FindIt - a Microservices Plattform to Organize Documents

Max Mayr, Max Zauner

July 9, 2018

Abstract

Throughout this semester we had to plan and implement a project. My project partner and I both have a very strong background in web technologies thus we decided to use this skills and improve them by realizing this project. Our decision was to create a microservice platform that allows users to easily upload, organize and find their documents. In this report, we explain the architecture and technologies that we used to implement the described platform.

microservices, kafka, event-sourcing, event-based, docker

1 Introduction

1.1 Motivation

For our semester project we decided to build software that help us and other users to easily organize their documents. As we both already have several years of experience with web technologies we decided to use technologies and patterns that are new to us or that we haven't used before. Our intention was to further improve our skills and to get in touch with new technologies while actively using them.

Furthermore we recently noticed that numerous web-service providing companies were having severe scaling problems. Some companies took no immediate action at all which resulted in temporarily unstable and unreachable services while other companies tried to limit incoming requests by blocking new user registrations. Either way companies were loosing a lot of money by not being able to correctly scale their platforms according to their users needs. Thats why we decided to focus our project on scalability and thus decided to build a microservice architecture to crate the platform.

1.2 Goal

The goal of this project is to create a prototype of a fully scalable microservice architecture that, as stated before, allows users to easily upload, organize, access and find their private documents and files. The prototype should at least enable the user to upload documents and afterwards search them through the user interface. The search should also search the content of the uploaded file. The platform should be provided as a web application.

This document is intended to explain all the details and architectural design decisions of the created system.

2 Microservices

At the time of writing this report the term "microservice" is still mostly a buzzword, but more and more companies are adapting microservice design paradigms or are already trying to convert their monolithic backend structures to a microservice architecture.

One of the pioneers of microservice adoption is Netflix who is currently responsible for over 30%[1] of all Internet traffic. The company uses a large scale, service oriented architecture to power their platform and provide their service to over 800 different kinds of devices. According to Netflix an average API call spreads out into six internal microservices.

Other big companies like Ebay, Amazon and Linkedin are using microservices in production already as well. Even startups are now realizing that they need to use a microservice architecture in order to being able to scale their product if the attention to it is increasing exponentially.

[1] microservices.io/patterns/microservices.html

2.1 Why the paradigm shift?

Nowadays an ordinary video can gain millions of views in just a few minutes after it has been uploaded. The same request behavior also applies to services that are available on the web. Of course scaling a web service is not a new problem but the traditional way of scaling a web service, which was mostly vertical scaling or adding more servers that run the same instance of the application, does not work very well anymore in the scale that is necessary nowadays as more and more people have access to the web and also actually use it in their day to day lives. Furthermore, one of the biggest problems in addition to the scaling problem is maintaining code bases and decrease release circle times.

To compensate the steadily increasing scaling need of web applications, programmers tried to solve the scaling problems by looking at existing software solutions. Thats the reason why the idea behind micro-services is copied or inherited from the UNIX-philosophy[2] which states the following:

Do One Thing and Do It Well

This means microservices should only focus on a specific part of the system and try to do that part as good, fast, and reliable as possible.

Additionally micro-services should be guided by the following principles:

- Keep it as small and simple as possible.
- Don't mix domain objects and follow domain-driven design.
- Provide and use common interfaces that are self documenting.
- Version your micro-service to allow stepwise migrations.
- Never share a database with another service.
- Never communicate with another service over another way as the interface.

Other principles that are good to know or are considered as best practice:

- Only one team should be responsible for a specific microservice.
- Microservices should use the technologies that are most suited for the task and the responsible team.
- Keep it stateless if possible.
- Use Continuous Integration and Deployment.

[2]en.wikipedia.org/wiki/Unix_philosophy

Many other additional guidelines and principles exist in the microservice environment but the ones listed above are the most common and agreed ones.

2.2 Data management

The most difficult problem to solve in the microservice environment is the management of data and relations. Usually data is tightly coupled but due to the guidelines it is now necessary to separate domain models to several different microservices in order to keep everything scalable and atomic. Therefore a lot of solutions have been invented to sync database changes to multiple services.

The difficult part to solve is not the synchronization by itself but how to synchronize data in a controlled way. A good example would be the change of a user's name. The new user name must be distributed to all services that need the user's name to function correctly. Therefore the integrity across all microservices must be guaranteed all time. For example the emailing service that sends an invoice should always use the most recent username when sending an invoice. The question is what happens if the update statement in on of the services throws an error or gets a timeout? How should the error be handled and how can an erroneous change be rollbacked across all microservices?

2.2.1 Pattern: Shared database

The solution that seems most obvious is to just ignore the guidelines and use a shared database[3] across multiple services so all services that use the database can make use of the transaction mechanism.

This pattern can make sense in some cases but as already mentioned it is totally against the guidelines as a change in the database schema could break other microservices if changes are not properly tested. Also through the fact that transaction can and will lock out changes of other services a deadlock or bottlenecks can occur that crash all the involved microservices.

The reason the microservice architecture is used is to decouple certain parts of the whole service so the it can still work if some microservice has stopped working. By coupling every microservice to one monolithic database can negate the whole microservice pattern in case of a database failure.

[3]microservices.io/patterns/data/shared-database.html

2.2.2 Pattern: Database per Service

Having a database as a service is the recommended approach[4] for microservices. Each service has it's on database and no other service is allowed to access the database. Because the data is separated it's quite more difficult to query all data that is needed in one API call. That is also the reason why this pattern is often combined with paradigms like the API-Composition[5] or the Command Query Responsibility Segregation[6] (CQRS) Pattern. Those Patterns allow to query data over multiple microservices.

By using the mentioned patterns it is possible to query data over multiple microservices but atomic transactions are still a problem. Now either the old approach with two-phase commits can be used or the Saga pattern can be considered. Because two-phase commits are frowned since years already as the pattern always creates problems in conjunction with microservices the saga pattern should be used instead.

2.2.3 Pattern: Saga

The Saga pattern[7] describes a series of local transactions, where each transaction that has completed triggers the next transaction. If any transaction fails it is just necessary to revert the transactions in the opposite direction by reverting every local transaction with new local transactions. Generally it can be said that every rollback or compensation transaction therefore triggers the next/previous one.

To use the Saga pattern an event system is needed that handles the triggering of actions. This can be done with the following patterns: Event Sourcing[8], Application events[9], Database triggers[10] or transaction log trailing[11].

The most common pattern to handle events is the Event Sourcing pattern.

[4]microservices.io/patterns/data/database-per-service.html
[5]microservices.io/patterns/data/api-composition.html
[6]microservices.io/patterns/data/cqrs.html
[7]microservices.io/patterns/data/saga.html
[8]microservices.io/patterns/data/event-sourcing.html
[9]microservices.io/patterns/data/application-events.html
[10]microservices.io/patterns/data/database-triggers.html
[11]microservices.io/patterns/data/transaction-log-tailing.html

2.2.4 Pattern: Event Sourcing

With the Event Sourcing pattern[12] every entity consists of a series of state-changing events. This means that for every entity there must me a created event. Also, for every change on the entity a change-event must be appended to the list of events. A write operation is therefore always atomic since the database is the log itself. Multiple producers can create events and push them to the log and multiple consumers can consume them if needed. The event sourcing pattern allows the system to update an entity in all microservices at the same time in one transaction because everything is based on the same log.

3 Design

This section contains all the motivation why we chose a particular application design pattern.

3.1 Architecture

We decided at the beginning that we want to build a fully scalable document management solution. To achieve this we are utilizing the microservice pattern. To not run into any problems we split up our domain objects and separated them as much as possible.

Furthermore we decided to utilize the Event-Sourcing pattern to stay flexible with how our services work and how our database is stored. This way we can change a microservice all the time and just replay all events to check if it works as planned. The beauty of the event-sourcing pattern is the fact that any service can easily be completely rebuilt with the most recent state by rebuilding the internal database with the events from the log.

To scale the platform easily we are using Docker[13] and docker-compose scripts. Docker is a lightweight virtualization software that makes it possible to run so called containers on an operating system. Containers in comparison to virtual machines use far less resources but still are completely isolated from other containers. Additionally Docker allows to define internal networks in the code. By having an docker internal network it is possible to create complex architectures for internal services. From a developers point of view the main advantage of using

[12] microservices.io/patterns/data/event-sourcing.html
[13] docker.com

containers is that once we have a working container build running - such a build is called “image” - docker guarantees that the image can be run the same way on any system that is capable of running docker containers.

Docker also provides so called compose scripts that allow the description of multiple containers, networks and volumes in a single file that can be scaled with just a single line of code. Once the system is running it is possible to instruct compose to start additional containers of a specific image. This way we can easily scale a service if there is demand for it.

Every service and database has an image which gets created on demand by using a Dockerfile. A Dockerfile describes the container that each service provides and consists of building instructions.

3.2 Services

After our foundation was fixed and our domain objects where known we had to think about our services. The services are designed to have small and clear interfaces that are accessible over REST-endpoints. All write operations create an event and are consumed asynchronously by the microservices that are designed to consume specific events. Thus all write operations are truly asynchronous as the time the event is consumed is undefined and depends on the load of the consuming microservices.

Of course by using this pattern we are facing many kinds of other problems that need to be solved but it’s still the best way to keep the platform scalable and as this project is a proof of concept we just try to produce a minimum viable product.

3.2.1 User-Service

The user service is responsible for everything related to a user. This includes signup, login, roles, and profiles of users. We decided to also include the authorization logic in this microservice because the user service is really small and splitting it up later would not be a problem at all.

Once a user signs in or changes his profile the user service sends an event so other microservices are aware that a new user has been created. This is a good example how event sourcing can be used to keep user-data consistent over multiple microservices.

3.2.2 File-Service

This service manages everything concerning the uploading and retrieving of files. Because we plan to use an external storage provider this service will perform the only write operation in the system that is immediately answered because we will have to somehow await confirmation from the storage provider that the file is saved.

After the confirmation the request from the frontend client will be answered as successful but the file service then sends out an event telling other services that a new file is available for a specific user. To keep the user experience acceptable the user should not wait until the file has been processed by the whole system.

3.2.3 Metadata Extraction-Service

Our metadata extraction service is responsible to extract information out of the uploaded documents. It listens to events from the file service and once it consumes a file-upploaded event it then downloads the file in order to analyze it. The extracted content will be sent out as an event for every microservice that is consuming the extracted metadata.

3.2.4 Fulltext Search-Service

The fulltext search service provides a public interface to search through all the files that belong to the user that is currently logged into the system. It consumes the events from the metadata service and updates it's database with new incoming data.

3.2.5 API-Gateway

To provide a common interface to all our microservices we will use the API-gateway pattern[14]. This pattern describes a service that discovers what kinds of services are running in the background and routes requests to them if possible. That way the internal microservice architecture is hidden and all public requests have to access the gateway first before they can be routed to the actual microservice.

The API-gateway acts as a gatekeeper denying invalid or unauthorized requests to different aspects of the microservices.

[14]microservices.io/patterns/apigateway.html

4 Implementation

This section describes what technologies we used to implement the so called “findit microservice platform”.

4.1 Architecture

As explained we are using the Event Sourcing pattern, for this we use Apache Kafka[15] as our immutable log. Kafka is a project started at LinkedIn and now gets further developed at Apache since 2012. It provides our immutable log that contains all write operations of the platform. This means we can also identify whom did what at a given time in the past.

Kafka internally has topics that describe the kind of event. Topics consist out of partitions which store the messages together and ensure the correct order.

As mentioned before we are using Docker to start and connect our microservices. Due to the newest version of docker-compose it's also possible to provide a Docker Swarm[16] (a connected network of servers running a specialized Docker software). This way Docker runs containers on the available hardware without the need to manually deploy containers on each server.

Figure 1 shows the full architecture of the findit platform as it is implemented in its last version. The green container marks the docker network which contains all microservices. Communication between the services is not explicitly plotted in the diagram as the services are all in the same internal docker network allowing them to internally communicate with each other. The Kafka instance and the AWS-S3 instance are separate, as they are external services that need to be called via an interface.

4.2 Services

During the planning of our services we noticed another advantage of microservices. Because everything is separated we could use a different programming language for every microservice. But due to the short time frame of this project - just half a year for two people - we decided to used technologies we were already familiar to which is Node.Js[17] and Java.

[15]kafka.apache.org

[16]docs.docker.com/engine/swarm

[17]nodejs.org/en

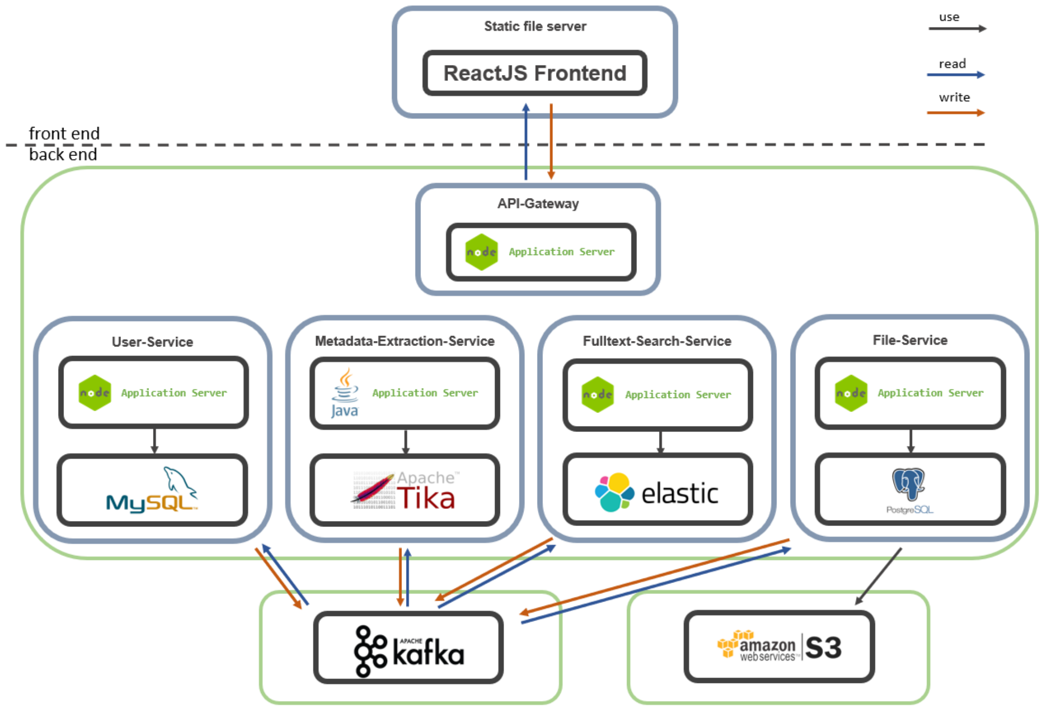

Figure 1: Findit platform overview

4.2.1 User-Service

As stated in the design our user service allows a user to signup to the platform. During signup the service checks if the mail is valid and not already in the database. After that the password gets hashed and a new USER_CREATED event with the e-mail and hashed password is created and distributed using Kafka. As a response the user just gets a success message and no token or cookie as just the event creation was a success and the event might not yet have been consumed.

The service itself listens to that event and updates its local database with the new entry. The local database is used as a view of the current state of the event log. This way we can quickly check if a user exists and what role the user has without needing to go through the complete log.

Now if the newly signed up user logs into the platform the service checks the local database to see if there exists a user with that mail. If the user exists in the database the user object is returned and the password gets compared with the stored salted hash. If this procedure also succeeds a JWT[18] (json web token) with

[18]jwt.io

the user id and role as payload gets signed and sent as a response to the request.

Additionally the user service also has some additional service and management routes that allow an admin to query all users for example or create and manage roles.

All entries in the whole system are using UUIDs[19] (Universally Unique Identifier) as the standard unique id. This is also required to keep everything scalable throughout the platform.

ⓘ Facts

Main-Technologies: Node.JS, TypeScript, Routing Controllers[a], TypeORM[b]
Database: MySQL
public routes: /login, /signup
private routes: /users, /roles

[a]github.com/typestack/routing-controllers
[b]github.com/typeorm/typeorm

4.2.2 File-Service

As mentioned already we are utilizing an external storage provider in order to keeping multiple uploads of files scalable. If we would save them locally we would run into storage scaling problems fairly soon. After looking through the available options we decided to use Amazon S3[20] because we had existing know-how on how to use S3. It is important to mention that the implementation supports different storage providers as everything is just coupled using an interface and can be exchanged easily.

Once the file is uploaded we send out an FILE_UPLOADED event that the file-service also consumes itself to update its local database in the service. The event contains the user id, original file name and the file id which is again a UUID to identify the file in every service.

The file service contains a public interface to query for files of a specific user.

[19]de.wikipedia.org/wiki/Universally_Unique_Identifier
[20]aws.amazon.com/de/s3

ⓘ Facts

Main-Technologies: Node.JS, TypeScript, Routing Controllers[a], TypeORM[b]
Database: MySQL
private routes: /file

[a]github.com/typestack/routing-controllers
[b]github.com/typeorm/typeorm

4.2.3 Metadata Extraction-Service

The metadata extraction service is listening to the FILE_UPLOADED event created by the file-service. Once a new event is consumed, the extraction-service requests the file from the file service and downloads the file. The file is then analyzed with Apache Tika[21] in order to extract all the content and other infos about the file. Tika is a toolkit that allows the extraction of metadata and text from all kinds of file types using a standardized Interface. Once this is done the extracted content and metadata is sent as a METADATA_EXTRACTED event and thus published to the rest of the system.

This is the only service that has neither a public nor an internal interface as everything is managed through consuming and producing events.

ⓘ Facts

Main-Technologies: Java, Spring Boot, Apache Tika

4.2.4 Fulltext Search-Service

The extracted data from our files is currently just stored in the immutable log without a possibility for any user to access it. Therefore the fulltext search service listens to METADATA_EXTRACTED events and saves the extracted content into Elasticsearch[22]. The search-service provides a REST-route that abstracts an internal query to Elasticsearch and returns all results according to the search input.

[21]tika.apache.org
[22]elastic.co/de/products/elasticsearch

To keep this service scalable we used the recommended approach from Elastic which was to create an index for every individual user. This way we can later also adapt the system to shutdown an index once a user logs out of the platform and activate it again on login. Additionally every request to elastic from the interface is always restricted to a specific index/user, so it's impossible to read the content of files of other users.

Facts

Main-Technologies: Node.JS, TypeScript, Routing Controllers[a]
Database: Elasticsearch
private routes: /search

[a]github.com/typestack/routing-controllers

4.2.5 API-Gateway

Our API-gateway is utilizing the docker socket which emits events when containers are started or go down. This way we know that something happens, but not what happens or what the container provides. To get this information we are using docker labels in our compose scripts. The labels contain information about what routes this service provides and if it needs authentication.

Example:

```
user-service:
    build: ./user-service
    depends_on:
      - user-db
    labels:
      - "api_routes=/login;/signup;/users;/roles;/user;/role"
      - "secure=no"
    networks:
      - be-network
```

Now it is possible to parse all information that is needed from the docker socket to route requests to the different services. If a service needs authentication the gateway tries to read the JWT from the request and if it is set the signature is checked for validity as well. If the JWT is valid the payload is parsed and appended as additional headers to the request which is then proxied to the service the request was directed to. By parsing the JWT on the gateway the internal services do not have to check if each request is authorized or not.

ⓘ Facts

Main-Technologies: Node.JS, TypeScript

4.3 Frontend

Probably the most important aspect of any system is the frontend as it represents and grants access to all the functionality the microservice architecture system provides. In our case we could not allocate enough resources to implement a fully featured frontend as the focus was to get the microservice architecture to work.

The frontend uses React[23] in order to build a singlepage application for the web that accesses the functionality of findit via the previously explained gateway API. Users can log in, upload files and search files they already uploaded by fulltext search.

In order to decrease the time spent on the website design we are utilizing the Material-UI[24] for React.

The deployment is done via a project called Create-React-App[25] which is a CLI tool that lets users create React applications with one command and also build the application very easily via the CLI. No setup time is needed to get the whole build and deployment toolchain up and running.

ⓘ Facts

Main-Technologies: React.JS, JavaScript(ES6), Material-UI Build-CLI: Create-React-App

5 Setup Manual

This section contains instructions to setup and launch the system.

[23]https://reactjs.org/

[24]https://material-ui-next.com

[25]https://github.com/facebook/create-react-app

Requirements

CPU: i7 4th Gen
RAM: $\tilde{8}$GB
HDD: 20GB free memory
OS: Linux/Windows/OSX
Software: Node 8.9.4 LTS, Docker, docker-compose

To start the system:

- Fill out docker-compose with AWS secrets and db users and passwords
- Setup docker-machine by starting Docker Toolbox.
- Navigate to docker-compose script
- Run docker-compose build && docker-compose up

To shutdown the system:

- Run docker-compose down
- Run docker-machine stop

Useful commands:

- Show logs of a specific service: docker-compose logs [service-name]
- Restart specific service: docker-compose restart [service-name]
- Launch another instance of a service: docker-compose scale[service-name]=[instance-count]

5.1 docker-compose

```
version: '3'

services:
  zookeeper:
    image: bitnami/zookeeper:latest
    ports:
      - "2181:2181"
    networks:
      - be-network
    environment:
      - ALLOW_ANONYMOUS_LOGIN=yes

  kafka:
    image: bitnami/kafka:latest
    ports:
      - "9092"
    depends_on:
      - zookeeper
    networks:
      - be-network
    environment:
      - KAFKA_ZOOKEEPER_CONNECT=zookeeper:2181
      - ALLOW_PLAINTEXT_LISTENER=yes

  elasticsearch:
    image: bitnami/elasticsearch:6-master
    ports:
      - "9200:9200"
      - "9300:9300"
    volumes:
      - elastic_data:/bitnami
    networks:
      - be-network

  user-db:
    image: mysql:8.0
    volumes:
       - user_data:/var/lib/mysql
    networks:
      - be-network
    environment:
      MYSQL_ROOT_PASSWORD: **********
      MYSQL_USER: **********
      MYSQL_PASSWORD: **********
      MYSQL_DATABASE: findit

  file-db:
    image: mysql:8.0
    environment:
      MYSQL_ROOT_PASSWORD: **********
      MYSQL_USER: **********
      MYSQL_PASSWORD: **********
      MYSQL_DATABASE: fileservice
    networks:
      - be-network
    volumes:
      - file_data:/var/lib/mysql

  file-service:
    build: ./file-service
    depends_on:
      - file-db
      - kafka
    labels:
      - "api_routes=/file"
      - "secure=yes"
    networks:
      - be-network
    environment:
      HTTP_HOST: 8081
      KAFKA_HOST: zaunerserver.ddns.net:49312
      DB_HOST: file-db
      DB_PORT: 3306
      DB_NAME: fileservice
      DB_USER: **********
      DB_PASSWORD: **********
      AWS_ACCESS_KEY: **********
      AWS_SECRET_ACCESS_KEY: **********
      AWS_BUCKET: **********
      AWS_REGION: **********

  user-service:
    build: ./user-service
    depends_on:
      - user-db
      - kafka
    labels:
      - "api_routes=/login;/signup;/users;/roles;/user;/role"
      - "secure=no"
    networks:
      - be-network
    environment:
      HTTP_HOST: 8081
      KAFKA_HOST: zaunerserver.ddns.net:49312
      JWT_SECRET: **********
      DB_HOST: user-db
      DB_PORT: 3306
      DB_NAME: findit
      DB_USER: **********
      DB_PASSWORD: **********

  fulltext-service:
    build: ./fulltext-service
    depends_on:
      - kafka
      - elasticsearch
    labels:
      - "api_routes=/search"
      - "secure=yes"
    networks:
      - be-network
    environment:
      HTTP_HOST: 8081
      KAFKA_HOST: zaunerserver.ddns.net:49312
      ELASTIC_URL: elasticsearch:9200
      ELASTIC_LOG_LEVEL: debug

  metadata-extraction-service:
    build: ./metadata-extraction-service

  api-gateway:
    build: ./api-gateway
    ports:
      - "8080:8080"
    environment:
      HTTP_HOST: 8080
      JWT_SECRET: **********
    networks:
      - be-network
    volumes:
      - /var/run/docker.sock:/var/run/docker.sock

volumes:
  file_data:
  user_data:
  elastic_data:

networks:
  be-network:
```

6 Sample request

Figure 2 shows how a file upload request from the findit frontend application looks like.

First the frontend sends the file to the findit backend which upfront leads directly to the Api-Gateway. The gateway checks if the request is authorized by checking the auth token that has to be sent with every api reqeust. Once the token is verified the gateway proxies the request to the File-Service.

The File-Service then uploads the file to the currently used storage engine and waits for the confirmation of the storage engine if the upload was successful. Once the upload is confirmed to being successful, the File-Service simultaneously sends a “FILE_UPLOADED” event as well as returning a success message to the HTTP-request so the frontend can indicate upload success to its user.

Once the “FILE_UPLOADED” event has been sent there are two consumers in this case. The first consumer is the File-Service itself which connects the uploaded file to the user in order to being able to later connect the file’s id to a specific user. The second consumer is the Metadata-Extraction-Service which tries to extract as much data from the provided file as possible. The extraction service can access the file through the “FILE_UPLOADED” event data. Once the extraction is complete, the service then sends a “METADATA_EXTRACTED” event.

The “METADATA_EXTRACTED” event is then consumed by the Fulltext-Service which saves the data that is contained in the extraction event into an elastic search cluster which makes the data searchable.

The user only gets notified once the file has been uploaded but all other actions that were described are happening in the system internally and asynchronously. Events are only consumed once the consuming services have enough resources to process events. If events cannot be consumed immediately they stay in the log marked as unconsumed. Once a service consumes the event later on, it can mark the event as consumed, notifying other instances of the same service that the event has been processed.

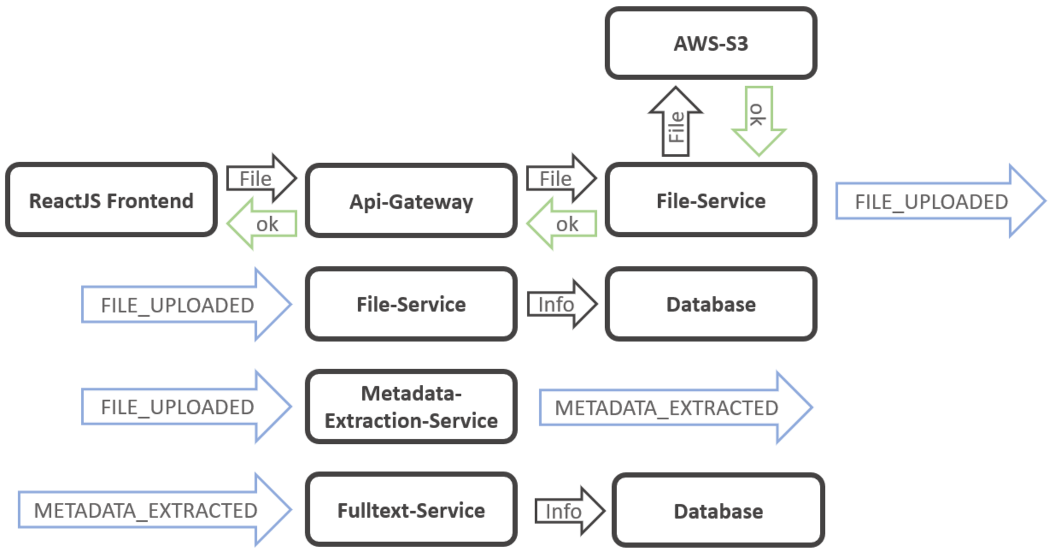

Figure 2: Sample file upload request iterating the system

7 Challenges

The biggest challenges in this project were to combine all the used technologies. We had to read through dozens of documentations to understand why certain aspects of the system were not working as intended. Apache Kafka is the hardest and most complex part in this system it just stores and distributes events but in order to sync services and commit events the right way a lot of trial and error was necessary including setting data types to raw bytes and converting them to actual data types afterwards just to name one problem.

Additionally all the docker scripts and images were though to setup as we never created any images before by ourselves. For this project everything had to be "dockerized" in order to work so this was definitely a challenge.

As shown in Figure 1 the Kafka instance is an external service. We tried to include the full Kafka service into our docker-compose file but it just would not work at all. There is an internal problem with resolving internal network addresses and our usecase is so specific that there were no resources we could use to solve this problem. Our solution to the problem is to have the Kafka service as external service that all the microservices can connect to. In the future the Kafka service should be included in the internal docker network as well.

8 Future Work

This project could be further extended to include a real metadata service that contains general metadata like tags, locations, authors and filenames. Such a service would probably use a graph database like neo4j as storage engine as this would allow to find out connections between individual documents.

Furthermore a mail- and log service should be implemented to send emails to the user on signups and other occasions. The log service should be responsible to collect all log data from all services and to provide an easy to search interface for it. For this service I would again recommend an Elasticsearch database connected to an Kibana instance. Maybe even with Logstash to complete the ELK-stack. But this is maybe viable for the next project.

Before putting this system in production some versioning of every microservice should be implemented. This should just be done globally for one microservices to prefix all routes with vX (e.g. v1). That would enable easy breaking changes on one microservice without breaking existing website or API functionality.

9 Conclusions

Our conclusion for this project is that its just too big for a semester project for two people. We had to rush a lot of the implementations and couldn't set up automatic API documentation generation and other important aspects that every microservice platform should use. But we felt we had to be able to show something in order to get a good grade. If we would do such a project again we would plan everything more carefully at the beginning by spending multiple weeks for that step alone.

Nevertheless this was the biggest project ever done by us both and it really shows that microservices are one possible future of future architectures. The ideas definitely are here to stay as also server-less backends are splitting up responsibility for a specific domain entity to different functions.

Event Sourcing is definitely a good way to solve data integrity problems in microservice scenarios. Because the events also serve as a audit log of every action done it's easy to find out bugs and solve them.

We want to say we are proud at ourselves as we initially planed that this should be more of an experiment and not a usable product but we managed to provide a fully working platform for multiple users.