# Distributed System Assignment2 - MicroServices

Nick Wils

May 29, 2022

**Abstract**

You can find the code of my project with the following link: github_link.
To run the code you need to unpack zip or pull from the github link and then run the "run.sh" script from inside the directory.

## 1 Introduction

In this report I will decompose the given problem into microservices. After which I will explain how I implemented a subset of the features in this problem.

## 2 Decomposing in Microservices

In this section I will explain how I decomposed the problem. To do this I will go over all the microservices and give the following information:

- Features: Which features are included in this microservice.
- Database: The database that is used, and the microservices that share this database.
- Dependencies: The dependencies of the microservice, if there are any.
- Communication channels: which communication channels it has with other microservices.
- Explanation: An explanation on the choices I made.

I split the problem up into 9 microservices, with 5 different databases. You can find an overview of the decomposition on Figure 1 in the appendix. You can see the name of the microservice, the features it contains and which database it uses. Arrows from one microservice to another means that there is a (partial) dependency or some form of communication needed. While databases are sometimes used by multiple microservices, they should (if possible) use different tables to limit coupling.

## 2.1 Login

This microservice implements the login/registration.

### 2.1.1 Features

This microservice implements the following features:

- 1

Authorization is a very important and sensitive process. Because of this there is in a real world scenario added security. So this feature has a clear functionality that might have extra security. This is why this microservice has only 1 feature.

### 2.1.2 Database

This microservice uses the database: DB_User.

This database is used to store user information. That is all the groups, users and friends. The following microservices also use this database, as they also store user information (in which group they are and which friends they have):

- Friend
- Group

Because of the added security it might be a good idea to store the login microservice in a personal database.

### 2.1.3 Dependencies

This microservice stands alone and only needs access to the database.

### 2.1.4 Communication channels

There is 1 communication channel.

- Friend $\rightarrow$ Login : Is a given username in the database?

## 2.2 Friend

This microservice implements friend requests.

### 2.2.1 Features

This microservice implements the following features:

- 2

When splitting on functionality, it was a difficult decision if friends and grouping features should be in the same microservice. I decided to split it for the added graceful failure. When the Friend microservice is down, you can still watch movies with friends that are already in a group. Additionally you can also still make new groups, (you just can't add friends to the group). When the microservice Group is down, you can still add friends.

### 2.2.2 Database

This microservice uses the database: DB_User.

This database is used to store user information. That is all the groups, users and friends. The following microservices also use this database, as they also store user information (in which group they are and the users that exist):

- Login
- Group

### 2.2.3 Dependencies

This microservice has one dependency beside the database. It has to know if a user exists before adding the user as a friend. But this dependency makes sense, If the Login is down, then you can't log in, so it doesn't matter that you can't add friends.

It has a dependency on microservice Login.

### 2.2.4 Communication channels

There are 2 communication channels.

- Friend $\rightarrow$ Login : Is a given username in the database?
- Newsfeed/Search $\rightarrow$ Friend : Which friends does the user have?

## 2.3 Group

This microservice implements grouping functionality.

### 2.3.1 Features

This microservice implements the following features:

- 8, 9

When splitting on functionality, it was a difficult decision if friends and grouping features should be in the same microservice. I decided to split it for the added graceful failure. When the Friend microservice is down, you can still watch movies with friends that are already in a group. Additionally you can also still make new groups, (you just can't add friends to the group). When the microservice Group is down, you can still add friends. Features 8 and 9 both have a grouping functionality.

### 2.3.2 Database

This microservice uses the database: DB_User.

This database is used to store user information. That is all the groups, users and friends. The following microservices also use this database, as they also store user information (who are friends and the users that exist):

- Login
- Friend

### 2.3.3 Dependencies

This microservice has one dependency beside the database. It has to know if the user is friends with a given name. If this dependency is not fulfilled, the only functionality that goes away is the ability to add friends to a group.

It has a dependency on microservice Friend.

### 2.3.4 Communication channels

There is 1 communication channel.

- Group → Friend : Is the user friends with a name?

## 2.4 Watch

This microservice implements streaming functionality.

### 2.4.1 Features

This microservice implements the following features:

- 3

Watching a movie has a clear functionality. Furthermore it might need different libraries for its implementation. This is why it has only one feature.

### 2.4.2 Database

This microservice uses the database: DB_Movie.

This database is used to store movie information. That is a list of all the movies with additional information and streaming information. The following microservice also uses this database, as they also need movie information (which movies there are):

- Search

It is possible that this microservice needs a personal database, as it is possible that the information needs to be stored in a different way then information for a search.

### 2.4.3 Dependencies

This microservice has one dependency beside the database. When a user watches a part of a movie and comes back later, the user will probably want to start from where he left off. To make this possible the Logs microservice needs to work. If this dependency is not fulfilled then the user will not be able to start from where he left off.

It has a dependency on microservice Logs.

### 2.4.4 Communication channels

There are 2 communication channels.

- Watch $\rightarrow$ Logs : Where should the movie start?

## 2.5 Search

This microservice implements a search bar with filters .

### 2.5.1 Features

This microservice implements the following features:

- 4, 5, 6, 7

This microservice contains the most features. The functionality that all these features implement is a search on the available movies. It is possible that feature computer-generated tags need to be in another microservice, depending on which libraries are needed for feature 6. When thinking about graceful failure, you can consider moving feature 4 to another microservice as it is crucial to the website that you can see a list of all the movies. It is also possible that the search functionality needs a lot of updates to give the most accurate movies back. Taking all this in consideration, I still decided that with the information I currently have, it makes the most sense to keep all the features in the same microservice.

### 2.5.2 Database

This microservice uses the database: DB_Movie.

This database is used to store movie information. That is a list of all the movies with additional information and streaming information. The following microservice also uses this database, as they also need movie information (streaming information):

- Watch

### 2.5.3 Dependencies

This microservice has three dependencies beside the database. Providing a list of all movies, a simple search bar and computer generated tags will always work and have no dependency other than a running database. But to filter for recommendations of friends, you need to know which friends you have and what they recommend. I also added the dependency on Logs as you probably want to use the information of what movies you have already watched in the search bar.

It has a dependency on microservices: Friend, Rating and Logs.

### 2.5.4 Communication channels

There are 2 communication channels.

- Search $\rightarrow$ Logs : What movies did the user already watch?
- Search $\rightarrow$ Friend : What friends does the user have?
- Search $\rightarrow$ Rating : What do friends recommend?

## 2.6 Log

This microservice implements the logging of viewing activity.

### 2.6.1 Features

This microservice implements the following features:

- 14

Logging everything viewing history of all users is a clearly defined functionality. This is clearly something that can stand alone, as it just keeps track of what everybody is watching and logs this information.

### 2.6.2 Database

This microservice uses the database: DB_Log.

This database is used to store logging information. That is all the information regarding viewing activity: what movie, when, how long, in which group, ...

### 2.6.3 Dependencies

This microservice has no dependencies beside the database. It works stand-alone.

### 2.6.4 Communication channels

There are 4 communication channels.

- Watch → Logs : Where should the movie start?
- Search → Logs : What movies did the user already watch?
- Advertisement → Logs : What did a user watch?
- Newsfeed → Logs : What did a set of users recently watch?

As you can see these channels give slight variations on the same information. It can be that some channels merge.

## 2.7 Rating

This microservice implements feedback of a movie in the form of rating and recommendations.

### 2.7.1 Features

This microservice implements the following features:

- 10, 11

Rating a movie and recommending it to friends have a similar functionality. They might even be done in the same form, where do you recommend this movie is a checkbox. It can be more complex, but the base functionality is very similar and I don't see a reason why these should be in different microservices.

### 2.7.2 Database

This microservice uses the database: DB_Review.

This database is used to store review information for each movie and any user reviews. It also stores recommendations.

### 2.7.3 Dependencies

This microservice has no dependencies beside the database.

### 2.7.4 Communication channels

There are 3 communication channels.

- Newsfeed → Rating : What does a set of users recommend?
- Advertisement → Rating : What does a user like?
- Search → Rating : What do friends recommend?

## 2.8 Newsfeed

This microservice implements the news feed.

### 2.8.1 Features

This microservice implements the following features:

- 12

When splitting the features up in functionality, I had to consider that advertisement is linked to Newsfeed. But advertisement is a paid service, which means that there are probably more outside dependencies. Furthermore there are probably more updates required on the advertisement service, to keep up to date with security. Lastly it makes more sense to split due to graceful failure. If advertisement is down, that should not compromise the whole news feed. That is why I choose to split these features up in two different microservices.

### 2.8.2 Database

This microservice uses the database: DB_Feed.

This database is used to store news feed information. This could be a list of ads for example. The following microservice also uses this database, as they also store news feed information:

- Advertisement

Depending on the implementation, this database is unnecessary.

### 2.8.3 Dependencies

This microservice has three dependencies beside the database. The news feed shows among other things what friends recently watched. This means you need to know which friends you have and what they watched. It can also be interesting to add to the news feed what friends recommended or liked.

It has a dependency on microservices: Friend, Rating and Logs.

### 2.8.4 Communication channels

There are 4 communication channels.

- Newsfeed → Friend : Which friends does the user have?
- Newsfeed → Rating : What does a set of users recommend?
- Newsfeed → Logs : What did a set of users recently watch?
- Advertisement → Newsfeed : Show this ad.

## 2.9 Advertisement

This microservice implements the advertisement functionality.

### 2.9.1 Features

This microservice implements the following features:

- 13

When splitting the features up in functionality, I had to consider that advertisement is linked to Newsfeed. But advertisement is a paid service, which means that there are probably more outside dependencies. Furthermore there are probably more updates required on the advertisement service, to keep up to date with security. Lastly it makes more sense to split due to graceful failure. If advertisement is down, that should not compromise the whole news feed. That is why I choose to split these features up in two different microservices.

### 2.9.2 Database

This microservice uses the database: DB_Feed.

This database is used to store news feed information. This could be a list of ads for example. The following microservice also uses this database, as they also store news feed information:

- Newsfeed

Depending on the implementation, this database is unnecessary.

### 2.9.3 Dependencies

This microservice has three dependencies beside the database. First of all the advertisement needs to be shown, that's what people pay for. That's why there is some dependency on Newsfeed. But you could still pay for advertising when the website is down, the ad would be saved and shown at a later moment. Furthermore it will probably be interesting for advertisers to specify to whom the ad is shown. For this you could use information stored in the logs and ratings.

It has a dependency on microservices: Newsfeed, Rating and Logs.

### 2.9.4 Communication channels

There are 3 communication channels.

- Advertisement → Newsfeed : Show this ad.
- Advertisement → Logs : What did a user watch?
- Advertisement → Logs : What did a set of users recently watch?

# 3 Implementing Microservices

With the explained decomposition there are four different microservices to implement:

- Login
- Friend
- Group
- Search

Login, Friend and Group all have the same database, only Search has another database.

I implemented each service one by one and tested their graceful failure each time a new microservice was finished. I used django to implement the microservices.

For the database, I choose to use the official image of postgres for this. Because I need two different databases, I added a script I found on the following website. This made it possible to add multiple databases in one container.

I used the following ports for the host:

- 5433 : postgres
- 8001 : Login
- 8002 : Friend
- 8003 : Group
- 8004 : Search

I will now go over all the API endpoints. For testing purposes I also implemented for most microservices. These can be accessed as follows:

- Login : GET :8001/api/register
- Friend : GET :8002/api/friend
- Group : GET :8003/api/group

These will return all the info in one of there tables.

For the API's I worked with adding information in the body this time. This information should be in json format.

### 3.0.1 LOGIN: Register User

**POST :8001/api/register**

Add an entry in the User table with user and password. Only if username is not already in the table.

| Required body in json | |
|---|---|
| username | string(required) Example: nick<br>Name of the user. |
| password | string(required) Example: azerty<br>Password of the user. |

### 3.0.2 Response 201

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "boolean",
        "description": "Successful operation."
    },
    "required": [
        "response"
    ]
}
```

### 3.0.3 Response 400

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": False
}
```

### 3.0.4 LOGIN: Login user

**GET :8001/api/login**

Check if the user is in the database with the correct password for login.

| Required body in json | |
|---|---|
| username | string(required) Example: nick<br>Name of the user. |
| password | string(required) Example: azerty<br>Password of the user. |

### 3.0.5 Response 200

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "boolean",
        "description": "Successful operation."
    },
    "required": [
        "response"
    ]
}
```

### 3.0.6 Response 400

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": False
}
```

### 3.0.7 LOGIN: Check if user Exists

**GET :8001/api/user_exists**

Check if the user is in the database.

| Required body in json | |
|---|---|
| username | string(required) Example: nick<br>Name of the user. |

### 3.0.8 Response 200

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "boolean",
        "description": "If user exists."
    },
    "required": [
        "response"
    ]
}
```

### 3.0.9 Response 400

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": False
}
```

### 3.0.10 FRIEND: Add friend

**POST :8002/api/friend**

Add an entry in the Friend table with current user and friend. Only if friend exists and not already friends.

| Required body in json | |
|---|---|
| username | string(required) Example: nick<br>Name of the user. |
| friendname | string(required) Example: eline<br>Name of the friend. |

### 3.0.11 Response 201

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "boolean",
        "description": "Successful operation."
    },
    "required": [
        "response"
    ]
}
```

### 3.0.12 Response 400 or 500

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": False
}
```

### 3.0.13 FRIEND: Check if user is friends with friend

**GET :8002/api/friend_exists**

Check if the two users are friends in the database.

| Required body in json | |
|---|---|
| username | string(required) Example: nick<br>Name of the user. |
| friendname | string(required) Example: eline<br>Name of the friend. |

### 3.0.14 Response 200

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "boolean",
        "description": "If they are friends."
    },
    "required": [
        "response"
    ]
}
```

### 3.0.15 Response 400

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": False
}
```

### 3.0.16 GROUP: Create group

**POST :8003/api/group**

Add an entry in the Group table with the groupname. Only if groupname is not in the table already. Add the user as member of the group in the GroupUser table.

| Required body in json | |
|---|---|
| username | string(required) Example: nick<br>Name of the user. |
| groupname | string(required) Example: WINAK<br>Name of the group. |

### 3.0.17 Response 201

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "boolean",
        "description": "Successful operation."
    },
    "required": [
        "response"
    ]
}
```

### 3.0.18 Response 400

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": False
}
```

### 3.0.19 GROUP: Add friend to group

**POST :8003/api/group_user**

Add an friend as member of group in GroupUser table. Only if the group exists, user and friend are friends, user is in the group and friend is not yet a member.

| Required body in json | |
|---|---|
| username | string(required) Example: nick<br>Name of the user. |
| friendname | string(required) Example: eline<br>Name of the friend. |
| groupname | string(required) Example: WINAK<br>Name of the group. |

### 3.0.20 Response 201

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "boolean",
        "description": "Successful operation."
    },
    "required": [
        "response"
    ]
}
```

### 3.0.21 Response 400 or 500

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": False
}
```

#### 3.0.22 SEARCH: Get list of all the movie titles

**GET :8004/api/movielist**

Return a list of all the titles in the Movie table.

#### 3.0.23 Response 200

**Header**

```
Content-Type: application/json
```

**Body**

```
{
    "response": True
}
```

**Schema**

```
{

    "response": {
        "type": "list",
        "description": "list of all the movie titles"
    },
    "required": [
        "response"
    ]
}
```

# 4 APPENDIX

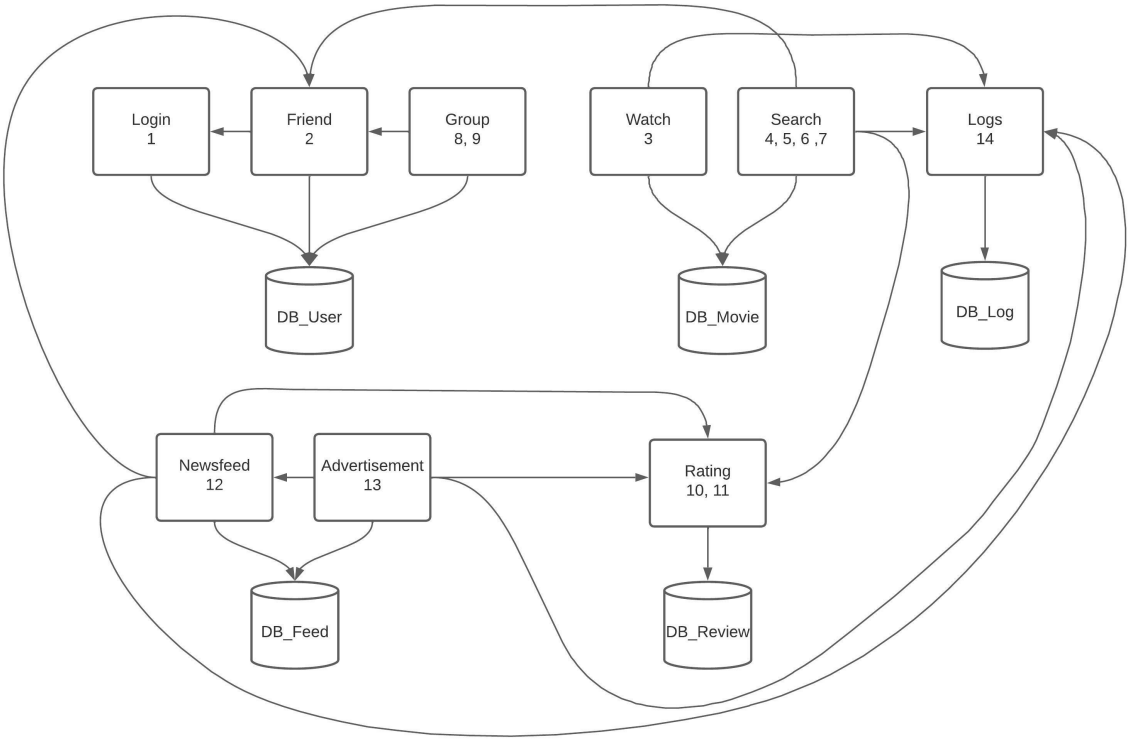


Figure 1: Microservice decomposition