Advanced Software Engineering
FINAL REPORT

Team number:	0203

Team member 1	
Name:	Alexander Grentner
Student ID:	11743246
E-mail address:	a11743246@unet.univie.ac.at

Team member 2	
Name:	David Kreisler
Student ID:	01569177
E-mail address:	a01569177@unet.univie.ac.at

Team member 3	
Name:	Fabian Schmon
Student ID:	01568351
E-mail address:	a01568351@unet.univie.ac.at

Team member 4	
Name:	Victoria Zeillinger
Student ID:	11809914
E-mail address:	a11809914@unet.univie.ac.at

Team member 5	
Name:	Michal Žák
Student ID:	11922222
E-mail address:	a11922222@unet.univie.ac.at

1. Final Design

Sadly our deploy failed (build and tests pass) on the day of the deadline when we wanted to record the video, and we couldn't fix it in time. We tried to document our development process as good as possible in this document and we described what each service can do. Everything was tested in the UI and additionally tested with postman as an End-to-End test.

1.1. Design Approach and Overview

After a careful evaluation of the problem domain we decided that designing a web app would provide the best possible solution to the problem domain. The main contributing factors to choosing a web app were:

1. The app needs to be used by multiple different users
2. It is not reasonable to assume that all users of the app will want to use the app from one machine and hence we need to enable access for multiple machines at different locations
3. The computational complexity of the app is not high enough to warrant a native app
4. The regular operation of the app requires an internet connection anyway (database access)

As data transfer endpoints will need to be defined when the app will be deployed using a microservice architecture, a technology for these also needs to be considered. Here the option was fairly clear, with us deciding to use a REST API. A REST API is the de facto standard of information exchange on the internet and we could not identify any reason to use a more complex and specialized solution.

To design our solution, we initially set out to use a waterfall software design model. As our team comes from the medical informatics sphere, where the waterfall model is commonly used due to its high structurization, we were all familiar with it and thought it would aid us most in creating a robust solution. However, in the end, we decided to adapt the waterfall model to be more similar to an agile approach. This allowed us to implement and quickly test our ideas in the code, while we are designing the diagrams, which should hopefully lead to a better solution.

The first step conducted while designing the app was to capture the system and software requirements. This was done by carefully analysing the provided domain description, asking stakeholders (tutors) about the problem domain, and also formalising certain assumptions.

We quickly identified several user roles, namely the administrator, responsible for maintaining the app, the organizer, responsible for creating new events and managing their existing events, the attendee, responsible for registering for events and bookmarking events and the unregistered user, who can register as either an

organizer or an attendee. These roles were afterward used as the basis for identifying the different use cases of the app.

After figuring out our rough requirements, some crud domain models were created. The first draft of the domain models consisted of seven domains: User, Organizer, Attendee, Admin, Event, Notification, and Administration. For each of these, we further identified several subdomains and defined the relations between the different domains.

These domains were also subjected to iterations. The next major change redefined the identified subdomains to be more in line with the SRs provided in the domain description

After gathering our requirements and getting a grasp on the problem domain, we set off to create a first draft of UML diagrams, based on the identified domains and requirements. These diagrams went through many iterations until they arrived at their current stages. During modelling, we focused our efforts on the class diagram, as we found that this would allow us to best represent the functionality of the app. Other UML models were however also used, namely the use case diagram to model our main use cases, a sequence diagram to model the interaction between classes, and the component diagram to visualise the coupling of the app.

During modelling, we stumbled upon several issues where we were unsure of the best design and modelling strategy. Hence we decided to diverge from the classic waterfall model and we started the implementation step earlier than anticipated. This allowed us to better familiarise ourselves with the used frameworks and to design models that more closely correspond to the best practices of the used frameworks.

The models created after this stage were later adapted to reflect DDD and formalised into the 4+1 View model.

We then adapted our monolithic design approach to microservices and therefore updated the 4+1 view model accordingly. We also implemented a layered architecture pattern for our microservices .

During the implementation and the model creation, we made sure to have frequent discussions about the current design, problems, and ideas for the project. We wanted to ensure that the information flow between team members is high. This also helped ensure that every subdomain is designed consistently and with the same standards.

1.1.1. Assumptions

As the problem domain was very open-ended, certain assumptions had to be made to clearly define the scope and delimit the use cases.

1. In our application, we have decided to omit the ability to create admins. The flow for creating admins cannot be the same as creating organizers and

attendees. This process is however nowhere described in the assignment and further discussion with the stakeholders is necessary to determine the exact needed steps.
As a temporary solution, we have decided to create default admin users, which can be used to demo their functionality.

2. We decided that each account can only have one role assigned to it (attendee, organizer, admin). This means, as the users are authenticated using an email and a password, that each email can be assigned only to one role. Allowing each account to only have one role assigned to it should reduce the complexity of our system and further ease the development of clear boundaries between the different roles on our system.
3. The Search-Service, which should list all saved events and provide filters for searching, were assumed to be independent of the role of the users. So, attendees and organizers can all see every event from every organizer and search with different filters. On the one hand, it leaves less room for errors regarding the role-based-authentication but on the other hand, we decide that in terms of usability, the organizers should also see events from other organizers to adapt their schedule if needed.
4. We further found that “Attendee” (as in the user) and “Attending” (as in attending an event) could confuse. To mitigate this, we have decided to use the verb register instead of attend, but stick to Attendance and Attendees.
5. In the notification domain we have decided that notifications will be sent out using emails. Email is a standard and to this date very common technology and as we require each user to register with their email, we don’t need to query for more information using this solution. Further developing a service capable of sending emails is common practice and allows us to use plenty of existing solutions.
6. Due to the logical conclusion, we decided that one organizer can only have one event inventory. The event inventory is for managing and viewing the own events, so two or more would not provide a benefit.
7. We made multiple assumptions regarding the feedback:
    a. A user can give multiple feedbacks on the same event
    b. Feedback can be identical in content, even one user giving two feedbacks for the same event
8. We assume that tags are predefined and therefore can not be created by any user or any role. This should be sufficient concerning usability and counteract problems that custom tags would possibly cause.
9. The search service can only search for events by one criteria at once. To avoid complicated queries or path variables we decided to simplify the search. Users can still look up an event based on every criterion but just need to check each criterion at once.

1.1.2. Design Decisions

1.1.2.1 Project-wide Design Decisions

Use Cases (Team)

The project use cases were first defined by functionalities and then adapted to fit the functional Scope Requirements. The tagging service was added to the use cases in the adaptation process, as well as the search service for events. At first, we modelled three use case diagrams, one for each role, but then decided to make one big use case model, with all three user types in it, to also show the connections between the actions of each user.

Designations (Team)

We found that “Attendee” (as in the user) and “attending” (as in attending an event) could confuse. To mitigate this, we have decided to use the verb register instead of attend, but stick to Attendance and Attendees.(See Assumption 4) Further we then used “sign in” as the term for a user registration.

Predefined Tags (Team)

We have opted to predefined tags to facilitate efficient processing searches and recommendation services. This approach allows users to utilize pre-existing tags, thereby enabling the recommender service to more efficiently process and aggregate the tags compared to customized tags. The use case that users can create their own tags did not arise from the requirements and is not relevant from the perspective of meaningful processing.

Unique Identifiers (Team)

In our microservice architecture, each service stores data separately and will require its own dedicated database. To identify data objects, we decided to use a String datatype-based ID. For generating primary keys, we found the UUID string generator to be much more suitable for entities. This is especially important in our case as it generates a much larger number of possible random strings, which helps to limit the risk of duplication.

Duplicating Data / Redundant Data (Team)

Some microservices need asynchronous access to the stored data of other microservices to maintain their functionality. The publish/subscribe pattern provides the data to other services and they store them in their own database. Therefore some data, like events or users, are stored in two microservices and are redundant. We needed to implement it like that, to ensure that even if one of the services is down, the other service can still function on its own.

H2 Persistence Database (Team)

As we used the H2 in memory database before and had already implemented some logic, we decided to keep using the H2 database and persisted locally, even though it has some restrictions. But it works well enough for our system.

Pub/Sub (Team)

For communication between services we facilitated the Publisher Subscriber pattern, by using RabbitMQ. This approach enabled loose coupling between the services and possible improvement of scalability within the application in general. Especially the communication within services is sufficient by using the publisher subscriber pattern. In the application microservices can be used as publishers and subscribers, depending on the usage and the required communication between other services, which allows better scalability and communication between microservices.

We identified following advantages:

1. Loose coupling

   In our application the Publisher-Subscriber-Pattern allows loose coupling, which reduces the direct knowledge of the Subscriber and the Publisher, which is not needed for communication.

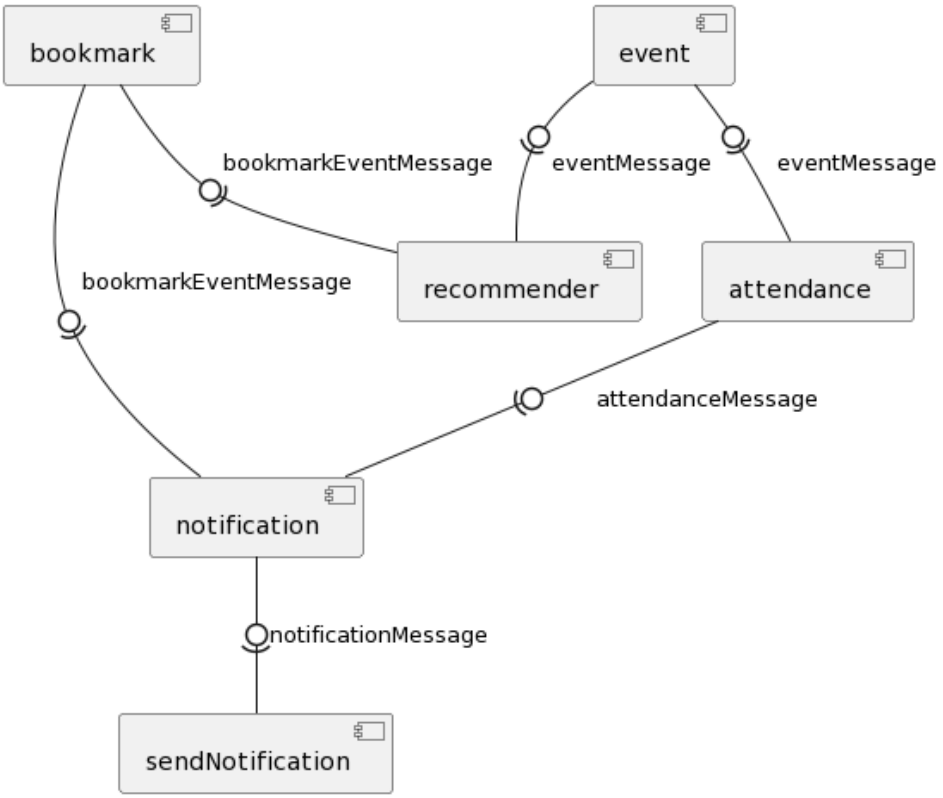
2. improved scalability and flexibility

   The communication can be easily extended by adding more subscribers, since direct communication from both, subscriber and publisher is not required.

3. asynchronous communication

   Publishers can publish communication to subscribers without the necessity of a response. On the other hand subscribers can process messages as soon as their publisher sends messages and don't have to request them actively.

The following diagram shows services which communicate using publisher subscribe.



API Gateway (Team)

We initially planned to omit an API Gateway microservice and instead allow the Frontend microservice to directly fetch data from our backend. However, after careful reconsideration and consultation with our supervisor, we have decided to separate the API Gateway as its own microservice.

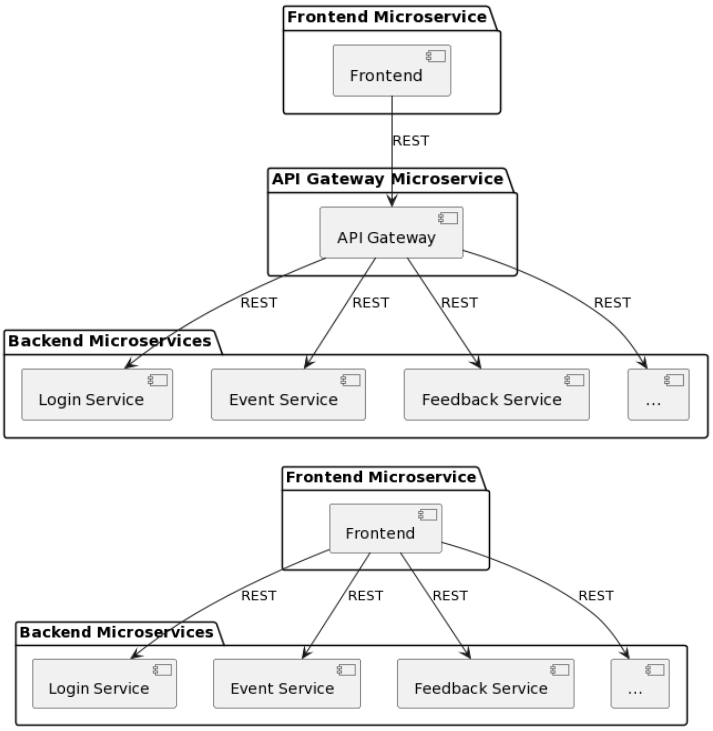
We have identified the following advantages of this decision:

1. Improved abstraction: The API Gateway serves as a single network address, acting as a facade for our system. As a result, the frontend only needs to be aware of the API Gateway's address, simplifying the communication process.
2. Streamlined Docker setup: By utilising an API Gateway, we only need to expose the API Gateway itself outside of our Docker setup. This enables us to shield all other services behind a network wall, enhancing security and reducing potential vulnerabilities.
3. Enhanced security configuration: Since the API Gateway is a trusted application, unlike the frontend, we can rely on it to handle our JWT authentication. Without an API Gateway, we would have to either find a way to trust the frontend with authentication or burden the microservices with the authentication process, leading to increased complexity and maintenance.

Nevertheless, we have also recognized the following disadvantages:

1. Increased complexity: Introducing an API Gateway adds an extra microservice that requires development and ongoing maintenance. This complexity should be carefully managed to ensure smooth operation.
2. Potential for errors: Introducing an additional layer in the system introduces a new potential source of mistakes. It is essential to pay close attention to the implementation and thoroughly test the API Gateway to minimise any errors or issues.

By weighing these advantages and disadvantages, we have concluded that implementing an API Gateway as a separate microservice outweighs the drawbacks and aligns better with our system's overall architecture and security requirements.



Frontend and User Interface (Team)

The user interface was implemented as a service named frontend by using angular. The web pages themselves are written in html, the logic is written in typescript. We used these technologies because the team was already familiar with them and had experience in other projects prior to this, using angular. We decided to keep the UI very simple and divided the scope into register and login and into the rest of the system. The Web-page components are in the app-layer of the frontend.

Components:

- The user registration and the user login are two form fields which only route the users to the system, if the authentication works. Both forms have input fields for the required parameters and a button to submit the input. When a user is already correctly registered, they can login and are automatically navigated to the homepage of the app.
- The app has a navigation bar on top with different buttons to navigate to the sites.
    - The homepage has an introduction text and additional buttons to all other pages.
    - The Event Inventory page provides the event inventory of the logged in organiser. All their events are listed and have buttons for updating an event and deleting an event. Further for every event it's shown how many attendees have already registered for it. If the user is not an organizer, the button is unclickable.
    - The add event page provides an input form, where a new event can be created. An organizer can type in all the parameters and then click the button to submit the information. The event will be added to the inventory of this organizer as well as to the Search Service list of all events. If the user is not an organizer, the button is unclickable.
    - The Search Service page provides a list of all events from every organizer and input fields to filter the events by different criteria. Every event has a button to either bookmark an event or register for it, as well as a button to give Feedback for an event.
    - The calendar page provides three export buttons, so an attendee can export the own bookmarked and registered events in three different formats, xml, json and ical. If the user is not an attendee, the Calendar button is unclickable.
    - The analytic report page provides two buttons, which either show an analytic report of all events of an organizer or about the feedback of all events.
    - The maintenance page provides a dashboard to the admins, where they can control which services are running and get logging information. If the user is not an admin, the button is unclickable.
    - The logout button navigates the user to the login page.
- The buttons in the navigation bar are only clickable, if the user has the matching role. Event Inventory, Add Event and analytic report are only clickable if the user has the role "organizer". Calendar is only clickable if the user has the role "attendee". Maintenance is only clickable if the user has the role "admin". All other pages can be accessed if the user is logged in.

The service layer of the frontend makes all HTTP-calls to the API Gateway. Other Layers include the dtos which have the data-models and global which has the

enumerations. Those parameters are used by the app components and the services so therefore, they are extra layers.

1.1.2.2 Microservice Specific Design Decisions

Event (Inventory) Microservice (Victoria Zeillinger/11809914)

The Event Inventory Service should provide the organizers with the management of their events. Therefore the whole Event Inventory Service is included or merged in the Event Service, because the functions and features are the same and the Event Inventory alone has no other functions. For the management, the microservice needs to store the event information and the organizer ID to fulfil all the CRUD methods. All that can be included in the Event Microservice and should not be outsourced into a separate microservice. Because otherwise the events will be redundantly stored. In other words, the Event Inventory Microservice is named Event Service Therefore also the Event model and database lies in the Event Microservice. Because of design decisions we made in the project planning phase, the whole microservice which includes the event inventory is called “Event Service” and corresponds to the SR4. Everytime an event is created, updated or deleted it will be published for the other services to fetch.

Search Service (Victoria Zeillinger/11809914)
The Search Service should provide the users with a filter for searching/browsing Events. Therefore it saves the events, which are added through the event service, in its own data storage redundantly. With a publish-subscribe pattern, everytime an organiser creates, updates or deletes an event it will be changed accordingly in the search-service database. So the users can access all the events, even if the event service is down. The endpoint of the search service only provides get methods to the api-gateway, so therefore no users can change events in this service. Users can search for events by different criteria: Event Name, Date, Capacity, Description, Type. But only one criteria at a time can be filtered, so that there are no nested queries and therefore it’s less error prone.

Bookmark and Tagging (Alexander Grentner/11743246)
We decided to split the Bookmark and Tagging use case in different microservices. The data is not dependent on each use case and can be used separately. This enables the usage of the booking and tagging separately as well as abstracting the concerns of the application. From a user point of view also a much higher usage of bookmarking is expected. Therefore to ensure a stable connection within the microservices which use the bookmark service on higher frequency is of higher priority.

CalendarService (Alexander Grentner/11743246)

The calendar export service operates by converting existing data fetched from other services into an exportable calendar format, rather than persistently storing the data.
The calendar uses data from services which store bookmarked events. We used the approach as a REST service since better aggregation and direct knowledge of other services is not necessarily required.
This approach ensures that data is not retained beyond the export process. The calendar service employs the factory pattern to generate the desired export calendar type, which is then returned to the frontend.
To initiate the export process, the user creates a user calendar, providing the necessary data, and generates a calendar object. This object is crucial for the subsequent export within the service. The service defines the available types, and based on these definitions, the calendar required by the user is returned. For user convenience, three export options are available: JSON, XML, and standard ICal. These options produce the respective exports as string outputs sent to the frontend. These formats were chosen due to their widespread usage in this application domain.

Attendance Service (Fabian Schmon/01568351)

The Attendance Service is a Java service designed around the domain model consisting of Attendees and EventCapacity entities. It uses the IAttendanceRepository to abstract away data access, allowing independence from specific database technologies. The service also uses dependency injection for the repository and a publisher object, facilitating testing and configuration flexibility. This service manages operations like registration, deregistration, and capacity management for event attendance. The register and deregister methods update the attendees list and use a publisher to notify about the changes. The service also provides methods to add, update, and delete event capacities. Additionally, it offers retrieval of attendee count for a particular event and a list of events a specific user is attending. The data persisted includes an Attendees entity for each event, storing the event's ID, capacity, and a list of attendee IDs. Changes in capacity and attendees are saved as part of these operations.

Recommender Service (Fabian Schmon/01568351)

The Recommender Service, implemented in Java, uses domain models like EventType and UserInterest, along with repositories IRecommenderRepository and IEventTypeRepository, to provide event recommendations based on user interests. It employs a publisher for recommendation notifications and a filter to streamline user interests. The service includes methods for adding, updating, and removing event types and users' interests. It generates a list of event recommendations for a specific attendee ID based on their interest in certain event types, essentially categorizing the events bookmarked by the user. Upon registering an interest or removing it, the

service updates the count of interests for each event type accordingly. The filter class assists in narrowing down user interests based on a predetermined threshold, producing a recommendation list of users exceeding this interest threshold. The service maintains persistence for event types and user interests with attributes like event ID, event type, and counts of user interests for various event categories. The decision to recommend based on user interests and event categories they've bookmarked allows the service to provide personalised recommendations, thus improving user engagement and satisfaction.

Login Service (Žák/ 11922222)

The biggest decision in the login service was the use of the authentication type. In our implementation, we have decided to opt for JWT. Considered alternatives were:

- a custom solution
- oauth

JWTs biggest advantage is its simplicity, compared to the other solutions. Developing our custom solution would further introduce a lot of security issues. Oauth would provide a more modern and more secure stack, compared to JWT, however we have decided against it as we felt it was unnecessarily complicated.

Notification Service (Žák/ 11922222)

Notification type

We have decided to use **emails to provide the notifications functionality** of the system (this is hinted at in the assignment document but not explicit). Alternatives would be:

- creating a notification pop up in the frontend of the system

The main advantage of sending an email compared to creating a notification pop up is that this solution is decoupled from the frontend, it uses an existing technology and through the registration, our solution already has access to the emails of registered users as the emails are given during the registration.

Email provider

We have further decided to **send emails via gmail**. Alternatives to this would be:

- setting up an email server ourselves
- using a different provider then gmail

Setting up an email server ourselves is fairly complicated and goes a bit out of scope of our project. Further, we feel it would be a bit unnecessary to set up an email server, if other, already set up, email servers exist.

We have chosen to use gmail as opposed to other competitors, as it is widely supported and well established. It further allows to simply export application

credentials which can be used for the project. Unfortunately this also means that a google account is needed to use our app. We tried to make the process of setting credentials for our app as simple as possible. For the context of this assignment, we further provide credentials created by us.

Service architecture

From a more architectural point of view, the decision was made to **split the notification micro services into two**. A service responsible for scheduling and creating notifications, the notification service, and a service responsible for sending notification, the send notification service. This decision was made, as other services (Recommender) require sending notification as well.

Feedback Service (David Kreisler/ 01569177)

The feedback service allows attendees and organizers to give feedback for an event. Feedback consists of the numerical grades from one to five and one open text field. The numerical grades are specified for different categories, namely food, location and overall. Feedback can only be created for an existing event. No constraints regarding the possibility to give feedback are implemented.

Analytic Report Service (David Kreisler/ 01569177)

There are two types of analytic reports, one summarising the feedback given on a specific feedback and one putting attendings and capacity in relation. The report regarding feedback can be seen by all users, the one about attendings and capacity can only be accessed by users who have the role of organiser. This data is fetched directly by the Analytic report service, since it does not have to be updated on a regular basis but will be fetched when a user accesses it.

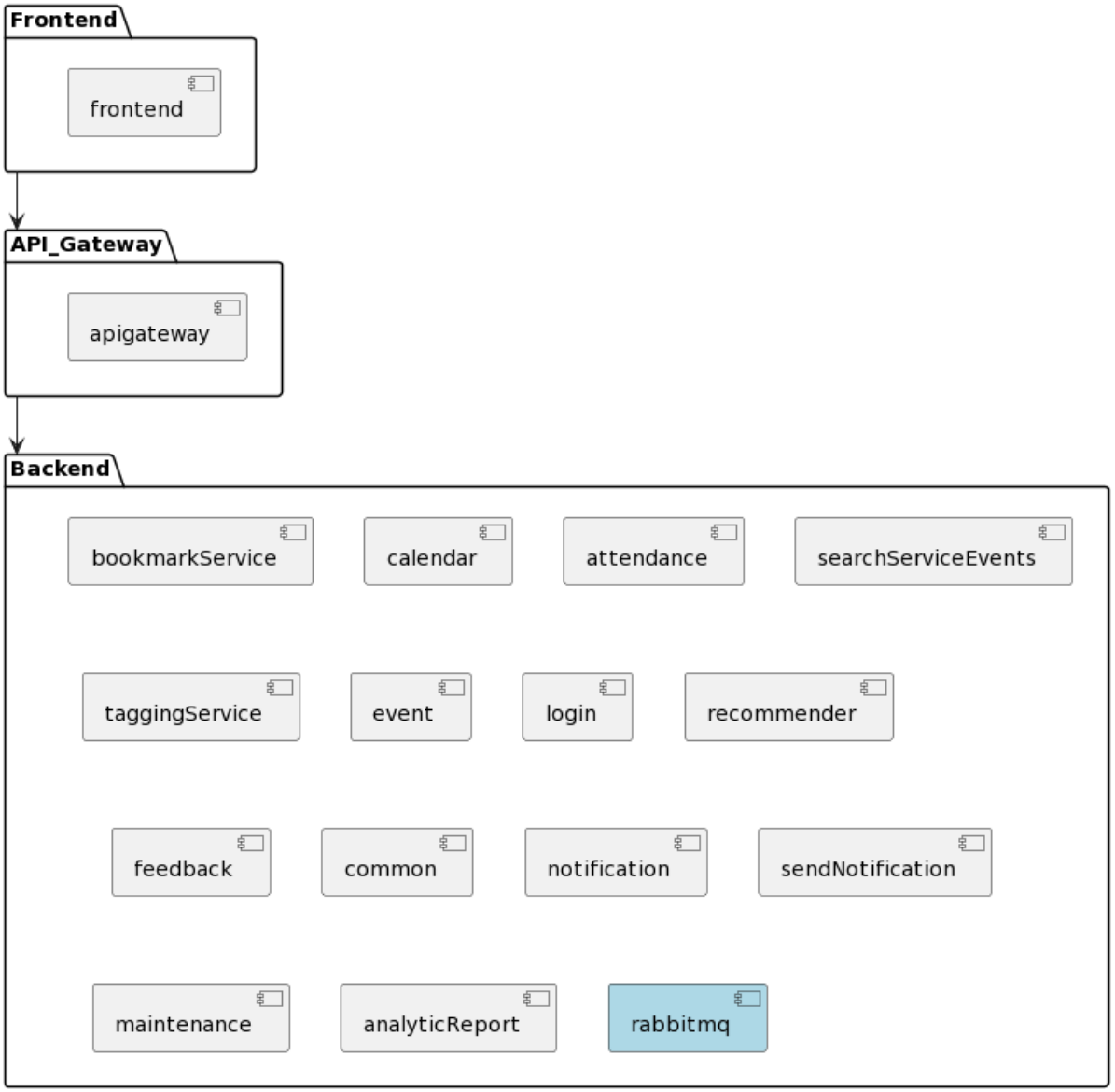
Maintenance Service (David Kreisler/01569177)

The maintenance service is supposed to give an overview of the system’s components and their availability. It is available only for the role of admin. It knows of all other services in the system and accesses their stati via Rest, since this will not happen on a regular basis. Still it is separated from the api gateway for future improvement and addition of functionality like acquiring logs.

1.1.3. Birds-eye view

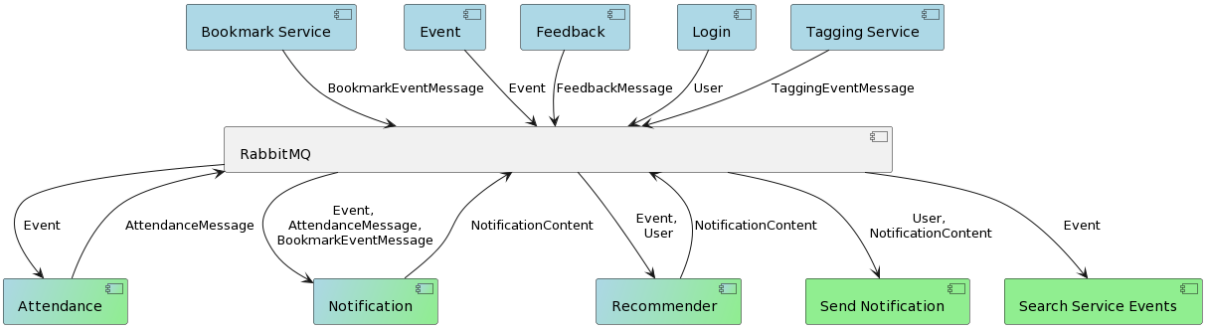
The following diagrams in this chapter show an overall context of the application by abstracting on different levels. The goal is to gain an overall perspective of the application.

Package Diagram System

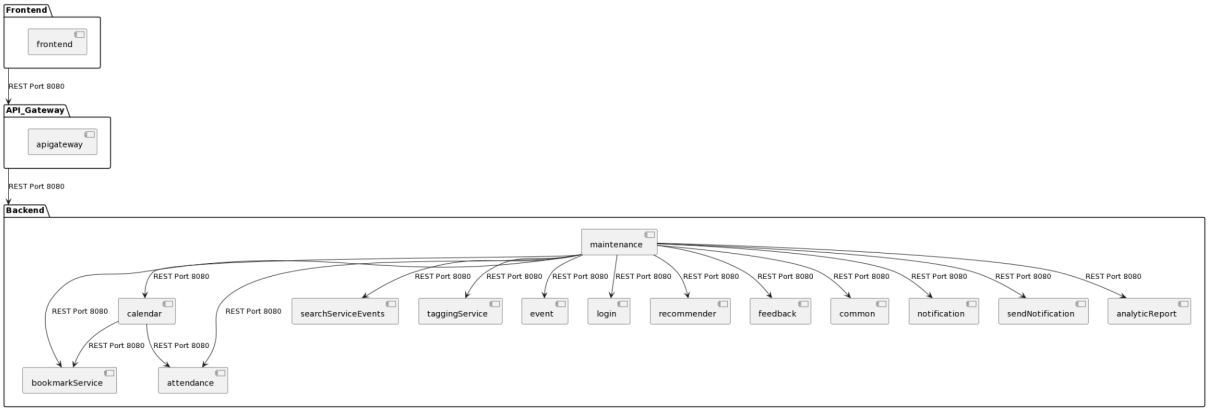


The package diagram above illustrates the implemented domains as packages, the dependencies of the API-Gateway, Frontend and the Backend with its microservices. The backend shows which services are provided within the application and its backend. The services in the backend can be seen as its own entities with its own environment. Further dependencies are abstracted in this version.

Component Diagram Communication

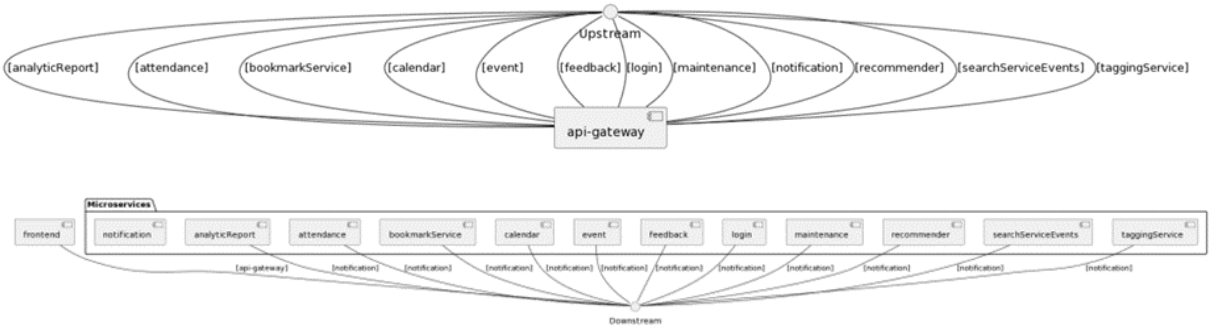


The diagram above shows the communication between services using the Publish Subscribe approach, facilitated with RabbitMQ. RabbitMQ is in this case an exchange which can be either a Subscriber or a Publisher. The services BookmarkService, Event, Feedback, Login and TaggingService provide messages at the RabbitMQ-Exchange. The services Attendance, Notification, Recommender, SendNotifiation and SearchServiceEvents subscribe at the RabbitMQ exchange to request messages provided. Additionally these services can also publish back on the rabbitMQ exchange.



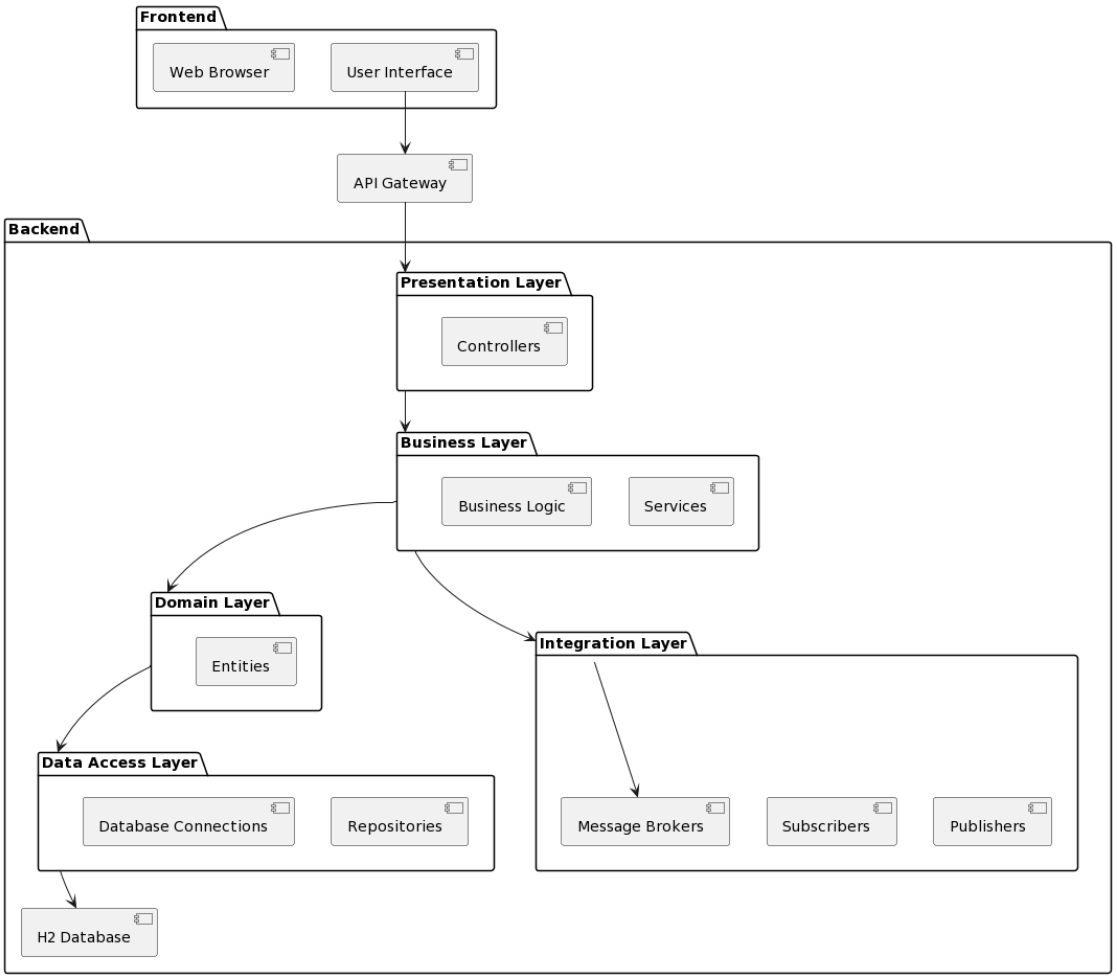
The plot above shows the REST based communication between the Microservices. Most of the microservices do not communicate via REST and are therefore abstracted in this model, but shown in the model above which shows the communication via RabbitMQ.

Bounded Context Map



This bounded context map shows the important upstream/downstream communication between the microservices and the api-gateway

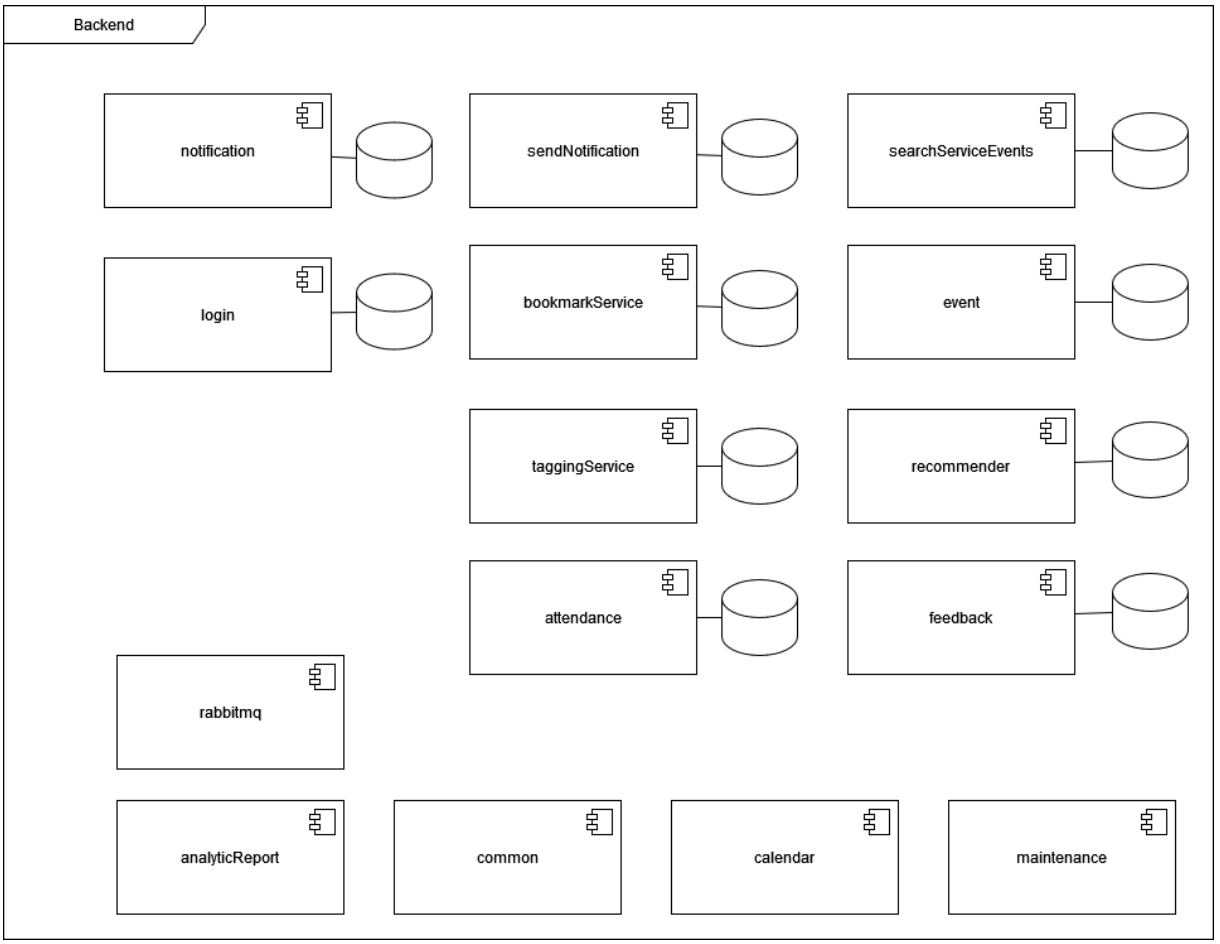
Layered Architecture Pattern



The diagram above shows the layers architecture which is used in this project. It is an overview and describes the architecture in the Microservices in general. Some of the Microservices do not have an integration Layer, because they do not need

asynchronous communication and others don't have a Data-Access-Layer, because they don't need to store data.

Local Databases



The above diagram shows the packages of the backend and their local h2 databases. As one can see the following packages use a database:

- notification
- login
- sendNotification
- searchServiceEvents
- bookmarkService
- event
- taggingService
- recommender
- attendance
- feedback

rabbitmq is used as message broke for the PubSub and common is used to transfer objects via the network. AnalyticReport, calendar and maintenance contain services

that do not store any data. Persistence is achieved by extracting the h2 database from the docker containers.

1.2. Architectural Design Decisions (ADDs)

1.3.1 RabbitMQ

Problem Statement

Our microservices need efficient, reliable, and scalable inter-service communication. REST API calls currently used present several challenges, including increased latency and tightly coupled dependencies.

Decision Drivers

- Decoupling: Minimising dependencies between microservices.
- Reliability: Ensuring no message loss.
- Scalability & Performance: Handling increased demand with minimal latency.
- Efficiency: Optimal resource utilisation.

Alternatives

- RabbitMQ: A robust, reliable message broker with support for publisher/subscriber pattern.
- Apache Kafka: A distributed streaming platform known for high-throughput and fault-tolerance.
- Google Pub/Sub: A simple, reliable, scalable messaging and streaming service.

Recommendation

RabbitMQ was chosen for its robustness, scalability, and publisher/subscriber support, which fits well with our event-driven services.

Decision Outcomes

Status: Decided

Chosen Alternative: RabbitMQ

Justification: RabbitMQ provides the needed reliability, decoupling, and performance.

Consequences: Improved system reliability and performance, with the trade-off of managing RabbitMQ infrastructure.

Assumptions

The system load will increase.

REST API communication can continue to be used for frontend communication.

1.3.2 H2 Database

In context of:

Some microservices need to store data.

The corresponding microservices are: login service, notification service, send-notification service, bookmark service, tagging service, attendance service and recommender service, feedback service. The included.

Corresponding Use Cases are (IDs - see 4+1 Scenario View): 1, 2, 3, 4, 5, 6, 9, 10, 11, 12, 13

facing:

Data should be stored from multiple microservices but each service should have their own database. It should be a persisted database.

we decided for:

As we used the H2 in memory database before and had already implemented some logic, we decided to just persist the H2 database.

to achieve:

Store the data locally and be able to fetch it again, even if the system, or single microservices, have a downtime.

accepting:

H2 is a relational database and sql based so it’s production ready. In comparison to other databases H2 has technical limitations. The whole database setup is not highly sophisticated, but it works well for the project scope.

1.3. Major Changes Compared to DESIGN

Common

DESIGN	FINAL
Monolith Architecture	Microservices Architecture Grund: weil ihr es wolltet

DESIGN	FINAL
Domains communicated through their Services in the monolithic architecture	Microservices communicate via a Publish/Subscribe Pattern (see more in Design Decisions) Microservices which need synchronous communication use REST-Calls to other Microservices. (see Calendar, Analytic Report)
“User Interface” was a Postman Script directly to Endpoints	An API Gateway is interposed between the Frontend (UI) and the Backend and forwards the requests. (see more Design Decisions: API-Gateway)
H2 in memory database	H2 persistence database (see design decisions)

Event Service (Victoria Zeillinger)

DESIGN	FINAL
Event Service and Event Inventory Service two different Microservices	Those two microservices were merged into the Event Service. Argumentation in: Design Decisions Event Service
Event Service saves all events. If updated and deleted then it’s changed in its own database.	When an event is created, updated or deleted a RabbitMQ Publisher provides the event (or change) in a queue.
Data is stored in the domain where it’s created	Duplicated data through Pub/Sub and necessity. (See design decision)

Calendar Service (Alexander Grentner/11743246)

DESIGN	FINAL
generating a Calendar Object from and export with second request	No calendar object is created, the data is fetched from other services via REST, frontend/API gateway sends calendartype to convert in certain type
getting data from an external object to export to calendar type object	fetching data from other services like bookmark and attendance

Bookmark/Tagging Service (Alexander Grentner/11743246)

DESIGN	FINAL
Bookmark and Tagging are combined in one area of responsibility	Microservices Architecture, implemented as a service, split in separate bookmark service and in tagging service

Attendance Service (Fabian Schmon/01568351)

DESIGN	FINAL
Can only return a List of all userIDs that are registered for a specific eventID	Can return a List of all userIDs for a specific eventID and a List of all eventIDs for which a given userID is registered.
Could only handle a mock userID and mock eventID	Can fully communicate with Event and Notification microservices
Only uses basic package structuring	Introduces layered architecture

Recommender Service (Fabian Schmon/01568351)

DESIGN	FINAL
Can only recommend the mock userID.	Introduces the concept of userInterest. Depending on the chosen filter, users will get recommendations for new Events if they surpass the interest threshold which depends on the number of bookmarked events with the same Eventtype.
Could not communicate with other services.	Can fully communicate with Event, Bookmark and Notification microservices.
Only uses basic package structuring	Introduces layered architecture

Login Service (Zak/11922222)

DESIGN	FINAL
User OAuth for authorization/authentication	Use JWT for authorization authentication
	general refactor

Notification Service

DESIGN	FINAL
	general refactor
	introduced layered architecture

SendNotification Service

DESIGN	FINAL
	general refactor
	introduced layered architecture

Maintenance Service

DESIGN	FINAL
	general refactor
	introduced layered architecture
provide uptime and logs	provide availability

Feedback Service

DESIGN	FINAL
	general refactor
	introduced layered architecture

AnalyticReport Service

DESIGN	FINAL

	general refactor
	introduced layered architecture

1.4. Development Stack and Technology Stack

1.4.1. Development Stack

Our application was developed using Java and Spring Boot in a IntelliJ development environment. To simplify data access and enable CRUD operations, we utilized Java JPA Repository, along with customised data querying methods that are explained in detail in the Technology Stack section.

We chose IntelliJ because it is widely used and provides more pre-installed features compared to other IDEs, reducing the need for plugin configurations and environment setup. Gitlab is used for code versioning as it is a well-established version control system. It is also easily integrated with IntelliJ and provides a user-friendly environment in addition to GitBash. Additionally, Gitlab offers a CI/CD development environment, which will be necessary for the release and future testing scalability. Furthermore, we can easily manage our dependencies using Maven.

Our application sends notifications via email and requires an external email service. For the task, we chose to use Gmail for convenience as it is a free service, easy to set up and widely used.

To facilitate containerization, we decided to use Docker, which is the most widely employed tool for this purpose. Depending on the operating system, Docker desktop can be utilised as an optional addition, though it is not mandatory.

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Gmail Email Service	current version provided by Google
Docker	23.0.5

Event Microservice (Victoria Zeillinger)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Search Service Event Microservice (Victoria Zeillinger)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Calendar Microservice (Alexander Grentner/11743246)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0

Docker	23.0.5

Bookmark Service (Alexander Grentner/11743246)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Tagging Service (Alexander Grentner/11743246)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Attendance Service (Fabian Schmon/01568351)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Recommender Service (Fabian Schmon/01568351)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Feedback (David Kreisler/01569177)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Maintenance (David Kreisler/01569177)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

AnalyticReport (David Kreisler/01569177)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Login Microservice (Michal Zak)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

Notification Microservice (Michal Zak)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0

Docker	23.0.5

Send Notification Microservice (Michal Zak)

IntelliJ	Community Ultimate 2022.3
Gitlab	current
Gitlab CI/CD	
Maven Build	4.0.0
Docker	23.0.5

1.4.2. Technology Stack

We chose to use Spring because it is widely supported, well-documented, and established in the Java community. This made it a great choice for creating a long-lasting application. Subsequently, we used the Spring Web and Spring Security Framework which provide packages for HTTP, REST, and JSON as well as Security Features which prevent the most common security breaches within the application. Because of the use of Spring and the requirement of a full stack application REST API is state of the art for this kind of application.

Our technology stack included various dependencies included in Maven. One of the most important packages we used was JPARepository, which enabled us to easily implement CRUD operations in our Repository. Additionally, it allowed us to customise database queries based on the Entities we implemented. We found that the JPA implementation was the most lightweight option when used in combination with our local database, which in our case was the h2 database. The database can be easily used as persistence data storage by reconfiguration. Although the h2 database is not the most sophisticated approach the database can easily be set up and the expected amount of data storage and complexity is enough.

For the CalenderFactory we decided to use the JSONFormatter for easier conversion of data formats within the CalenderService, the JSON Formatter also provides the conversion to other data formats.

We utilized OAuth2 for authentication, as it is considered the standard protocol for authorization. We opted for OAuth2 due to its scalability and widespread adoption. In conjunction with this, we utilized the JWT token service for authorization, which seamlessly integrates within the Spring environment and provides ease of implementation, and is very transparent in generating the token. The development stack for the authentication process using these frameworks is well documented in the community and therefore provides better support during the implementation.

For communication between the microservices and to facilitate the publish subscribe pattern we used RabbitMQ, which provides easy configuration both on the publisher as well as on the subscriber side and therefore allows easy integration in our system. By using already established technology it can be assured that the implementation ist stable and secure for this application and task for each microservice.

Mockito enables easy unit testing and allows more useful Test-Driven-Development within the Iterations. Therefore it is possible to adapt test cases easily within the development process which results in better code quality and persistent testing.

For the Frontend we decided to use Angular in combination with TypeScript. Both Frameworks are scalable for a bigger microservice environment and allow a quick setup within the IntelliJ IDE. Angular provides the most common design features which are sufficient for our application.

Packagename	**Version**
Spring	3.0.5
JPARepository	3.0.5
H2 database	3.0.5
JSON Formatter	20160212
OAuth2	Version 2
JUnit	3.0.5
JWT	4.4.0
Mockito	4.8.1

Packagename	**Version**
Spring	3.0.5
Spring Web, Spring Security	3.0.5
Typescript/Angular	the latest version
RabbitMQ	3.0.4

Event Microservice (Victoria Zeillinger/11809914)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0
spring-rabbit	3.0.4
h2	2.1.214
spring-security-crypto	6.1.0

Testing:

spring-boot-starter-test	3.1.0
mockito-core	5.3.1

Search Service Microservice (Victoria Zeillinger/11809914)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0
spring-rabbit	3.0.4
h2	2.1.214
spring-security-crypto	6.1.0

Testing:

spring-boot-starter-test	3.1.0
mockito-core	5.3.1

Calendar Service (Alexander Grentner/11743246)

Packagename	Version
Spring	3.0.5
JSON Formatter	20160212
Spring Web, Spring Security	3.0.5

Testing

JUnit	3.0.5
Mockito	4.8.1

Bookmark Service (Alexander Grentner/11743246)

Packagename	Version
Spring	3.0.5
JPARepository	3.0.5
H2 database	3.0.5

Testing

Mockito	4.8.1
JUnit	3.0.5

Tagging Service (Alexander Grentner/11743246)

Packagename	Version
Spring	3.0.5
JPARepository	3.0.5
H2 database	3.0.5
JUnit	3.0.5

Testing:

Mockito	4.8.1
spring-boot-starter-test	3.1.0

Attendance Service (Fabian Schmon/01568351)

Packagename	Version
Spring	3.0.5
JPARepository	3.0.5
H2 database	3.0.5
JUnit	3.0.5

Testing:

Mockito	4.8.1
spring-boot-starter-test	3.1.0

Recommender Service (Fabian Schmon/01568351)

Packagename	Version
Spring	3.0.5
JPARepository	3.0.5
H2 database	3.0.5
JUnit	3.0.5

Testing:

Mockito	4.8.1
spring-boot-starter-test	3.1.0

Login Service (Žák 11922222)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0
spring-rabbit	3.0.4
h2	2.1.214
spring-security-crypto	6.1.0
java-jwt	4.4.0

Testing:

spring-boot-starter-test	3.1.0
mockito-core	5.3.1

Notification Service (Žák 11922222)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0

spring-rabbit	3.0.4
h2	2.1.214

Testing:

spring-boot-starter-test	3.1.0
mockito-core	5.3.1

Send Notification Service (Žák 11922222)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0
spring-rabbit	3.0.4
h2	2.1.214
spring-boot-starter-mail	3.1.0

Testing:

spring-boot-starter-test	3.1.0
mockito-core	5.3.1
greenmail-junit5	2.0.0

Feedback (David Kreisler/01569177)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0
spring-rabbit	3.0.4
h2	2.1.214

Testing:

spring-boot-starter-test	3.1.0
mockito-core	5.3.1

AnalyticReport (David/Kreisler/01569177)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0
spring-rabbit	3.0.4
h2	2.1.214

Testing

spring-boot-starter-test	3.1.0
mockito-core	5.3.1

Maintenance(David Kreisler/01569177)

spring-boot-starter-web	3.1.0
spring-boot-starter-data-jpa	3.1.0
spring-rabbit	3.0.4
h2	2.1.214

Testing

spring-boot-starter-test	3.1.0
mockito-core	5.3.1

2. Updated Requirements

1. Login Service (Michal Zak)
2. Search Service (Victoria Zeillinger)
3. Event Service (Victoria Zeillinger)
4. Analytic Report Service (David Kreisler)
5. Notification Service (Michal Zak)
6. Recommender Service (Fabian Schmon)
7. Attendee Service (Fabian Schmon)
8. Feedback Service (David Kreisler)
9. Bookmark and Tagging Service (Alexander Grentner)
10. Calendar Service (Alexander Grentner)
11. Maintenance Service (David Kreisler)

ID	Domain	Requirement	Description	Use Case	Priority	Verification	Status
1.1	Auth Service	Role-based	A user can have one of three roles (organizer, attendee, admin)	1, 2	high	Test, UI	pass
1.2	Auth Service	Register	A user can register and create an account for the role attendee or organizer.	1	high	Test, UI	pass
1.3	Auth Service	Log in	A user can log in to the system based on their role	2	high	Test, UI	pass
2.1	Search Service	List all Events	Every user can see a list of every saved event.	3,4	high	UI, Test	pass
2.2	Search Service	Search for Events	Every user can search for every event by different criteria.	3,4	medium	Test, UI	pass
3.1	Event	Create Event	Every organiser can create and save events	5	high	Test	pass

			in their event inventory.				
3.2	Event	Update Event	An organizer can update the fields of their own events in their Event Inventory.	3	mediu m	Test, UI	pass
3.3	Event	List all events	All events of one organizer are listed in their Event Inventory	3,5	high	UI, Test	pass
3.4	Event	Delete Event	An organizer can delete their own event.	3	low	Test, UI	pass
3.5	Event	Event Capacity	An organizer can specify the maximum capacity.	3	mediu m	Test	pass
4.1	Analytic Report	View Analytic Report Event	An organizer can view a report about registered andcapacity from an event	8	mediu m	Test, UI	only visualizati on in frontend doesnt work
4.2	Analytic Report	View Analytic Report Feedback	Organizers and attendees can view a report about feedback from an event	9	low	Test, UI	only visualizati on in frontend doesnt work
5.1	Notificat ion Service	Schedules and orchestrate notifications	The notification service schedules new notification and also orchestrate them.	13	high	Test, UI, review	pass there are issues with other services not publishing properly
6.1	Recom mender Service	Orchestrate s notification attendees for new events	Attendees get notifications about new events based on their bookmarks	14	mediu m	Test	could fail if eventID is unknown

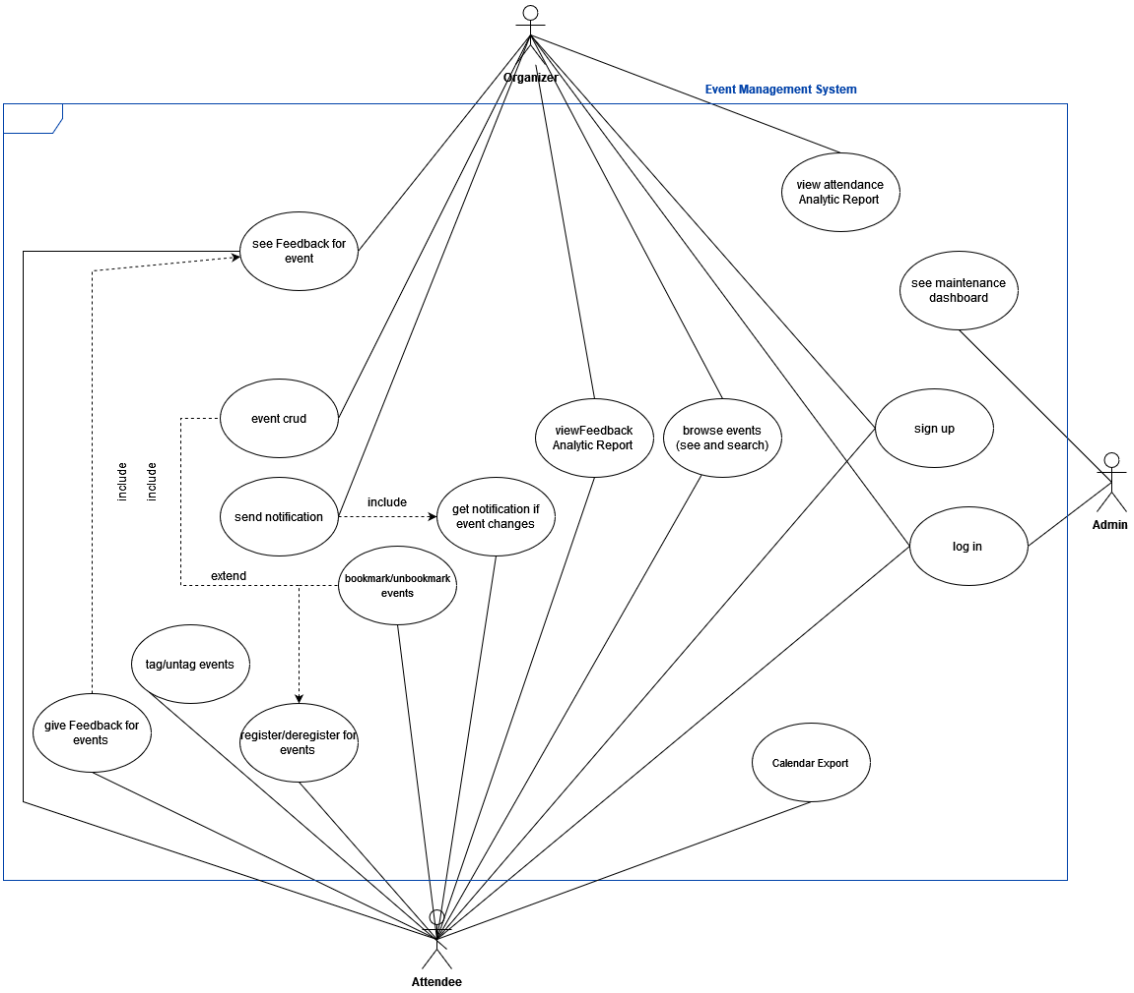
			and other criteria				
7.1	Attendance Service	Attendees can register	Attendees can register for an event, if there are vacancies	12	medium	Test, UI	could fail if eventID is unknown
7.2	Attendance Service	Attendees can deregister	Attendees can deregister for an event	12	low	Test, UI	could fail if eventID is unknown
7.3	Attendance Service	Organizer can send messages	Organizers of an event can send messages directly to an attendee	13	medium	Test, UI, review	
8.1	Feedback	Give Feedback	Attendees and organizers can give Feedback to an event with a rating and a text.	6	medium	Test, UI	pass
8.2	Feedback	See Feedback	Attendees and Organizers can see feedback given on an event already passed.	6	medium	Test, UI	only visualisation in frontend doesnt work
9.1	Bookmark and Tagging	Bookmark Events	Attendees can bookmark events if they are interested	10	medium	Test, UI	pass
9.2	Bookmark and Tagging	Unbookmark Events	Attendees can unbookmark events they have bookmarked.	10	medium	Test, UI	pass
9.3	Bookmark and Tagging	Get all Bookmarked Events for user	All bookmarked events per user	10	medium	Test, UI	not relevant for FINAL

9.4	Bookmark and Tagging	Tag Events	Attendees can Tag events with given tags.	11	medium	Test, UI	pass
9.5	Bookmark and Tagging	Untag Events	Attendees can untag already tagged events	11	medium	Test, UI	pass
9.6	Bookmark and Tagging	Get all Tags per user and Event	All tags per event and user	11	medium	Test, Ui	not relevant for FINAL
10.1	Calendar Export	Export bookmarked Event	Attendees can export a bookmarked event as json, xml or ical	16	medium	Test, UI, review	partly fulfilled event data is not fetchable in some cases
10.2	Calendar Export	Export registered Event	Attendees can export an event they are registered for as json, xml or ical.	16	medium	Test, UI, review	registered events are not fetchable
11.1	Maintenance Service	Dashboard	Admins can see a dashboard with health-information about the system	15	high	Test, UI	visualization in frontend doesnt work
12.1	Send Notification Service	Send Notifications	The Send notification service is responsible for sending notifications to email addresses.	8	medium	Test, UI, review	pass

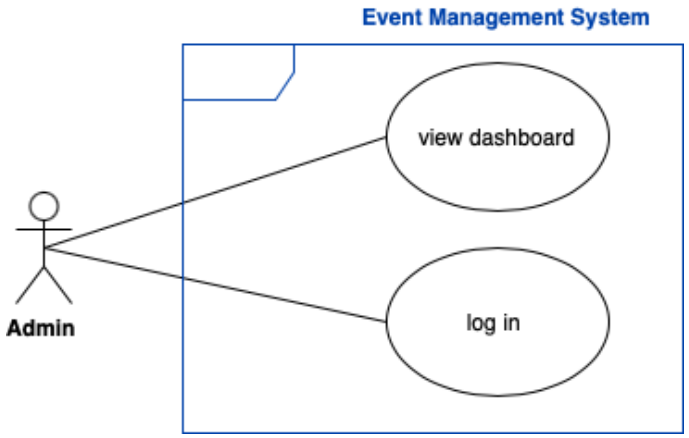
3. Updated 4+1 Views Model

3.1. Scenarios / Use Case View

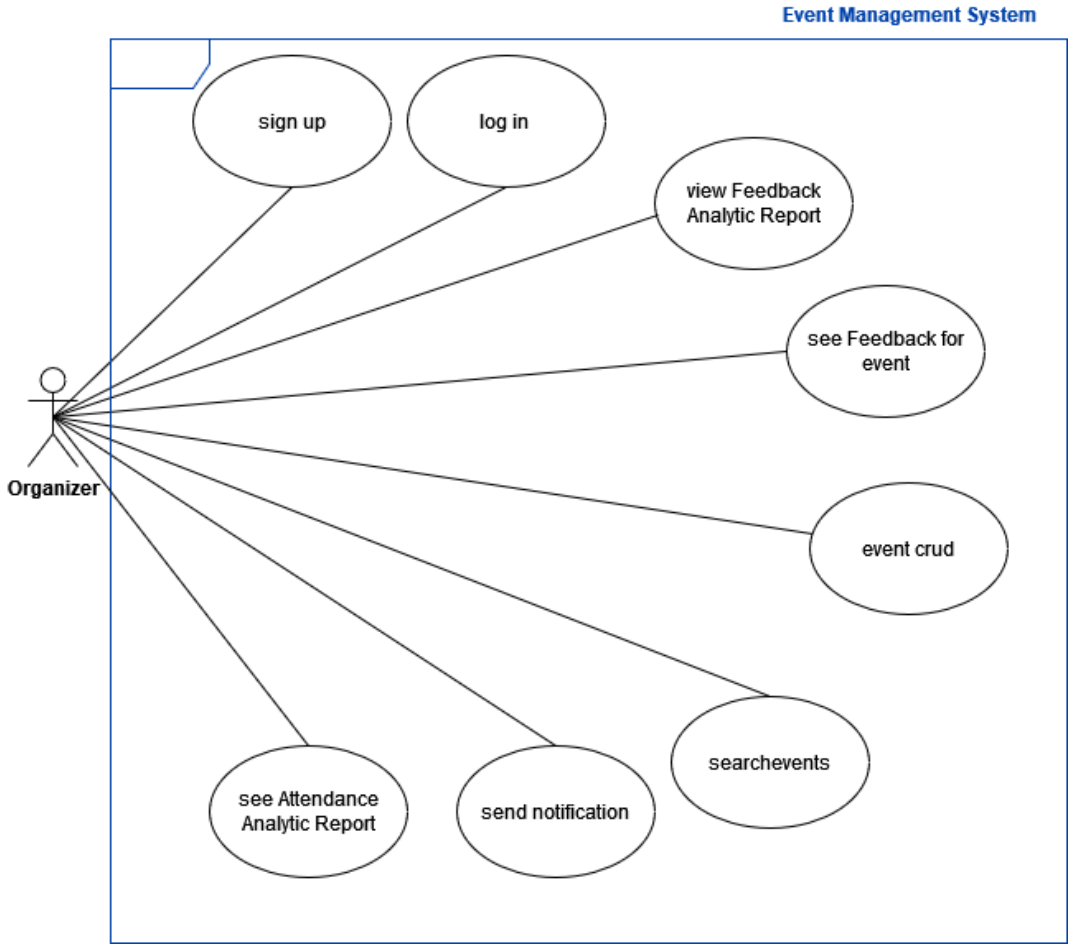
3.1.1. Use Case Diagram(s)



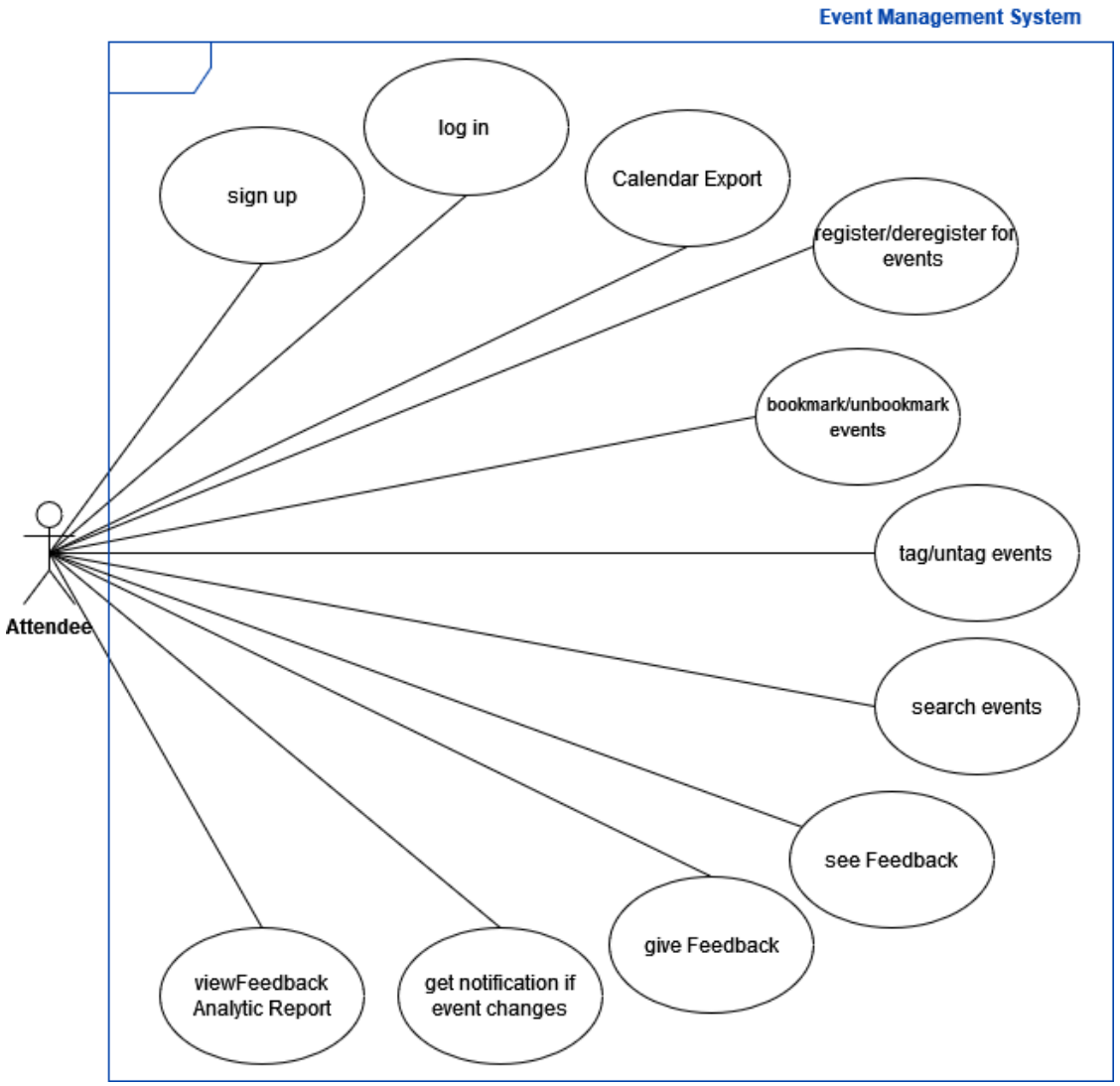
Use Case Diagram which represents the whole system at once. Some use cases or functionalities can be carried out by more than one user role. Some use cases are extended from others or included if there has to be one use case carried out first so that the other one can perform its functionality. The Use-Case “sign up” represents the user registration, thus creating a whole new user in the system. The Use-Case “register/deregister for Events” represents the registration for events from attendees. We use the term register instead of attending because it could get confused with the noun attending.



Use Case diagram representing the admin view and only functions an admin can carry out.



Use Case diagram representing the organizer's view and only functions an organizer can carry out.



Use Case diagram representing the attendee's view and only functions an attendee can carry out.

3.1.2. Use Case Descriptions

Use Case:	Sign up
Use Case ID:	1
Actor(s):	Unregistered user
Brief Description:	An unregistered user can create an account with their data and choose a role, either attendee or organizer
Pre-Conditions:	The user owns an email which is not saved in the system
Post-Conditions:	User is created
Main Success Scenario:	User is signed in and can log in with saved credentials.
Extensions:	
Priority:	high
Performance Target:	high
Issues:	

Use Case:	Login with existing user
Use Case ID:	2
Actor(s):	attendee, organizer or admin
Brief Description:	An already registered user can log in to the system with

	their credentials
Pre-Conditions:	Credentials must be correct, User is created
Post-Conditions:	User is logged in and can use the system
Main Success Scenario:	1. User access the login page of the system 2. User enters their credentials 3. Credentials are validated by the system 4. An access token is issued to the user
Extensions:	
Priority:	high
Performance Target:	medium
Issues:	

Use Case:	Event CRUD
Use Case ID:	3
Actor(s):	organizer
Brief Description:	An organizer can create any event and read, update and delete their own event with name, capacity, type, description, and date.
Pre-Conditions:	User has an “organizer” account on the system. User is logged in.
Post-Conditions:	Event is created/updated and saved with a link to this organizer. It can be found or deleted by organizerID.
Main Success	Newly created or existing events updated and saved.

Scenario:	Found event by criteria or deleted event.
Extensions:	
Priority:	high
Performance Target:	high
Issues:	

Use Case:	Search Events
Use Case ID:	4
Actor(s):	organizer, attendee
Brief Description:	Organizers or attendees can see all events and search for every event by different criteria.
Pre-Conditions:	Events had to be created in event service and saved in search service.
Post-Conditions:	One or more Events are returned
Main Success Scenario:	Only events with the right criteria are viewed.
Extensions:	
Priority:	high
Performance Target:	medium

Issues:	What happens with spelling mistakes

Use Case:	give Feedback
Use Case ID:	5
Actor(s):	Attendee
Brief Description:	Attendees can give Feedback for events they have attended.
Pre-Conditions:	The event has to be concluded already and the user has to be registered.
Post-Conditions:	Feedback should be linked to event and user
Main Success Scenario:	Feedback is created and saved.
Extensions:	Event CRUD
Priority:	medium
Performance Target:	medium
Issues:	

Use Case:	see Feedback
Use Case ID:	6

Actor(s):	Organizer, attendee
Brief Description:	Organizers and attendees can see the feedback which was given for an event.
Pre-Conditions:	Event for which the feedback should be shown has had to be created.
Post-Conditions:	Feedbacks were shown.
Main Success Scenario:	Organizer or Attendee can see every Feedback for the chosen event
Extensions:	give Feedback
Priority:	medium
Performance Target:	medium
Issues:	

Use Case:	View Attendance Analytic Report
Use Case ID:	7
Actor(s):	Organizer
Brief Description:	Organizers can view the Analytic Reports for an event. The report contains information about attendees who are registered for an event and the capacity.
Pre-Conditions:	The event for which the report should be displayed has had to be created. Only the event organizer can view the report.

Post-Conditions:	An analytic report is shown.
Main Success Scenario:	Organizer can view the Report to the linked Event.
Extensions:	EventCRUD
Priority:	medium
Performance Target:	low
Issues:	

Use Case:	View Feedback Analytic Report
Use Case ID:	8
Actor(s):	Attendee, Organizer
Brief Description:	Attendees and organizers can view the Analytic Reports for an event. The report contains feedback information.
Pre-Conditions:	The event for which the report should be displayed has had to be created.
Post-Conditions:	An analytic report is shown.
Main Success Scenario:	Attendees and organizers can view the Report to the linked Event.
Extensions:	EventCRUD
Priority:	medium

Performance Target:	low
Issues:	

Use Case:	Bookmark/Unbookmark Events
Use Case ID:	9
Actor(s):	Attendee
Brief Description:	Attendees can bookmark events if they are interested. They can also unbookmark it again.
Pre-Conditions:	User has to have the role of “attendee”. User has to be authenticated and logged in. Event has to be created in advance to be bookmarked.
Post-Conditions:	Bookmarked Event is linked to an attendee with its eventId.
Main Success Scenario:	Attendees can bookmark/unbookmark the events.
Extensions:	Event CRUD
Priority:	high
Performance Target:	medium
Issues:	

Use Case:	Tag/Untag Events

Use Case ID:	10
Actor(s):	Attendee
Brief Description:	Attendees can tag events with predetermined tags. They can also untag them again.
Pre-Conditions:	User has to have the role of “attendee”. Event has to be created, has to be tagged or untagged for each case. User has to be logged in and authenticated
Post-Conditions:	Tag is linked to Event and Attendee. The tag is taken from an Enum.
Main Success Scenario:	Attendees can tag the events.
Extensions:	Event CRUD
Priority:	high
Performance Target:	medium
Issues:	

Use Case:	register/deregister for Events
Use Case ID:	11
Actor(s):	Attendee
Brief Description:	Attendees can register for Events. If they are registered, they can also unregister again.

Pre-Conditions:	Event has to be created and the attendance count for that event must be below its maximum capacity in order to register. Users need to be already registered in order to deregister.
Post-Conditions:	Register/Attendance is linked to Event and Attendee.
Main Success Scenario:	Attendee is registered to Event and this registration is saved. Organizers can see the current attendance count.
Extensions:	Event CRUD
Priority:	medium
Performance Target:	medium
Issues:	

Use Case:	Send notification
Use Case ID:	12
Actor(s):	Organizer, attendee
Brief Description:	Organizers can send Notifications to Attendees of their own events.
Pre-Conditions:	Users have to be registered by email. Sender has role organizer. The recipient has the role of “attendee”.
Post-Conditions:	Attendee received an email from the organizer.
Main Success	Organizer sent an email to an attendee who is registered

Scenario:	for an event. Attendee receives this email.
Extensions:	
Priority:	high
Performance Target:	high
Issues:	What if the external mail server isn't working?

Use Case:	get notification if event changes
Use Case ID:	13
Actor(s):	Organizer, attendee
Brief Description:	Attendees receive a notification if an event changes.
Pre-Conditions:	Organizer updated an event
Post-Conditions:	Attendee received an email.
Main Success Scenario:	Attendee received an email if an event was updated.
Extensions:	
Priority:	medium
Performance Target:	high
Issues:	What if the external mail server isn't working?

Use Case:	Get recommender notification
Use Case ID:	14
Actor(s):	attendee
Brief Description:	Attendees get notifications about new events they might like, based on their bookmarks and the chosen recommender filter
Pre-Conditions:	Users have to be registered by email. The recipient has the role of “attendee”. The recipient has enough bookmarks with the same eventtype as the new event.
Post-Conditions:	Attendee received an email.
Main Success Scenario:	Attendees get emails with an event recommendation
Extensions:	send notification
Priority:	medium
Performance Target:	medium
Issues:	What if the external mail server isn’t working?

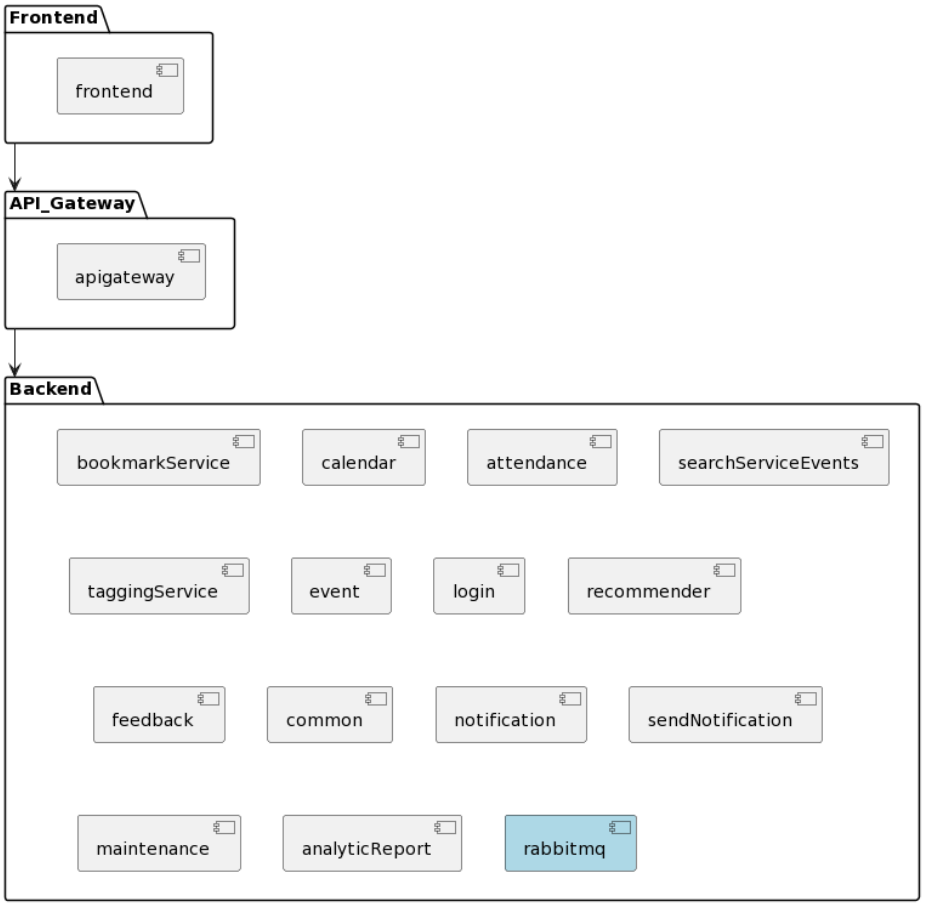
Use Case:	View maintenance dashboard
Use Case ID:	15
Actor(s):	admin

Brief Description:	Admins can view a dashboard with the health of each service.
Pre-Conditions:	User has the role “admin”. The system is running.
Post-Conditions:	Information about all services is viewed.
Main Success Scenario:	Admins can see the health of the system and can check if a service isn't working.
Extensions:	
Priority:	medium
Performance Target:	medium
Issues:	What if all Services aren't running?

Use Case:	Calendar Export
Use Case ID:	16
Actor(s):	attendee
Brief Description:	Attendees can export events to a calendar
Pre-Conditions:	User has the role of attendee. Events are registered or bookmarked.
Post-Conditions:	JSON, XML, or Ical is created and downloaded on the users device
Main Success Scenario:	Attendee exported an event as one of three formats, which is available on device.

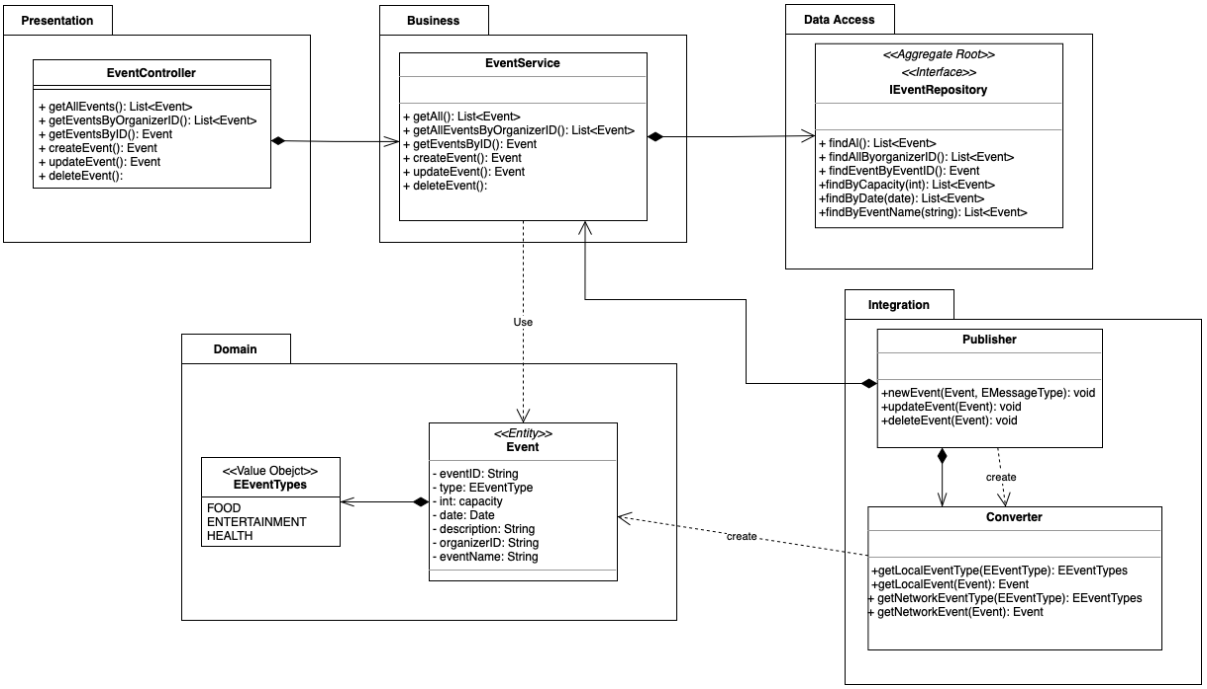
Extensions:	
Priority:	medium
Performance Target:	medium
Issues:	

3.2. Logical View



The logical view describes the main logical components of the application. By abstracting the dependencies within the services. The main components to differentiate in the services are in this case Frontend, the Api-Gateway and the further backend services.

Event Service (Victoria Zeillinger/11809914)

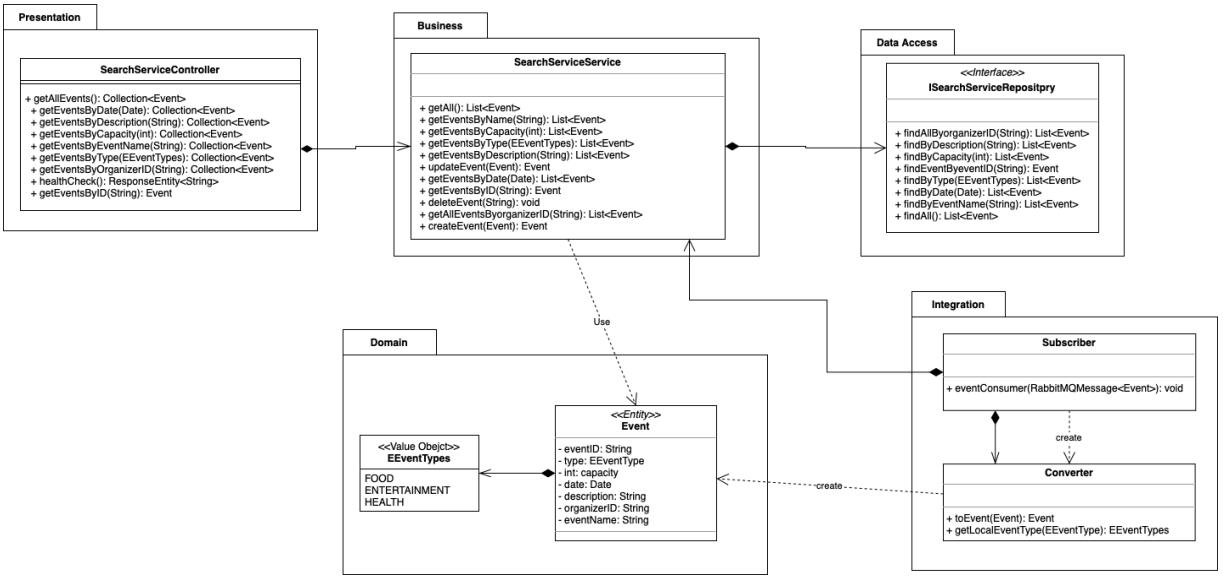


The Event Microservice is responsible for the Event CRUD. Events can be created, updated, read, and deleted. They are stored in the Event Entity and have a relation to the organizer by saving their ID. The types an event can have are predefined by an enum, EEventTypes. Every other service that needs an event can subscribe to the Publisher where every change, create, update and delete, will be provided asynchronously .

DDD-Building Blocks:

- Entity
- Value Object

Search Service Event (Victoria Zeillinger/11809914)



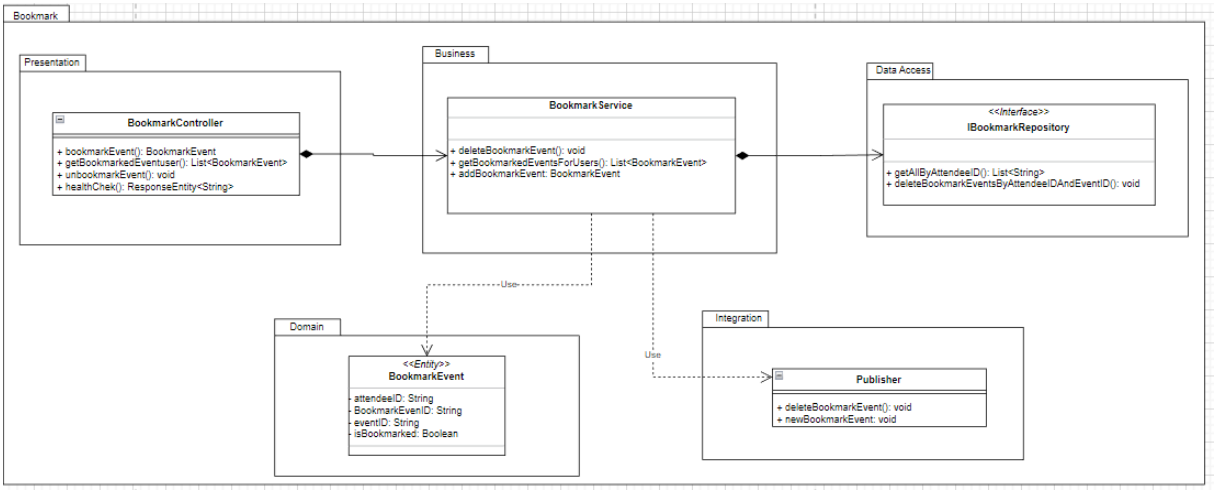
The Search-Service-Event Microservice responsible for filtering and providing the users with the events. Everytime an event is created or changed in the Event Microservice, the subscriber gets this Event change and saves it in the local data storage. Therefore the events are duplicated and synchronised asynchronously. The types an event can have are predefined by an enum, EEventTypes. The Controller only provides GET-Methods, whereas the Service includes all the CRUD with input from the Subscriber.

DDD-Building Blocks:

- Entity
- Value Object

BookmarkService (Alexander Grentner/11743246)

The Bookmark service is structured using the model-view-controller pattern. The package includes a Controller that defines endpoints, a service that implements the logic, and requests to the database. To enable persistent storage, a BookmarkEntity is created that represents a bookmarked event. Additionally, the Publisher is used to publish to other subscribing services.

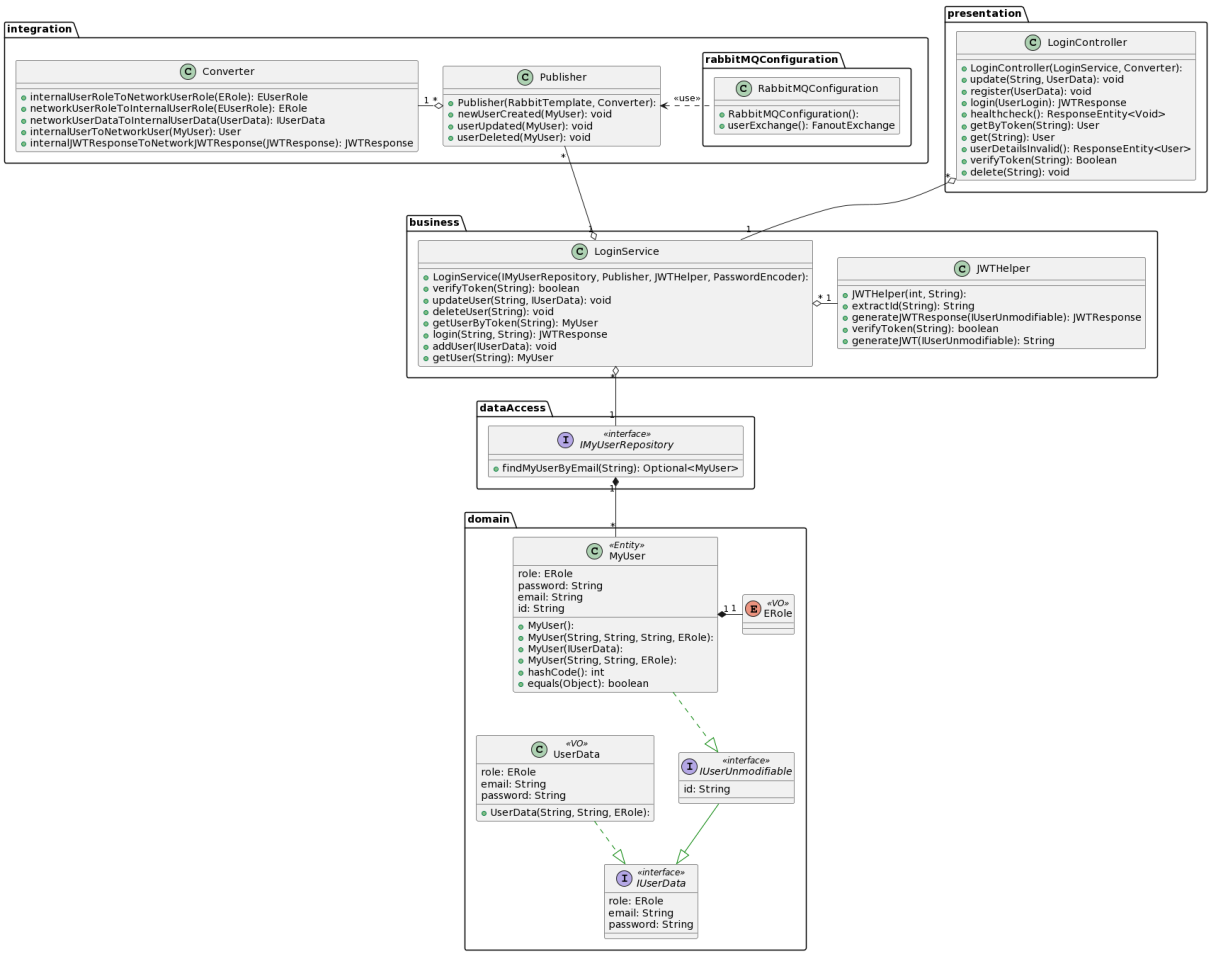


DDD-Building Blocks:

- Entity
- Aggregate Root

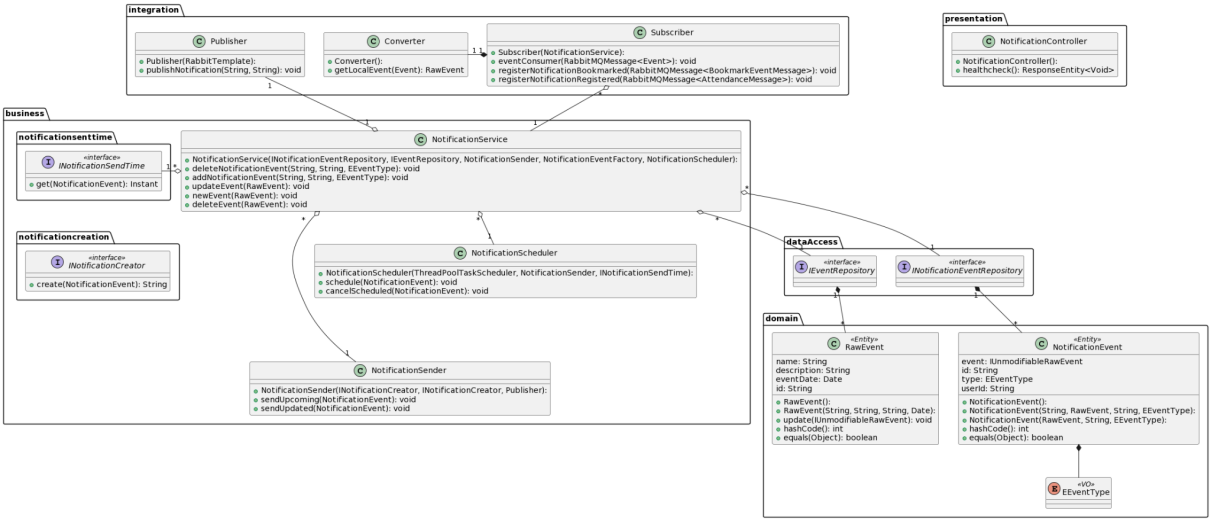
Login

The following class diagram depicts a high level overview of the Login Service. It receives REST request in the controller, routes them to the LoginService, which orchestrates all backend operation, and publishes any changes to its domain models via the publisher.



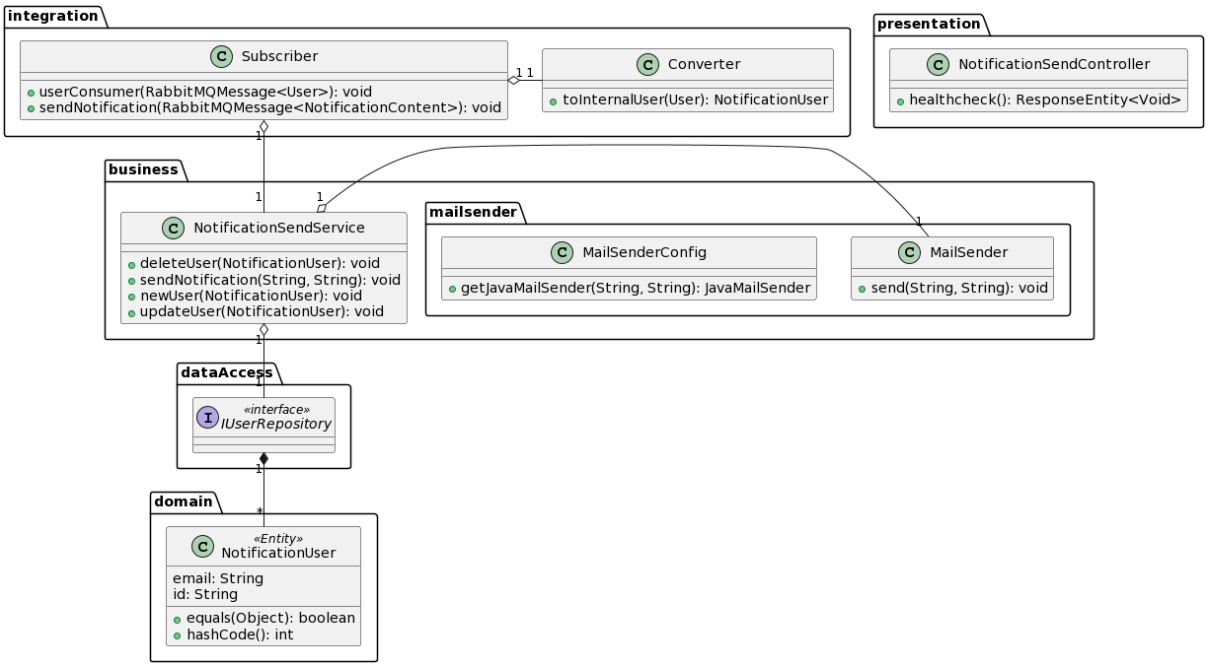
Notification

The following depicts a high level class diagram of the notification service. It is triggered by a message coming into the Subscriber. The subscriber forwards this request to the NotificationService, which orchestrate all backend events



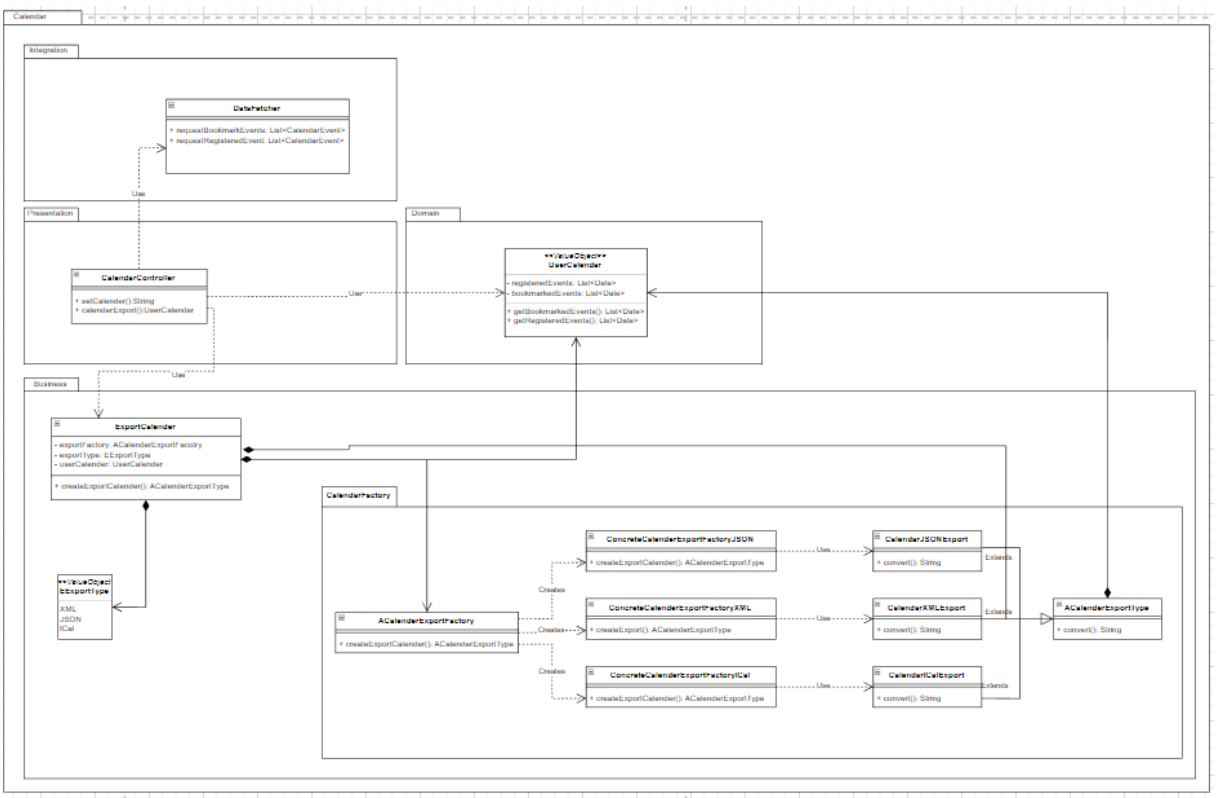
SendNotification

The following is a simple class diagram, showing a high level overview of its structure. The functionality is triggered by a message coming into the Subscriber, which redirects it into the Service class, which orchestrates all backend activity.



CalendarService (Alexander Grentner/11743246)

The CalendarService provides the functionalities for exporting a calendar in with certain defined types. The creation of the calendar is facilitated by using a factory for each calendar type which should be exported.



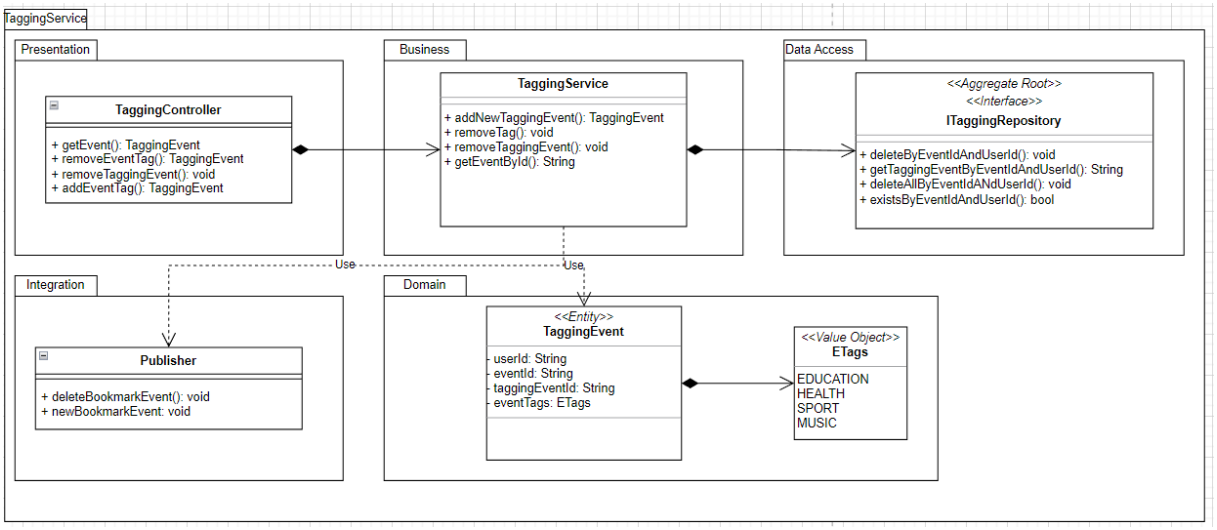
These export types are defined as an enumeration and include JSON, XML, and ICAL formats. The user's saved events are represented within the userCalendar and therefore require data from other services or packages.

DDD-Building Blocks:

- Value Object

TaggingService (Alexander Grentner/11743246)

The Tagging service is designed and implemented using the Model-View-Controller pattern. By also using a Publisher for other subscribing mircroservices.

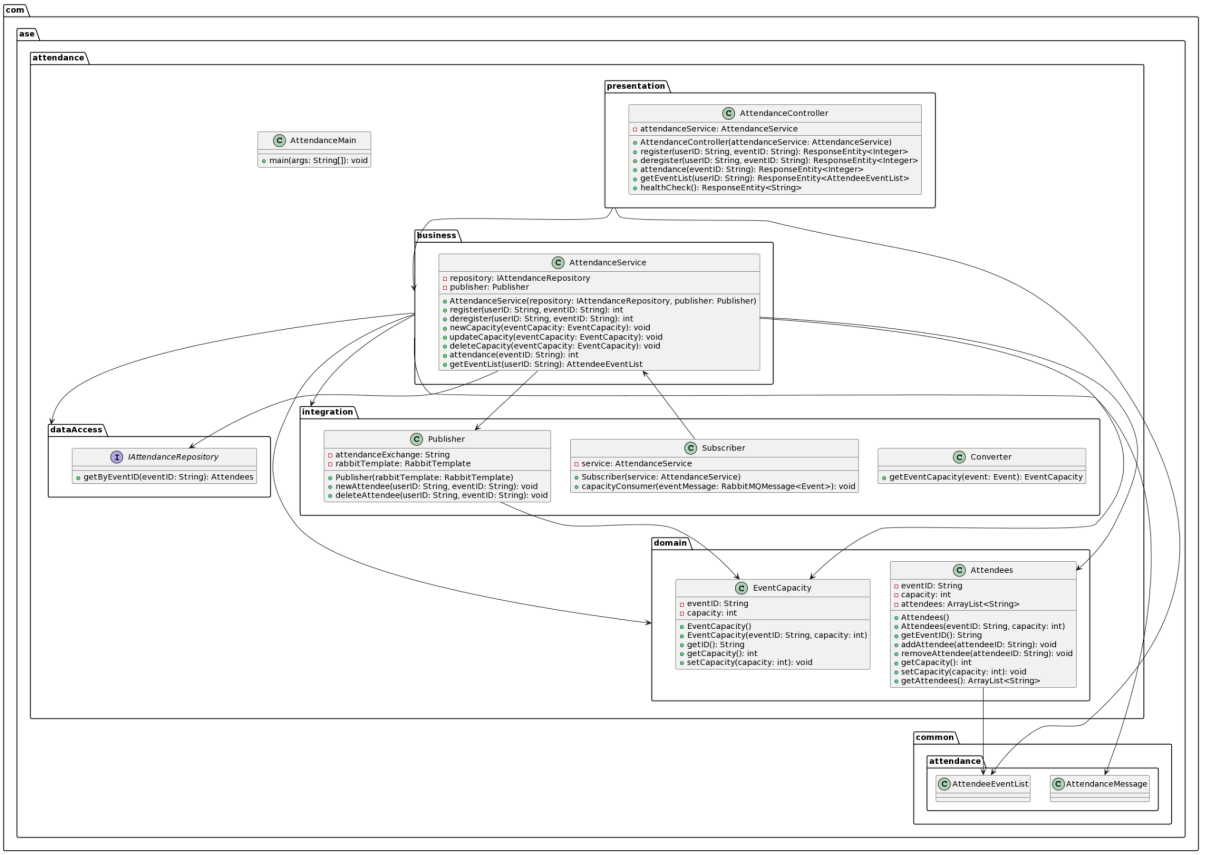


The TaggingController contains endpoints for accessing the TaggingService and its logic. In addition to creating and deleting events, the TaggingService also provides a method for updating the tags of TaggingEvent objects, with a focus on the tags within the event. The Tags themselves are defined using an Enum. The data stored persistently is defined by a tag, userId and eventId.

DDD-Building Blocks:

- Entity
- Value Object
- Aggregate Root

Attendance Service (Fabian Schmon/01568351)



The logical view of the attendance service represents the high-level structure and organization of the components involved in managing event attendance. It focuses on the logical relationships between the classes and modules of the attendance service.

In the attendance service, the core domain concepts are represented by the Attendees and EventCapacity classes in the com.ase.attendance.domain package. Attendees store information about the attendees of an event, including the event ID, capacity, and a list of attendee IDs. EventCapacity represents the capacity of an event.

The business logic of the attendance service is encapsulated in the AttendanceService class in the com.ase.attendance.business package. It provides methods for registering and deregistering attendees, managing event capacities, retrieving attendance counts, and obtaining a list of events attended by a user. The AttendanceService depends on interfaces defined in the com.ase.attendance.dataAccess package, such as IAttendanceRepository, for accessing and manipulating data related to event attendance.

Integration with external systems, such as RabbitMQ for message-based communication, is handled in the com.ase.attendance.integration package. The Publisher class is responsible for publishing messages related to attendee changes,

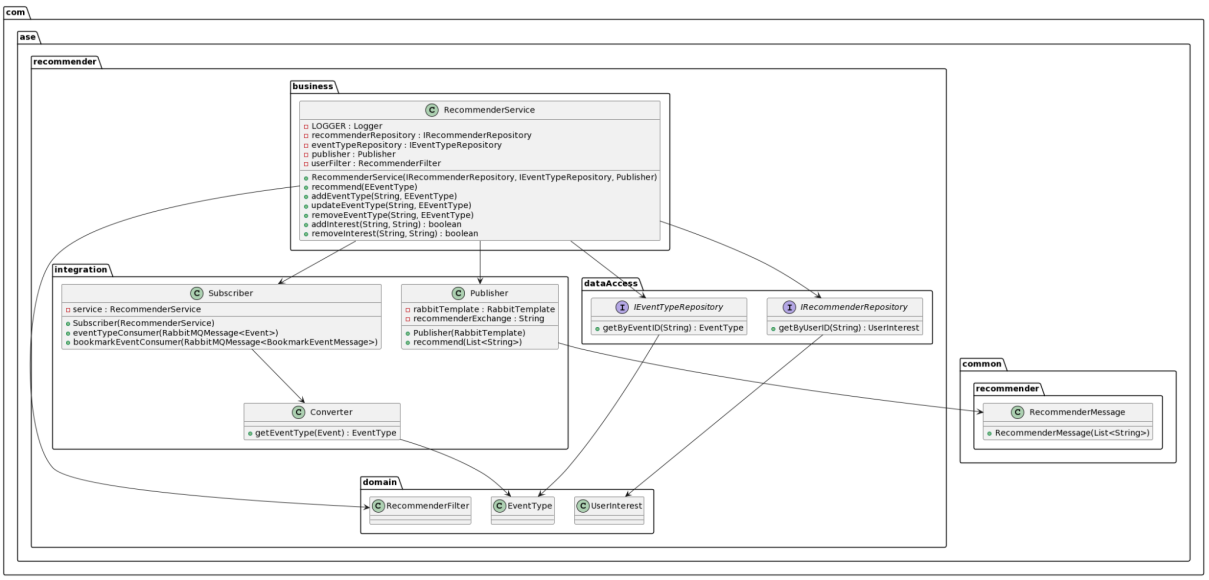
while the Subscriber class consumes messages and triggers corresponding actions in the AttendanceService. The Converter class assists in converting message objects to domain objects.

The presentation layer is represented by the AttendanceController class in the com.ase.attendance.presentation package. It provides RESTful endpoints for registering, deregistering, and retrieving attendance information. Additionally, a health check endpoint is available for monitoring the service's availability.

Overall, the attendance service follows Domain-Driven Design (DDD) principles, with the following DDD blocks used:

- Entities: Attendees and EventCapacity classes represent core domain concepts and hold state.
- Repository: IAttendanceRepository interface defines the contract for accessing Attendees entities.
- Service: AttendanceService class encapsulates the business logic and operations related to event attendance.
- Value Objects: AttendanceMessage and AttendeeEventList classes are used as value objects for communication between components.
- Aggregate: Attendees and EventCapacity can be considered aggregates that encapsulate related entities and enforce consistency.

Recommender Service (Fabian Schmon/01568351)



The logical view of the recommender service represents the structure and organization of the components involved in providing event recommendations to users. It emphasizes the logical relationships between the classes and modules of the recommender service.

The core domain concepts are represented by the EventType, UserInterest, and RecommenderFilter classes in the com.ase.recommender.domain package. EventType stores information about different types of events, UserInterest records the interests of a user, and RecommenderFilter is used to filter recommendations based on a user's interests.

The business logic of the recommender service is encapsulated in the RecommenderService class in the com.ase.recommender.business package. This service provides methods for adding, updating, and removing event types and user interests, as well as recommending events based on user interests. RecommenderService depends on interfaces defined in the com.ase.recommender.dataAccess package, such as IEventTypeRepository and IRecommenderRepository, for accessing and manipulating data related to event types and user interests.

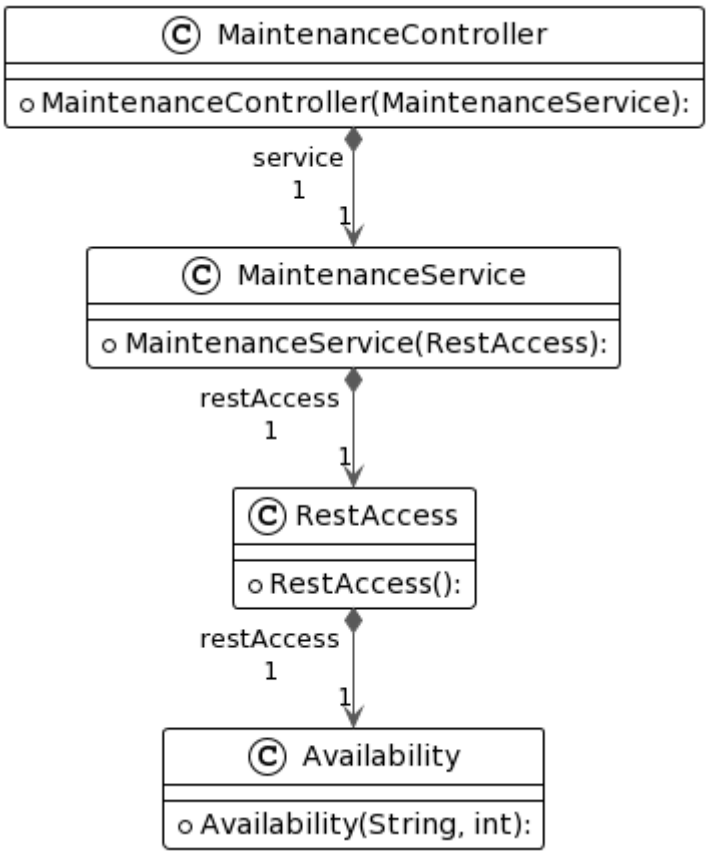
Integration with external systems, such as RabbitMQ for message-based communication, is handled in the com.ase.recommender.integration package. The Publisher class is responsible for publishing recommendation messages, while the Subscriber class consumes messages and triggers corresponding actions in the RecommenderService. The Converter class assists in converting RabbitMQ messages to domain objects.

In the com.ase.common.recommender package, the RecommenderMessage record is used for wrapping recommended user IDs list, which can be serialized and sent as a message.

Overall, the recommender service follows Domain-Driven Design (DDD) principles, with the following DDD blocks used:

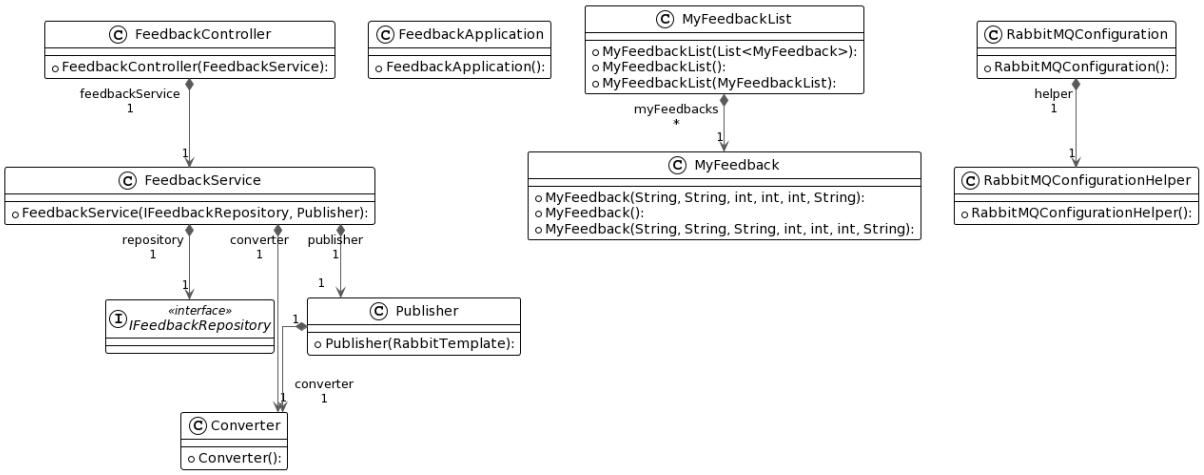
- Entities: EventType and UserInterest classes represent core domain concepts and hold state.
- Value Objects: RecommenderMessage and RecommenderFilter classes are used as value objects for communication between components.
- Repository: IEventTypeRepository and IRecommenderRepository interfaces define the contract for accessing EventType and UserInterest entities, respectively.
- Service: RecommenderService class encapsulates the business logic and operations related to user interest and event recommendation.
- Aggregate: EventType and UserInterest can be considered aggregates that encapsulate related entities and enforce consistency.

Maintenance Service



The logical view of maintenance service depicts the structure and organization of the service.The service accesses all other services using the RestAccess class, which returns an Availability object for each Service. This object consists of a hostname (string) and an http response code (int). No data is persisted and no message queue is used.

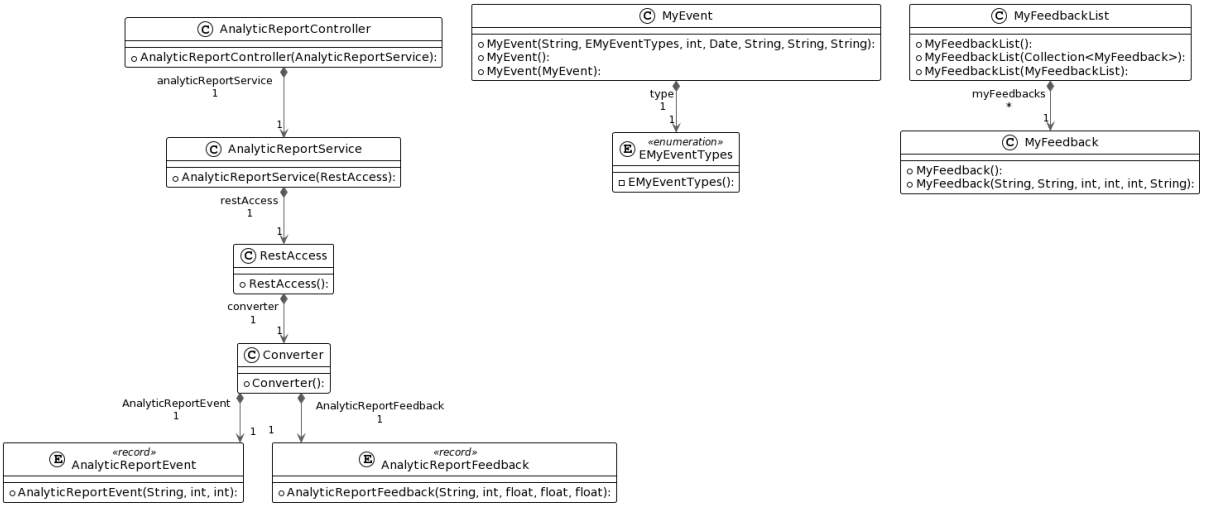
Feedback Service



The above diagram depicts the logical view of the feedback service. An entity (MyFeedback) is persisted, a publisher is used to inform interested parties about

newly created feedbacks. A converter is used to send feedbacks via the network. A wrapper class is used, surrounding the MyFeedback class, used to send packed feedbacks.

AnalyticReport Service



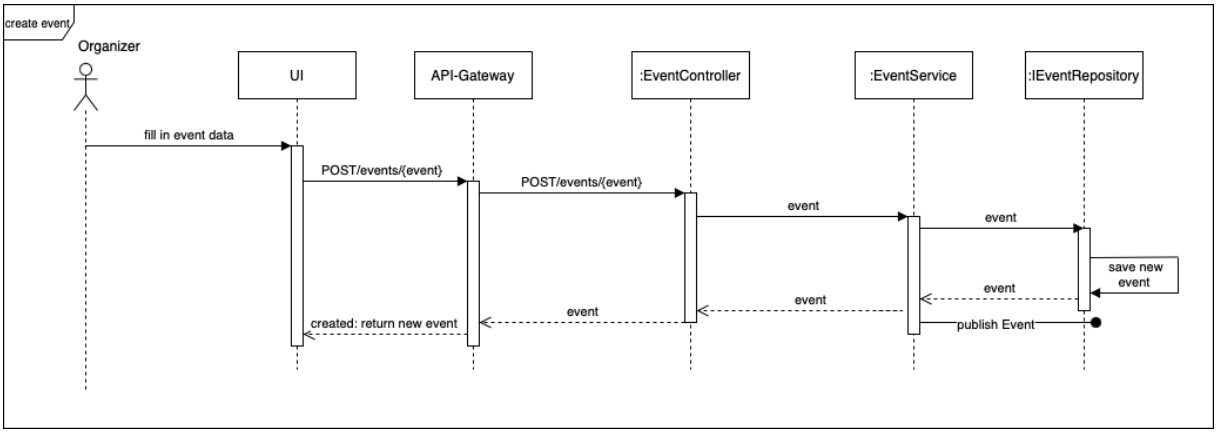
The logical view depicted above show the structure of the analytic feedback service. Important to note is that this service implements an MyEvent class, which uses a EMyEventTypes enum, aswell as a MyFeedback and a MyFeedbackList class. These are used to map the data acquired through the network to local objects.

3.3. Process View

Event Microservice (Victoria Zeillinger/11809914)

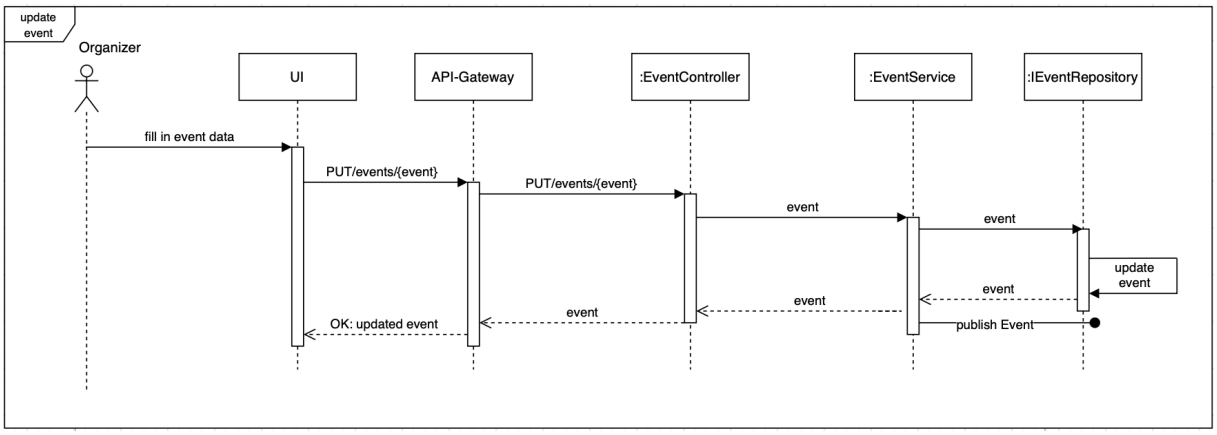
The following sections highlight the processes conducted by the Event service.

*... not mandatory field

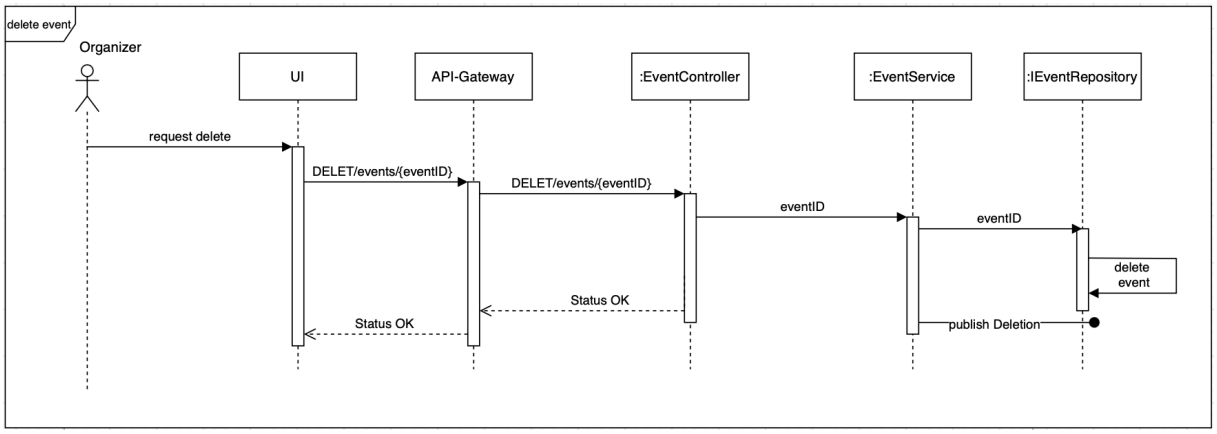


The sequence diagram above highlights the process of an event being created by an organizer. The process is initiated by the organizer filling in the event details (name, description*, capacity, date, type*) and ends, in the happy scenario, with the UI

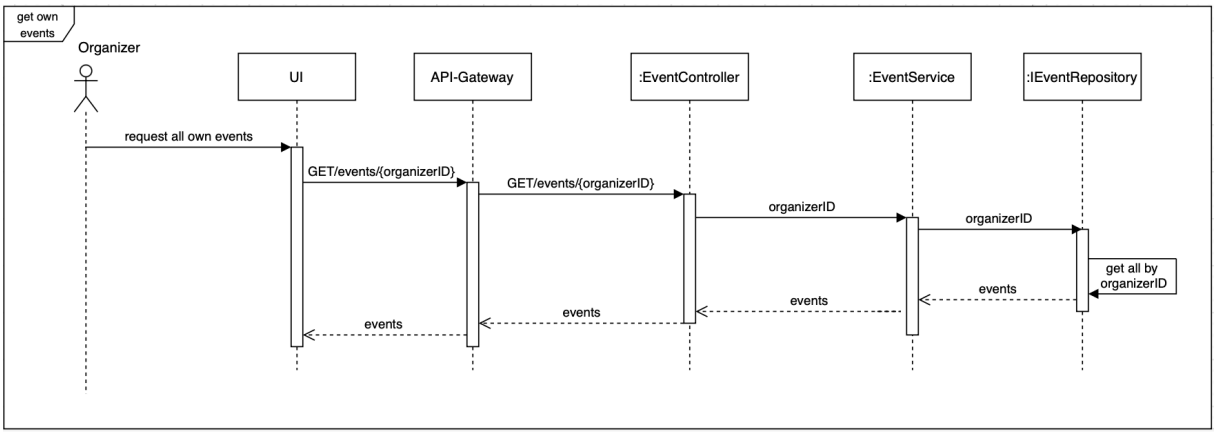
receiving a created and saved event. The EventService also publishes the event to a queue.



The sequence diagram above highlights the process of an event being updated by an organizer. The process is initiated by the organizer filling in the new event details which they want to change (name, description*, capacity, date, type) and ends, in the happy scenario, with the UI receiving an updated and saved event. The updated event is published to the queue.



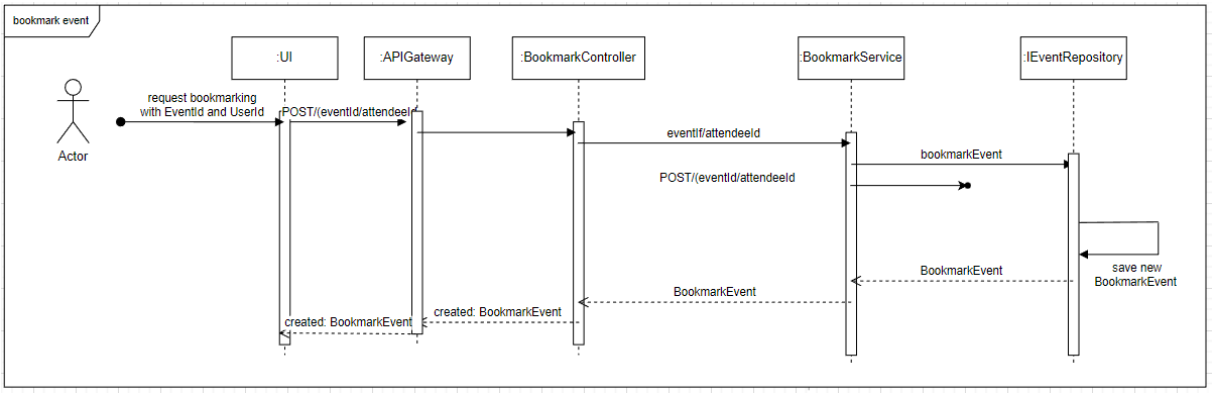
The sequence diagram above highlights the process of an event being deleted by an organizer. The process is initiated by the organizer requesting the deletion and ends, in the happy scenario, with a status update. The deleted event is published to the queue.



The sequence diagram above highlights the process of a request to get all events. The process is initiated by the organizer starting the request and ends, in the happy scenario, with the UI receiving all events which have this organizerID.

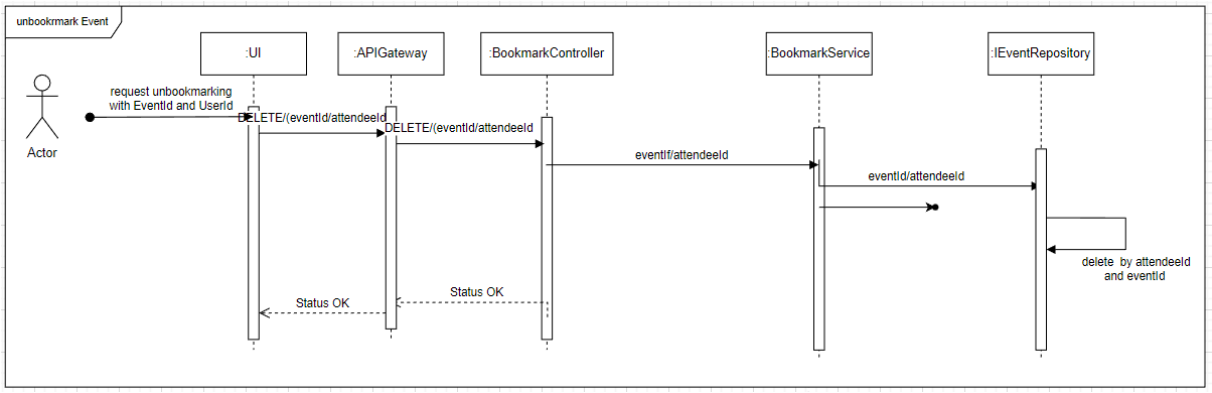
BookmarkService (Alexander Grentner/11743246)

Bookmark event



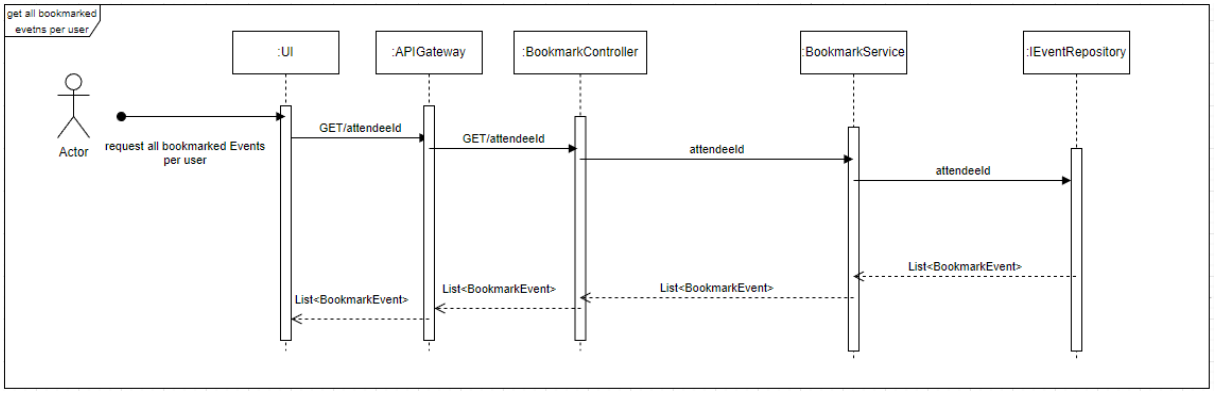
To bookmark an Event the User sends a POST request to the Controller and with the eventId and its attendeeId. This Triggers the Bookmark Service, which handles the creation of a new BookmarkEvent for the database. The Service also returns the successfully created BookmarkEvent within the process. Additionally the BookmarkService class which handles the business logic publishes a bookmarking message for other services.

Unbookmark Event



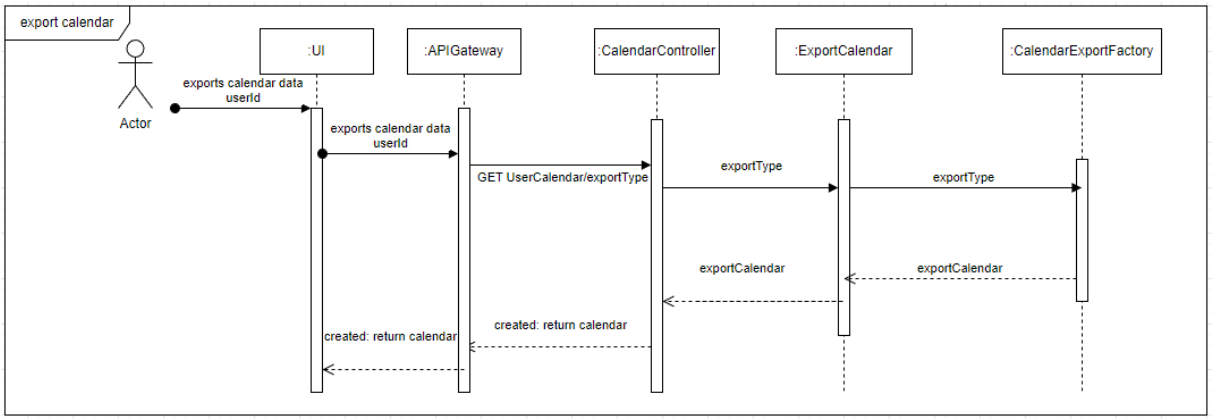
By unbookmarking an event the event will be deleted in the database, in addition to the deleting and the handling in the business logic a BookmarkEventMessage will be published to subscribers.

Get all bookmarked events per user



The BookmarkService is responsible for handling a GET request with the attendeeId as a parameter to retrieve a list of events bookmarked by each user from the database. The retrieved list of bookmarked events is then returned to the user for further processing within the services.

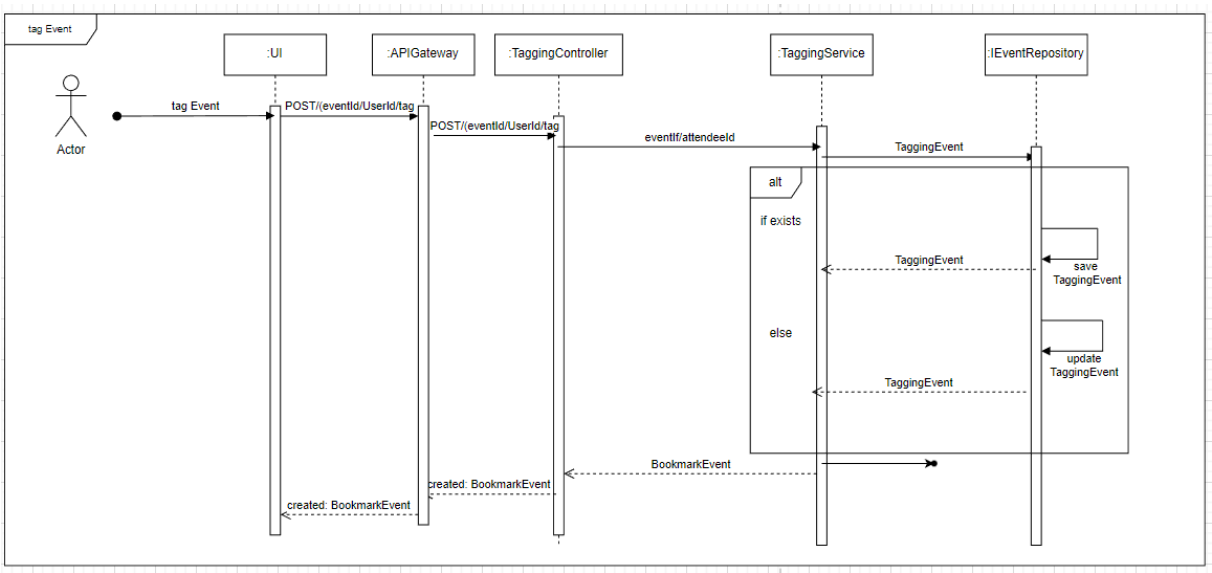
CalendarService (Alexander Grentner/11743246)



The calendar handles the data provided by the controller endpoint and propagates it further to the factory, which is in this diagram abstracted to one CalendarExportFactory but implemented as a much more complex structure.

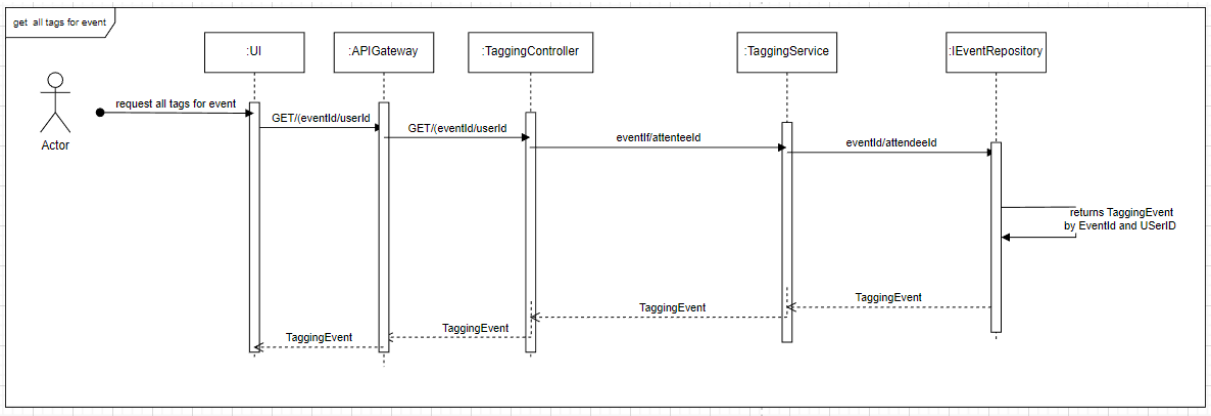
TaggingService (Alexander Grentner/11743246)

Tag event



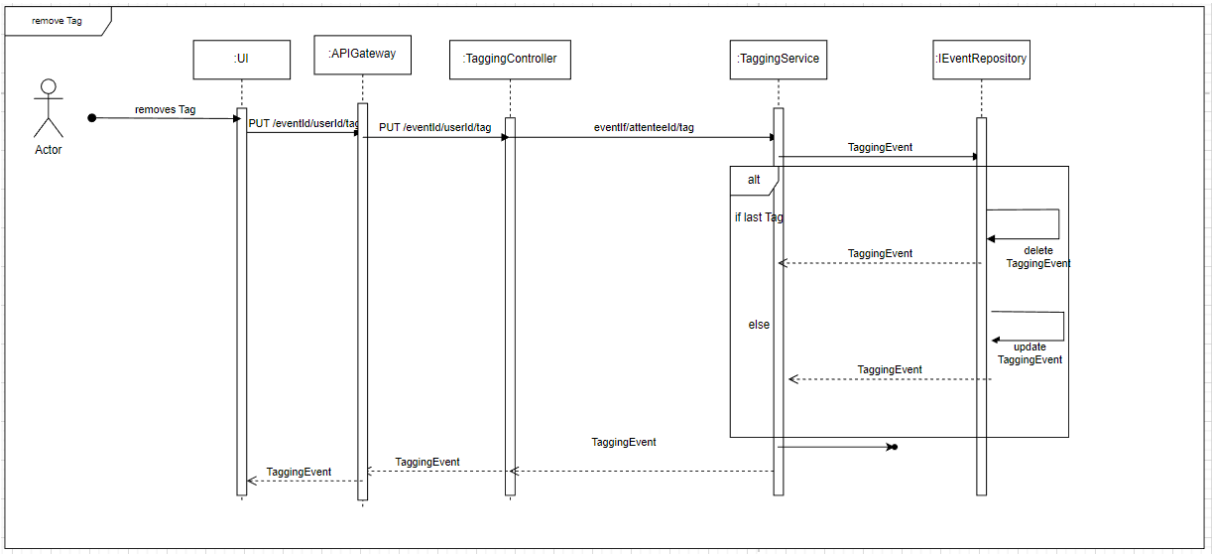
The endpoint is responsible for creating or updating the TaggingEvent, depending on whether it already exists. If the event has not been tagged previously, a new TaggingEvent will be created. Otherwise, the existing TaggingEvent will be updated in the database. Additionally, a TaggingEventMessage is triggered for publishing changes in the database for other services.

Get all tags for event



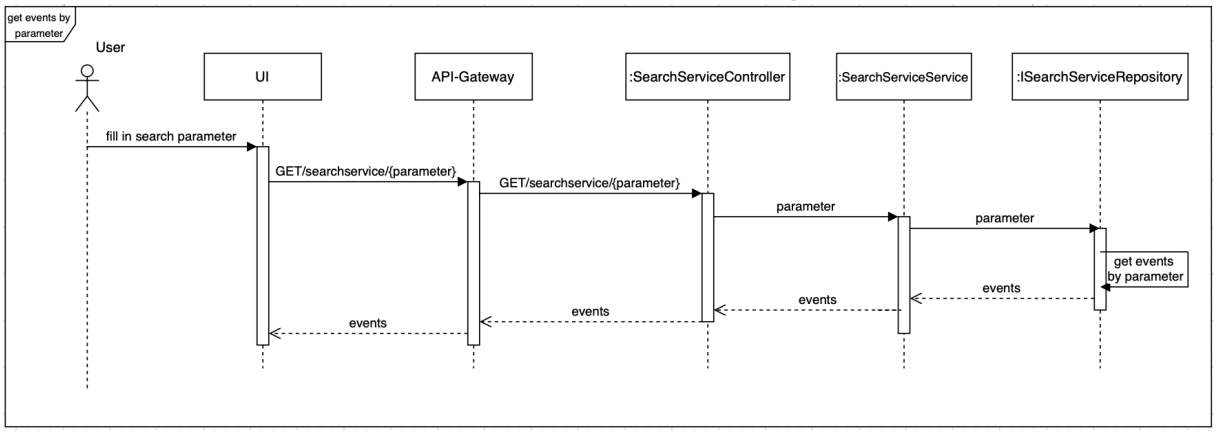
The GET request is propagated to the service, which will use the Repository to search for TaggingEvents associated with the specified eventId and userId.

Remove Tag



For removing a tag the Event will be updated with a new list of tags, which removes the requested tag. The service checks whether the tag is the last one in the TaggingEvent. If it is, the entire TaggingEvent is removed from the database to prevent unused datasets in the database. At the end of the updating process a message will be published via the publisher to inform possible subscribers to the service.

Search Service Microservice (Victoria Zeillinger/11809914)

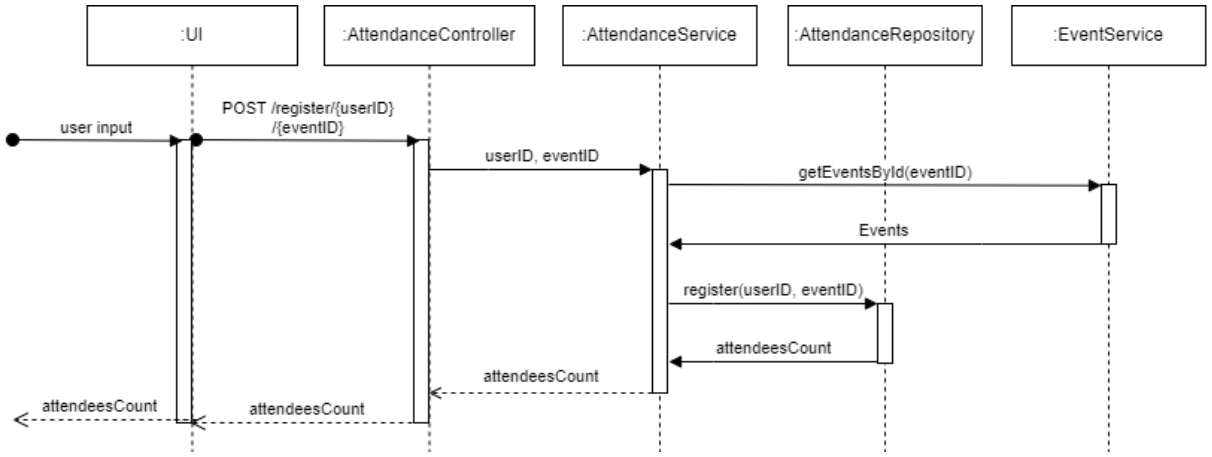


The sequence diagram above highlights the process of searching events based on different criteria. The process is initiated by the a user starting the request and filling in the search parameter (which can only be one of the fields of the event) and ends, in the happy scenario, with the UI receiving all events which have this criteria.

Attendance Service (Fabian Schmon/01568351)

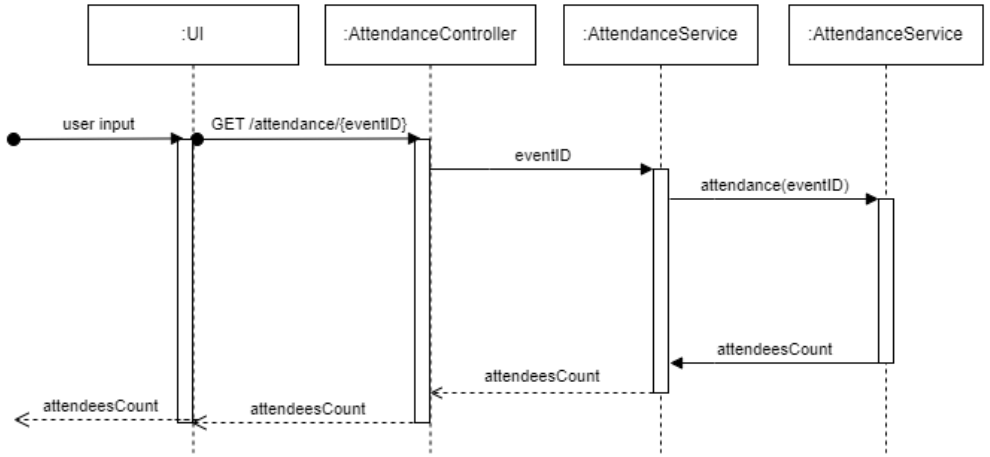
Register User for an Event

This sequence diagram illustrates the communication between the UI, the controller, the attendance service, and the event service for the three endpoints: Attend, attendance and deregister.



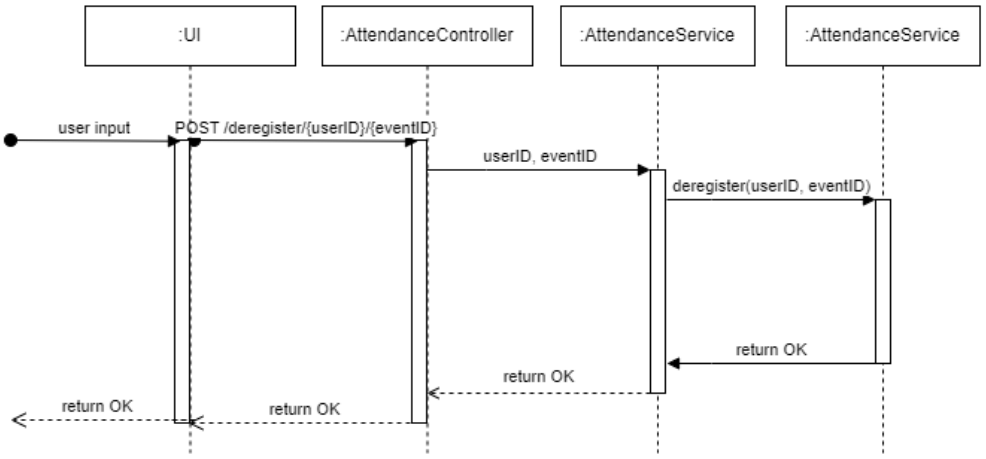
When one user wants to attend an event, a POST request with the userID and eventID gets sent to the controller which makes a request to the event service in order to check if the maximum capacity allows for one more participant.

Get Attendance Count for Event



To get the current attendance count, a GET request with the corresponding eventID is sent to the controller. In the end, the attendance count gets returned.

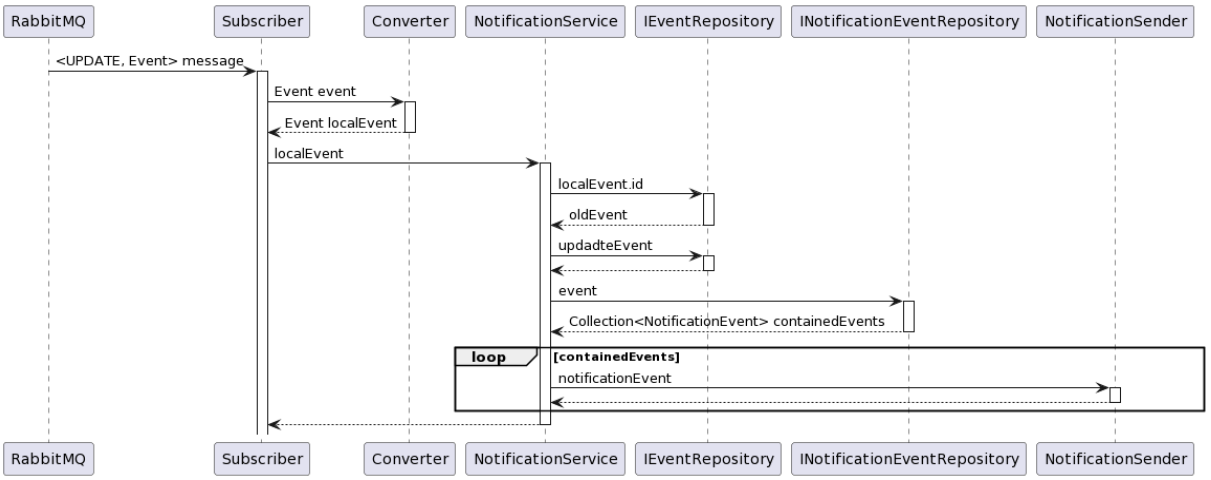
Deregister User for an Event



If an attendee wants to deregister from an event, a POST request with the userID and eventID must be sent to the attendance controller

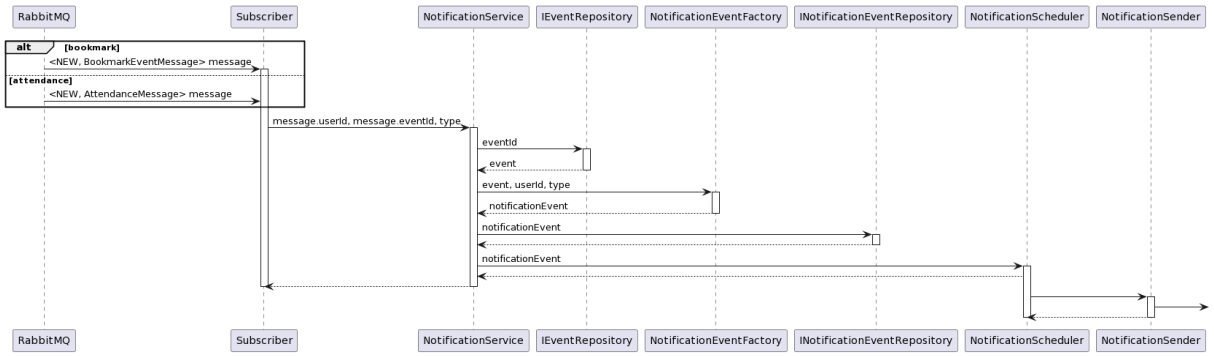
Notification Microservice (Žák, 119222222)

Event update:



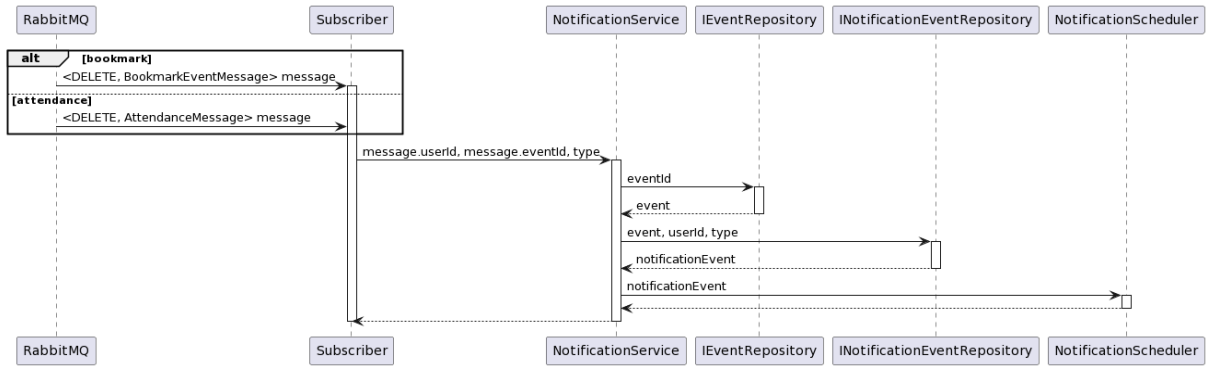
The above interaction is triggered by **RabbitMQ sending a message that an event was updated**. It updates the event copy and sends a notification to all users, which were scheduled to receive a notification about this event.

Register for a notification



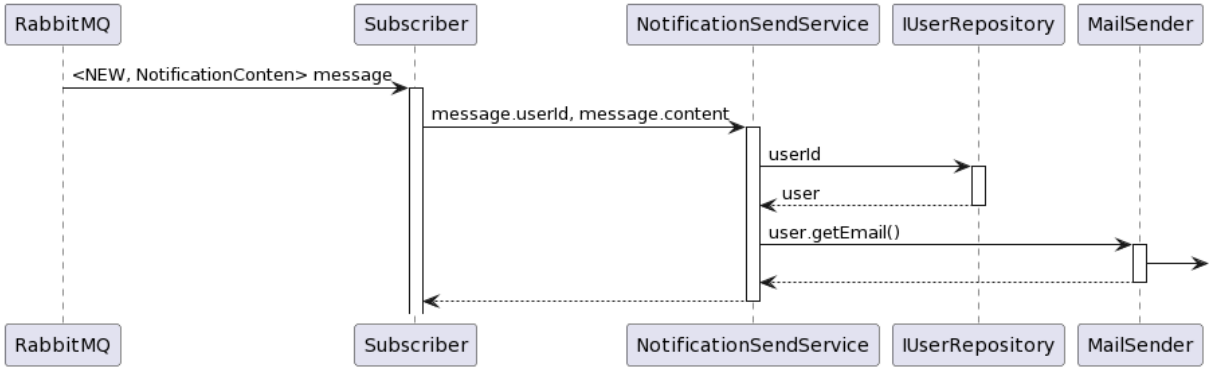
The above interaction is triggered by RabbitMQ sending a message that a new notification should be scheduled. This can be triggered by either a user bookmarking a new event, or by user marking an event as attending. This schedules a notification and sends it at the appropriate time.

Delete a registered notification



This sequence is triggered by RabbitMQ sending a message, that a notification should be unscheduled. This is triggered by a user un-bookmarking or un-markding an event as registered. The result of this is removing the scheduled notification from the system.

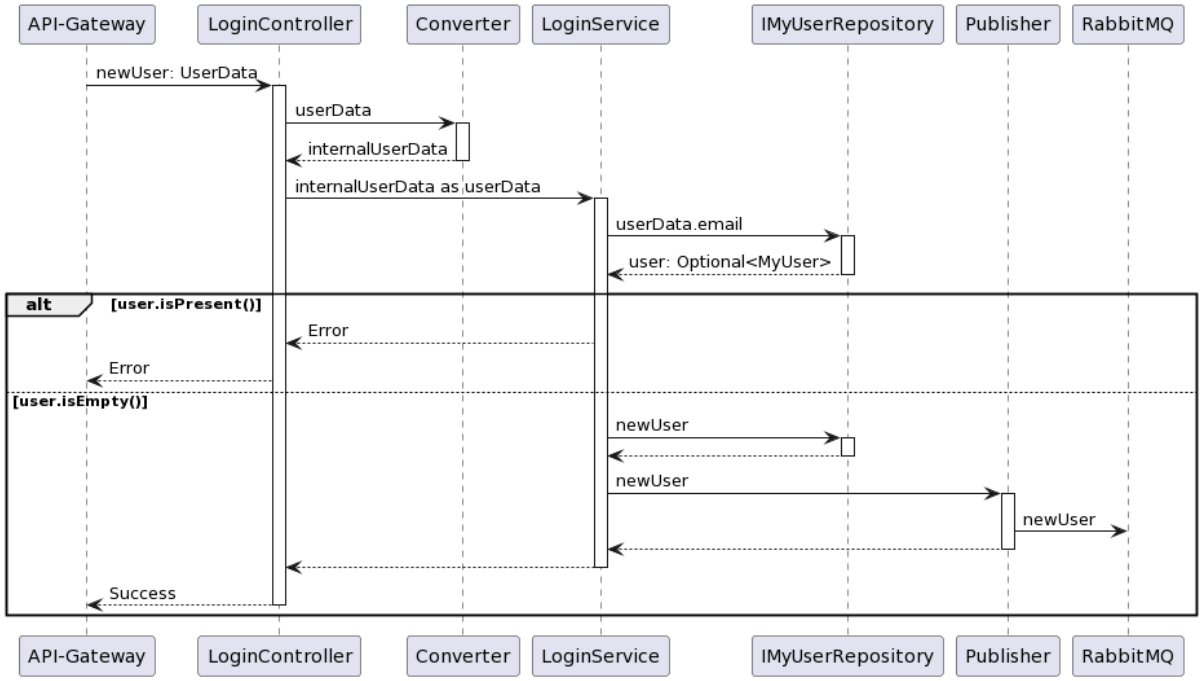
Send notification service



The above diagram highlights the main functionality of the notification service, the sending of notification. It receives notification content, containing a userId and a content. The userId is used to get the associated user and get the email-address of this user. A notification is then sent to the email-address with the specified content.

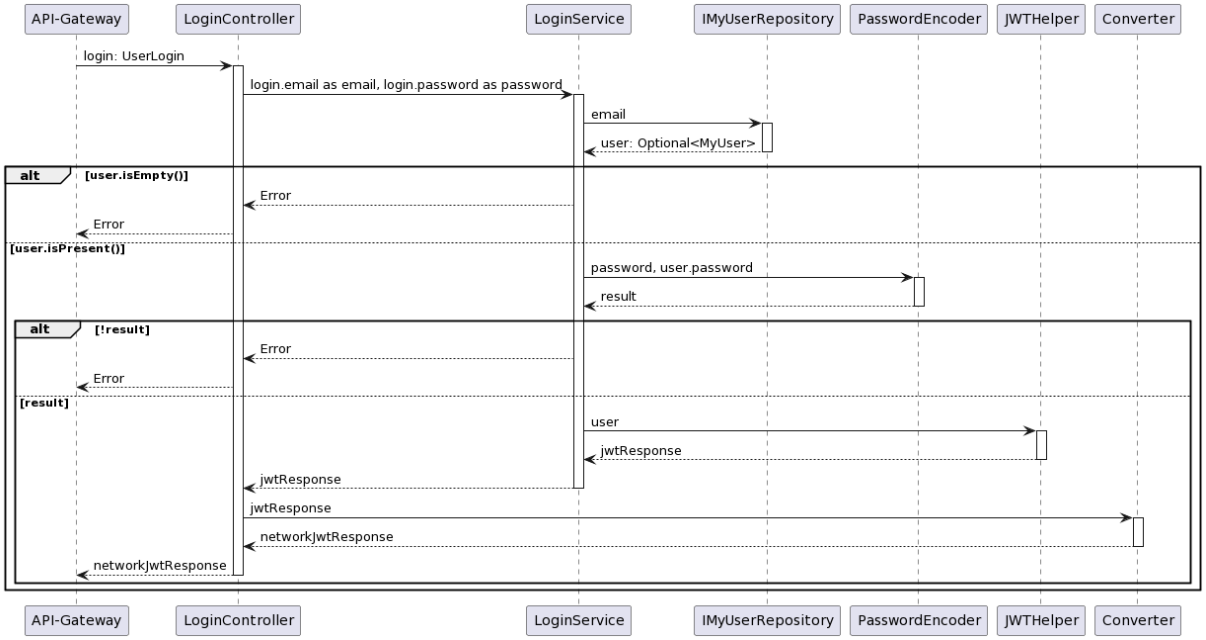
Login Service

Register new user



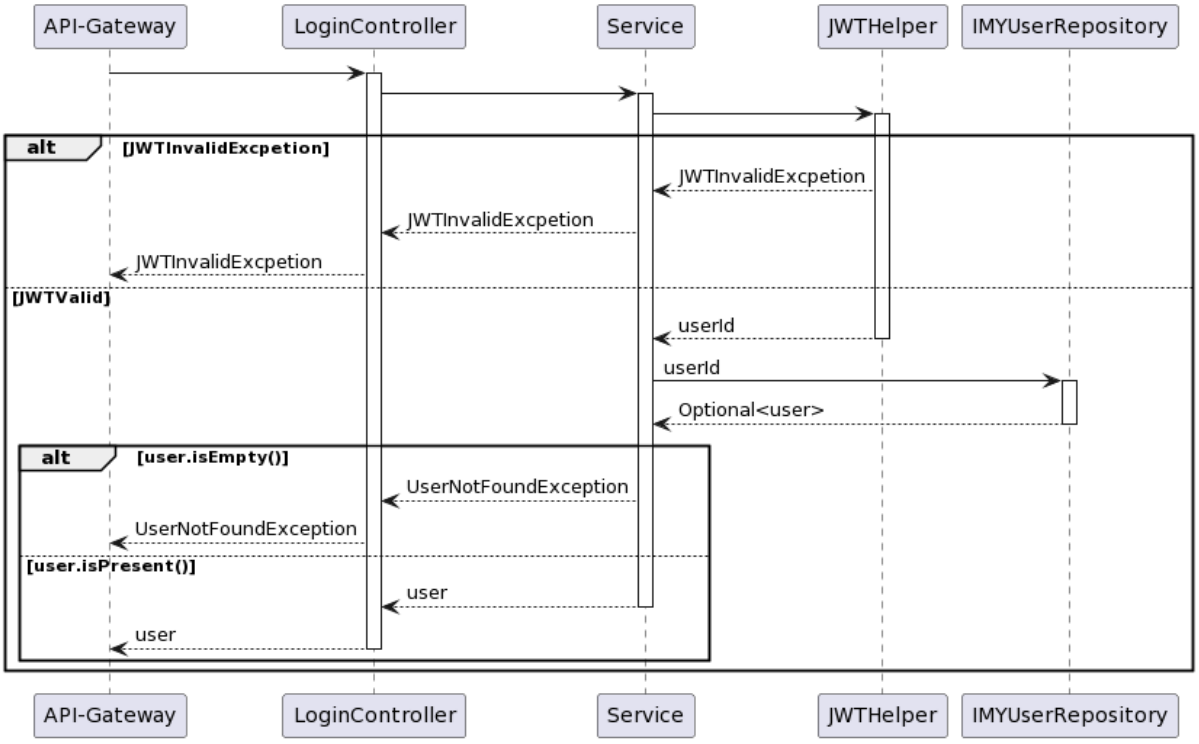
This highlights the sequence of operation when creating a new notification. The API-Gateway sends the userData of the newly created user. It is first checked whether the requested user already exists. If the user does not exist, they are saved into the database and a message is published to RabbitMQ.

Login user



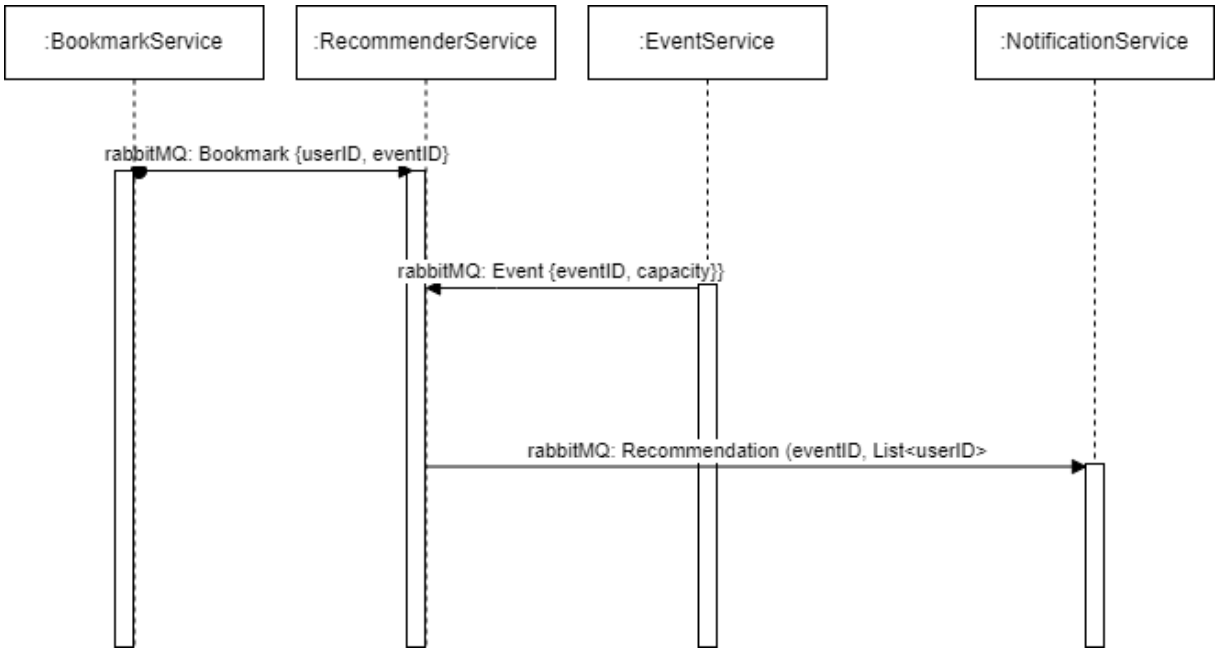
This sequence diagram highlights how a user authenticates (logs-into) our app. This is triggered by the API-Gateway sending a request with user login data. The services then tries to query the email from the database, and get the associated object. If the object is not found an error is returned to the API-Gateway. If a user is present, the provided password is checked against the password of the user. If the passwords match, a JWT is returned, otherwise an Error is thrown.

Get user by token

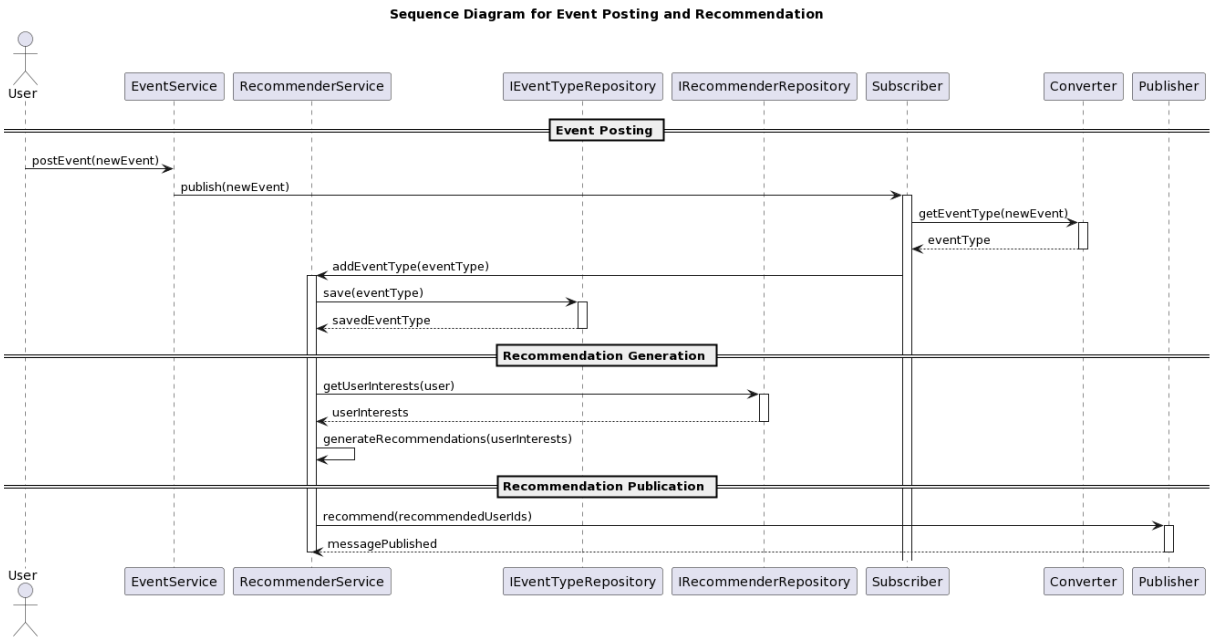


This diagram highlights how a user gets authorised. The API-Gateway sends a JWT into the service. The userId is then extracted from the JWT. This ID is after used to get the full user object from the repository. If everything went right, a user object is returned to the API-Gateway.

Recommender Service (Fabian Schmon/01568351)



The sequence diagram of the recommender service shows the rabbitMQ communication between the recommender service, the bookmarking service, and the notification service. The recommender service subscribes new bookmarks and events and publishes recommendations to the notification service based on its internal filter logic.

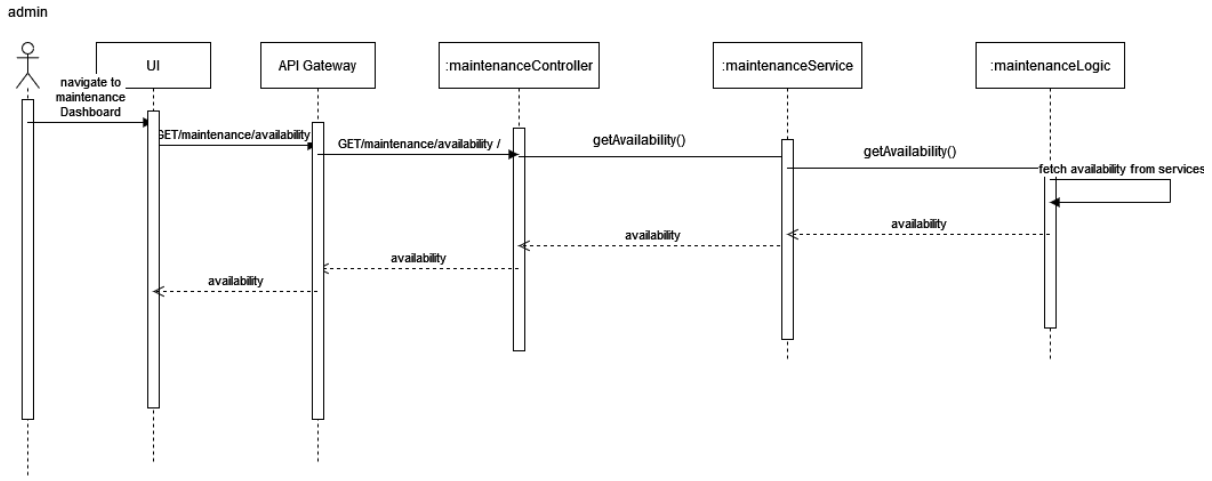


This variation of recommender service sequence diagram depicts the internal service communication involved in the posting of a new event and the subsequent recommendation generation within the system.

- The interaction initiates with a User posting a new event via the EventService.
- Upon receiving the new event, the EventService communicates this event to the Subscriber. The Subscriber, in turn, converts the received Event into an EventType using the Converter.
- This EventType is then passed to the RecommenderService, which stores the EventType information in the EventTypeRepository.
- Following this, the RecommenderService fetches the user's interests from the RecommenderRepository to generate event recommendations based on these interests.
- Once the recommendations are generated, they are sent to the Publisher, which is responsible for publishing these recommendations.

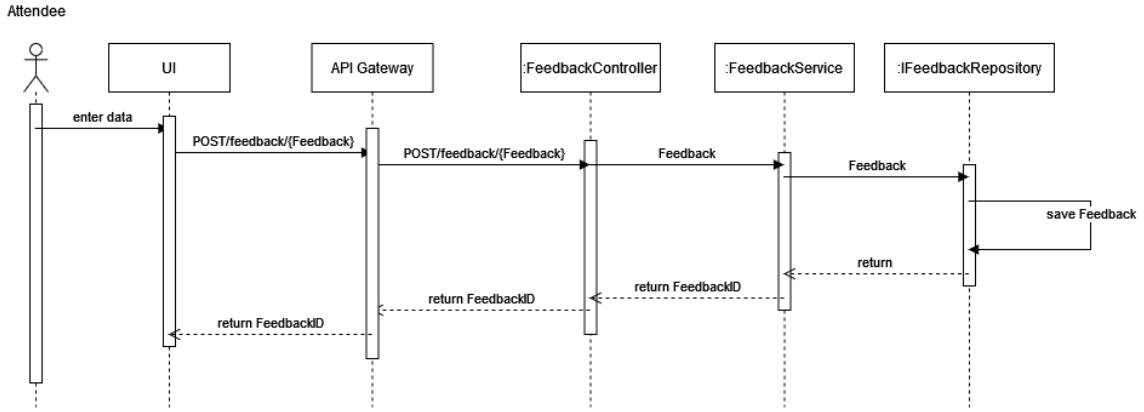
In essence, this sequence diagram displays the flow of data and interactions between different components when a new event is posted and recommendations are subsequently generated and published. It's a reflection of the coordinated workings of multiple services - EventService, Subscriber, Converter, RecommenderService, EventTypeRepository, RecommenderRepository, and Publisher - within the system for successful event posting and recommendation dissemination.

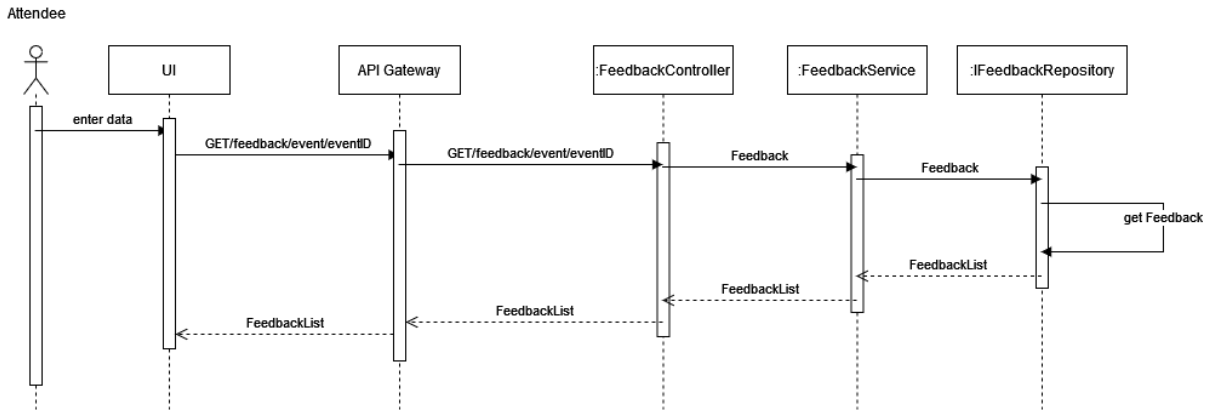
Maintenance Service



This sequence diagram shows the process the system undertakes when the maintenance report is acquired. The only difference to the design is the addition of the api gateway. Instead of accessing the controller directly from the frontend (via REST) the api gateway forwards the request and returns the result.

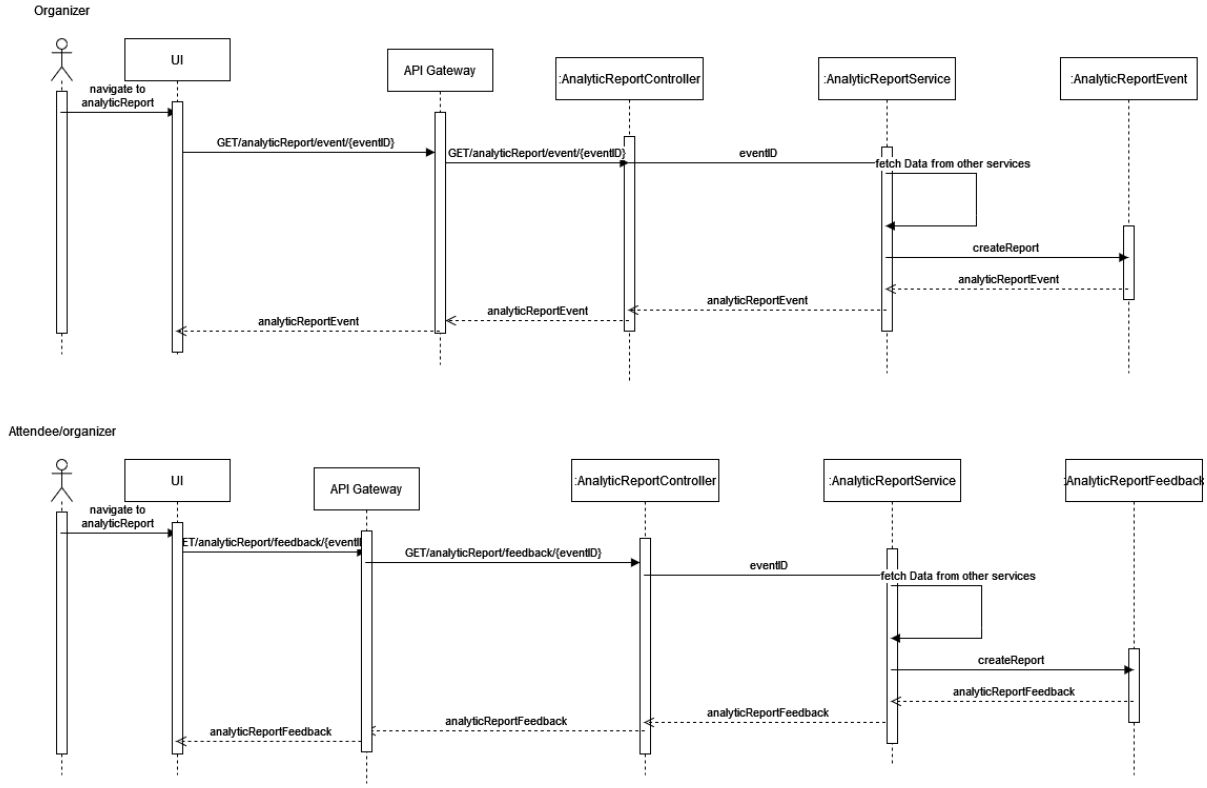
Feedback Service





These two sequence diagrams show how an user can create and fetch feedbacks. The only difference to the design is the addition of the api gateway.

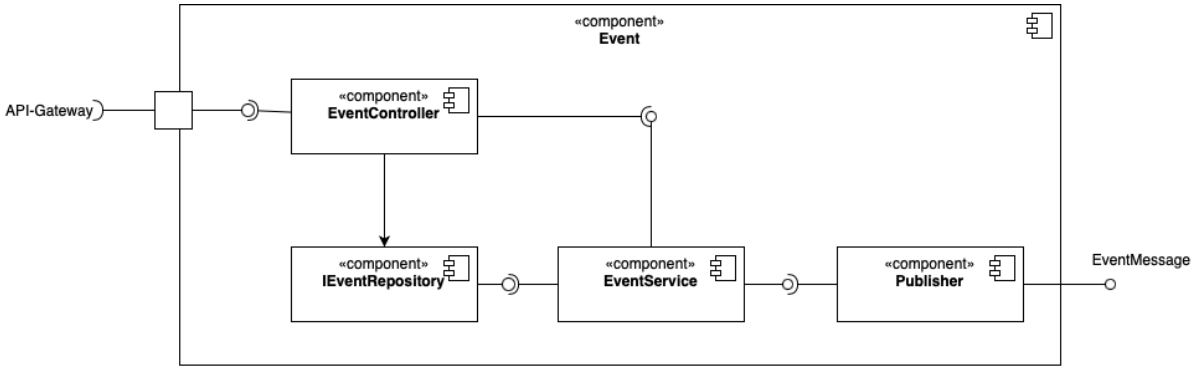
AnalyticReport Service



This sequence diagram shows the acquisition of the analytic reports, one for event and one for feedback. The api gateway is put in between the UI (Frontend) and the controllers again, abstracting the backend from the frontend.

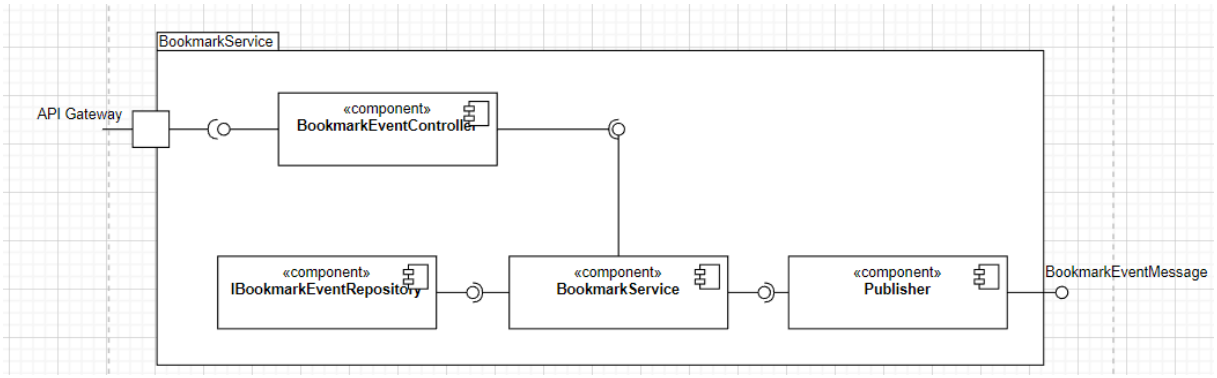
3.4. Development View

Event Microservice (Victoria Zeillinger)



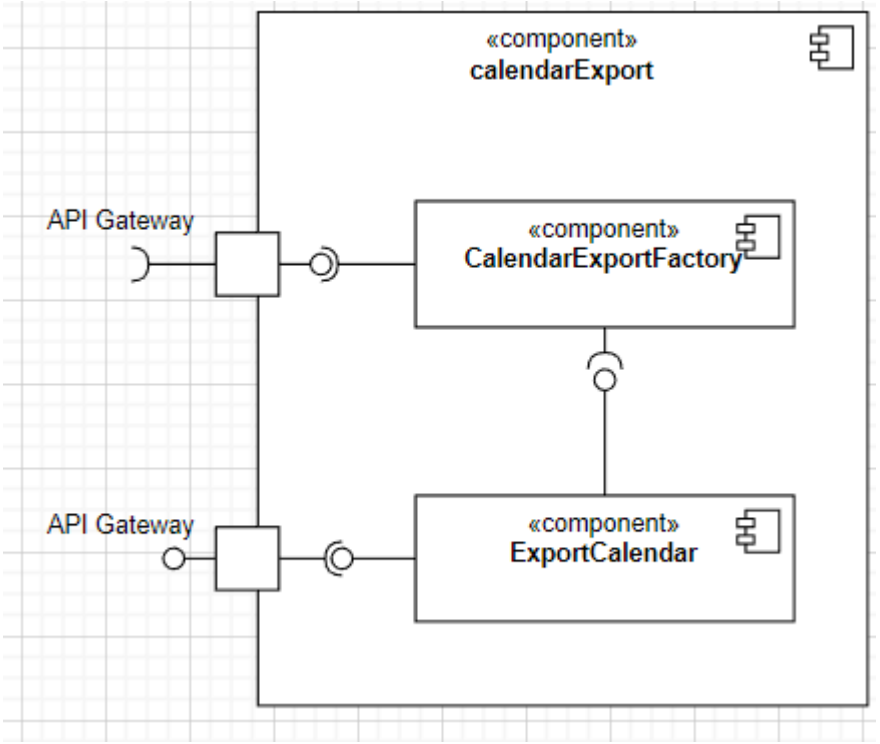
This component diagram shows the event domain. The controller gets a request from the API-Gateway and if an Event is saved, updated or deleted the publisher provides a message to the queue. Several services have relations to the event domain, this is shown in the package diagram in 3.2.

BookmarkService (Alexander Grentner/11743246)



The component diagram shows the bookmarkingEvent, which gets the necessary information via the API-Gateway. Furthermore the BookmarkService publishes and BookmarkEventMessage to possible subscribers.

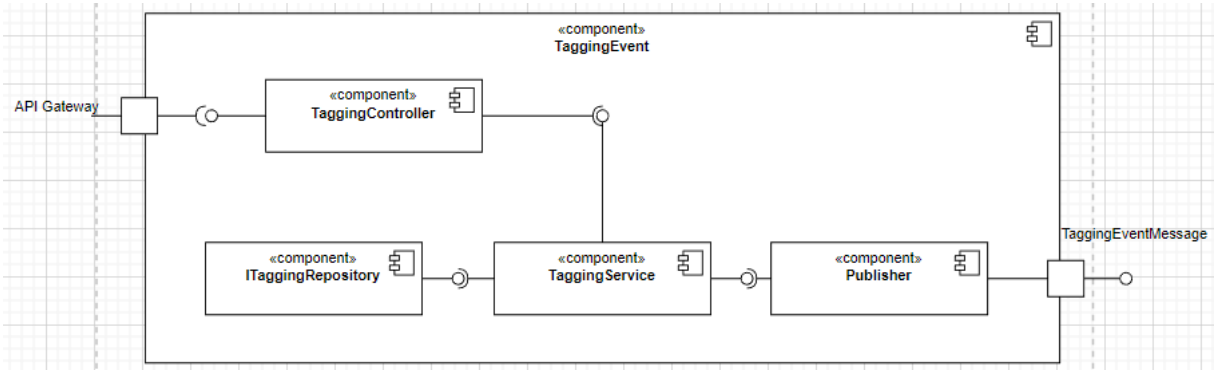
CalendarService (Alexander Grentner/11743246)



The component diagram shows the calendar service which gets calendar data and creates a calendar object in the exportable format. The information for the service is provided via an API-Gateway and will be returned back to the API-Gateway.

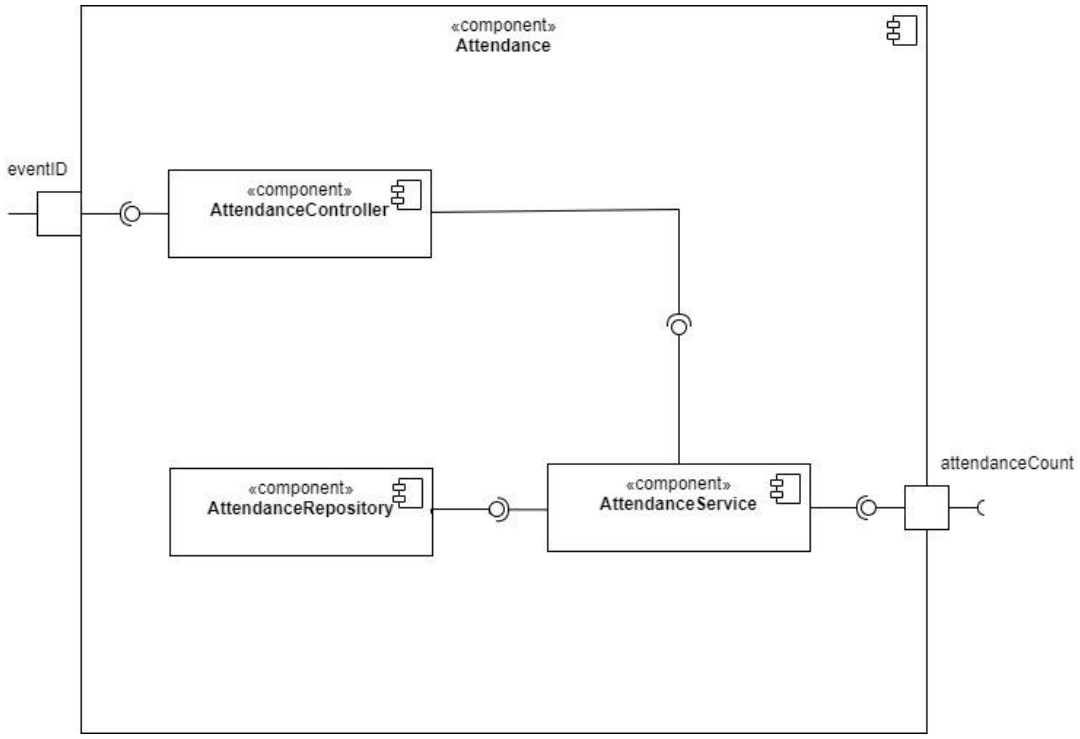
The creation of the exportable calendar with the factory is in this diagram abstracted in one component.

TaggingService (Alexander Grentner/11743246)



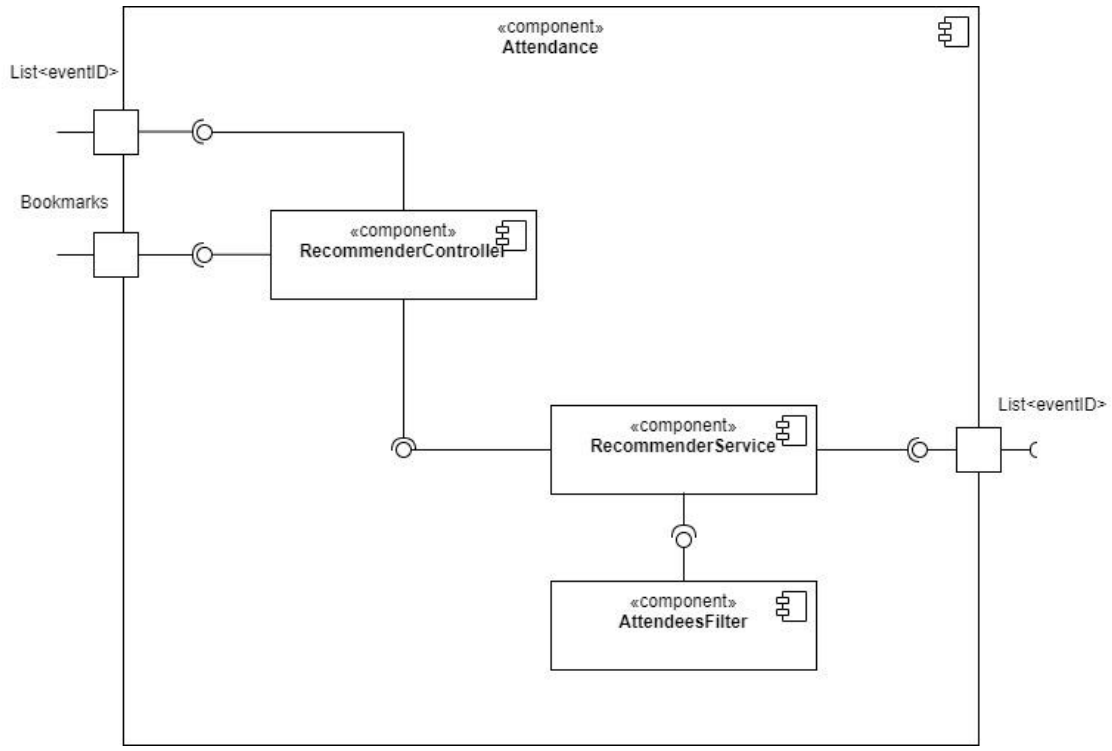
The component diagram gets its information via an API-Gateway, which accesses the endpoint on the controller. The service can then propagate a processed message via a publisher as a TaggingEventMessage to other services.

Attendance Service (Fabian Schmon/01568351)



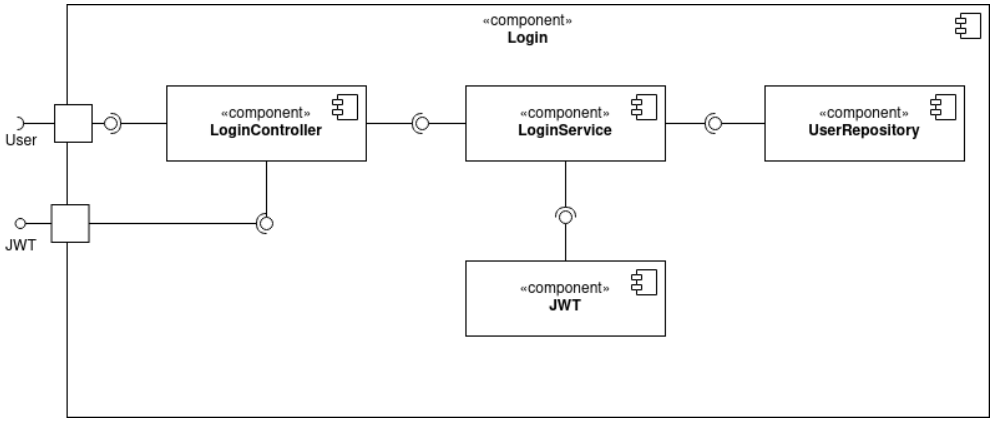
The component diagram of the attendance services shows the three main components controller, repository, and service as well as the input endpoint eventID and the output endpoint attendanceCount.

Recommender Service (Fabian Schmon/01568351)



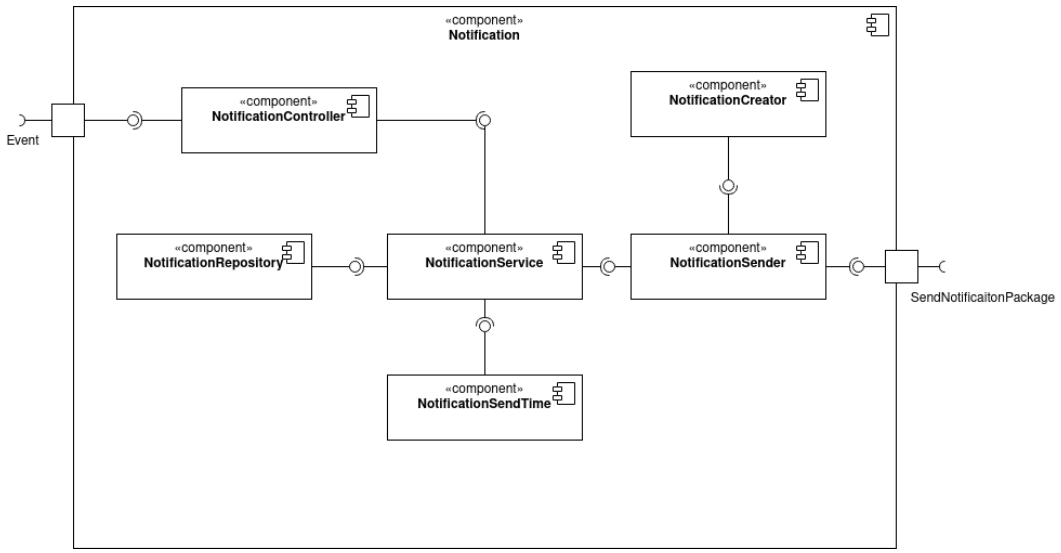
The component diagram of the attendance services shows the three main components controller, service, and filter. The whole attendance “blackbox” is connected by the input variables eventIDs and Bookmarks and the outputs a List of eventIDs

Login Domain



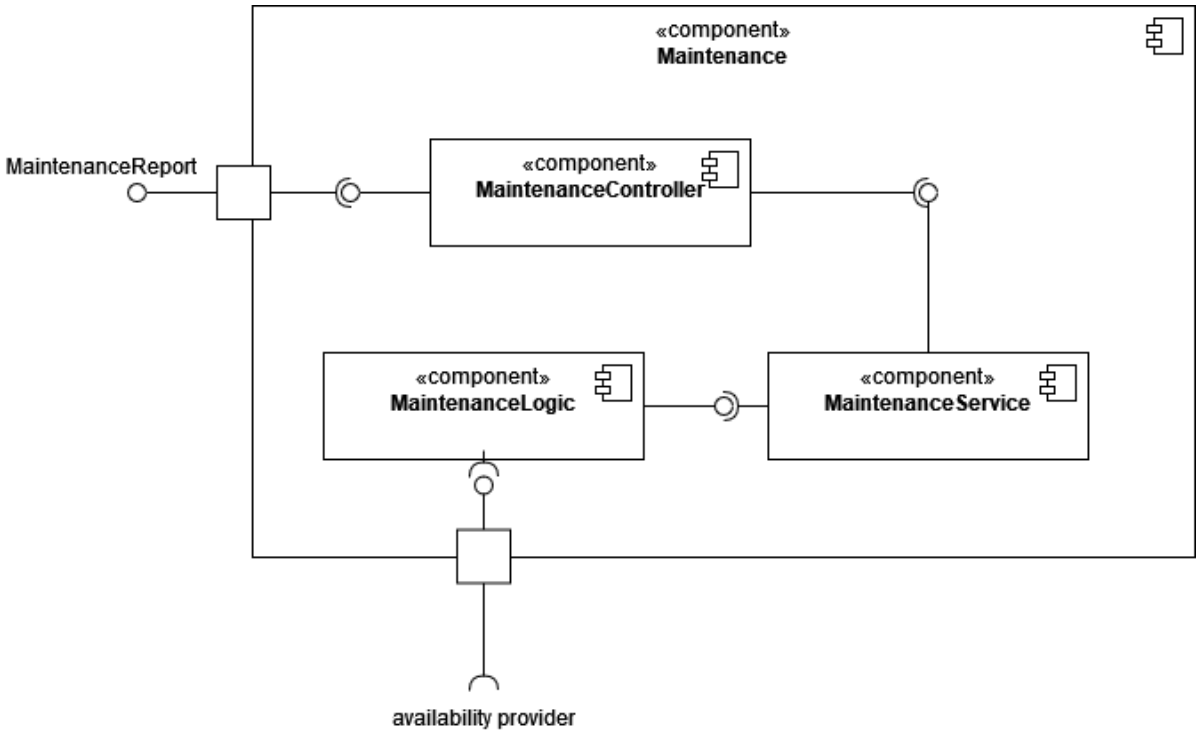
This components diagram shows the structure of the login domain. It needs to receive Users and issue JWT.

Notification Domain



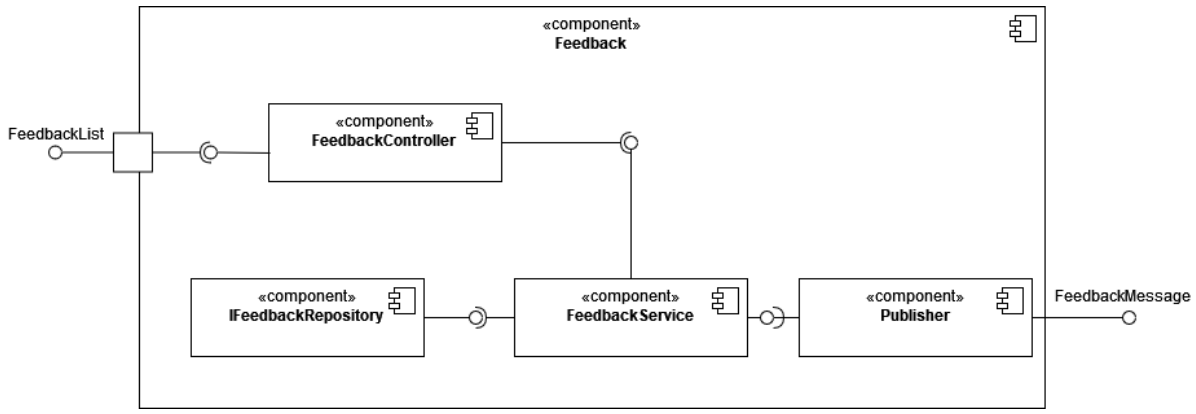
For the notification domain to function it needs an Event, about which will be notified, and a “sendNotificationPackage” (which is responsible for actually sending the notification). The internal connections of the components are also modelled.

Maintenance Domain



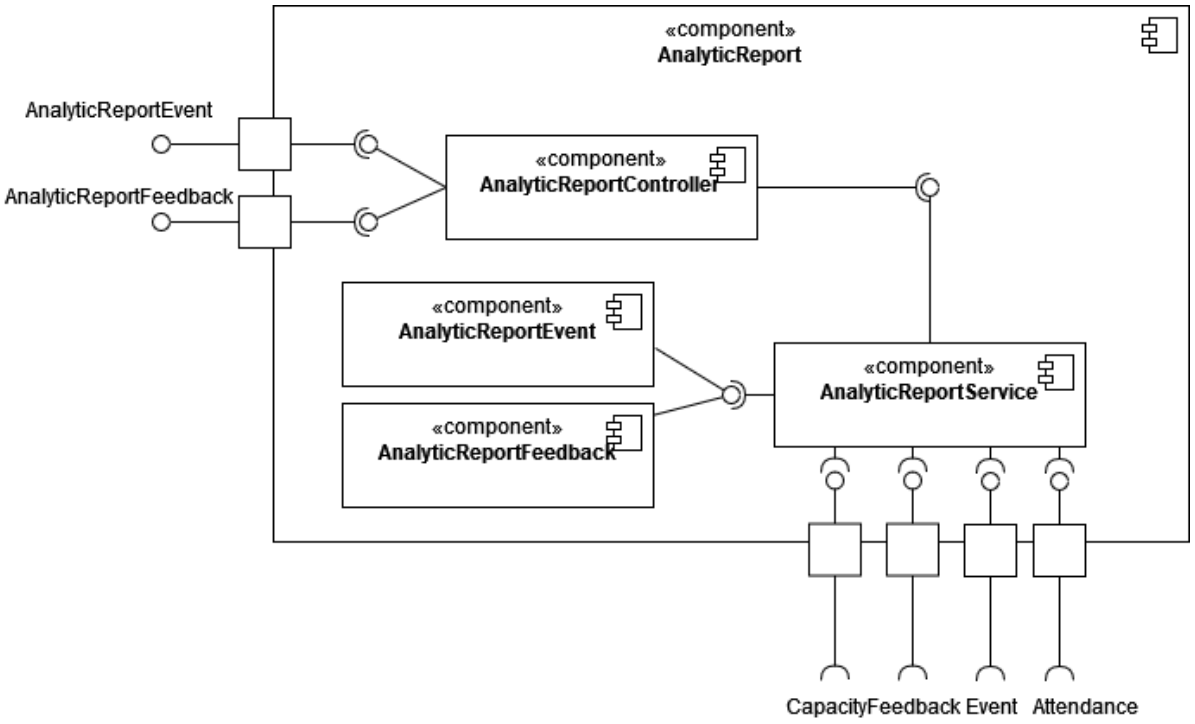
The maintenance domain provides the maintainance report, consisting of the availabilities of all services. Availabilities are provided by the services themselves

Feedback Domain



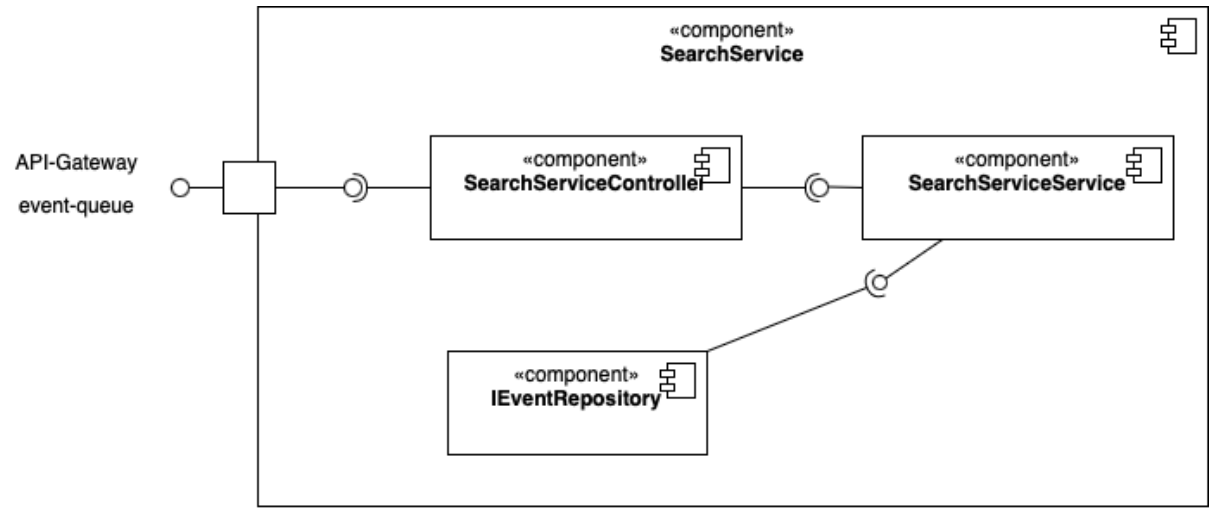
The feedbackdomain provides an interface where feedbacks can be aquired and create. When feedbacks are created a FeedbackMessage is sent to all interested parties

AnalyticReport Domain

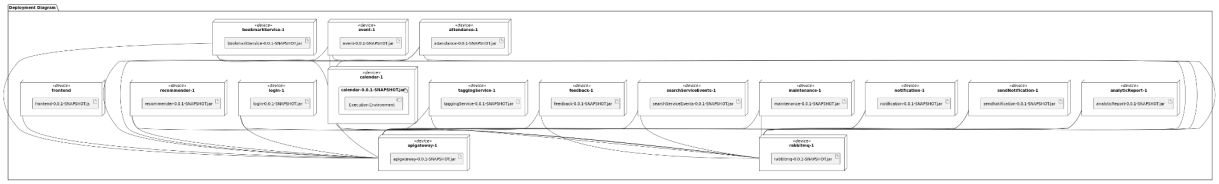


The analytic report provides two itnerfaces to aquire two differen reports. It needs 4 interfaces to acquire the following: event Capacity, Feedback, Event, Attendings.

Search Search Events (Victoria Zeillinger/11809914)



3.5. Physical View



The deployment diagram showcases a microservices-based architecture, with each microservice represented by a node encapsulating a specific application within an execution environment. The services include event management, search, feedback, attendance, login, bookmarking, recommendation, analytics, notification, maintenance, and tagging. All services communicate asynchronously via a message broker, rabbitmq-1, and are exposed to the frontend through the apigateway-1. The frontend acts as the user interface of the system, while apigateway-1 serves as the main entry point for routing incoming requests to the appropriate services. Several services are also directly interconnected, indicating specific dependencies or communication among them.

4. Continuous Delivery

CI/CD and Testing Strategy:

The pipeline consists of three stages: build, test and deploy. The services are thoroughly tested at each stage in combination with the other services. Ideally, the pipeline should be initiated by merging the development branch to facilitate testing, the merging and deployment should only be processed if the pipeline was successful. For improved data management we used an exploration time of one hour. The pipeline should ideally complete the entire process of building, testing, and deployment within this timeframe. Moreover, we utilized the provided runner, which enhances the building process and enables the pipeline to conduct tests in a subsequent stage without the need for rebuilding each microservice. This description is transferable to every microservice, as the pipeline works the same.

SR4 event microservice (Victoria Zeillinger/11809914)

The event microservice is built by executing the corresponding maven command within the script. To ensure comprehensive testing, local tests are executed and checked for successful completion. Testing was conducted by using integration-tests and unit-tests for testing the functionality of the presentation layer (controller endpoints) and the business layer (service) in combination with the database. Further the integration layer (publisher) is also mock-tested.

SR5 search-service-event microservice (VictoriaZeillinger/11809914)

The search-service-event microservice is built by executing the corresponding maven command within the script. To ensure comprehensive testing, local tests are executed and checked for successful completion. Testing was conducted by using integration-tests and unit-tests for testing the functionality of the presentation layer (controller endpoints) and the business layer (service) in combination with the database.

SR6 bookmarkService and taggingService (Alexander Grentner/11743246)

The bookmark and tagging service is built by executing the corresponding Maven command within the script. To ensure comprehensive testing, local tests are executed and checked for successful completion.

Testing was conducted by using integration tests and unittest of the bookmarkService and the taggingService. The unittests test each functionality of the services in combination with the database, mainly focused on the business logic. The integration tests mainly focus on the controller endpoints and the process within the overall project.

SR7 Attendance Service (Fabian Schmon/01568351)

The Attendance Service includes Java Unit Tests and all endpoints were tested with POSTMAN scripts.

SR8 feedbackService (David Kreisler/)

The feedbackService mainly conduct unittests for the businesslogic

SR9 calendar (Alexander Grentner/11743246)

The calendar is built by executing the corresponding Maven command within the script. To ensure comprehensive testing, local tests are executed and checked for successful completion.

Testing was conducted by using integration tests and unittest of the calendar. The unittests. Integration Tests of was only required for one used endpoint, which mocks data of the necessary REST calls to other services. The calendarservice does not persist data which therefore does only require unittest regarding the business logic in transferring the mocked calendar data within the tests.

SR10 analyticReport (David Kreisler/)

Testing was done using unit tests for the business logic.

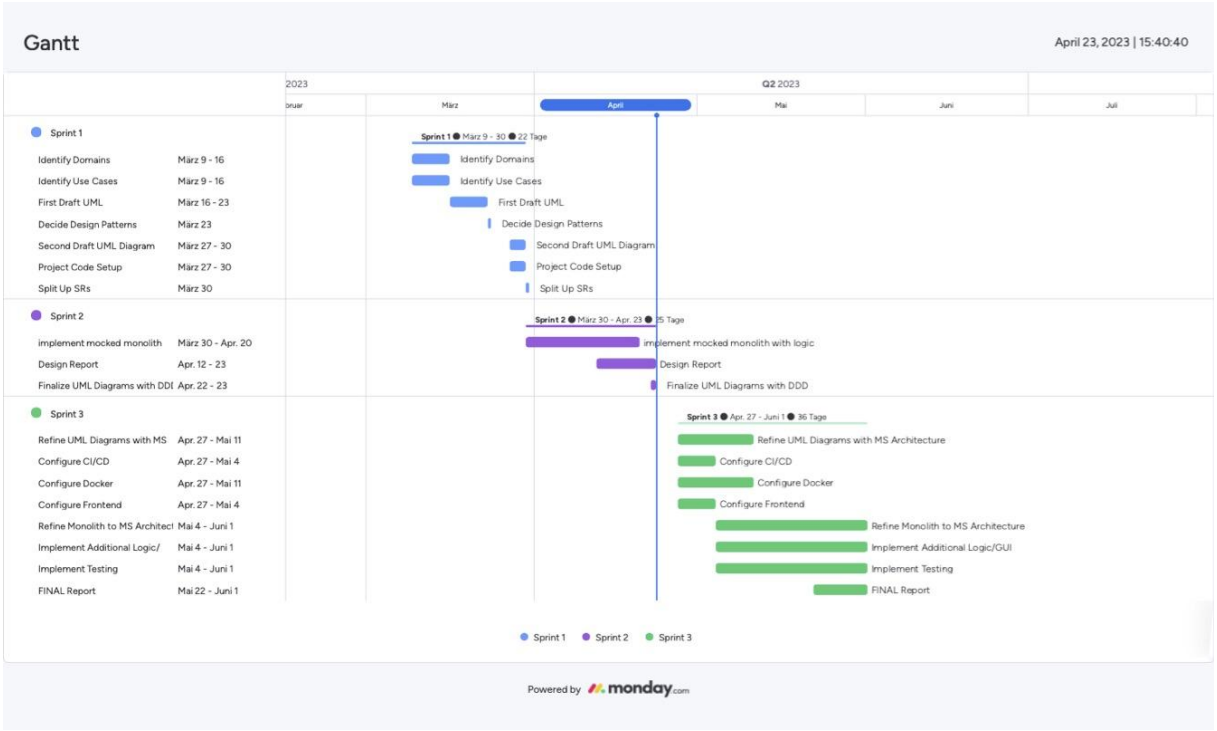
SR11 Recommender Service (Fabian Schmon/01568351)

The Recommender Service includes Java Unit Tests and all rabbitMQ messages were triggered and tested with POSTMAN scripts.

5. Team Contribution and Continuous development method

Overall, the workload was equally distributed on the project and on the infrastructure setup. Although, the commit history might lead to another assumption because of the inconsistent and undistributed number of commits. The reason is that the setup of the CI/CD pipeline was a trial and error process, which required to commit a new gitlab-ci.yml file to test a functional pipeline. The number of git commits are therefore not a valid parameter to measure work distribution within the team.

5.1. Project Tasks and Schedule



The Gantt chart above shows the main tasks of the semester projects. We decided to structure the tasks in three sprints because in the initial phase of the project, it became clear that we worked with an agile project management model. The first sprint mainly consists of the design decisions, based on understanding the project scope and the domains, as well as the setup of our IDE. The second sprint consists of the implementation process of the monolith and the writing of the design report. The design report isn't divided into more details, because it would go beyond the scope of this chart. The third sprint consists of the refining and adaptation process from the monolith into a microservice (MS) architecture as well as adding the GUI, the CI/CD and completing the project. The writing of the FINAL Report is the last task of sprint 3 and as explained above it's also not split up into more detail.

5.2. Distribution of Work and Efforts

Contribution of Member 1:

Alexander Grentner/11743246:

- SR6: bookmarkService, taggingService
- SR9: calendar
- CI/CD initial setup
- Deployment Diagram

estimated time: 240 hours

Contribution of Member 2:

David Kreisler/01569177:

- SR3: maintenance service (split from login service, because the workload of this SR was too big)
- SR8: feedback
- SR10: analytic report

estimated time: 240 hours

Contribution of Member 3:

Fabian Schmon/

- SR7: recommender service
- SR11: attendance service
- Deployment Diagram

estimated time: 240 hours

Contribution of Member 4:

Victoria Zeillinger/11809914

- SR4: Event Domain with Event Inventory
- SR5: Search Service
- Frontend setup

estimated time: 240 hours

Contribution of Member 5:

Michal Zak/

- SR3: Login Service (split from maintenance service, because the workload of this SR was too big)
- SR12: NotificationService
- RabbitMQ setup

- API-Gateway
- CI/CD helping

estimated time: 240 hours

Summary SRs and Team Members

SR	Team-Member
SR1	joint
SR2	joint
SR3	Michal Zak, David Kreisler (divided into User Authentication and Login from Zak and Maintenance Service from Kreisler
SR4	Victoria Zeillinger
SR5	Victoria Zeillinger
SR6	Alexander Grentner
SR7	Fabian Schmon
SR8	David Kreisler
SR9	Alexander Grentner
SR10	David Kreisler
SR11	Fabian Schmon
SR12	Michal Zak

Although the workload was divided evenly, the git-commits from each team member do not reflect it. There were some issues with commits that weren't squashed, so the commit-numbers vary strongly.

6. How-to documentation

How to start the project:

To run the application you have to do the following steps in the command line in the folder ./implementation:

- docker-compose build (Can take up to 30 minutes)
- docker-compose up

The dockerfile doesn't have any requirements and dependencies, but you have to run docker on your desktop.

Then you should open a browser of your choice and type in 'localhost:4200' where you should be directed to the login page of the frontend. You can click "register" to sign up with a new user and then you should login with your newly saved credentials.

CI-CD TEAM WIDE

As the ci-cd is run at every commit, we have decided to combine this section into one in this document.

The pipeline has two stages:

- build -> builds all of the applications
- test -> tests all of the applications

We have considered a third, deploy stage, which would test the deployment of the docker-compose, but after letting this file build the jars and not use the jars from the previous stages, the building took too long and would time out the pipeline.

Victoria Zeillinger/11809914

As the command-line client was nowhere mentioned in the assignment sheet, and therefore was not implemented because of lack of knowledge, here is a way to run the tests of event and searchServiceEvents locally. You have to open the command line and navigate to their folders ../event ../searchServiceEvents. There you can run the command "mvn test" and it will run the test files. For Events there are Integration tests, which test the Endpoint Controller, Unit Tests, which test the business logic service and publisher tests which mock-test the rabbitMQ publisher queue. If all tests passed, you will get "tests-passed" message. If tests fail, you will get a "tests-fail" message.

To deploy the project, you can execute the instructions of "How to start the project" above. Keep in mind that each service, Event and SearchService is containerized with each a dockerfile (you can view it in each folder of the microservices). These dockerfiles are then executed while running docker-compose. The docker-compose.yaml has a section for every microservice, for event and searchServiceEvents it's those two:

```
event:

 image: event

 build:
```

```
    context: .
    dockerfile: event/Dockerfile
  networks:
    event-rabbit:
    event-apigateway:
    event-calendar:
  environment:
    <<: [ *common-variables, *endpoint-variables ]
  volumes:
    - ./_persistence/event:/persistence
```

```
searchServiceEvents:
  image: searchserviceevents
  build:
    context: .
    dockerfile: searchServiceEvents/Dockerfile
  networks:
    searchServiceEvent-rabbit:
    searchServiceEvent-apigateway:
  environment:
    <<: [ *common-variables, *endpoint-variables ]
  volumes:
    - ./_persistence/searchService:/persistence
```

They define the networks (which are also in the docker-compose), build an image and persist the database for this service.

Alexander Grentner/11743246

Running the test was mainly conducted by merging feature branches to ensure code quality throughout the development process. The tests were conducted by using the “mvn test” command, as well as the “mvn build” within the project. The command line client was not mentioned in the assignment sheet and was therefore not a priority of the project. Additionally, local tests with other services were conducted with postman or fullstack tests. The containerization was implemented by docker and executed on the container. By using the socker command docker-compose build, docker-compose up and docker-compose down, depending on the process. The docker compose file is provided for each microservice. Additionally, fullstack tests were used with a local environment and the docker environment, depending on the wanted concern of the test in combination with the frontend. Tests in the pipeline were mainly conducted by mvn and also integrated as this in the stage and building blocks of the CI/CD, additionally to the the building.

In the following the example shows the main part in the docker compose, with its dependency to build the docker image within the application by using docker-compose.

```
taggingService:
  image: taggingservice
  build:
    context: .
    dockerfile: taggingService/Dockerfile
  environment:
    <<: *common-variables
  networks:
    taggingService-rabbit:
    taggingService-apigateway:
  volumes:
    - ./_persistence/DB-taggingService:/persistence
```

```
bookmarkService:
  image: bookmarkservice
```

```
  build:
    context: .
    dockerfile: bookmarkService/Dockerfile
  environment:
    <<: [*common-variables,*rabbitmq-exchanges]
  networks:
    bookmarkService-calendar:
    bookmarkService-rabbit:
    bookmarkService-apigateway:
  volumes:
    - ./_persistence/DB-bookmarkService:/persistence
```

```
calendar:
  image: calendar
  build:
    context: .
    dockerfile: calendar/Dockerfile
  environment:
    <<: *common-variables
  networks:
    bookmarkService-calendar:
    event-calendar:
    attendance-calendar:
    calendar-apigateway:
```

Each microservice can be run via docker by using the docker file. The concept is for each microservice individually the same. A jar file must be created by using mvn install. The microservice can be run individually with the docker by building an Image if necessary and running the application from the docker file.

Fabian Schmon/01568351

Every major change in the attendance service and recommender service was tested on the CI/CD pipeline and Java Unit tests with mock-ups. A continuous evaluation of all API endpoints was critical because both services rely highly on other services like event, bookmark and notification service All endpoints can easily accessed and tested through POSTMAN with the paths:

- http://localhost:8080/api/v1/attendance

for example:

/register/{userID}/{eventID}

/deregister/{userID}/{eventID}

- http://localhost:8080/api/v1/recommender

The testing could be further improved by implementing comprehensive integration tests.

The following script shows the main part of the docker-compose.yml that is needed to correctly setup the attendance and recommender services:

attendance:

image: attendance

build:

context: .

dockerfile: attendance/Dockerfile

environment:

<<: *common-variables

networks:

attendance-rabbit:

attendance-apigateway:

attendance-calendar:

```
recommender:
    image: recommender
    build:
    context: .
    dockerfile: recommender/Dockerfile
    networks:
  recommender-rabbit:
  recommender-apigateway:
```

Michal Zak/11922222

As the command-line client was nowhere mentioned in the assignment sheet we do not provide an implementation of it.

My microservices can be found under the path:

- implementation/sendNotification
- implementation/notification
- implementation/login

To run the tests install maven and run mvn test.

Ideally you would execute the microservices from docker-compose (consider the requirement was to provide a docker-compose which builds the entire app and not one which would be able to set up a microservice each. As ALL of my microservices depend on at least rabbitmq running, you cannot really just run them independently). If you still want to run each microservice individually, you can execute the associated Dockerfile and forward port 8080. Potentially also run docker run -it --rm --name rabbitmq -p 5672:5672 -p 15672:15672 rabbitmq:3.11-management to start rabbitmq.

David Kreisler / 01569177

No command line client was implemented, since it was not mentioned in the assignment. Microservices can be found under implementation/maintenance, implementation/feedback, implementation/analyticReport. To run the tests install maven and run mvn test. Additionally testing can be performed using Postman. Each microservice runs in their own docker container and can be started independently.