WeRec

Ramilya Yusupova (Backend Developer)

Sesegma Tsydypova (Project Manager, Backend Developer)

Sergey Zhurbey (Backend Developer, Tester)

Andrey Kan (Frontend Developer)

Mahmoud Hossameldin (Frontend Developer)

Project description

WeRec is a personal video feed creator.
Use web client to customize video feed, invite parents, friends, or kids to join WeRec-bot in a messenger (such as Telegram, Whatsapp, etc.) and share created feeds with them.
This will allow to control the amount and the quality of content User's beloved people watch or to recommend others User's personal video collection.

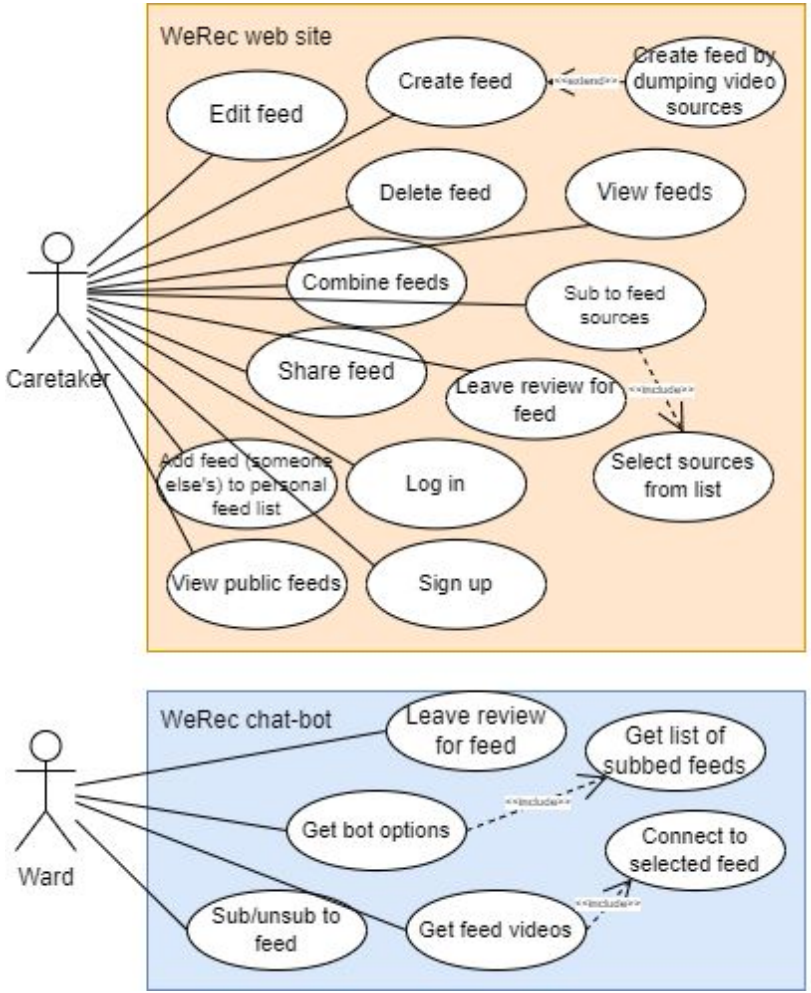
- First–end users can set up a custom video feed by specifying videos source and via web app connect it to a chatbot for selected messenger
- Second-end users then can access the chat bot and consume verified content
- Users can create a personal account to manage their feeds

Team: Ramilya Yusupova, Sesegma Tsydypova, Sergey Zhurbey, Andrey Kan, Mahmoud Hossameldin
Repository: https://github.com/hse-wasd-team/werec
This report:
https://docs.google.com/presentation/d/1q4Uw0llppfLLT7tamvll7eBABjQIQ9iNKEdnLbRpBf4/edit#slide=id.g1bc71fecfc4_0_9

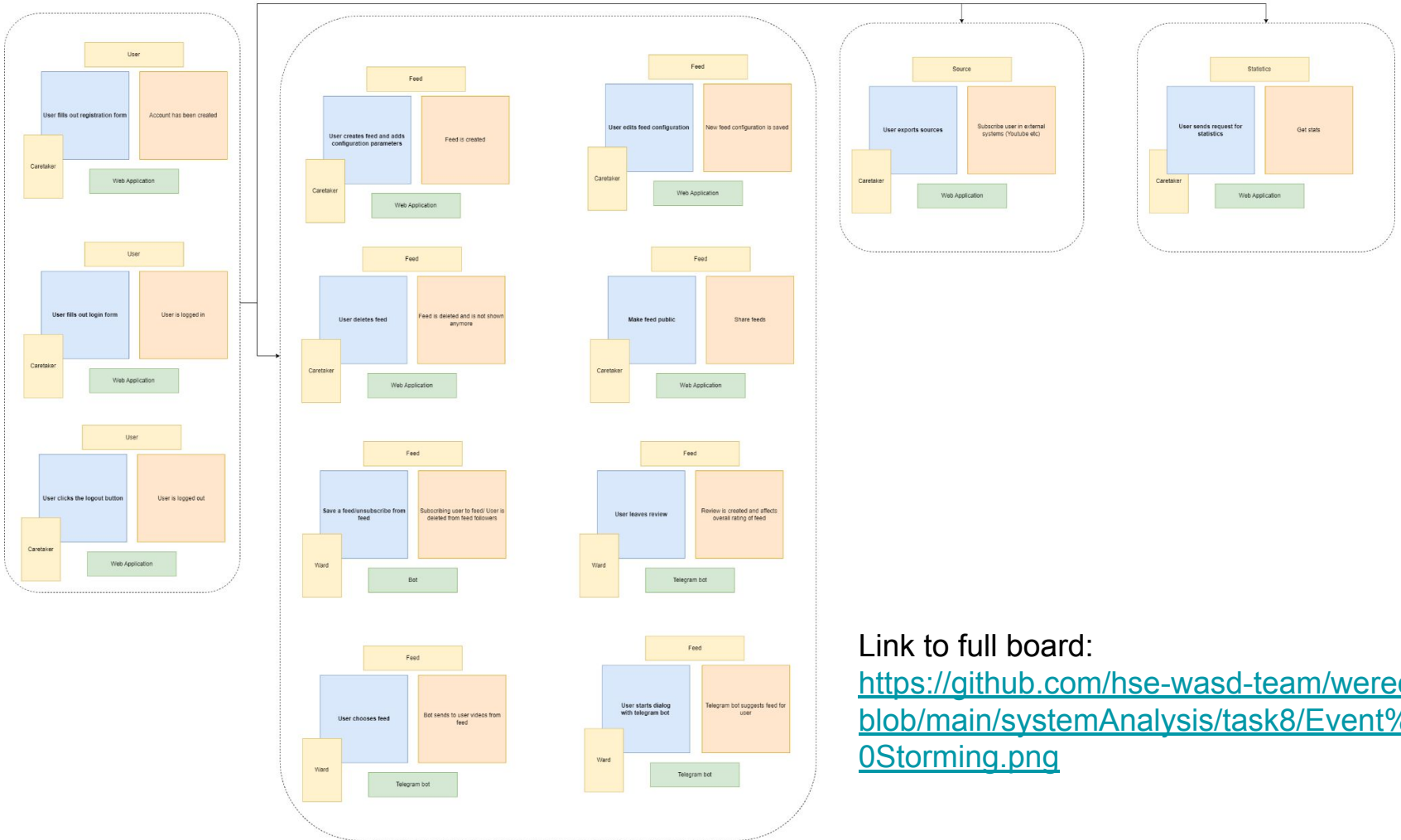
Use case Diagram

Detailed textual description of use cases:
https://github.com/hse-wasd-team/werec/wiki/Use-Cases-description

Image:
https://github.com/hse-wasd-team/werec/blob/main/systemAnalysis/task5/useCaseDiagram.png



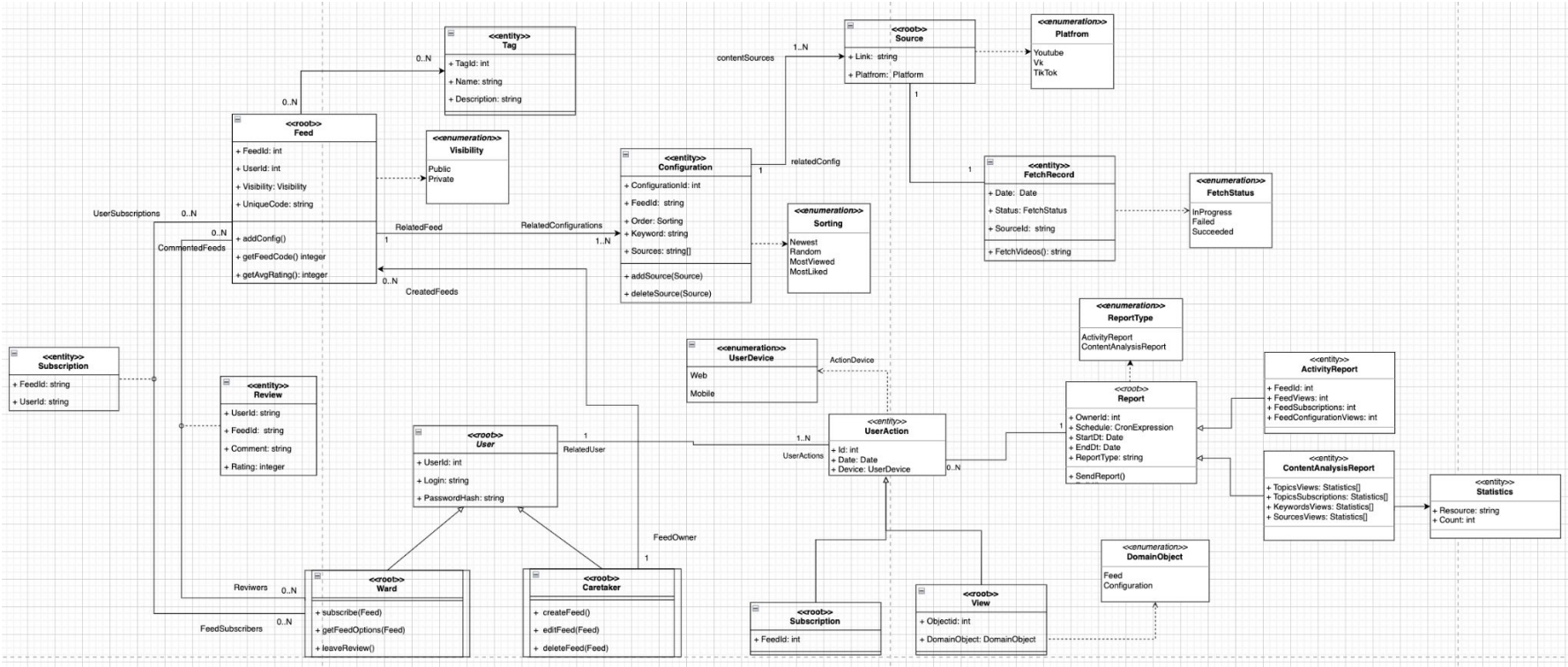
Event storming



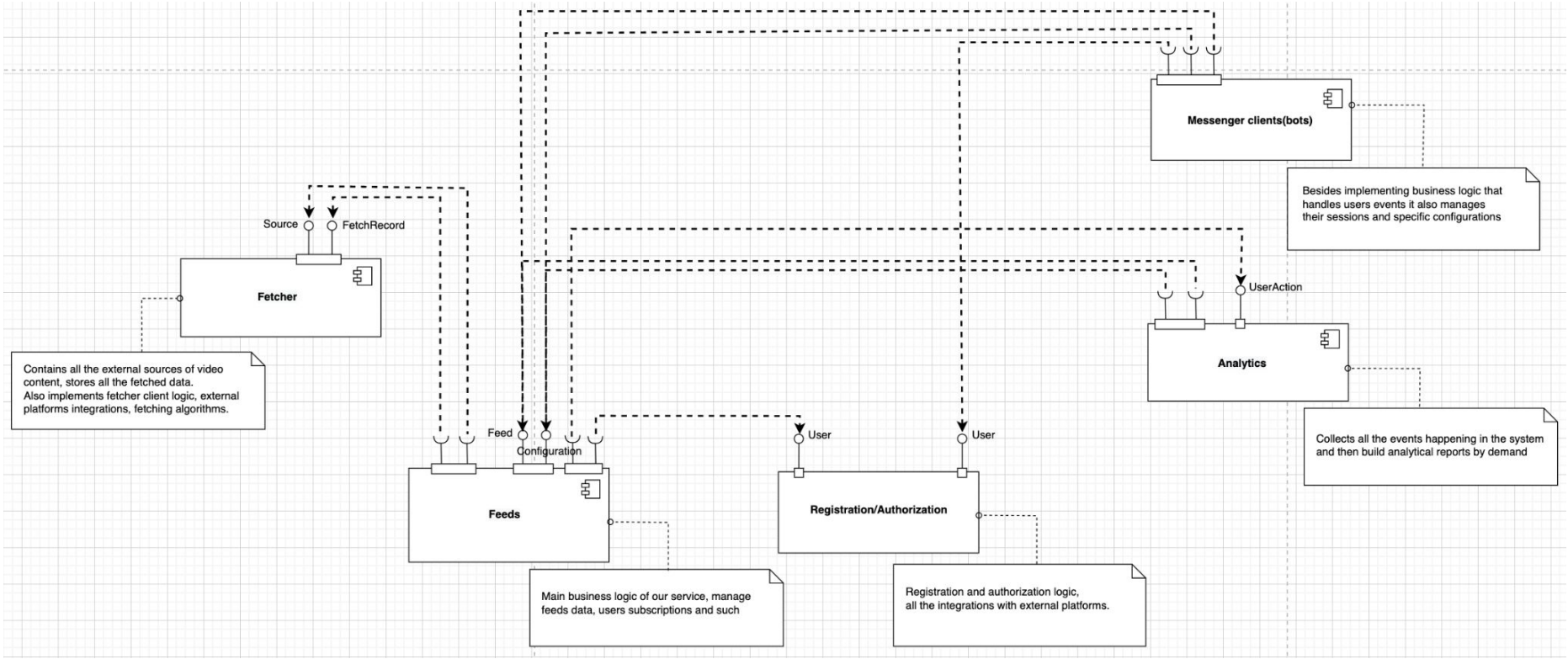
Link to full board:
https://github.com/hse-wasd-team/werec/blob/main/systemAnalysis/task8/Event%20Storming.png

Detailed class diagram

Image: https://github.com/hse-wasd-team/werec/blob/main/systemAnalysis/task8/Updated%20Class%20Diagram.png



System architecture



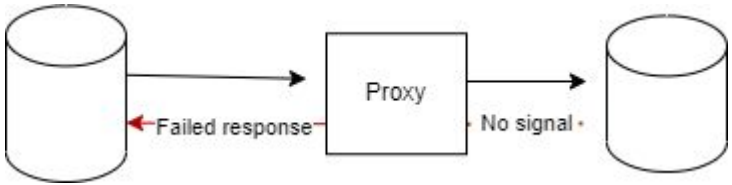
Design principle #1: Proxy service with Circuit breaker

Since there is a big amount of requests between microservices and to external services, we consider situation when a service crashes and does not respond. Sometimes it is better to return error right away instead of waiting for response. In critical situation it would save us some performance points and keep users aware that something happened.

For example, authorization service will also connect to Google OAuth 2.0 service and in case of getting no response from server, request will automatically terminate.

As for internal connections the same scheme will be applied.

Note: It is crucial to handle exceptions cause there can be different causes of request timeout.



но я бы конечно еще доработал, + более конструктивный рассказ, нужно раскрыть в чем суть принципа и дизайн кейса

Design principle #2: RESTful API design

Points following from RESTful design:

- Granularity of response - all of our endpoints return atomic data related to different entities, it directly affects design of our microservices interfaces;
- Allocation of resources - defines data allocation, it also defines the structure and entities structure, how everything is stored in the database;
- Client-server design which follows from RESTful design also brings the separation of concerns to our services, splitting apart client and server logic.

This principles leads to these patterns across our microservices:

- Interface separation - entities that are not related to logic of specific microservice and its entities are separated to other microservices;
- Single responsibility - entities in each microservice follow single responsibility principle and it is encouraged by higher structure of the system which comes from RESTful design.

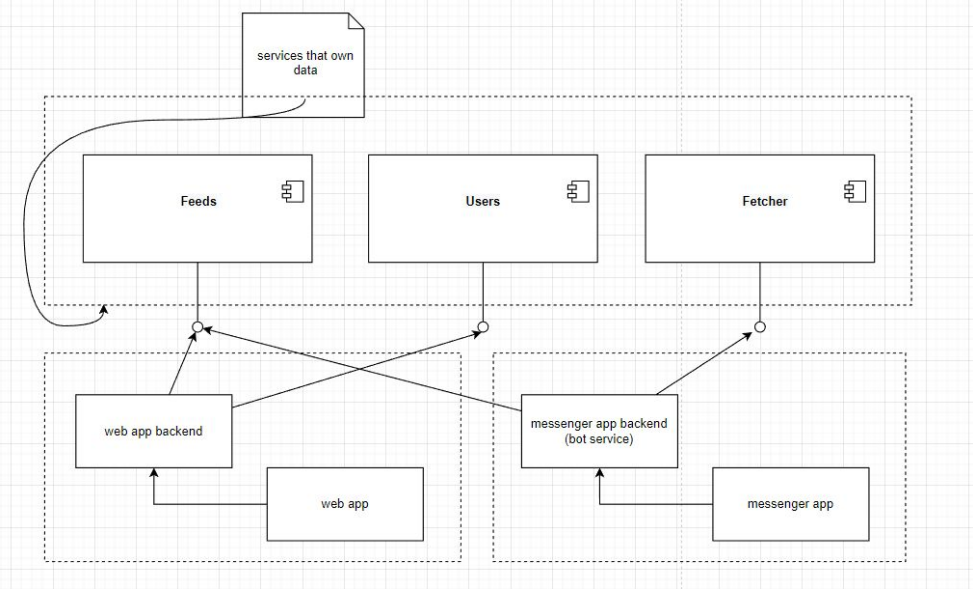
Design principle #3: Backend for frontend

Problem:

APIs have to support multiple clients, hence the increase of services complexity with the risk of building a monolith

Solution:

Separate backend for each client application



Solution stack

Implementation

- Api definition: OpenAPI
- Connection server: not necessarily needed for asynchronous asyncio framework. Nginx kinda solves problems from the same category
- App frameworks: asp.net for C#, aiohttp for Python, Gin for Golang
- Serialization/state format: json, marshmallow library for marshaling(in case of python), Json.NET for c#
- Deployment: Yandex Cloud (free trial)

Asynchronous interactions

- Apache Kafka to fetch data between microservices, Amazon SQS for long running tasks(building analytical reports)

Testing tools

- pytest, hamcrest for python testing
- Internal tools and mock library for golang
- xunit for .net unit tests

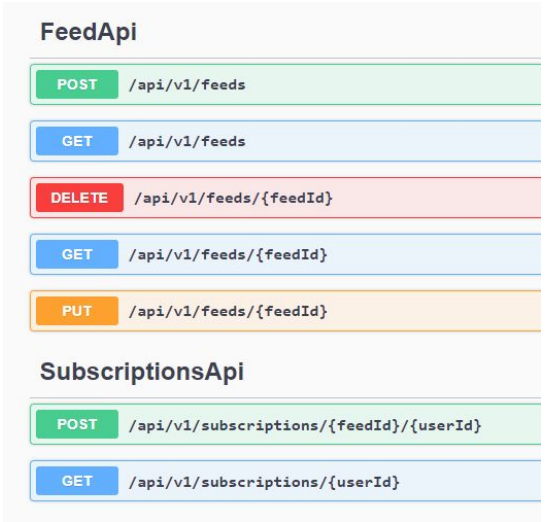
Operations

- CI/CD: github actions, octopus/terraform
- Delivery methods: docker
- Monitorings: Prometheus, Grafana, Sentry, Graylog

API usage service #1: Feeds

https://github.com/hse-wasd-team/werec/blob/main/backend/services/feeds/docs/specification.yml

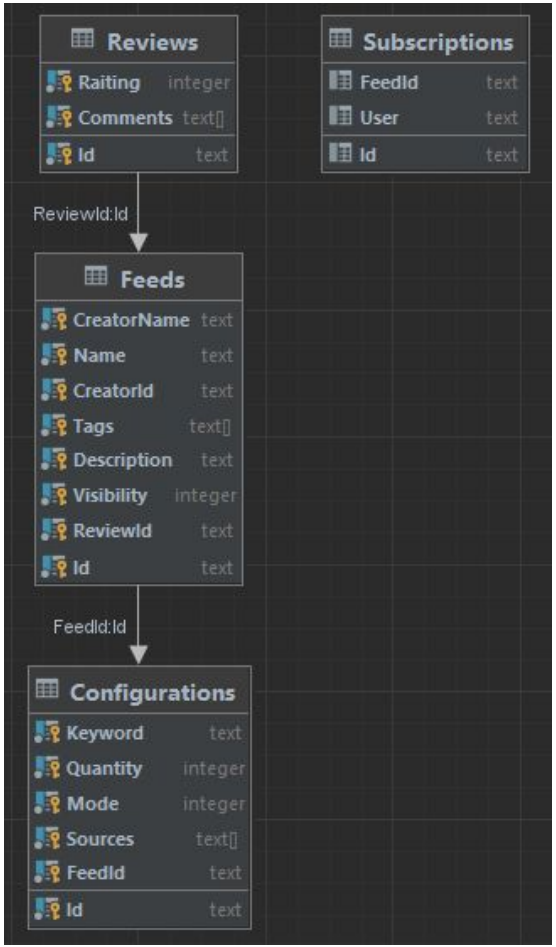
- Manages feeds
- Manages subscriptions for feeds (calls user service)
- Call analytics service to post user actions data, simple scenario:
  - User via bot subscribes to the feed
  - Bot service calls feeds service
  - Feed service verifies user data by calling user service
  - Feed service then saves the subscription
  - Feed service notifies analytics service about new user action
  - Analytics service then can add this data to the report
  - Finally, feed owner can view the report about how many users have subscribed to their feed



Feeds service, physical schema

```
create table if not exists "Feeds"
(
    "Id"          text not null
        constraint "PK_Feeds"
            primary key,
    "CreatorName" text,
    "Name"        text,
    "CreatorId"   text,
    "Tags"        text[],
    "Description" text,
    "Visibility"  integer,
    "ReviewId"    text
        constraint "FK_Feeds_Reviews_ReviewId"
            references "Reviews"
            on delete restrict
);

create index if not exists "IX_Feeds_ReviewId"
    on "Feeds" ("ReviewId");
```

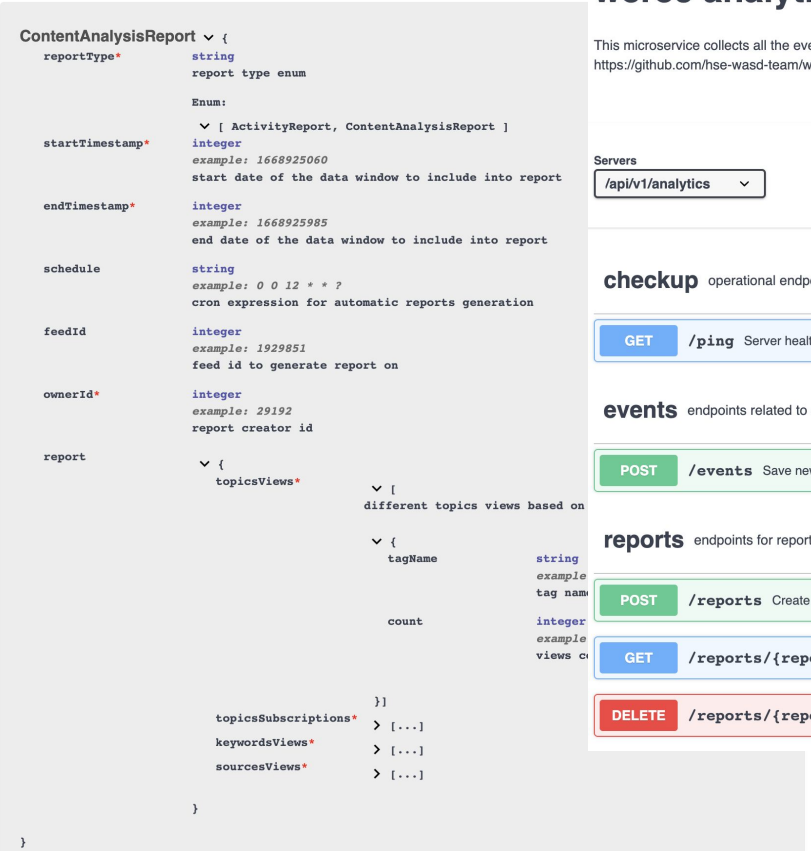
```
create table if not exists "Reviews"
(
    "Id"       text not null
        constraint "PK_Reviews"
            primary key,
    "Raiting"  integer,
    "Comments" text[]
);

create table if not exists "Subscriptions"
(
    "Id"     text not null
        constraint "PK_Subscriptions"
            primary key,
    "FeedId" text,
    "User"   text
);
```

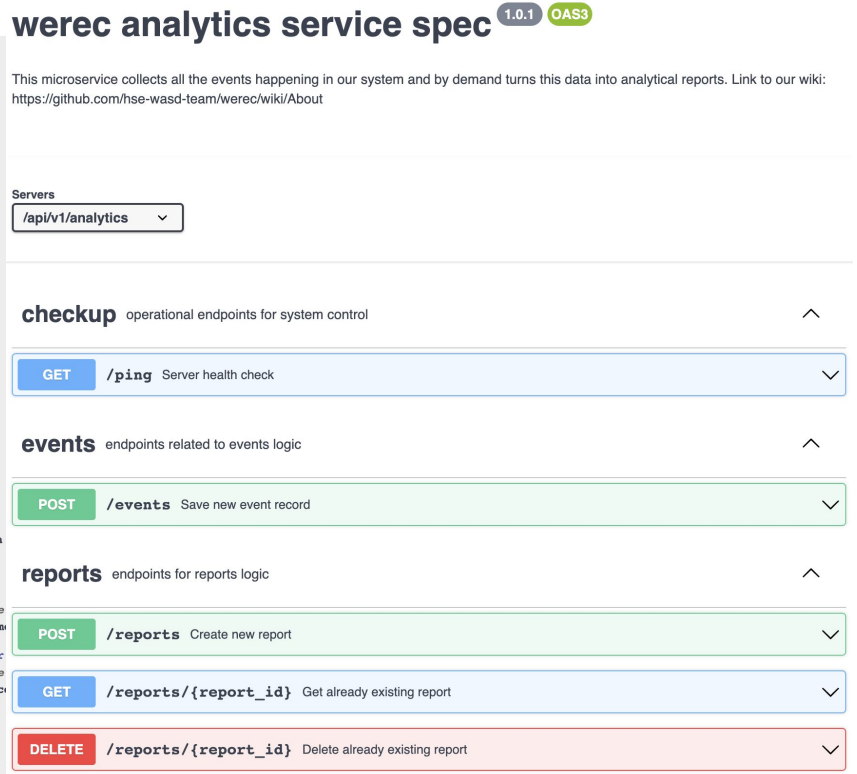


API usage service #2: Analytics

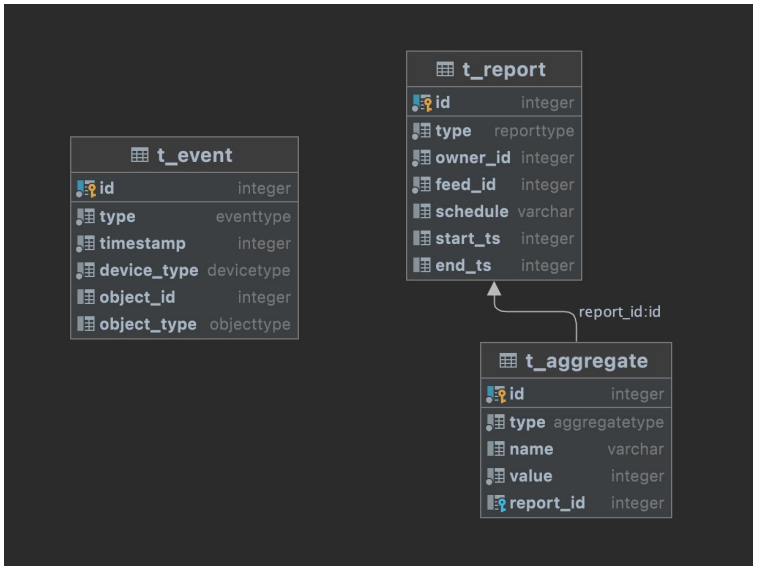
Link to specification source files: https://github.com/hse-wasd-team/werec/tree/main/backend/services/analytics/docs/openapi

Analytics service is in charge of collecting all the events happening in our distributed system. For that, every time when something is happening, active component sends post /events query to the analytics service. Then by demand (manual call or cronjob) it may build different analytical reports. post /reports to create new report, get and delete for corresponding operations





Analytics service, physical schema



analytics.t_report

```
CREATE TABLE t_report (
        id SERIAL NOT NULL,
        type reporttype NOT NULL,
        owner_id INTEGER NOT NULL,
        feed_id INTEGER NOT NULL,
        schedule VARCHAR,
        start_ts INTEGER,
        end_ts INTEGER,
        PRIMARY KEY (id)
)
```

analytics.t_event

```
CREATE TABLE t_event (
        id SERIAL NOT NULL,
        type eventtype NOT NULL,
        timestamp INTEGER NOT NULL,
        device_type devicetype NOT NULL,
        object_id INTEGER,
        object_type objecttype,
        PRIMARY KEY (id)
)
```

analytics.t_aggregate

```
CREATE TABLE t_aggregate (
        id SERIAL NOT NULL,
        type aggregatetype NOT NULL,
        name VARCHAR,
        value INTEGER NOT NULL,
        report_id INTEGER,
        PRIMARY KEY (id),
        FOREIGN KEY(report_id) REFERENCES t_report (id)
)
```

There is a class inheritance that is implemented by SQLAlchemy used for different report classes

API usage service #3: Authorization

https://github.com/hse-wasd-team/werec/blob/main/backend/services/auth/api/swagger.yaml

Authorization service is responsible for managing users and creating JWT Tokens for authorization.

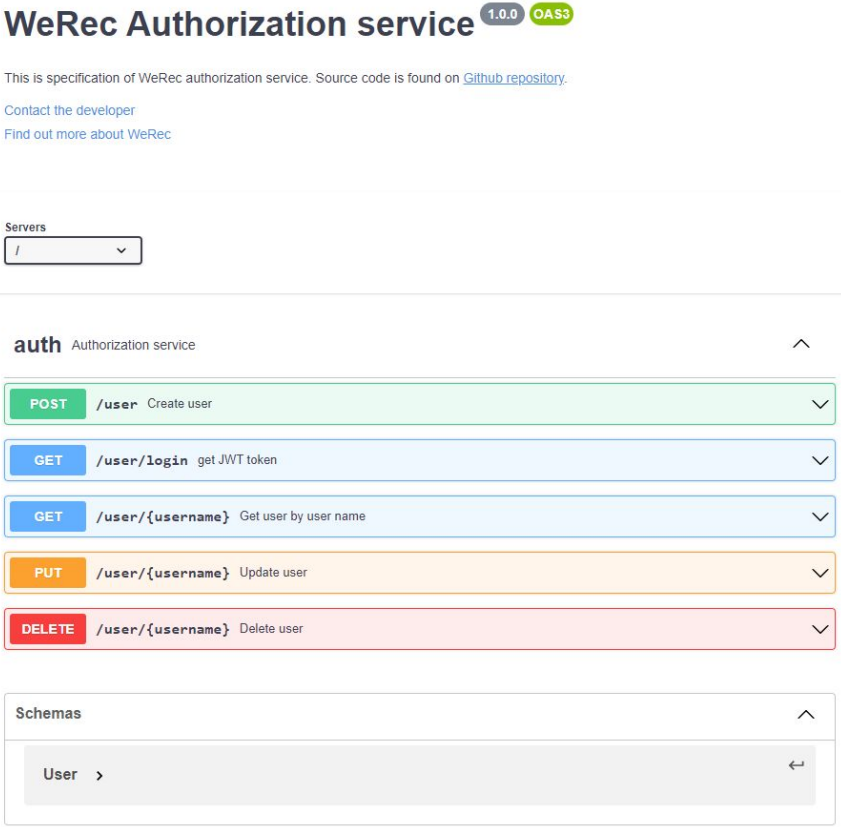
Service connects to API Gateway which authenticates web requests against system.

User {

id	integer($int64) example: 10
username	string example: theUser
email	string example: user@gmail.com
password	string example: password

}

example: OrderedMap { "password": "password", "id": 10, "email": "user@gmail.com



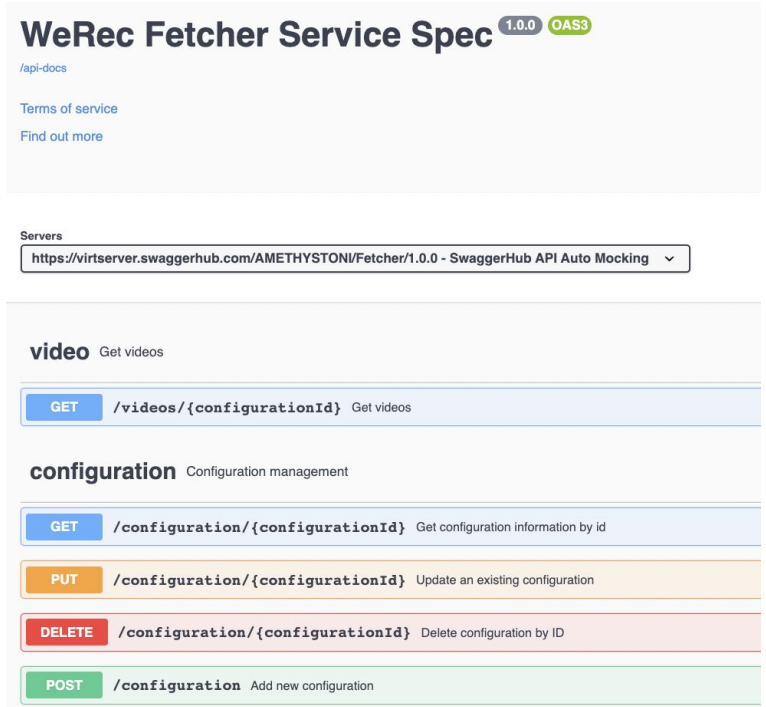
Auth service, physical schema


```
CREATE TABLE "public.Users" (
      "id" serial,
      "name" VARCHAR(255) NOT NULL UNIQUE,
      "created_at" DATETIME,
      "password" VARCHAR(255) NOT NULL,
      "email" VARCHAR(255) NOT NULL,
      "isActive" BOOLEAN NOT NULL,
      CONSTRAINT "Users_pk" PRIMARY KEY ("id")
) WITH (
  OIDS=FALSE
);
```

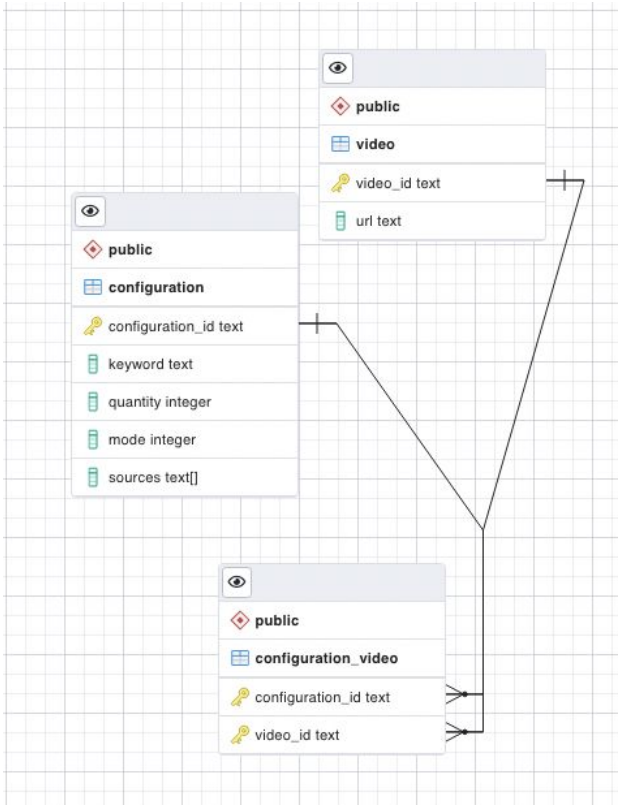
API usage service #4: Fetcher

https://github.com/hse-wasd-team/werec/blob/main/backend/services/fetcher/api/openapi.yaml

Fetcher service is responsible for managing configurations and fetching videos from platforms such as youtube based on configuration id. It connects to Youtube API and retrieves channel videos based on channel id, which we can retrieve via Youtube API knowing configuration source links.



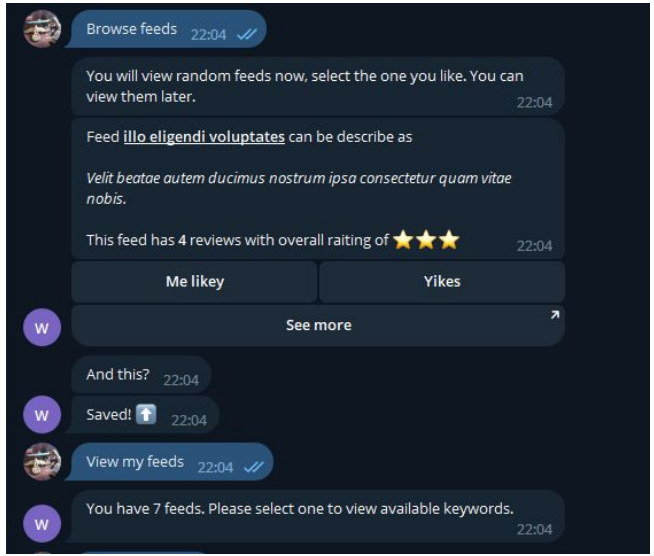
Fetcher service, physical schema




```
CREATE TABLE IF NOT EXISTS public.configuration
(
    configuration_id text COLLATE pg_catalog."default" NOT NULL,
    keyword text COLLATE pg_catalog."default" NOT NULL,
    quantity integer NOT NULL,
    mode integer NOT NULL,
    sources text[] COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "Configurations_pkey" PRIMARY KEY (configuration_id)
)

CREATE TABLE IF NOT EXISTS public.video
(
    video_id text COLLATE pg_catalog."default" NOT NULL,
    url text COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT video_pkey PRIMARY KEY (video_id)
)

CREATE TABLE IF NOT EXISTS public.configuration_video
(
    configuration_id text COLLATE pg_catalog."default" NOT NULL,
    video_id text COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "configuration-video_pkey" PRIMARY KEY (configuration_id, video_id),
    CONSTRAINT configuration_foreign_key FOREIGN KEY (configuration_id)
        REFERENCES public.configuration (configuration_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID,
    CONSTRAINT video_foreign_key FOREIGN KEY (video_id)
        REFERENCES public.video (video_id) MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
        NOT VALID
)
```

Service #5: Bot service

- Manages connections from messengers
- Store temporary data (like user choices)
- Not callable for other services (except /metrics)
- Call feed service to subscribe user to the feed
  - User selects the feed
  - Bot service calls feed service
  - Feed service process action
  - User can view previously subscribed feed
- Call analytics service to post data of user actions



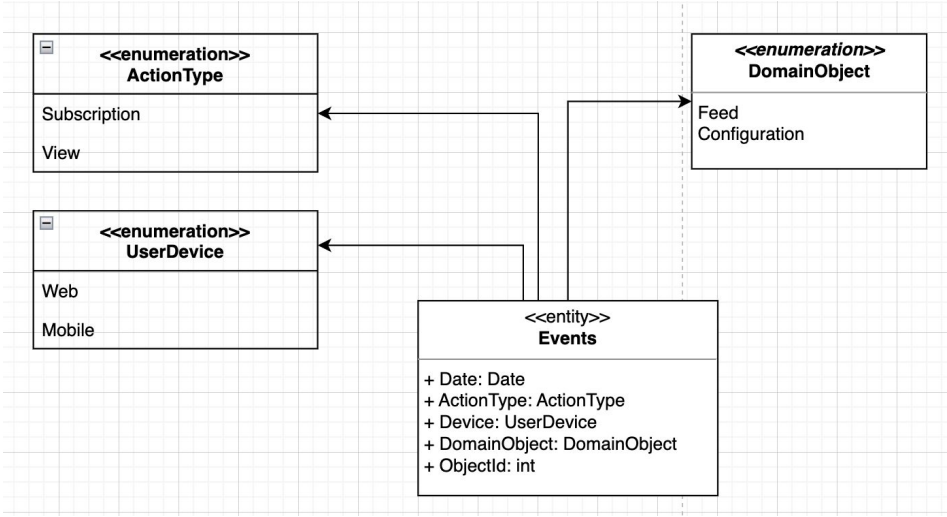
Design case #2: Event sourcing (first design case is circuit breaker)

Problem:

Making requests to our database we want to create different types of events for Analytics service(for example when querying recommendations by keywords or just viewing some feeds). Most of these operations need to be done reliably and in one transaction, none of the events should get lost.

Solution:

These scheme is very similar to saga pattern and a good option for solving it is event sourcing pattern. We will create new table for all types of events that happen in feeds service and every time some action triggers we will create new row in this table in one transaction.



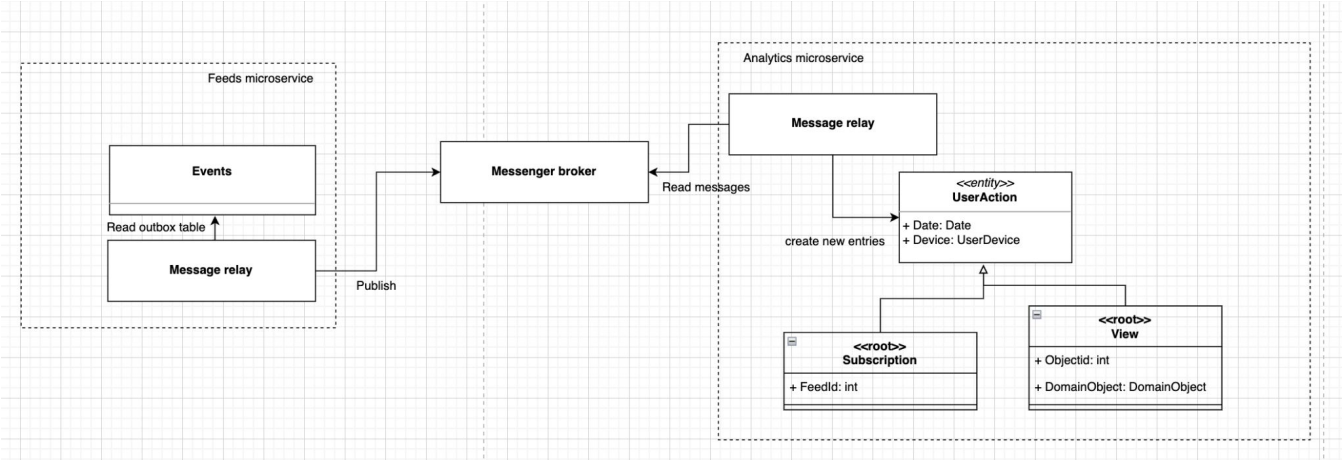
Design case #3: Transactional outbox

Problem:

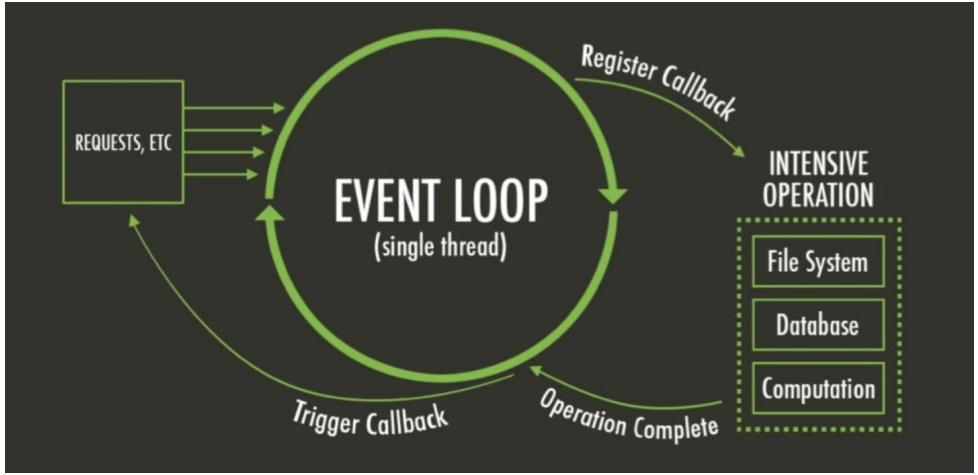
As the previous design case states we use event sourcing pattern for managing events in Feeds microservice. Event sourcing defines the way we work with those events locally in source microservice, how we create and store them, but to actually deliver these events to analytics microservice we need some other transportation solution.

Solution:

A good option for this problem is transactional outbox pattern. Table defined by event sourcing will take a role of outbox, events created here will be published to message broker(apache kafka) and then consumed by analytics microservice where we will save it to another table and use it to build analytical reports.



Design case #4: Event loop



Problem:

Microservices design of our system implies that we have many services talking to each other. That means that we have a ton of external requests and most operations will be IO intense(opposite to CPU intense). Using threads or processes to solve such a problem doesn't usually scale well.

Solution:

To fasten the system we will use asynchronous operations, which is one of the best solutions for such a problem for its simplicity and low resource consumption. Some languages have embedded support for this pattern, for example in Python we would just use *async* and *await* keywords to use this scheme(applied in Analytics microservice).

In two words this pattern works as follows: when we plan to do any kind of IO operation we add it to the queue and before returning to its execution we wait for the resource. Event loop handles events in the queue in its order and executes operations when resources are available.

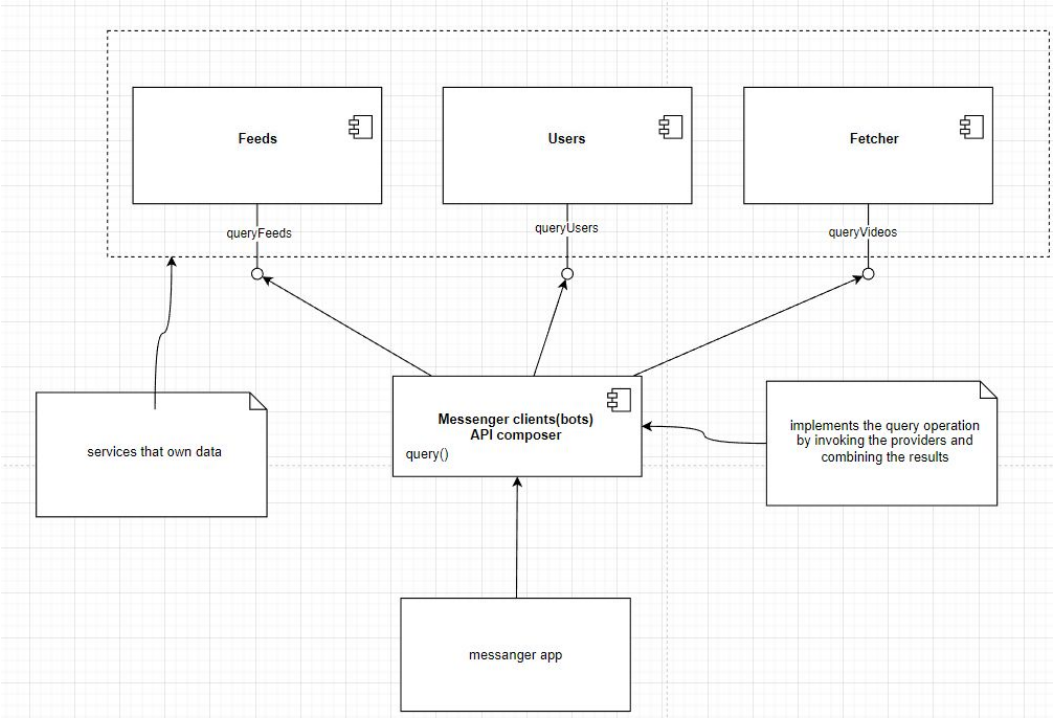
Design case #5: API composition

Problem:

Data is distributed amongst several microservices

Solution:

Implement a query in bot service that acts as an API Composer, which invoking the services that own the data and performs an in-memory join of the results.



Chidamber & Kemerer object-oriented metrics

	WMC	DIT	NOC	CBO
FeedRepository	5	1	0	1
SubRepository	2	1	0	1
FeedApiController	5	2	0	0
SubscriptionsApi	3	2	0	0
Feed	5	1	0	2
FeedConfiguration	4	1	0	2

WMC = number of methods defined in class

DIT = maximum inheritance path from the class to the root class

NOC = number of immediate sub-classes of a class

CBO = number of classes to which a class is coupled

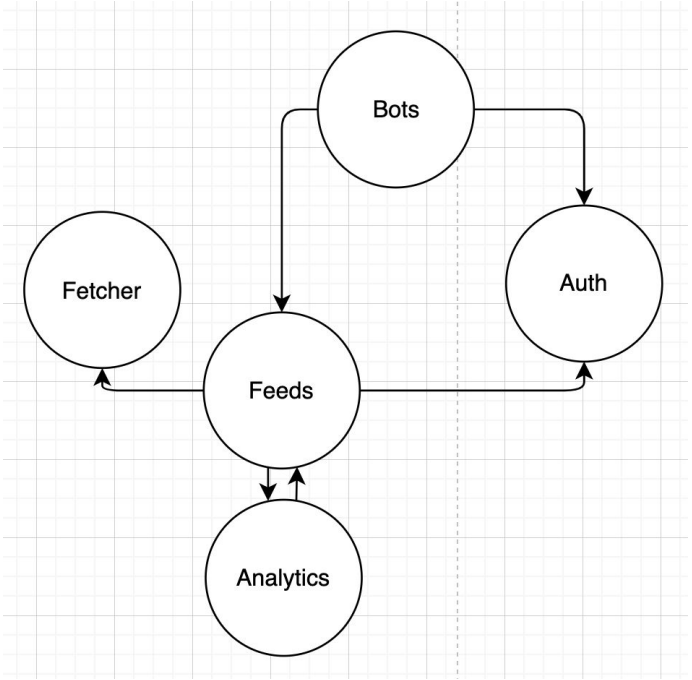
Code of feed service: https://github.com/hse-wasd-team/werec/tree/main/backend/services/feeds/WeRecWebApp

Design complexity, microservices coupling metrics

	Fetcher	Analytics	Feeds	Bots	Auth
SIY	0	1	1	0	0
AIS	1	1	2	0	2
ADS	0	1	3	2	0

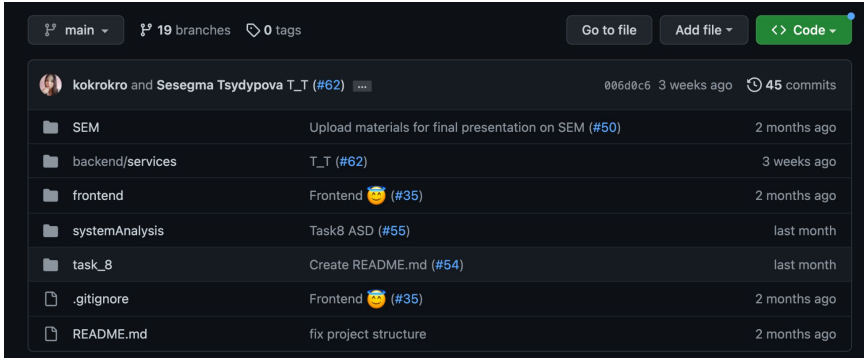
Notes:

1. SIY metric != 0 for analytics and feeds as they depend on each other;
2. AIS(S) equals to the number of service that depend on S;
3. ADS(S) equals to the number of other services that S depends on.



Microservices dependency graph

Repository structure

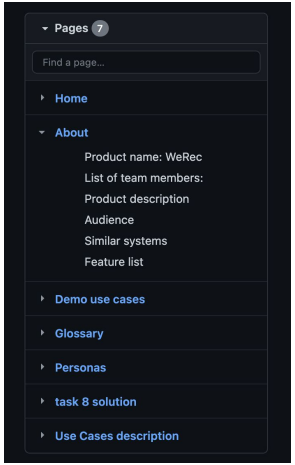


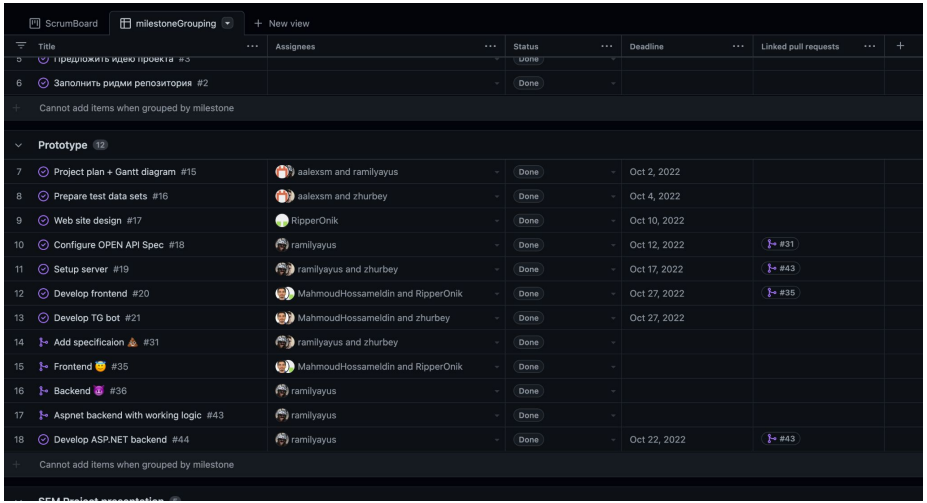
Main repository folders:

1. backend/services/* - folder containing microservices source code, each lives in a separate folder
2. frontend - frontend prototype source code
3. systemAnalysis - all the figures related to all the tasks

repo link:
https://github.com/hse-wasd-team/werec

1. For storing our docs we used github wiki, the structure is on the right
2. To plan our work we took a use of github projects, for each task we opened an issue which then was added to our project boards





Team and roles

Ramilya Yusupova
Backend developer

@ryusupova
rryusupova@hse.ru

Feeds service

Andrey Kan
Frontend developer

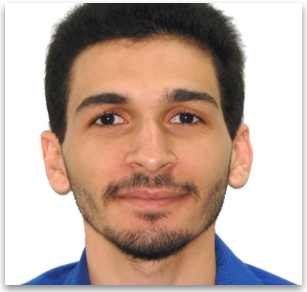
@AndrewKah
adkan@hse.ru

Fetcher service

Sergey Zhurbey
Backend developer, Tester

@ZhurbeySA
sazhurbey@edu.hse.ru

Analytics service



Mahmoud Hossameldin
Frontend developer

@mahhdn
mkhossameldin@edu.hse.ru

Bot service

Sesegma Tsydypova
Backend developer

@pepegma
syutsydypova@edu.hse.ru

Auth service