Micro services Distributed Systems Assignment 2

Jason Liu
Science
UAntwerpen
Middelheim, Antwerp
Jason.Liu@student.uantwerpen.be

***Abstract*— Different services that interact with each other, but are not reliant on each other.**

***Index terms*—Documentation, Report**

I. Introduction

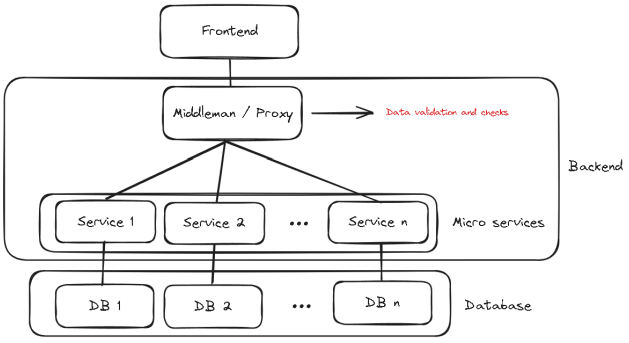
In this report we talk about why and how we chose to implement the different services used in this project.

The project itself is about a calendar application where users can interact with each other by creating events and inviting registered users. Not only that, users can create public events, where others can participate with no regards to invitation.

II. Architecture

Our architecture is based on multiple layers to fully seperate the coupling between every service and frontend.

Our achitecture has the standard three layers (frontend, services (backend) and database). We add an additional layer called middleman that acts as our proxy.



A. Reasoning for middleman

Backend services should be interpretted as services not implemented by us. So, services don't know the existence of each other and only can know at the point of connection.

This could be done fully in the frontend, but this makes the frontend reliant on the services and checks need to be done in the frontend. To avoid the reliability in frontend, we pass the given data to a middleman and validate the data in the middleman.

While it is true that the services and frontend are now reliant to the middleman, this middleman can be fully implemented by us, not knowing who implemented the services nor the frontend. As long as the service makes use of the same output/input as the middleman.

One could make a middleman for each service to split-up the reliability to multiple services, but because of the lack of time and it is out of scope for this project, we only make use of one.

III. Middleman or Proxy

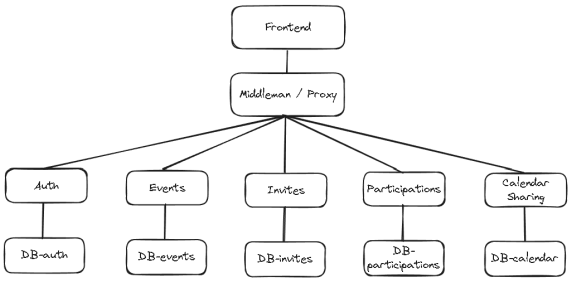
A proxy or middleman, is an application that acts as an intermediary between a client requesting a resource and the server providing that resource. It improves privacy, security, and performance in the process. [1]

In our context we add extra checks by accessing other services from this middleman/proxy to ensure validity of data.

IV. Micro services

The micro services are fully independent. They act as a service on their own without knowing the existence of the other services.

In our context we chose five different services, each with their own reasoning.



A. Authentication / Users

First we have the authentication service that also maintains our user service.

This service provides the following functionality:

1. User authentication. This provides user the options to login and register in the system.
2. User acount. We store the user credentials in teh database for users, to remember they have an account in our system.

This service stores the basic user information, needed for the application. Other services are not directly reliant on this service, but makes use of the user id to identify whom the data belongs to.

1) *Why do you group those features together in a micro service?:* These features are grouped because they have similar connection based on what it is needed for the user database.

Main reason why I grouped these features is that the authentication and user database are reliant on each other. This lessens the unneeded travel cross-network.

2) *What are the consequences if a service fails?:* If this service fails, the frontend should still be working and nothing regarding frontend would change. The main problem that would occur, is that no functionality would work in the frontend, because there is no possible way to bypass login page.

In case the login can be bypassed some how, users will still not be able to do anything because the other services makes use of an user to be able to do anything. An user for instance can't participate to an event if the user might not exist.

3) *Why does your approach scale well for large user bases?:* This approach scales well for large userbases because the user database can be fully seperated from the other services. This means that the user database can be expanded with no cause of trouble for other services.

Scaling can also be done regarding performance, by adding multiple instances of this service. We can seperatly scale this service with no regards to other services to improve user performance.

4) *Improvements?:* I do want to add that these two services can also be seperated to be fully decoupled. This adds the possibility to have different authentication services without relying on the service that maintains the user database. This makes the application even more scalable, but adds more maintainability regarding the middleman between those services.

I didn't do this because authentication was out of scope for this project, but just mentioning that it could have been done, in case it was needed.

B. Calendar sharing

This service provides the option to share calendars cross users.

This service stores the user id from the owner and the id of the user whom the owner shares with.

1) *Why do you group those features together in a micro service?:* I only have one feature in this service. I chose to make this a seperate service because other services are not directly reliant on this service.

2) *What are the consequences if a service fails?:* The biggest concern is that we would not be able to watch other people their shared calendars anymore. Not only that, we won't be able to share our own calendar anymore.

3) *Why does your approach scale well for large user bases?:* In case user amount scales, the anount of users one user can share with also scales. It is important that we can scale this service seperatly and easiliy without affecting the other services.

This approach makes the service fully decoupled from the other services. In other words, we can create multiple instances of this service without affecting the others.

C. Invitations

This services maintains all invitations and their current status. The user can check if the invitation they got, is still pending or not. When answered this inviation will not be vissible anymore for the user.

This service makes use of the user id and stores the status of the invitation. Not only that, it makes use of the event id to identify which event it belongs to and the inviter id indicating by whom the user got invited.

1) *Why do you group those features together in a micro service?:* These features are grouped in one single service because again they are not needed for other services.

Other service might need information from this service, but their is no direct correlation. This means it can be fully independent with no regards to other services.

The grouping of this service is based on that all data that the features need, are based on the same database they need to access.

2) *What are the consequences if a service fails?:* If this service fails, the invitations would be not working anymore in the frontend. This causes the users to not be able to respond/ participate to any private events.

Not only that, users will not be able to invite other users anymore.

3) *Why does your approach scale well for large user bases?:* Same reasoning as for calendar sharing, when users scale, the amount of invitations a user can make also scales.

D. Participations

This service maintains all participations regarding an event. This makes it possible for users to participate to an event and users can check who participates to an event.

This service needs the user id of the participator, the event it will participate or not, and the status if it will participate.

1) *Why do you group those features together in a micro service?:* All features are depended on one singular item of the

database. Same reasoning as other services, this service is not directly reliant on other services or vice versa.

2) *What are the consequences if a service fails?:* Users will not be able to participate anymore to events. While status can change for an event, the status of participation will not be tracked if the service fails. This is because they are not reliant on each other.

3) *Why does your approach scale well for large user bases?:* Because the amount of invites can scale, the same amount of scaling can happen when all users will accept their invites. Not only that, the user can participate public events, so if we count that also in, this could scale double the amount.

So, our implementation making partipation fully seperate, makes it possible that we can scale our service horizontally without affecting the other services. In case the user make many request to participate multiple events.

E. Events

This service keeps track of all events that are made by users.

These event store multiple information, they have a title, a description, a boolean indicating if the event is public or not and a date for when the event happens.

Other services rely indirectly on this service, making use of the event id.

1) *Why do you group those features together in a micro service?:* This service got grouped with these features, because of the same reasoning I gave for the other services. It makes of a singular database, and other services are not directly reliant on this service.

2) *What are the consequences if a service fails?:* In case the service fails, the option to make events will not work. This cascades to users not being able to make invite, because there is no event to make invites for. Same for participation.

3) *Why does your approach scale well for large user bases?:* Same as the reasoning I gave for calendar sharing, the amount of events one user can make can be infinite.

Especially because the events will not be deleted when expiration date is hit, the data will be kept and will scale really fast. Our implementation makes it so that events can be scaled seperatly in case more users make events.

F. Seperation of invitation and participation

There is one correlation that are closely related to each other, and that is invitation and participation. Why did we not make it one? As mentioned users can also participate to public events (supported in our implentation), so it does not need an invitation.

Because of this we need two seperate services, to maintian both invitation and participations.

V. Databases

For every service their is a seperate database. We did this, because if we only had one database, all services will fail in case the database fails.

VI. API

The api can be accessed by going to the service host, on port 8000 and route '/docs'

API docs url:

1. 'http:\backend-auth:8000/docs'
2. 'http:\backend-events:8000/docs'
3. 'http:\backend-participations:8000/docs'
4. 'http:\backend-invitations:8000/docs'
5. 'http:\backend-calendar:8000/docs'

Every endpoint is documented when reaching this url.

VII. Bugs

Some bugs I noticed during the make of the project, is that after participating an event from the invitation list, you have to manually reload the page to so any effect.

VIII. Technology

I made use of FastApi to build the services. Flask/Jinja for the frontend. Python as the language.

References

[1] Wikipedia, "Proxy server." [Online]. Available: https://en.wikipedia.org/wiki/Proxy_server