Distributed Systems Assignment 2: Microservices

Orfeo Terkuçi
Faculty of Science
University of Antwerp
Antwerp, Belgium
Orfeo.Terkuci@student.uantwerpen.be

***Abstract*— Explanation of the architectural design and microservice decomposition, alongside documentation about the API endpoints for each microservice.**

***Index terms*—Assignment, Report, Microservices**

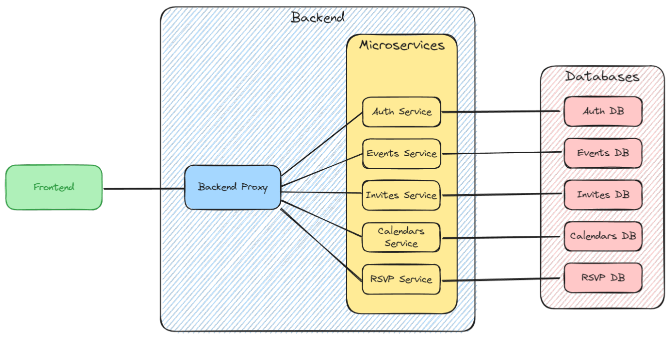
I. Introduction

In this report I will be going over my architecture and design choices. I will explain my microservice decomposition, as well as go over each microservice separately, explaining what features each microservice implements, what kind of data it stores, and what connections it has to other microservices. I will also go over the documentation of the API endpoints of each service.

II. Architecture

I decided to go for a standard three-layer architecture:

1. Front-end
2. Back-end
3. Databases



The frontend is the access point for the user. The backend is where all the requests sent by the frontend are processed. Each microservice has its own separate API in the backend. There is also a proxy [1] API for the backend. This proxy receives all the requests from the frontend, does any necessary checks (such as: does this user exist? Are the parameters correct?) and delegates them to each separate microservice. The last layer is the databases layer. This layer is only accessible by the backend, and in the backend only by the corresponding microservice (i.e. only the events microservice has access to the events database).

A. *Why use a proxy?*

The reasoning behind the choice for a proxy is to reduce the coupling of the microservices with each other and have them behave more like external independent entities. Instead of doing checks that rely on the availability of other services in the microservice itself, we do this check on the proxy, together with any input sanitation. This also reduces the amount of unnecessary calls to a microservice in the event of a faulty input.

III. Frontend

The frontend is implemented in Flask using Jinja templates. This was provided to us.

A new GUI element for indicating participation to public events was added to the implementation. When you go to see the details of a public event in the `/event/{eventID}` route, if the event is public you will be able to say that you will "Participate", "Maybe participate", and if you change your mind later you can also rescind your participation.

IV. Databases

Each service has its own PostgreSQL database. Having only one database would result in a bottleneck of communication and a Single Point of Failure for all the microservices (if the database is down, ALL the services fail). In this implementation, if one of the databases is down, for the most part only the microservice that is dependent on it loses functionality. Schemas for all the resources:

- Auth:
  - Users table `(id, username, password)`
- Events:
  - Events table `(id, title, description, date, organizer, isPublic)`
- Invites:
  - Invites table `(eventID, username, Status)`

    - Status can be either `"YES", "MAYBE", "NO" or "PENDING"`
- Calendar sharing
    - Shared Calendars table `(sharingUser, receivingUser)`
- RSVP [2]
    - RSVP responses table `(eventID, username, Status)`
    - Status can be either `"YES", "MAYBE" or "NO"`

V. Microservices

I decided on 5 microservices for the implementation of all the features:

1. Authentication Service
2. Events Service
3. Invites Service
4. Calendar Sharing Service
5. RSVP (Public event responses) Service

A. *Authentication Service*

This service handles the registration of new users as well as the login of existing users into the system.

There is also a `/users` endpoint for development purposes, so that you can easily check all the users in the backend (for security purposes only the usernames and user-IDs are returned).

1) *Why group these features together?:*

It made sense to group all the user functionality together. An Auth Database was created to contain all the user information (id, username, password). The Authentication (login and register) functionality makes use of only this database, so it was put in a microservice that has a direct connection with this database. Every other microservice that needs user information has to make a request to this microservice.

2) *What are the consequences if the service fails?:*

Login and Register functionality is unavailable. Every other service that needs user information to process certain requests (i.e. to create an event, we need to know if the organizer exists) is also hindered to some extent. The backend proxy receives an exception, catches it, thus recognizing that the service is down, and returns a 500 response to the frontend. This way, the frontend remains functional.

3) *Why does this approach scale for large user bases?:* The Authentication Database, containing all the user information is separate from all other microservices. Failure of other services does not affect the authentication service. It can also be scaled horizontally by adding more instances of the service.

B. *Events Service*

This service handles the CRUD [3] operations for events. Each event has an automatically assigned ID, a title, description, date, organizer and is either public or private.

The organizer element of the event is the username of a user in the Users database. The check for the existence of this user during the creation or modification of an event is done by the backend proxy.

1) *Why group these features together?:* An Event Database was created to store all the events. The Events Service makes use of this database and is (mostly) independent of other services. It only has a dependency on the Authentication Service for the creation and modification of events (it does a check on whether the organizer exists). The rest of the functionality is independent.

2) *What are the consequences if the service fails?:* Functionality regarding events is unavailable. This includes the Creation, Fetching, Updating and Deletion of events. Just like in the case of other services failure, the backend proxy receives an exception, catches it, thus recognizing that the service is down and returns a 500 response to the frontend. This way, the frontend remains functional.

3) *Why does this approach scale for large user bases?:* The Events Service has a direct connection to the Events Database. The amount of communication that goes through this service is limited to only relevant requests (i.e. requests that have to do with CRUD operations on the Events resource). This increases performance on large user bases. It can also be scaled horizontally.

C. *Invites Service*

This service handles the CRUD operations for invitations. Each invite contains an Event ID, Username and Status ("YES", "MAYBE", "NO", "PENDING"). Since the username is unique it was chosen as a suitable key.

The username element of the invite is the username of a user in the Users database. The check for the existence of this user during the creation or modification of an invitation is done by the backend proxy.

1) *Why group these features together?:* Invites could have been grouped together with events in the Events service, but that meant that if the events service went down, the invites functionality would be unavailable as well. It also made more sense to consider invites as a separate resource, and thus a separate database and microservice was created. Invites can be for both public or private events, but willing participation to public events was put on a separate microservice. The reasoning behind that will be discussed shortly. When the user responds positively to an invite ("YES" or "MAYBE"), the event will appear on his calendar.

2) *What are the consequences if the service fails?:* Functionality regarding invites is unavailable. This includes the Creation, Fetching, Updating and Deletion of invites. Just like in

the case of other services failure, the backend proxy receives an exception, catches it, thus recognizing that the service is down and returns a 500 response to the frontend. This way, the frontend remains functional.

3) *Why does this approach scale for large user bases?:* Following the same reasoning as for the other services, the Invites service has a direct connection to the Invites Database. The amount of communication that goes through this service is limited to only relevant requests (i.e. requests that have to do with CRUD operations on the Invites resource). This increases performance on large user bases. It can also be scaled horizontally.

D. RSVP (Public event responses) Service

This service handles the CRUD [3] operations for responses to participation to public events. An RSVP contains the same elements as an invite, with the difference of not being able to have a `PENDING` status: Event ID, Username and Status ("YES", "MAYBE", "NO"). This differs from invites, because a user can participate to a public event with or without an invite, although he can also be invited to one.

The username element of the invite is the username of a user in the Users database. The check for the existence of this user during the creation or modification of an RSVP is done by the backend proxy.

1) *Why group these features together?:* Merging this functionality into the Invites service was considered, but it was decided against. The reasoning behind it is that if the invites service goes down, less functionality would be lost . This means that the user would still be able to see and respond to all the PUBLIC events he can participate in in his home page.

2) *What are the consequences if the service fails?:* Functionality regarding participation to public events is unavailable, but the user can still respond to his invites (granted that the invites service has not gone down as well). Just like in the case of other services failure, the backend proxy receives an exception, catches it, thus recognizing that the service is down and returns a 500 response to the frontend. This way, the frontend remains functional.

3) *Why does this approach scale for large user bases?:* Following the same reasoning as for the other services, the RSVP service has a direct connection to the RSVP database. The amount of communication that goes through this service is limited to only relevant requests (i.e. requests that have to do with CRUD operations on the Invites resource). Since this service only handles responses to public events WITHOUT an invitation, it also serves as a kind of load balancing for the invites service, relieving it of an extra burden . This increases performance on large user bases. It can also be scaled horizontally.

E. Calendar Sharing Service

This service handles the calendar sharing functionality: A user can share their calendar with another user. Sharing is not a symmetric operation (e.g.: Even if user A shares their calendar with user B, user A cannot necessarily see user B's calendar). Two-way share is possible (user B just has to share their calendar with user A).

Each calendar share is a tuple `(sharingUser, receivingUser)` containing both users' usernames. A check of whether these users exist is done during the creation of a calendar share. This check is done by the backend proxy.

1) *Why group these features together?:* The calendar sharing feature did not quite fit with the other services. It could have been placed in the Authentication service, but that meant more coupling and it was decided against.

2) *What are the consequences if the service fails?:* Functionality regarding sharing and checking whether a user shares their calendar with another user is unavailable. Since no other service is reliant on the calendar sharing service, no other functionality is lost. The same way with the other services, the backend proxy receives an exception, catches it, thus recognizing that the service is down and returns a 500 response to the frontend. This way, the frontend remains functional.

3) *Why does this approach scale for large user bases?:* Following the same reasoning as for the other services, the Calendar Sharing service has a direct connection to the Calendar Sharing database. The amount of communication that goes through this service is limited to relevant requests. This increases performance on large user bases and can be scaled horizontally.

VI. Backend proxy

As explained previously, the usage of the proxy reduces the amount of calls to the microservices and increases their independence, this to the added cost of an additional bottleneck (if the backend proxy goes down, all the services are unavailable). All the requests from the frontend go through this proxy, the inputs are sanitized and any necessary additional checks are done: Does the user exist?; Does the invite exist?; etc. If one of the checks fail, an appropriate response is returned to the frontend and no unnecessary request is made to the corresponding microservice.

VII. API documentation

The API documentation is available at `http://localhost:8000/api/docs` while running the service.

VIII. Tech stack

I am using Flask and Jinja for the frontend, Fast API as my API framework of choice for each microservice (as well as the backend proxy), and PostgreSQL for the databases.

References

[1] "Proxy pattern," [Online]. Available: https://en.wikipedia.org/wiki/Proxy_pattern

[2] "RSVP," [Online]. Available: https://en.wikipedia.org/wiki/RSVP

[3] "CRUD," [Online]. Available: https://en.wikipedia.org/wiki/Create,_read,_update_and_delete