OurSpace

Projektdokumentation "OurSpace"

Noel Hug, Mischa Strauss, Masha Madani

Context

Objective

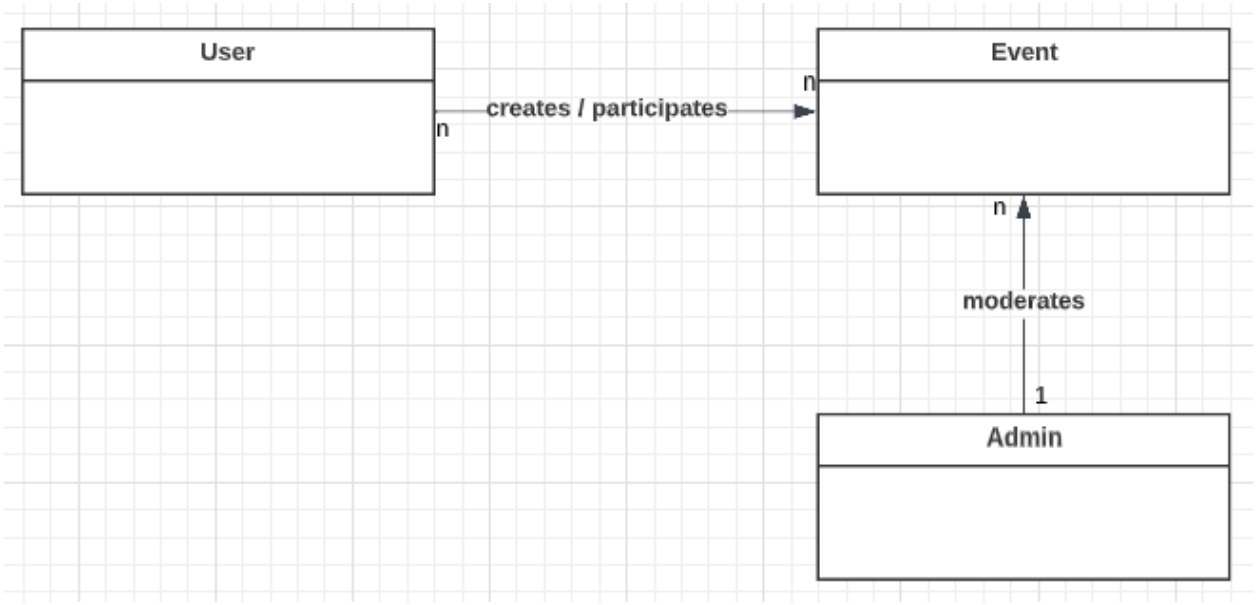
Our goal is to enable users to extensively manage events, including creating, editing, and deleting events, as well as adding event participants. It should also be possible to enroll in other people.

Background

In the production of a social media application, called “OurSpace”, we have been given the specific task to implement event functionalities.
A user can publish an event (for example a dinner party), and anybody interested can sign up.

Admins make sure that everyone is safe on the platform by deleting harmful content and blocking users that often publish such harmful content.

All of these features together result in a domain model like this:



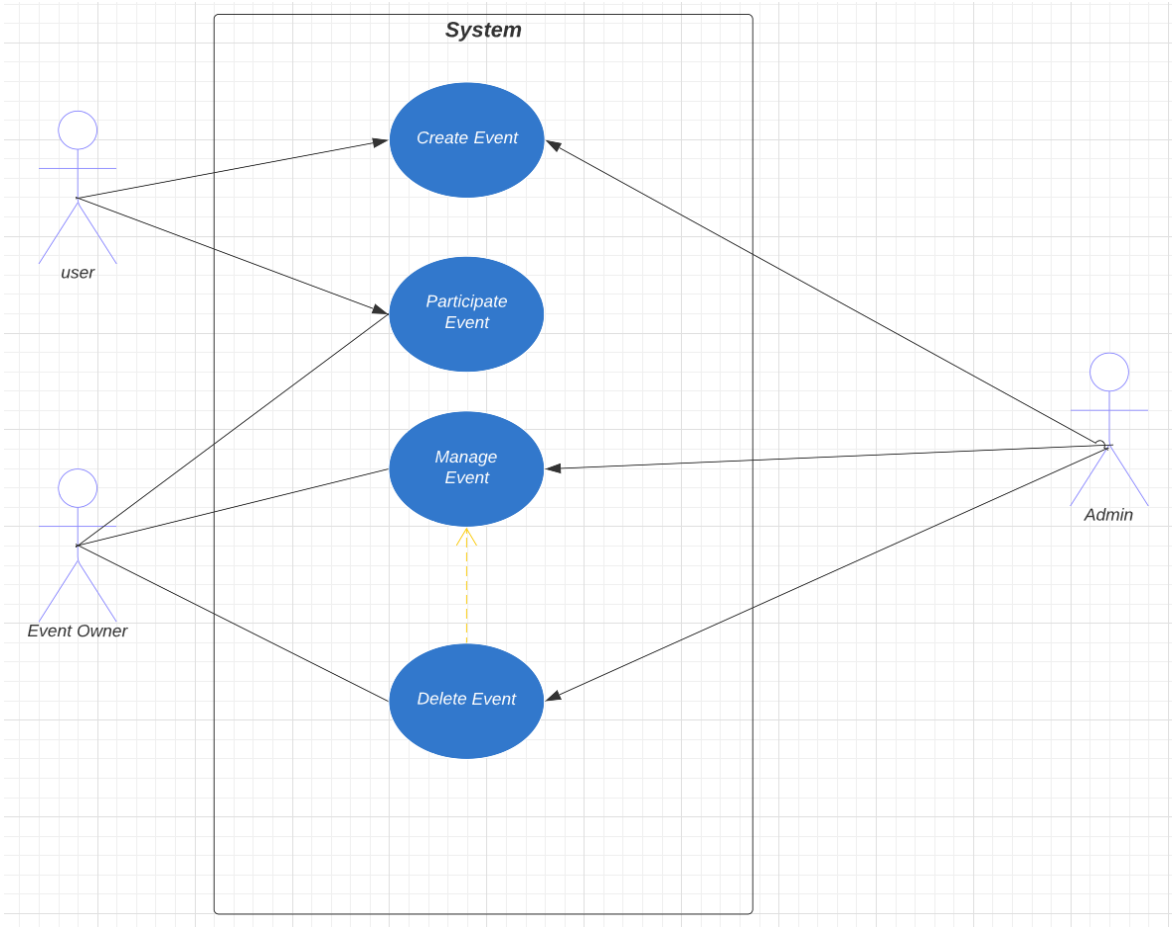
Picture one: Domain Model

Design

Overview

With this project, the focus is on the application functionalities. Quality attributes like performance are not important. So our idea is to keep it simple and minimalistic.
This application is designed as a monolith. There are no requirements regarding scalability and reliability, and for the sake of the speed of development, this single-server solution was chosen.

As our main task is to implement event management functionalities, this use case diagram visualizes hwo to create an event:

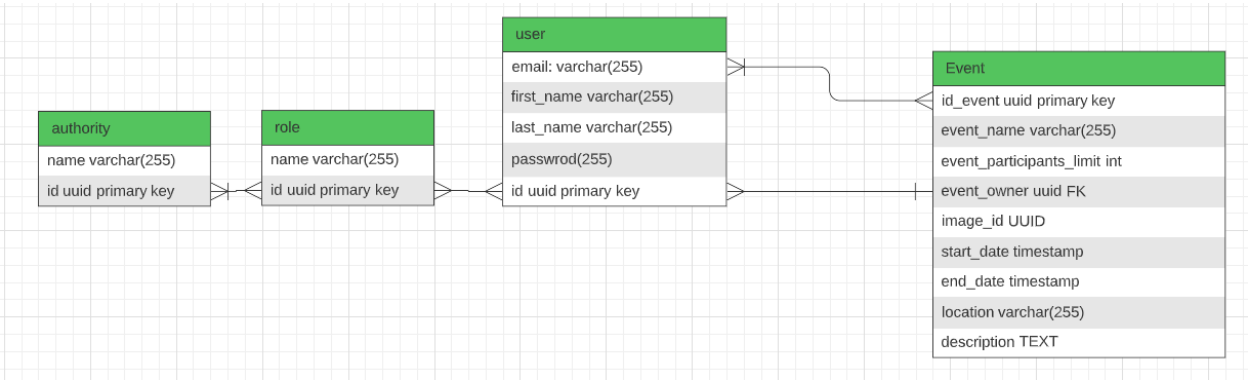


Picture two: Use case diagram

“Create Event” use case definition:

Actor(s)	User	Admin
Description	The User creates an event	The Admin creates an event
Preconditions	- User has to be logged in - Preconditions met - Is not banned	- Admin needs to be logged in as Admin - Same preconditions as normal user
Normal course	1. User logs in 2. User clicks on create event button 3. User fills in form to create event 4. Send event to backend 5. Put data into db	1. Admin logs in 2. Admin clicks on create event button 3. User fills in form to create event 4. Send event to backend 5. Put data into db
Alternative course	If the user doesn’t fill out all the forms, an error should be shown	If the user doesn’t fill out all the forms, an error should be shown

The data gets persisted in a postgresql database. The ERD in the database looks like this:



Picture three: ERD concept (not final)

Another important feature is the ability to enroll in an event (aka. Say that you want to participate). The use-case for that feature looks like this:
“Enroll in event” use case definition:

Actor(s)	User	Admin
Description	The User enrolls into an event	The admin tries to enroll

Preconditions	- User has to be logged in
	- Preconditions met
	  - Event number of participants not exceeded.
	  - Is not banned
		- Admin needs to be logged in as Admin
		- Needs admin rights

Normal course	6. User logs in 7. User clicks on enroll button 8. Gets confirmation screen	6. Admin logs in 7. Admin tries to enroll in an event 8. Can’t enroll
Alternative course	- If the number of participants gets exceeded, an error message should be shown to the user - For the banned part, it should be redirected to the banned error page	- The admin shouldn’t be able to see the UI button, but if the admin still calls the API somehow, a descriptive error message should be shown with steps to fix (clear local storage, …)

Detailed Design

Frontend

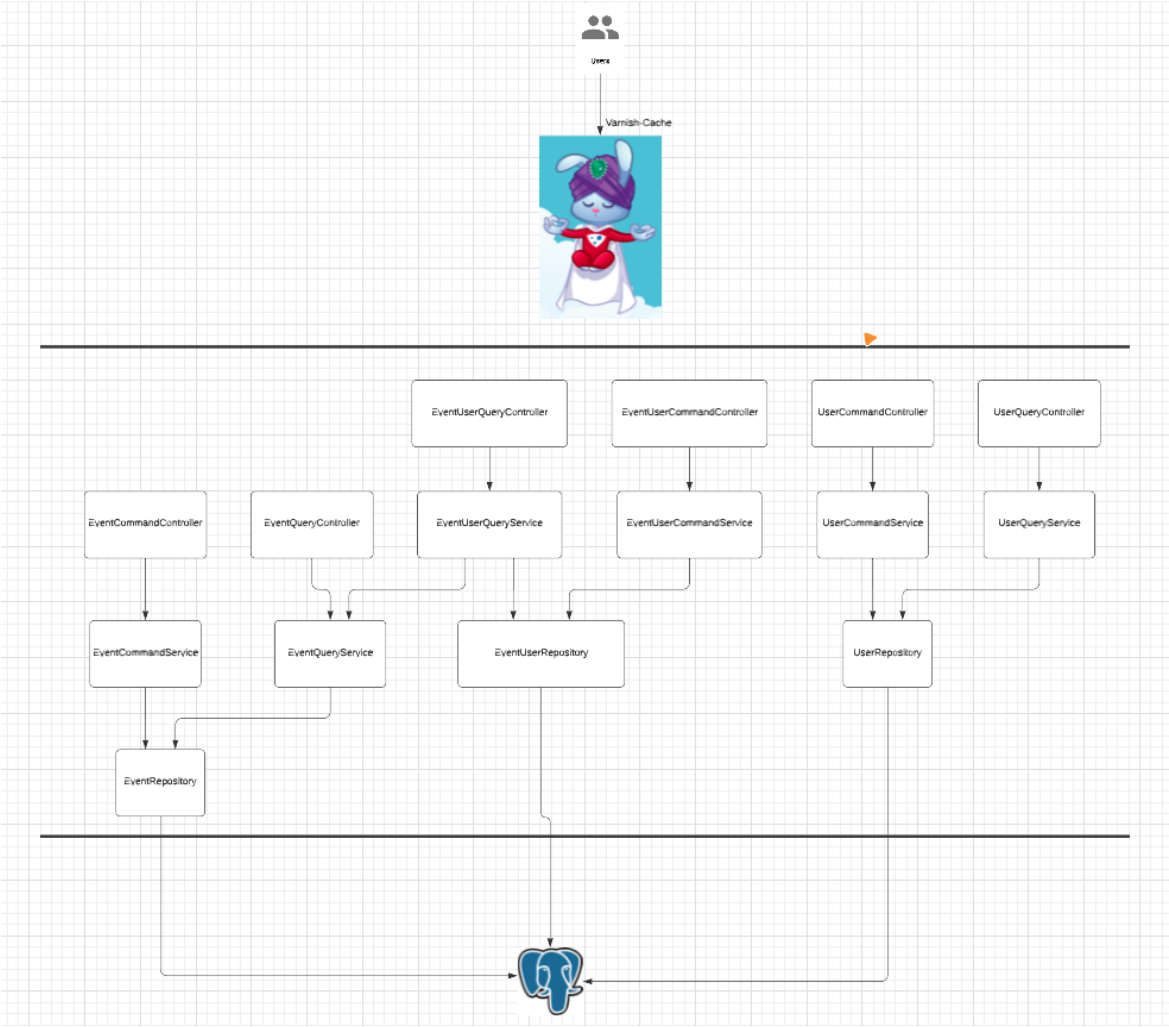
All in all, we have two pages to set up from scratch, whilst the login page and the homepage already exist. The first page is going to include all event management functionalities. Our idea is to make events visible through card elements, which when clicked on, show event information and actions. The second page holds all admin functionalities. Admin functionalities consist of:

- deleting events
- blocking users

The navigation will be placed in an appbar.

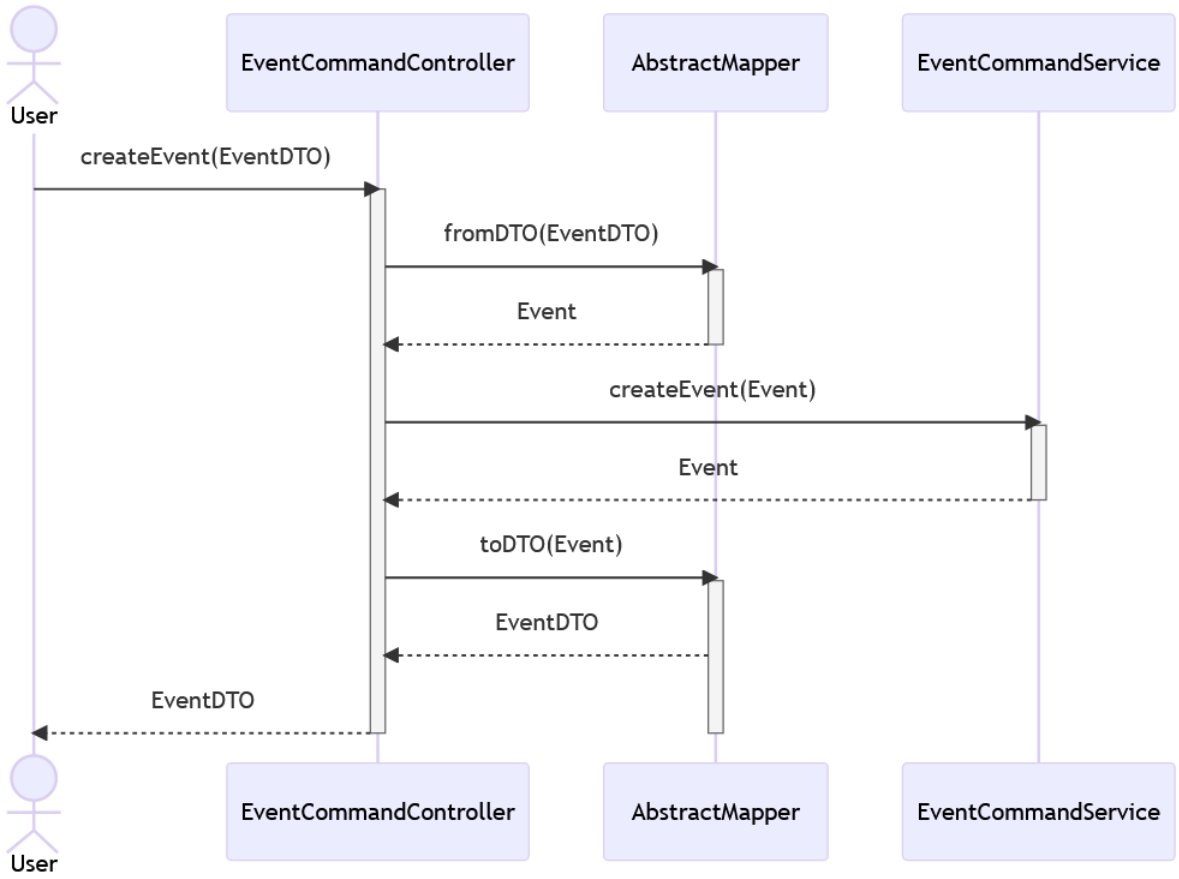
Backend

The backend part of this project would have the following components:



* [original diagram here]

A request on the “Create Event” endpoint / use case described as a sequence diagram:



Picture four: Sequence diagram

Architecture

Frontend

The application architecture is based on the atomic design principles.

Backend

All controllers have been split into two - one for Queries and one for Commands. This allows us to follow the CQRS pattern. This project also follows the layered architecture.
This is reinforced using architectural fitness functions.

Testing strategy

This project gets tested on many different levels:

- Integration tests (postman)
- Unit tests (JUnit)
- E2E testing (cypress)

All of the tests performed will be functionality tests, since the others (load, security, performance) are not a point of concern for this project.

No CI/CD pipeline is set up because it's not a requirement :).

We would test this use case extensively with cypress:

Actor(s)	User	Admin
Description	The User enrolls into an event	The admin tries to enroll

Preconditions	- User has to be logged in
	- Preconditions met
	  - Event number of participants not exceeded.
	  - Is not banned
		- Admin needs to be logged in as Admin
		- Needs admin rights

Normal course	9. User logs in 10. User clicks on enroll button 11. Gets confirmation screen	9. Admin logs in 10. Admin tries to enroll in an event 11. Can't enroll
Alternative course	- If the number of participants gets exceeded, an error message should be shown to the user - For the banned part, it should be redirected to the banned error page	- The admin shouldn't be able to see the UI button, but if the admin still calls the API somehow, a descriptive error message should be shown with steps to fix (clear local storage, …)

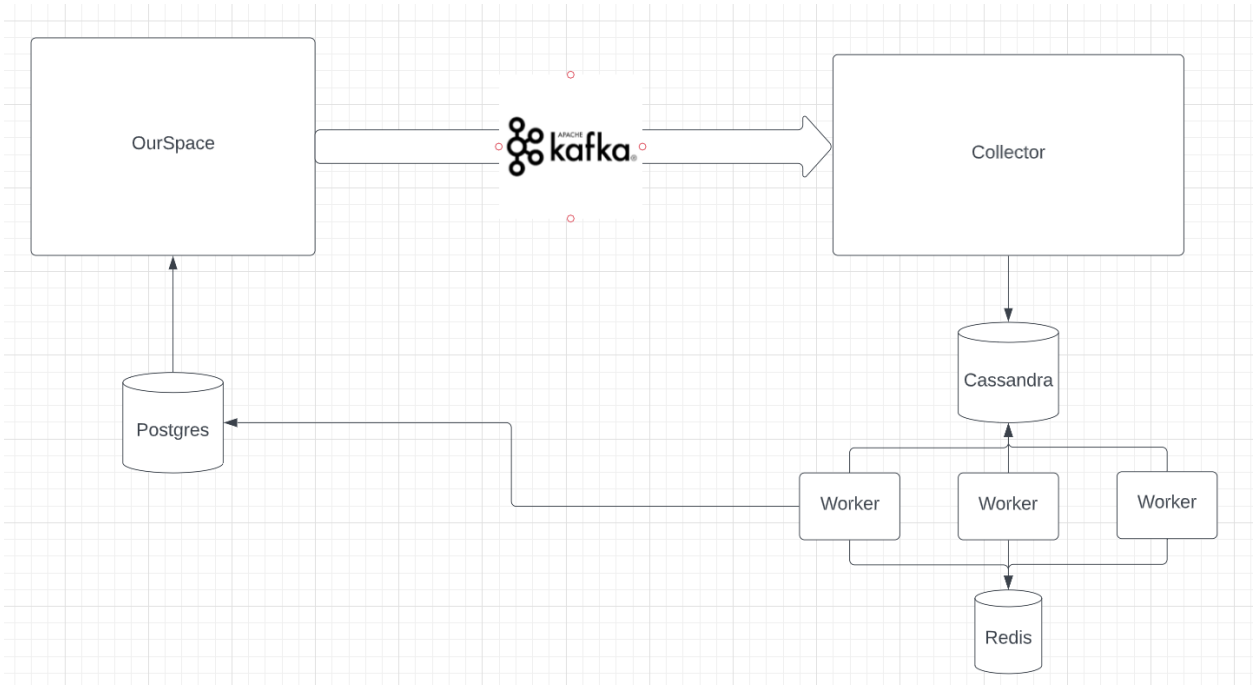
The reason why we chose to test this use case, is that it allows us to test our mechanisms for role management.

Future extensibility

One feature that will be added if time is found is a recommendation system.
It would use the Collaborative Filtering Algorithm.
We would split the system into two main parts, the “lookup” that searches for the kNN gets events from them, and a system that creates a graph for the kNN search.

Graph Creation

From a high level, the graph update system would look like this:



To update the graph we would collect all of the actions taken by the users in a cassandra database.
To this, the OurSpace monolith streams the events to the collector using apache kafka. If the user takes an action in OurSpace like, participating in an event or viewing / liking one, a message gets sent to kafka. This approach was chosen because we can avoid a situation where the original request (participate event, …) takes longer because the downstream Collector is fully occupied.

The next part of the system are the workers. They are responsible for the actual graph calculation.
Each worker is responsible for a range of users. To determine the range, we divide the keyspace (the ID’s of the users) by the number of workers.
Every worker has an ID between 0 and # of workers. This determines which fraction of the keyspace they are responsible for.

If we need a new worker, all the workers need to reevaluate their responsibilities. This update is communicated using service discovery. A tool that supports this is consul.
To remove a worker we do the same step of re-evaluating.
All the workers share a redis database. This is done to inorder to ensure consistency. We don't want multiple workers to evaluate a single user. The worker puts a 'lock' into redis for a user.
If a user is 'locked' no worker can work on it.

The workflow of a single worker would look like this:

1. Get all users in the database which are in the workers field or responsibility
2. Take the one with the oldest message
3. Look if the users is locked (redis)
4. If yes:
    - Take the next oldest
    - Go to step 3
5. Claim user lock in redis
6. Get all the messages from cassandra
7. Calculate LSH
8. Put LSH into postgres
9. Remove lock from redis

Because the lock of a user is global, we don't need to worry about the look of users.

Following this approach means that the services are very small (Collector, Worker). This approach was chosen, because they have different scaling needs. Taking an event and writing it into cassandra is a lot faster than re-calculating a huge amount of data. This means a lot more workers are needed compared to collectors.
Because the workers are mostly stateless, adding / removing workers is very cheap.