did this to introduce decoupling between all the services. For example with the API Gateway in place and assuming we are running multiple authentication services it becomes easier to implement a form of load-balancing without the GUI-service being aware of which exact micro-service was used to fulfill it's request. All communication occurs via the HTTP protocol.

Introducing the API Gateway does introduce a single point of failure since if the gateway goes down the entire system becomes unusable. But I believe it is worth it because of the scalability and failure resistance it add when we would start using multiple Calendar and Authentication micro-services.

For the sake of simplicity I made as little changes to the predefined API in the GUI-service as possible. The available endpoints of the API Gateway are:

- **POST /login:**
  Provide the functionality for logging into an existing account.
- **POST /register:**
  Provide the functionality for registering new accounts.
- **POST /authenticate:**
  Provide the ability for authenticating a user. For example this gets utilized in the Calender Micro-service for authenticating a users credentials before providing it's functionality. Wanted some form of authentication because I don't want to create data in the CalendarDB for users that don't exist.

  The way I implemented it currently is by cleverly reusing the /api/v1/login endpoint of the Authentication MS.
- **POST /event:**
- **GET /calendar:**
- **POST /calendar:**
- **GET /share:**

- **POST /share:**
- **GET //event/¡eventid¿:**
- **GET /invites:**
- **POST /invites:**

1.2 Authentication

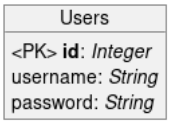

Figure 2: AuthDB Design

The Authentication Micro-service handles the functionality related to authentication. Registering accounts, logging in and even authentication form a nice bubble of functionality. By separating this in its own micro-service we allow the system to scale to accommodate for more users. For example if we need to be able to handle more authentication related requests we could spin up another Authentication MS which connects to the same user database (assuming the load balancing also gets added in the API Gateway). This ability to scale up the amount of Authentication Micro-services also increase the fault-tolerance of the system, if one service goes down we can just redirect the traffic to the ones which are still up. Currently however in my submission I only work with 1 Authentication Microservice, so if this service would fail all functionality related to authentication would no longer work and some of the functionality of the Calender Micro-service.

The endpoints (which only get called by the API Gateway) are:

- **POST /api/v1/login:**
  Implements the actual functionality for logging in and checking if the username exists and password is correct.
- **POST /api/v1/register:**
  Implements the actual functionality for registering a new account, does not allow account creation if username is taken.

1.3 Calendar

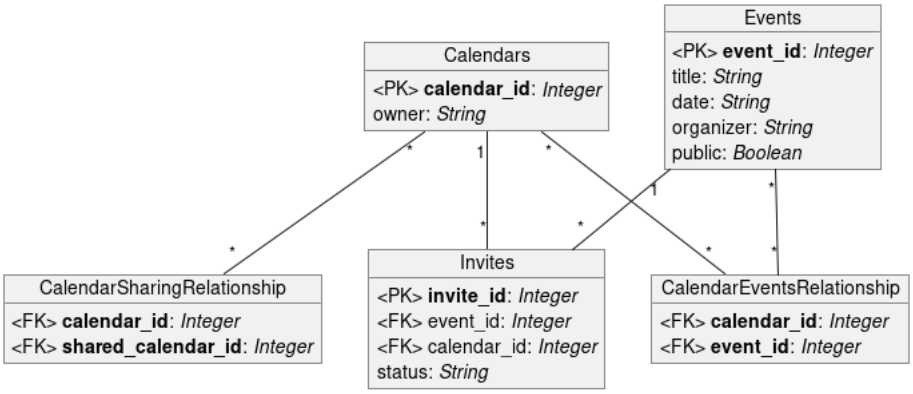

Figure 3: CalendarDB Design

The Calendar Micro-service handles functionality related to calendars, events and invites. The same reasoning behind grouping these features as with the Authentication Micro-service.

The endpoints (which only get called by the API Gateway) are:

- **POST /api/v1/event:**
- **GET /api/v1/calendar:**
- **GET /api/v1/share:**
- **POST /api/v1/share:**
- **GET /api/v1/event/¡eventid¿:**
- **GET /api/v1/invites:**
- **POST /api/v1/invites:**

2 Project Implementation

This project was my first experience experience fully setting up and orchestrating multiple containers. One of the things I really focused on was creating a good structure and set-up related to the docker files. I also used this project as an opportunity to learn GO, which is a language that shines in development of micro-services.

I work with environment files for nearly every service and secrets for sensitive data such as the database password files. I realize this structure and config setup is overkill for this project but I learned the importance of a good structure, in the course "Bachelor Eindwerk" so I wanted to invest the time to learn some good practices.

Another form of overkill in my project structure is the use of versioning in my GO packages. The reason I did this is because again this type of structure is well worth it in a real life setting. So I spent a fair chunk of time learning about the GO best practices for project and package structuring and am happy with the result for my first experience with GO.

3 Hours Spent

All things combined including writing this report I spent around 36 hours on this project. Maybe more with all the time spent learning new technologies.