Advanced Software Engineering
DESIGN 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. Design Draft

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:

- The app needs to be used by multiple different users
- 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
- The computational complexity of the app is not high enough to warrant a native app
- 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 analyzing the provided domain description, asking stakeholders (tutors) about the problem domain, and also formalizing 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 crude 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 modeling, 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 visualize the coupling of the app.

During modeling, we stumbled upon several issues where we were unsure of the best design and modeling 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 familiarize 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 formalized into the 4+1 View model.

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

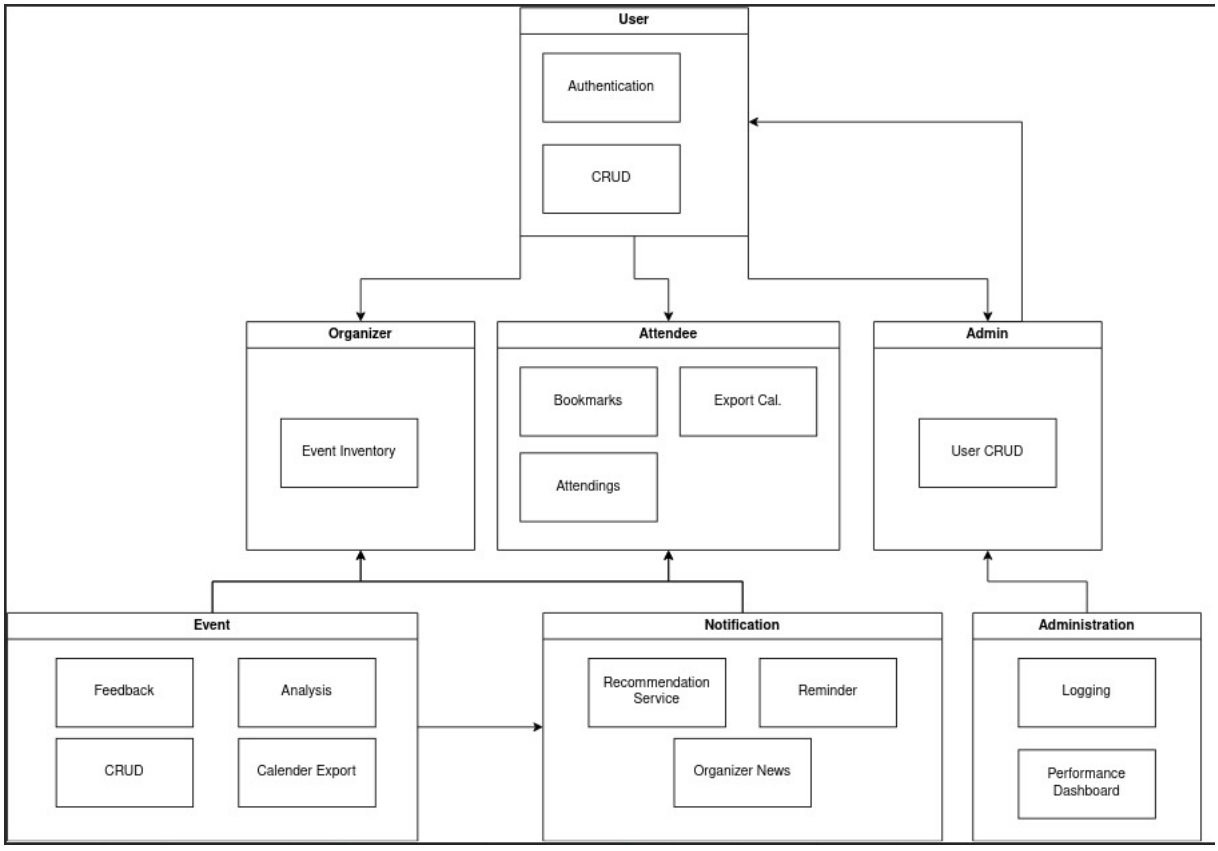
As the problem domain was very open-ended, certain assumptions had to be made to clearly define the scope and delimit the use cases.

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.

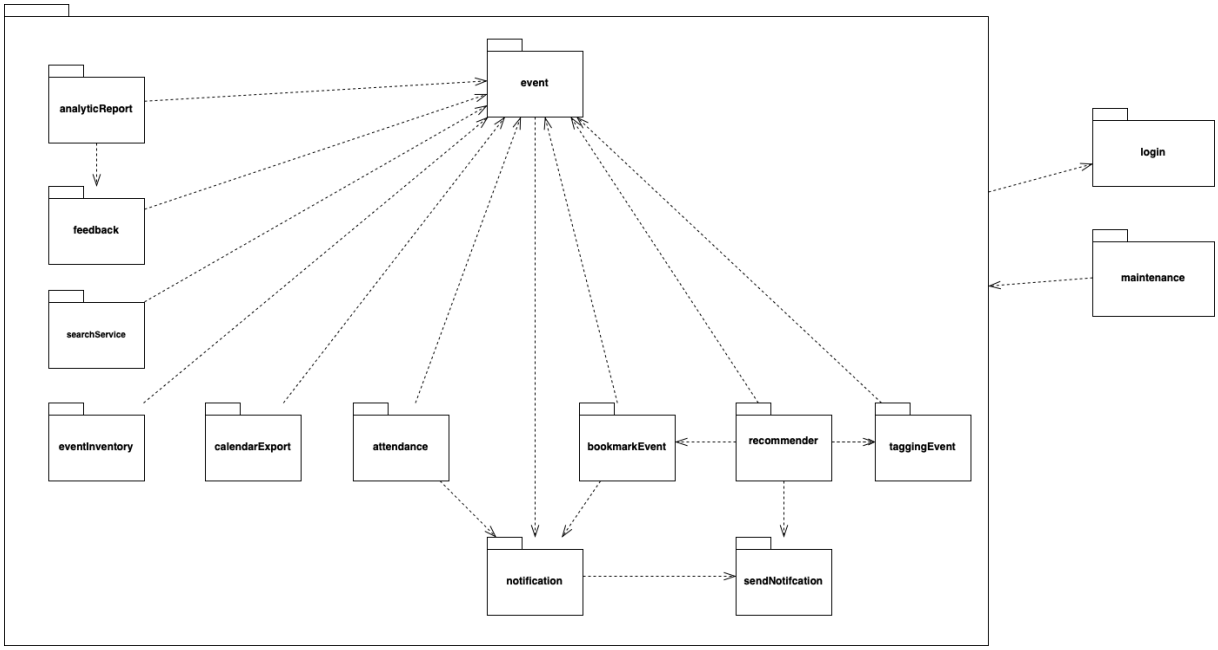
3. The Event-Search-Service, which should list all saved events and provide filters for searching, were assumed to be independent of the role of the users. So, attendees and organizers can all see every event from every organizer and search with different filters. On the one hand, it leaves less room 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 criterion 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

At the beginning of the design phase, we divided the whole project scope into seven domains. The first draft is shown in the following diagram. The seven main domains had subdomains corresponding to the services.



We found that this structure would result in highly coupled services, which we decided not to do. Instead, after some iterations, we used some subdomains as domains and merged others. This way the services can work more independently if the domains are separated and only save relations and not the whole objects from the other domains. The changes made and the resulting diagram is shown below.



The following changes were made to the domains so that they can fulfill their requirements and be independent.

- The Event Inventory was a subdomain of the organizer, but during the design process, it became clear that Event Inventory should be in its own domain which interacts with Events and Organizers individually. It saves the relations between an event and an organizer, but it does not save the event object because it would be redundant. The decision was made while keeping in mind that we will need individual microservices in the next phase.
- The registration/creation of users was relocated from the User domain to the Authentication Service. The User Domain became an abstract class that inherits to organizer, attendee, and admin.
- The feedback subdomain was taken away from the event domain and was made a domain itself. It contains information about the feedback and relations to the event and user domains.
- The analysis subdomain was renamed to analytic report and made a domain as well. It stores no information, it requests information from the event and feedback domain and prepares it to show to the organizer and attendee.
- The event domain now only contains the information about the event itself.
- The Recommender Service is its own domain now but is using the Send Notification Service from the Notification domain. This Service only sends an email to an email address that is given to the service.
- To register attendees to an event, there is an Attendance Service Domain which saves relations between the user and the event.
- The administration domain got replaced by the maintenance service domain. It is responsible for giving admin analytic information about the services. It has a connection to every service but does not store any data.

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 modeled 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.

Our Calendar Service has been implemented without the need for a persistent database. This service offers users from other domains the ability to create and convert existing calendar objects into an exportable format. Currently, these functionalities do not require persistent storage during the implementation phase.

However, this may change depending on the workflow and data provided for the calendar export.

We wanted to give the registered (= attending, see assumption 4) users the possibility to on the one hand give feedback about an event in a minimal amount of time, and on the other hand share their detailed opinion about the event if they wish to do so. Therefore we decided to provide 3 rating options. One for the location, one for the food, and one for the event overall. Additionally, we added the possibility to leave a comment of any length.

We decided to enable two types of analytic reports. One report was concerned with the feedback and rating of the event and the other shared information about the number of registerings, bookmarkings, and other information regarding the event. The first report can be accessed by all users, while the second one is only available for organizers.

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.

The Event Inventory is an extra domain besides the event domain, which only saves the relation between the inventory and the organizer. If an organizer creates their first event, their event inventory is automatically created.

For this version, we decided to implement a mocked backend but no GUI. Instead, we used Postman to test and mock the functions of the systems because the GUI would have been very simple with very few designs and would have had the same function as we can use in Postman.

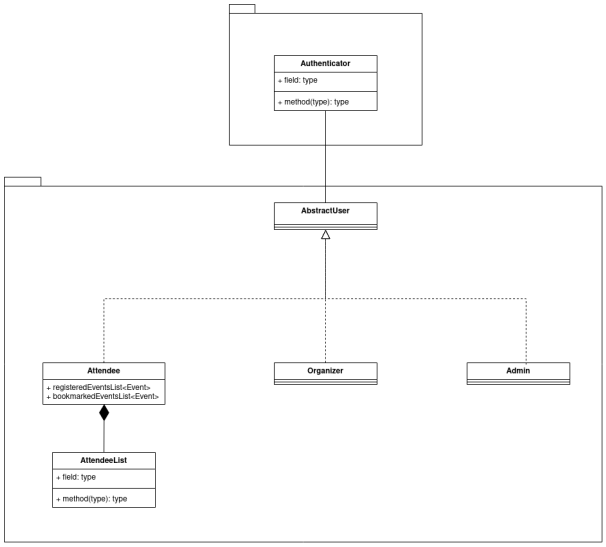
We have opted to predefined tags to facilitate efficient processing searches and recommendation services. This approach allows users to utilize pre-existing tags, thereby enabling the recommender service to more efficiently process and aggregate the tags compared to customized tags. The use case that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

The User will be able to control the final product by selecting buttons and by typing values in textboxes, for example, to add Event Information when creating a new Event. The user interface will be presented by the web app and consists of an HTML page that can be loaded on any common browser. The webpage will enable the user to interact with all microservices that are needed for fulfilling all the use cases like searching for Events and register to them, giving feedback, or exporting the personalized calendar.

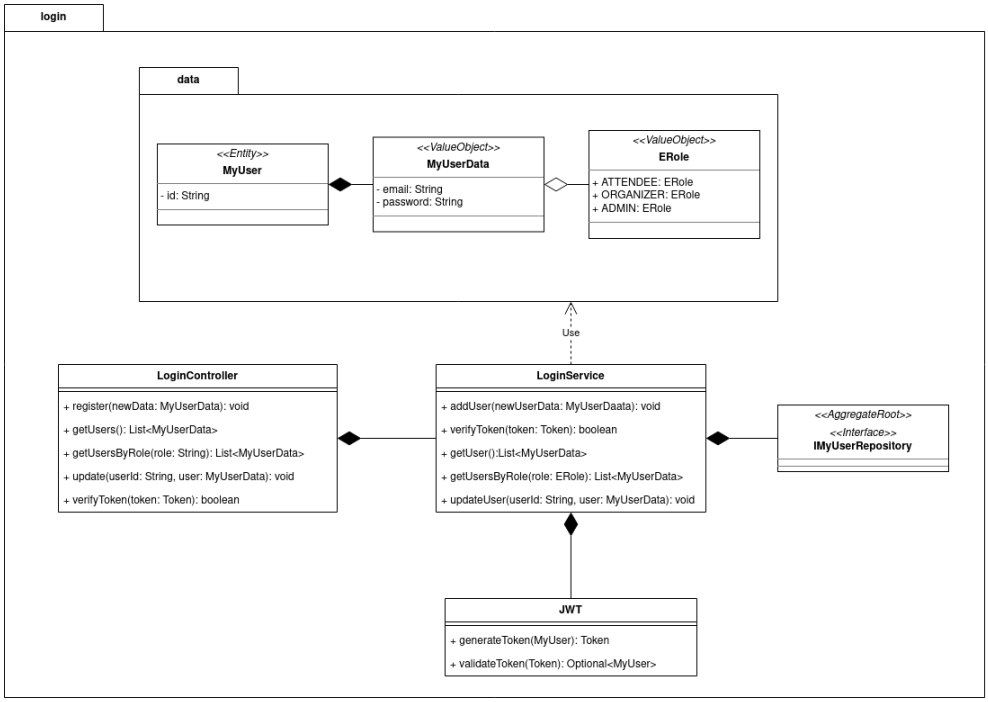
Subdomains Design Process

Login Service

In our first iterations of the design we did not consider the login service as is. The functionality, which was later (mostly) combined into the login service is modeled in the diagrams below. Initially, we wanted to have a whole service determined to serve users and some of their related data. After investigating OAuth2 and other security strategies, we have decided that this may not be the best solution to our problem.

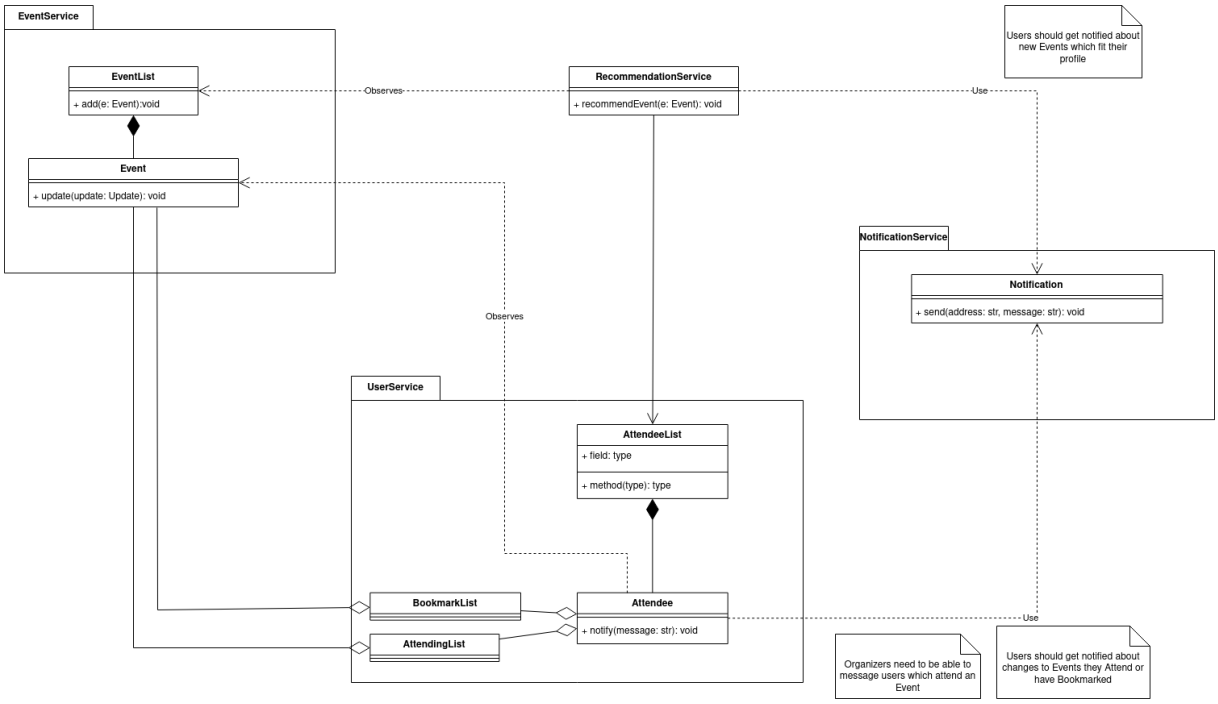


In the current iteration of the service, we have tried to put the authentication and authorization to the front, instead of the users like in the previous diagram. The service is still responsible for serving the users, but now it only saves the most relevant information of a user and delegates associated information, such as the attendee list to other services.

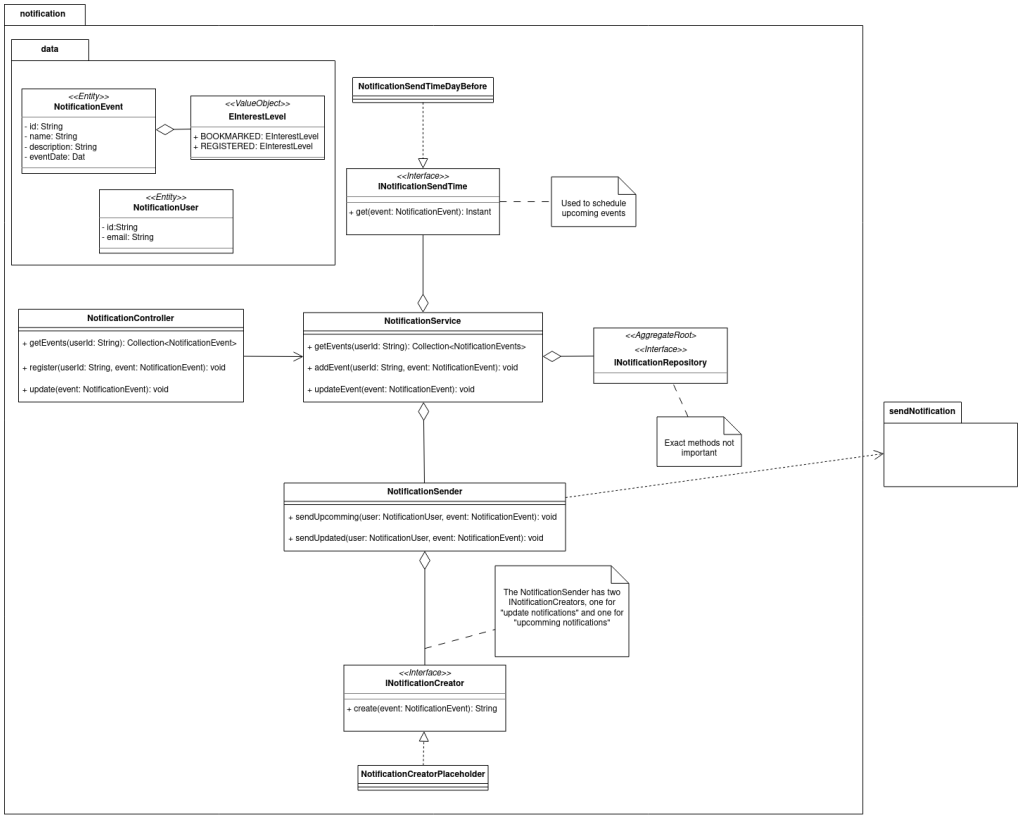


Notification Service

At first, it was fairly difficult to visualize the true functionality of the notification service. This is reflected in the diagram below, where we could only identify one needed class (we were aware that more helper classes would be needed even in this simple model, for simplicity's sake we have however omitted these). Our original idea was to model all the logic surrounding the notification (event got updated -> send notification and event is upcoming -> send notification) outside of the notification domain, inside relevant parts such as the UserService, and only let the notification service be responsible for sending notifications.

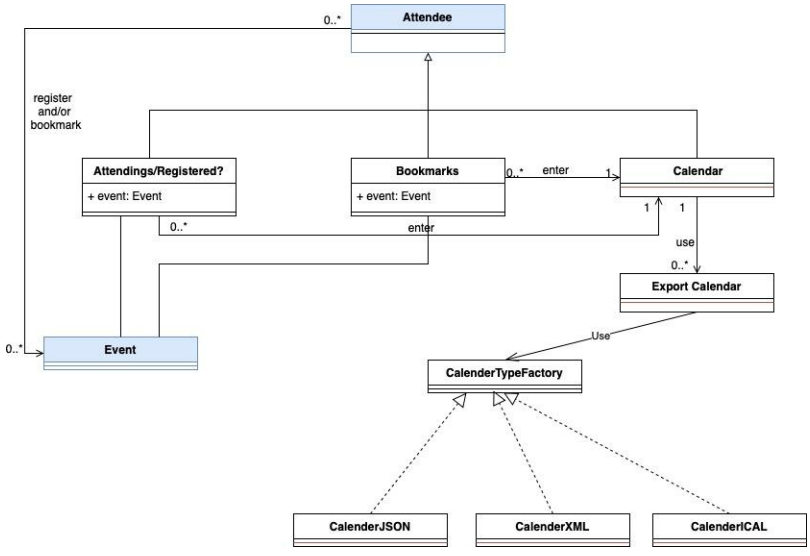


After a more careful evaluation of the problem domain and while also thinking a bit ahead to the needed microservice architecture we have adapted the diagrams quite heavily. The notification domain now also contains the logic for sending reminders and, given it receives the information that an event was updated, also the logic for sending updates about changed events. This decouples this functionality, which is not needed for a functional operation of the application, from the event domain.

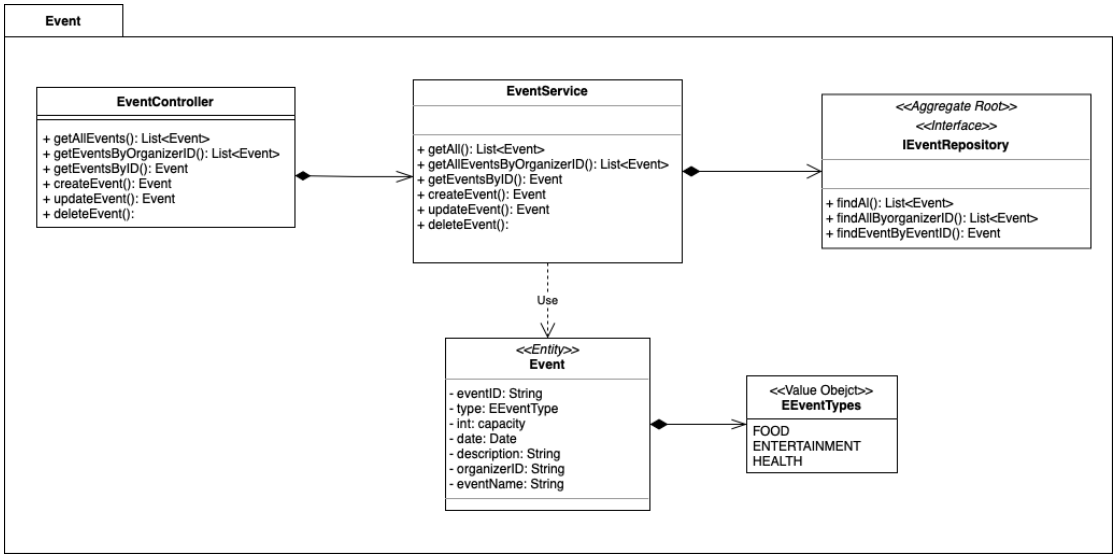


Event Service

In the first version of the class diagram, there was an attendee subdomain that contained the bookmarked and registered Events as well as the Calendar Export Service. After the next adaptation process, it became clear that attendees should only save the users' data and that the other functions should be in a domain of their own so that all the services can work as independently as possible. Also, the objects wouldn't be stored redundantly if each Service had its own domain. The Illustration below shows the first Attendee Subdomain, from which we developed the domains Event, Calendar Export, Attendance Service, Bookmark, and Tagging. In the following text, the iterations of the Event domain are shown. For the iterations of the other subdomains look at the next chapters.



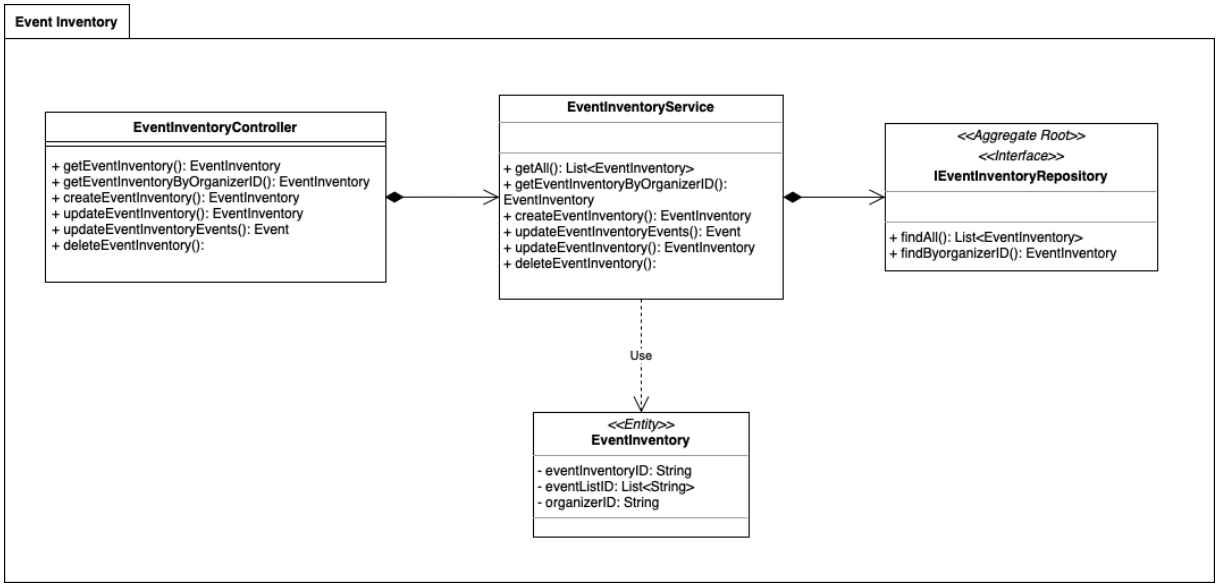
In the diagram below the newly adapted Event Domain is shown. It consists of an Entity where the table is stored, as well as Service, Repository, and Controller for the logic and endpoint communication. The enumeration EventTypes is for the predetermined types an event could have. For usage of the DDD-building blocks the class diagram of this subdomain was adapted in a third iteration, resulting in the diagram below.



EventInventory Service

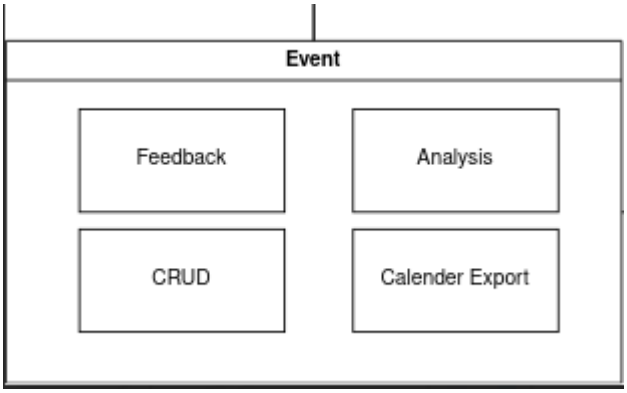
Originally the EventInventory was a Subdomain of the Organizer Domain. But, same as in the chapter Event Service, we decided to merge the organizer and attendee into the user domain and the other Services should become a domain on their own. In the diagram below the first draft of the Event Inventory Subdomain is shown.

In the second iteration, EventInventory became its own Domain and for usage of the DDD-building blocks the class diagram of this subdomain was adapted, resulting in the diagram below.

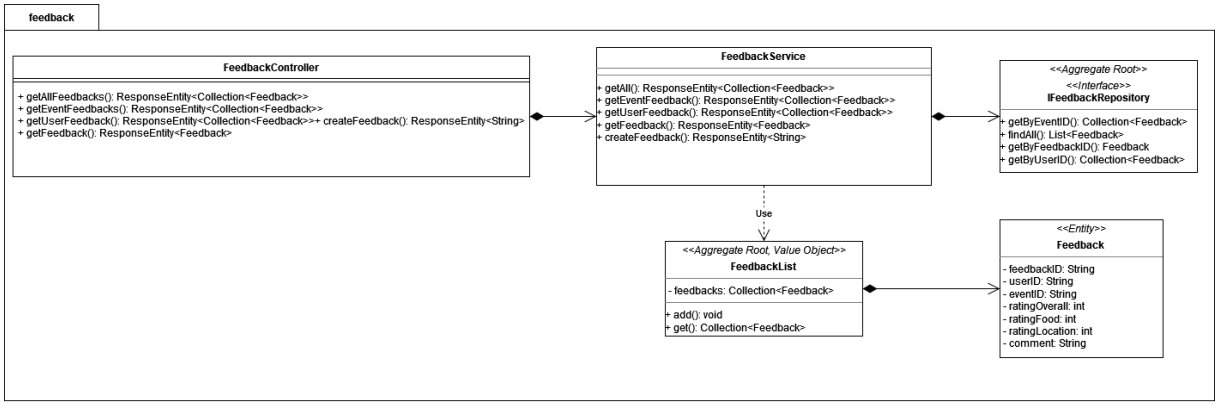


Feedback Service

At first the feedback was a subdomain of the event domain.



This would have meant that the feedback is highly coupled with the event. After more iterations, it became clear that the feedback should be pulled out and made its own domain to make every service have only one responsibility. The DDD building blocks were added in the third iteration, which is shown below.



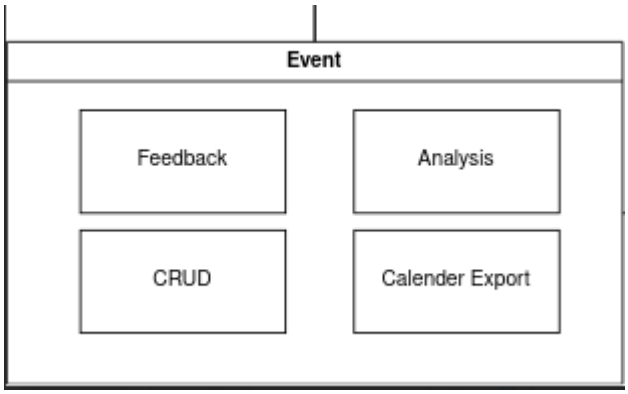
As can be seen, it is quite similar to the event class diagram. It consists of a repository, a service, a repository, and an entity. Where it differs is that it contains an

additional class to package multiple feedbacks into one object and does not need an enumeration. Packaging multiple feedback into one object was advantageous for the controller class.

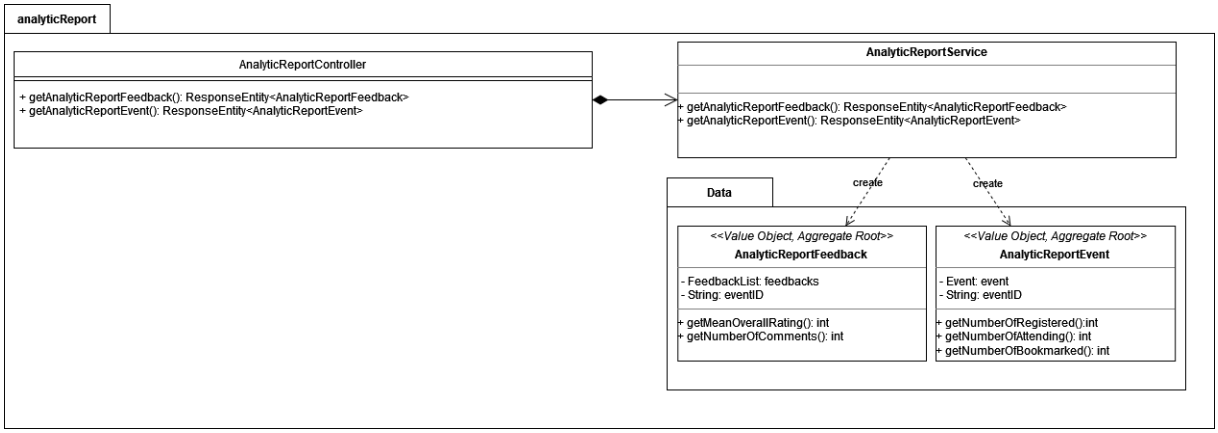
As already explained before the feedback is connected to the event and the user who created it by storing their ID. This way the feedback is strongly decoupled from the event.

Analytic Report Service

Just like the feedback domain, the analytic report domain started as a subdomain of the event domain. Again we decided to decouple it and make it its domain. This helps in improving adaptability for future changes that could need access to data stored outside of the event domain.

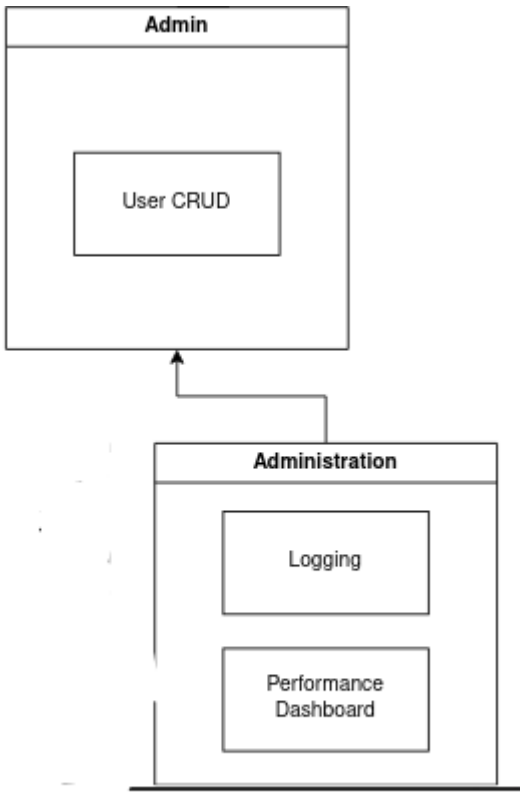


As can be seen below it is a very small domain that does not persist any data. Rather it gets its data from the other domains, currently only feedback and event. This makes it easily extendable.

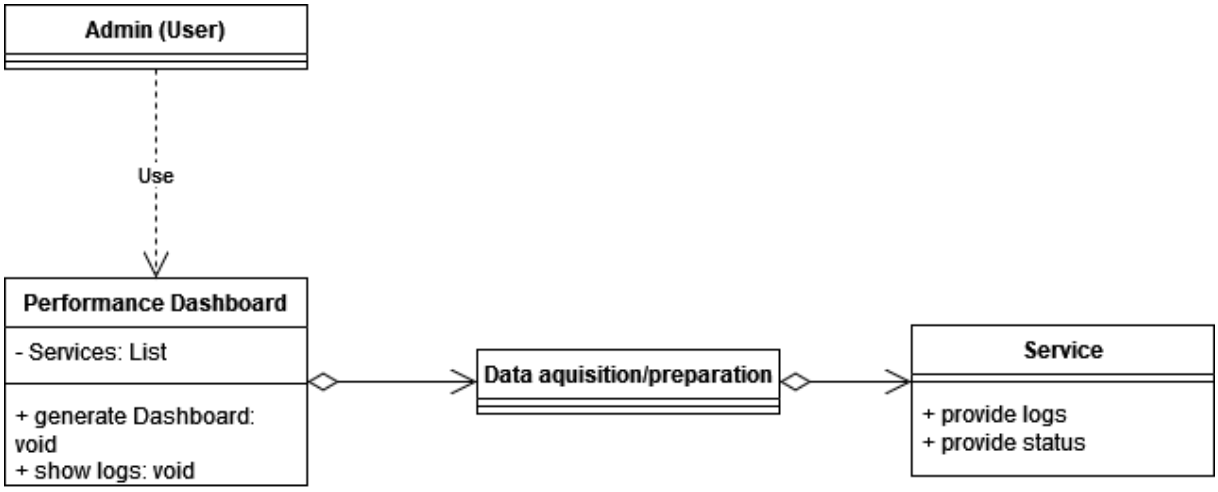


Maintenance Service

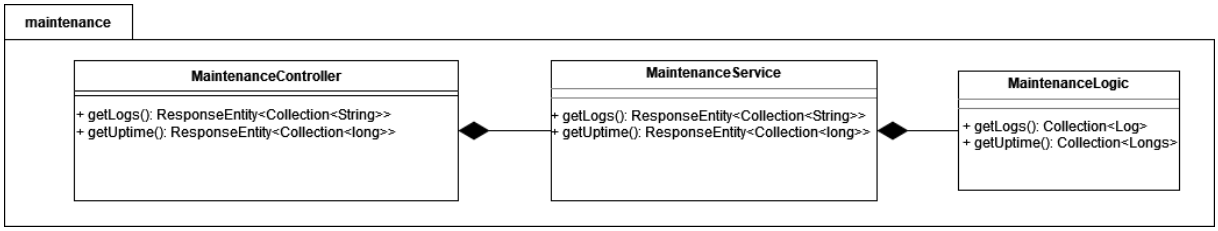
In the first iteration the maintenance service was split into two subdomains: logging and performance dashboard. We decided to combine both of those and make a domain out of them since they serve the same purpose of giving information about the system.



We pulled the acquisition of logs into the Performance Dashboard, which we renamed in a later iteration to maintenance service. This was before we switched to our more agile approach as a team, therefore it is kept very abstract. The DataAcquisition/preparation component in the middle would have to know about all existing services to gather the data from all of them.

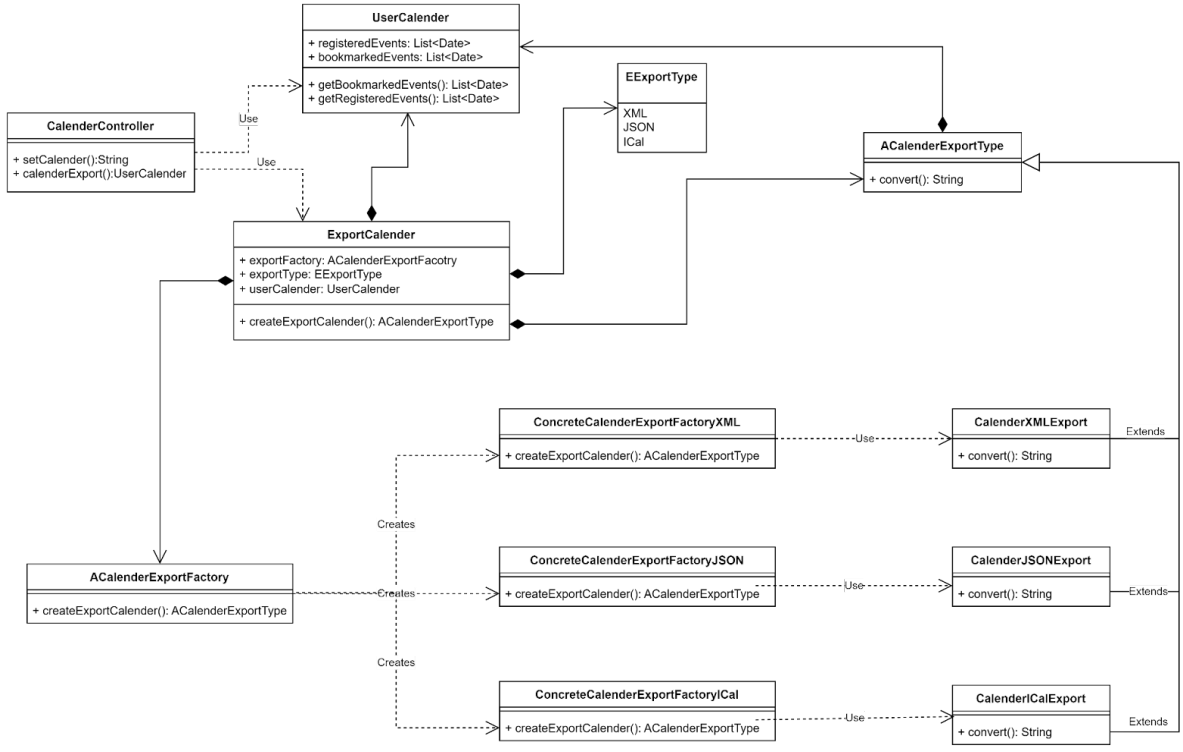


Our third iteration (shown below) shows the maintenance domain in isolation. It was adapted to the technology we decided to use.



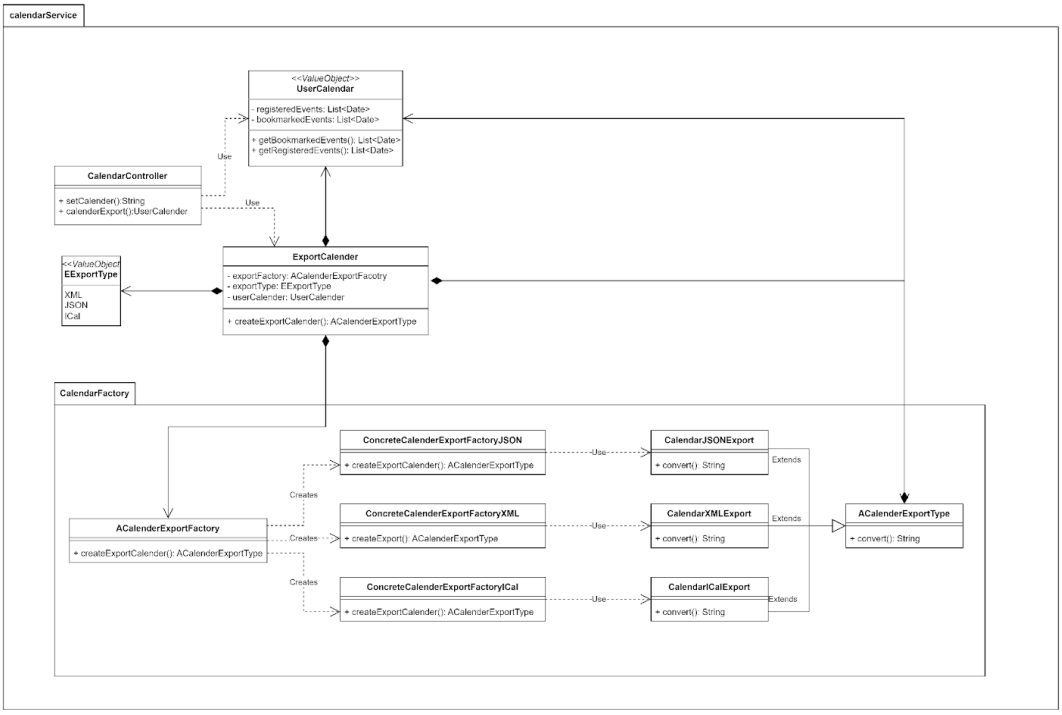
Calendar Service

At the start of the project, the calendar was initially conceived as a component of the Event service, as shown in the overview graphic above. However, as we worked on the project we defined the export types in more detail. To achieve this, we implemented a factory pattern for creating the calendar, which we continued to use throughout all design iterations.



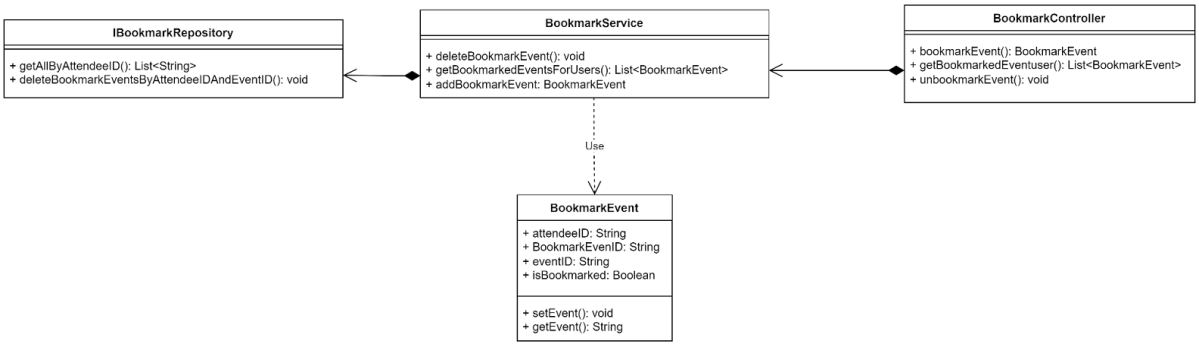
In the second iteration, we refined the class model and added more detail to the factory by specifying the export types and their return values during the export process. In the third iteration, we developed the architecture by defining the packages and DDD building blocks in more detail. We also improved communication and setup within the package.

As seen in the diagram above the improvement of the thirst and second iteration is clearly visible. The improvement of the second third iteration is focused on the DDD building blocks and the abstraction in packages.

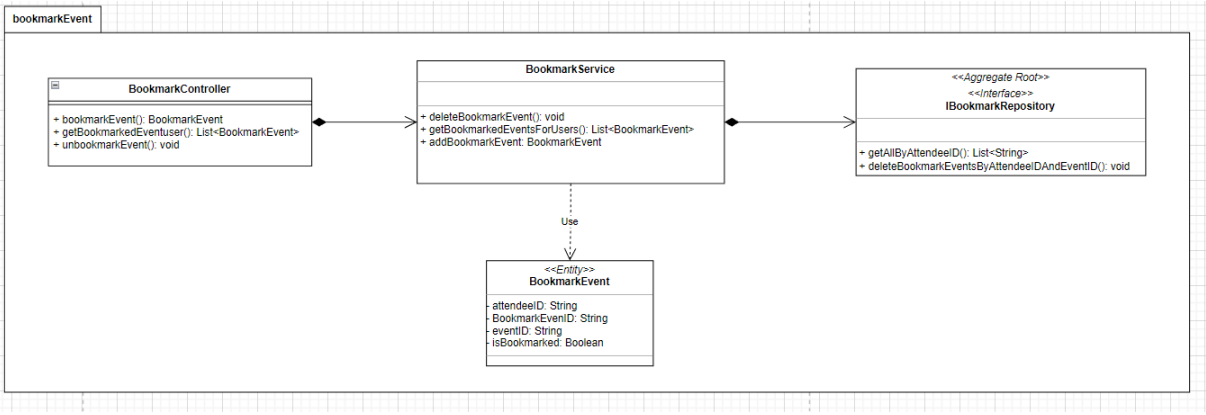


Bookmark Event

We improved the bookmarking process by abstracting the bookmarking service from the Event Domain. This enables better scalability in the future development process.

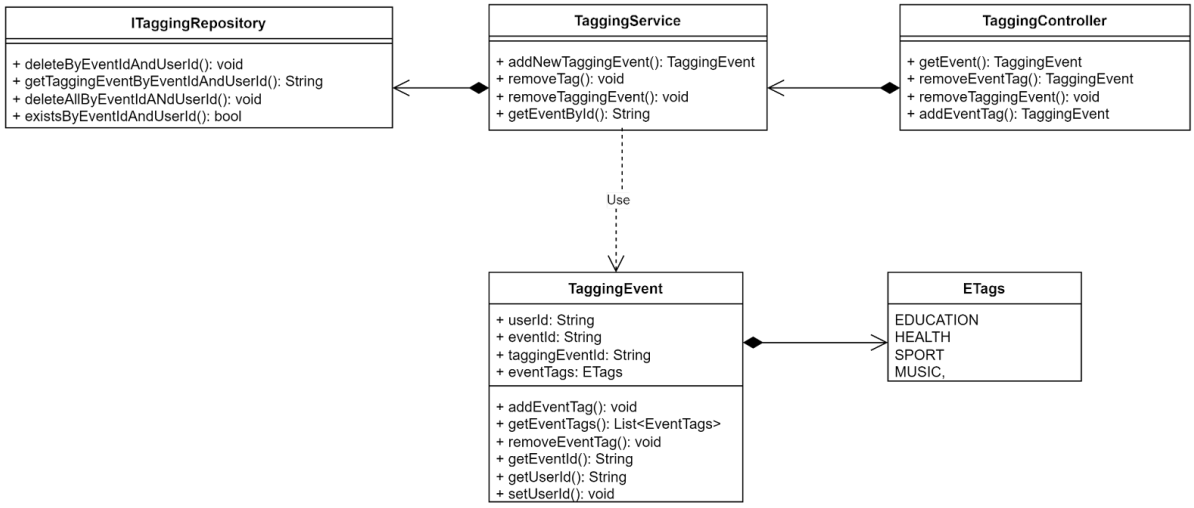


In the third iteration, we extended the Class Diagram with its DDD building blocks and changed the visibility of certain methods and fields.

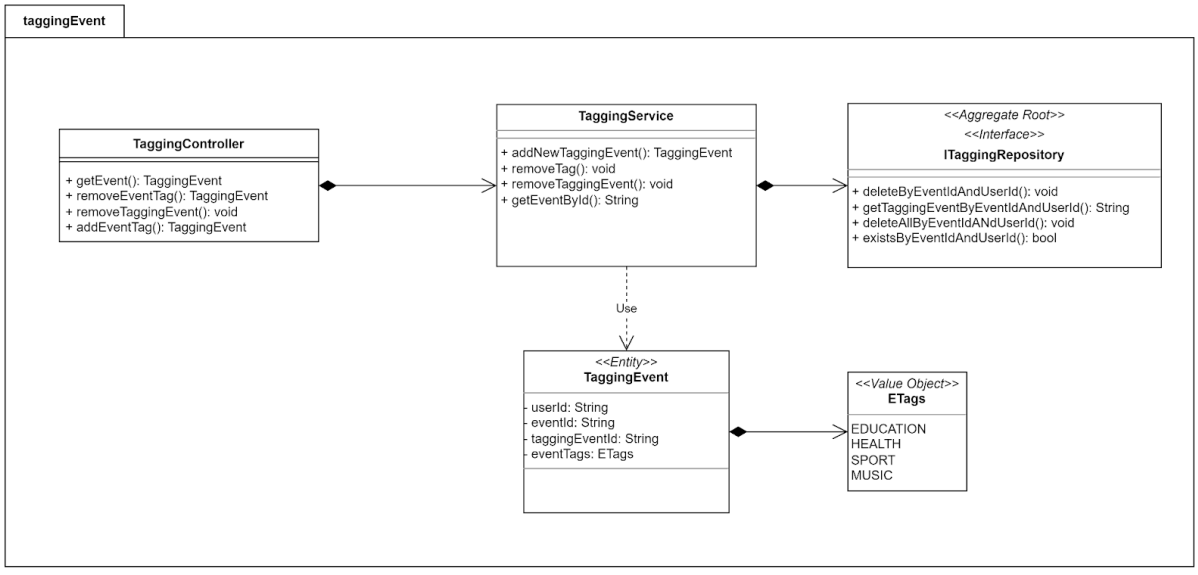


Tagging Event

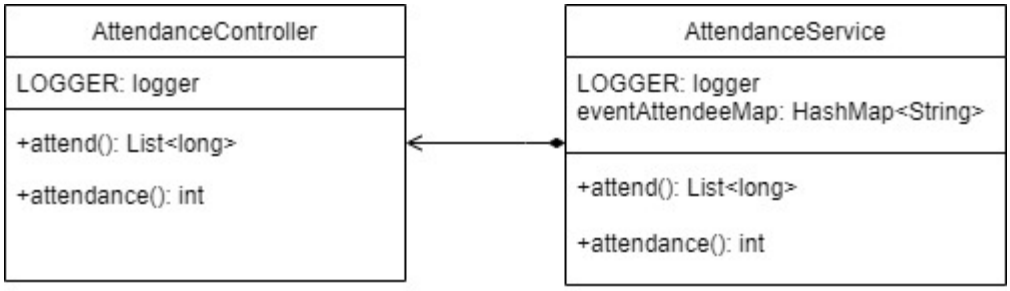
To improve scalability, we adopted an initial design process similar to the approaches mentioned above. We achieved this by abstracting the Tagging Service from the Event, which also involved abstracting predefined tags into an enumeration. Additionally, we assumed the responsibility of providing tags for the user, which we defined in this iteration.



In the further process, we defined the DDD Building Blocks and decided to implement them as a package.

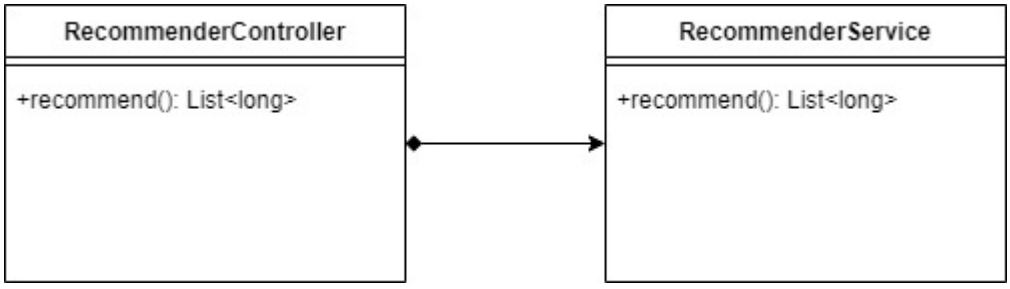


Attendance Service



The class diagram shown above was the first attempt to model the attendance service. It consists only of two classes: The controller which handles the requests, and the service which is responsible for doing the calculations. The final class diagram with an additional repository can be found in section 3.2: Logical View.

Recommender Service



This was the first iteration of the class diagram for the recommender service which only consists of two classes: The controller for handling the recommendation request and the service for generating the recommendation response. The final class diagram found under chapter “3.2: Logical view” includes an attendee filter class as well.

1.1.3. Design Overview

The current state of the design solution includes all the steps and decisions described in the chapters above and results in the package diagram which is shown in section 3.2.

For our Calendar Export Feature, we made a significant design decision to utilize the Factory Pattern. By implementing this pattern, we were able to create various types of calendars for the export feature, including JSON, XML, and ICal. With the Factory Pattern, our package can handle multiple calendar types and process them accordingly. It also allows an easy extension to other calendar types in a possible further implementation. The factory in our case is used in converting an already existing object into an exportable calendar type which can vary depending on the requirements of the user. The factory is suitable for this process because it creates a defined object in an instance with no further changes necessary.

To simplify and organize the required dependencies for the code, dependency injection is used. This design pattern separates the concerns of constructing objects

and using them. This has the advantage that a change of a class itself is not needed when some of its dependencies have changed.

Moreover, the design pattern of a model–view–controller to classify information depending on their responsibility in the groups model (managing data), view (representing models), and controller (receiving and passing input). This pattern is implemented by using Java MVC.

Finally, the publish/subscribe pattern will be used in the future for event-driven communication so that attendees, for example, can receive recommendations on newly created events. In the current implementation, the publish/subscribe pattern was omitted. The main reason for implementing the publish/subscribe pattern in the design deadline is to protect the team from potential team member exits, which we have done by implementing a more detailed solution than necessary for the mocked implementation.

1.2. Development Stack and Technology Stack

1.2.1. 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 customized 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.

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

Gmail Email Service	current version provided by Google

1.2.2. 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 customize 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.

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.

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

Packagename	**Version**
Spring	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
Spring Web, Spring Security	3.0.5
Typescript/Angular	we plan to use the latest version

2. System Requirements

In the following table all the identified functional requirements are listed, with the different columns representing:

- “ID”: a numeric ID with graded numbering to indicate related requirements
- “Domain”: assigns a requirement to a system domain
- “Requirements”, “Description”: give a brief overview of the functionality
- “Use Case”: shows the ID of the related use cases of chapter 3.1.2
- “Priority”: three levels; high, medium and low and should display the systemic relevance
- “Verification”: we define 3 types of verification:
    - “test”: unit or integration tests
    - “UI”: all testing with a user through a UI
    - “review”: all checks by one or more members of the team are meant

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

3.1	Event Inventory	Create Event Inventory	For every organizer one event inventory is created.	5	high	Test
3.2	Event Inventory	Manage Event Inventory	An organizer can manage their inventory by updating the events	3,5	medium	Test
3.3	Event Inventory	List all events	All events of one organizer are listed in their Event Inventory	3,5	high	UI, Test
3.4	Event Inventory	Search for events by different criteria	All events in an event inventory can be searched by different criteria.	3,5	low	UI
4.1	Event	Create Event	An organizer can create a new event.	3	high	Test, UI
4.2	Event	Update Event	An organizer can update the fields of their own events based.	3	medium	Test
4.3	Event	Event Capacity	An organizer can specify the maximum capacity.	3	medium	Test
4.4	Event	Delete Event	An organizer can delete their own event.	3	low	Test, UI
5.1	Analytic Report	View Analytic Report Registered	An organizer can view a report about registered/ bookmarked from an event	8	medium	Test
5.2	Analytic Report	View Analytic Report Feedback	Organizers and attendees can view a report about feedback from an event	9	low	Test
6.1	Notificatio n Service	Notify attendees	An organizer can send messages to attendees who are registered or have bookmarked	13	high	Test, UI, review

			one of their event			
7.1	Recommender Service	Notify attendees for new events	Attendees get notifications about new events based on their bookmarks and other criteria	14	medium	Test, UI, review
8.1	Attendance Service	Attendees can register	Attendees can register for an event, if there are vacancies	12	medium	Test, UI
8.2	Attendance Service	Attendees can deregister	Attendees can deregister for an event	12	low	Test, UI
8.2	Attendance Service	Organizer can send messages	Organizers of an event can send messages directly to an attendee	13	medium	Test, UI, review
9.1	Feedback	Give Feedback	Attendees can give Feedback to an event with a rating and a text.	6	medium	Test, UI
9.2	Feedback	See Feedback	Attendees and Organizers can see feedback given on an event already passed.	6	medium	Test, UI
10.1	Bookmark and Tagging	Bookmark Events	Attendees can bookmark events if they are interested	10	medium	Test, UI
10.2	Bookmark and Tagging	Unbookmark Events	Attendees can unbookmark events they have bookmarked.	10	medium	Test, UI
10.3	Bookmark and Tagging	Get all Bookmarked Events for user	All bookmarked events per user	10	medium	Test, UI

10.4	Bookmark and Tagging	Get all Tags per user and Event	All tags per event and user	11	medium	Test, Ui
10.5	Bookmark and Tagging	Tag Events	Attendees can Tag events with given tags.	11	medium	Test, UI
10.6	Bookmark and Tagging	Untag Events	Attendees can untag already tagged events	11	medium	Test, UI
11.1	Calendar Export	Export bookmarked Event	Attendees can export a bookmarked event as json, xml or ical	16	medium	Test, UI, review
11.2	Calendar Export	Export registered Event	Attendees can export an event they are registered for as json, xml or ical.	16	medium	Test, UI, review
12.1	Maintenance Service	Dashboard	Admins can see a dashboard with health-information about the system	15	high	Test, UI
12.2	Maintenance Service	Logfiles	Admins can see log files from services	15	high	Test, Ui

The following table lists all identified non-functional requirements. The functional :

- “ID”: A numeric ID with graded numbering to indicate related requirements
- “Domain”: to which domain the requirement belongs or which domain it influences if there is any
- “Name” and “Description”: give more details about the requirements
- “Priority”: three levels; high, medium and low and should display the systemic relevance

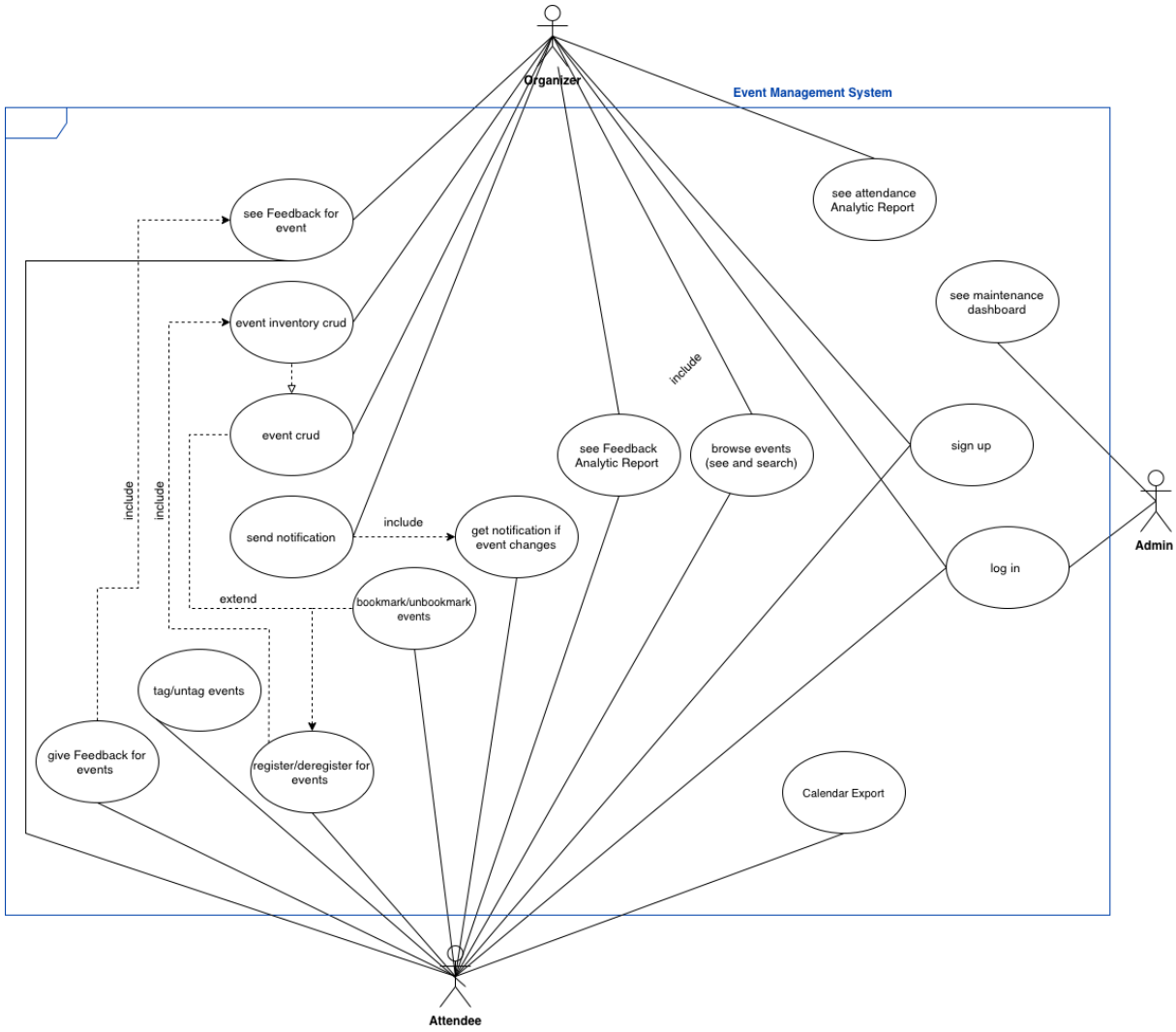
ID	Domain	Name	Description	Priority
1.1	Auth Service	Login-response time		low
1.2	Auth Service	Password validation		high

1.3	Auth Service	Token login		high
2.1		Event Driven Communication	Publish/subscribe pattern	medium
3.1		Micro-Services		high
4.1	Feedback	SQL-Injections in text fields		medium
5.1		Language English		high

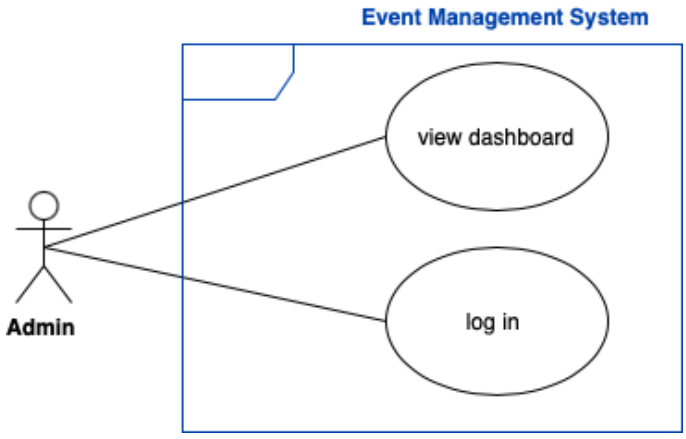
3. 4+1 Views Model

3.1. Scenarios / Use Case View

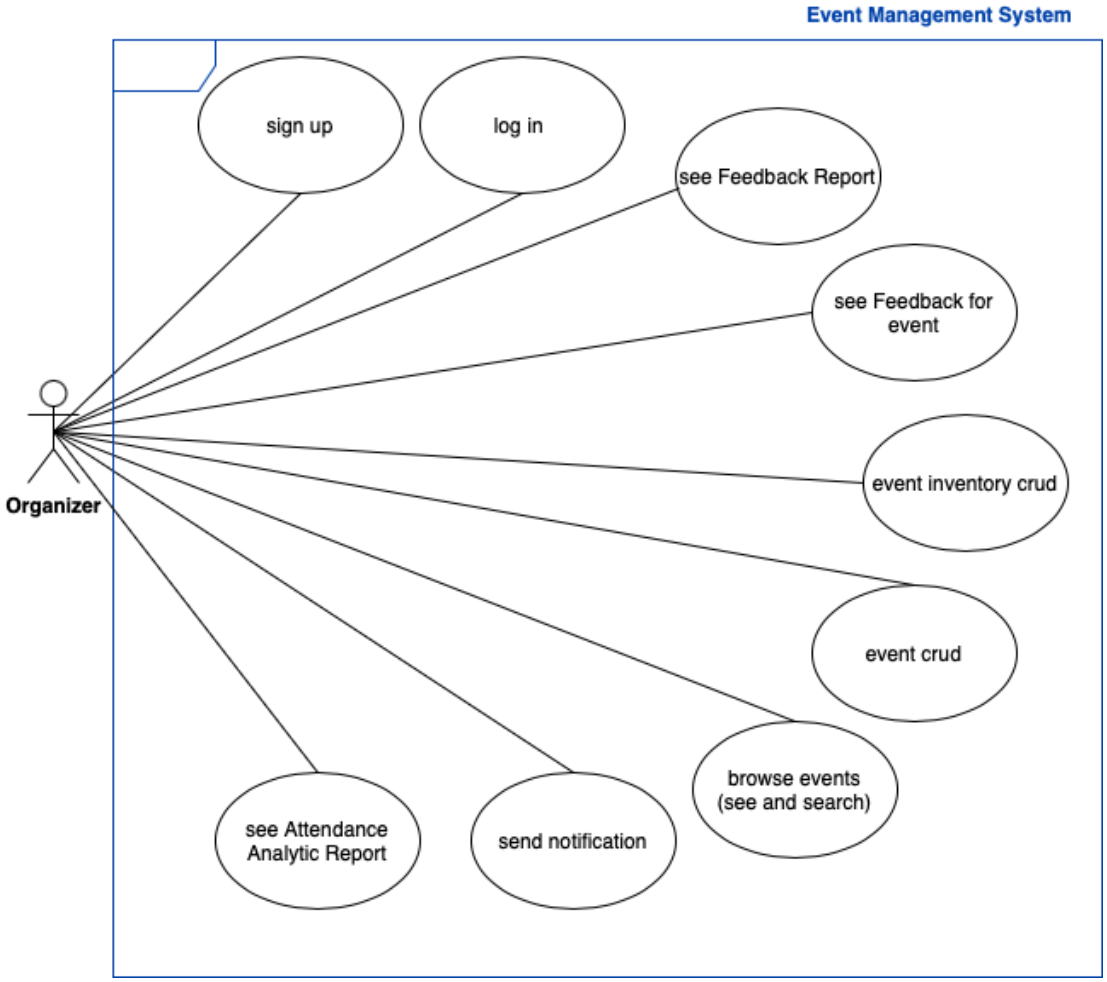
3.1.1. Use Case Diagram(s)



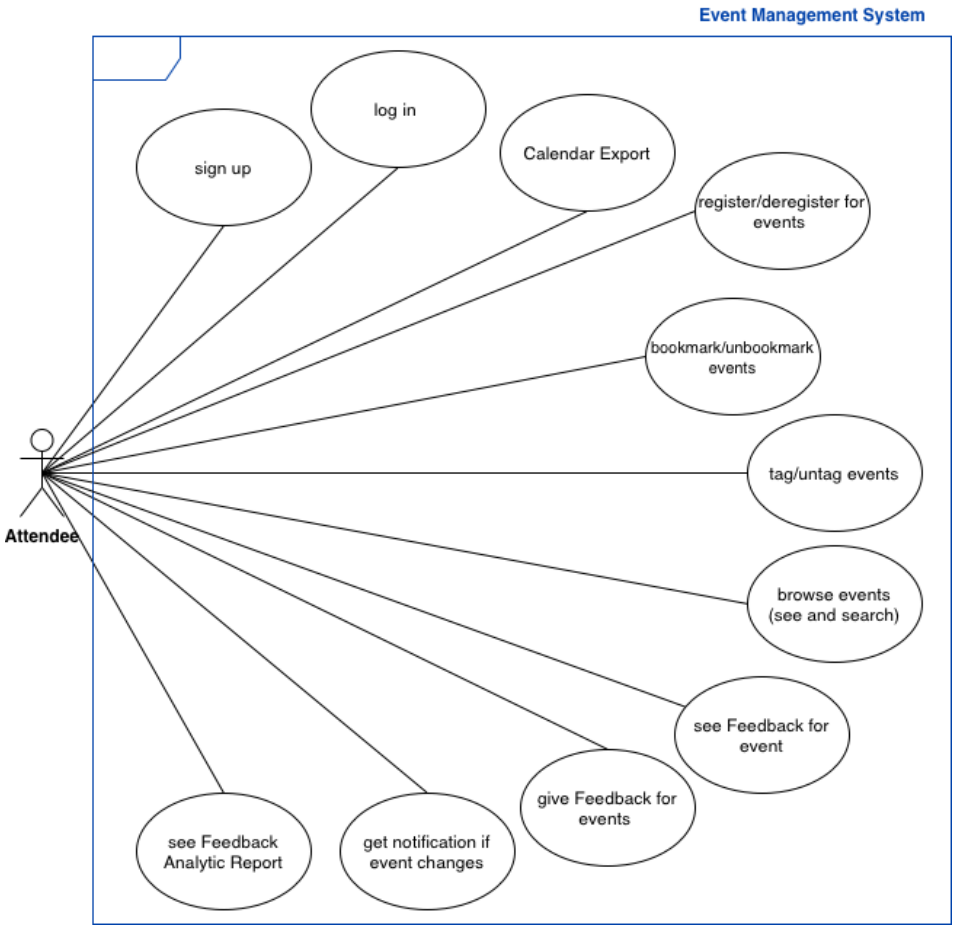
Use Case Diagram which represents the whole system at once. Some use cases or functionalities can be carried out by more than one user role. Some use cases are extended from others or included if there has to be one use case carried out first so that the other one can perform its functionality.



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:	Register a user
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 Registered 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 read, create any event and 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 Scenario:	Newly created or existing events updated and saved. 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 search for every event by different criteria.
Pre-Conditions:	Events had to be created and saved.
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:	Event Inventory CRUD
Use Case ID:	5
Actor(s):	Organizer
Brief Description:	Organizers have each one event Inventory, where they can manage their events (Event CRUD). The inventory can be created, updated, viewed, and deleted.
Pre-Conditions:	User must have the role of “organizer”.
Post-Conditions:	Event Inventory should be linked to the organizer
Main Success Scenario:	Event Inventory must be able to be accessed by the linked organizer.
Extensions:	Event CRUD
Priority:	high
Performance Target:	high
Issues:	

Use Case:	give Feedback
Use Case ID:	6
Actor(s):	Attendee
Brief Description:	Attendees can give Feedback for events they have attended.
Pre-Conditions:	A user must have the role of “attendee”. 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:	7
Actor(s):	Organizer, attendee
Brief Description:	Organizers and attendees can see the feedback which was given for an event.
Pre-Conditions:	A user must have the role of “organizer” or “attendee”. 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:	8
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, bookmarkings and information only visible to the organizer.
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:	9
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:	10
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”. Event has to be created, and has to be bookmarked or unbookmarked for each case.
Post-Conditions:	Bookmarked Event is linked to an attendee
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:	11
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.
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/unregister for Events
Use Case ID:	12
Actor(s):	Attendee
Brief Description:	Attendees can register for Events. If they are registered, they can also unregister again.
Pre-Conditions:	User has to have the role “attendee”. Event has to be created and the attendee has to either not be registered yet or be registered.
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 registered Attendees.
Extensions:	Event CRUD
Priority:	medium
Performance Target:	medium
Issues:	

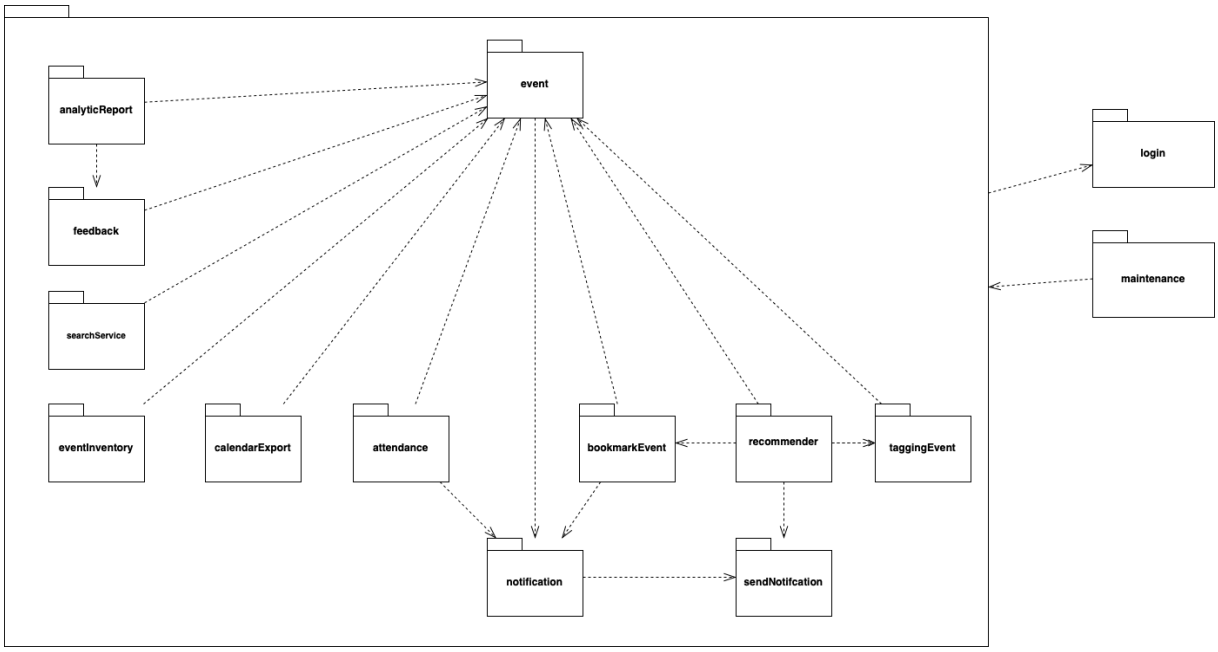
Use Case:	Send notification
Use Case ID:	13
Actor(s):	Organizer, attendee
Brief Description:	Organizers can send Notifications to Attendees of their own events.
Pre-Conditions:	Users have to be registered by email. Sender has role organizer. The recipient has the role of “attendee”.
Post-Conditions:	Attendee received an email from the organizer.
Main Success Scenario:	Organizer sent an email to an attendee who is registered 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 recommender notification
Use Case ID:	14
Actor(s):	attendee
Brief Description:	Attendees get notifications about upcoming events they might like, based on their bookmarks and registered events.
Pre-Conditions:	Users have to be registered by email. The recipient has the role of “attendee”. The recipient has bookmarks and/or registries.
Post-Conditions:	Attendee received an email.
Main Success Scenario:	Attendees get emails with upcoming events which
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?

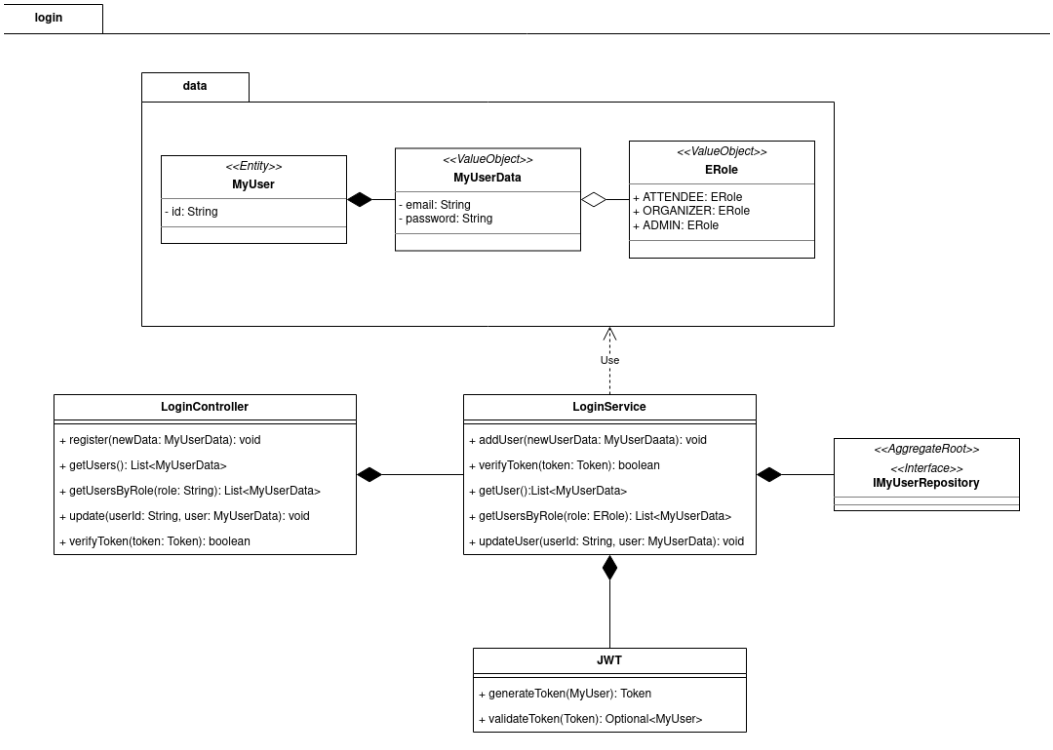
Use Case:	Calendar Export
Use Case ID:	16
Actor(s):	attendee
Brief Description:	Attendees can export events to a calendar
Pre-Conditions:	User has the role of attendee. Events are registered or bookmarked.
Post-Conditions:	JSON, XML, or ical is created
Main Success Scenario:	Attendee exported an event as one of three formats.
Extensions:	
Priority:	medium
Performance Target:	medium
Issues:	

3.2. Logical View



In this package diagram the whole project scope and the relations of the domains are shown. In the following chapters each domain is described in more detail with a class diagram. The package diagram shows which domains work together or have relations to one another. The arrowhead indicates that this domain gives data to the domain on the other end.

Login service



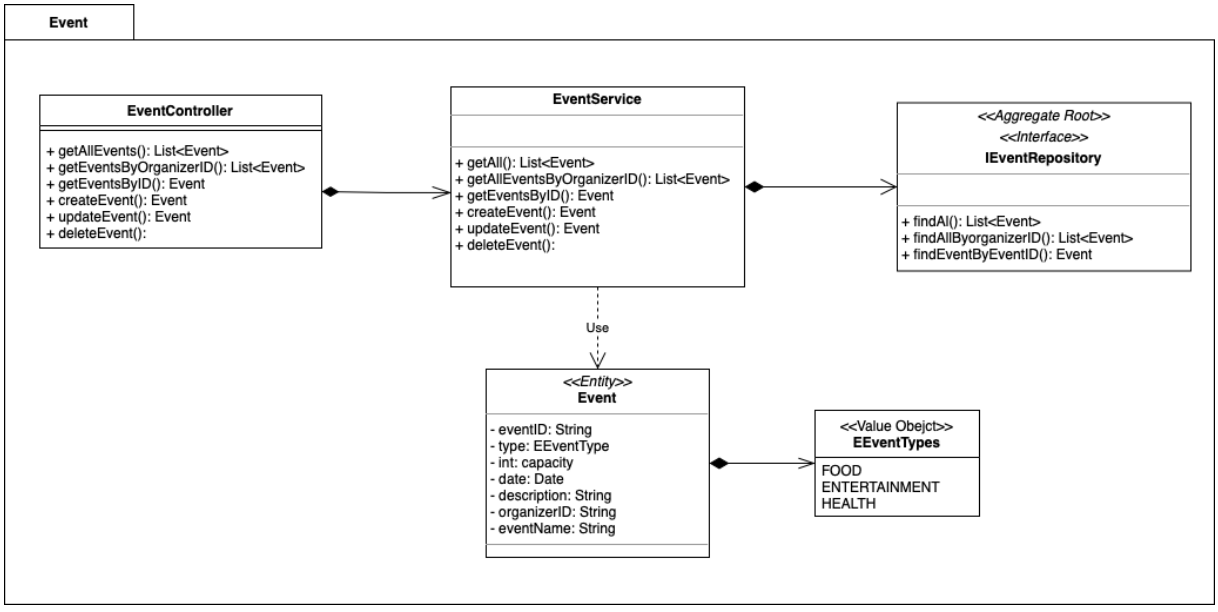
The login service is responsible for:

- User authentication (logging in and issuing tokens)
- User authorization (validating tokens and checking roles)
- User registrations
- User update

It issues JWT tokens, which can be used by the rest of the app to authorize the user. The JWT contains information about the corresponding user. It is further the central store of all users and can issue lists of users to services.

We suspect that this service will likely be subject to most changes in the next stages of development. The used implementation will depend on how well we can integrate the authorization with existing spring concepts, such as the security filter chain, as using these should make our code more concise and easier to read. Further, we struggle to assess how difficult it will be to use an authorization and authentication architecture that strictly follows the OAuth2 standard. Since the architecture of our app is not extremely complex, we may opt to use a more simplified version of the OAuth2 protocol.

Event Service



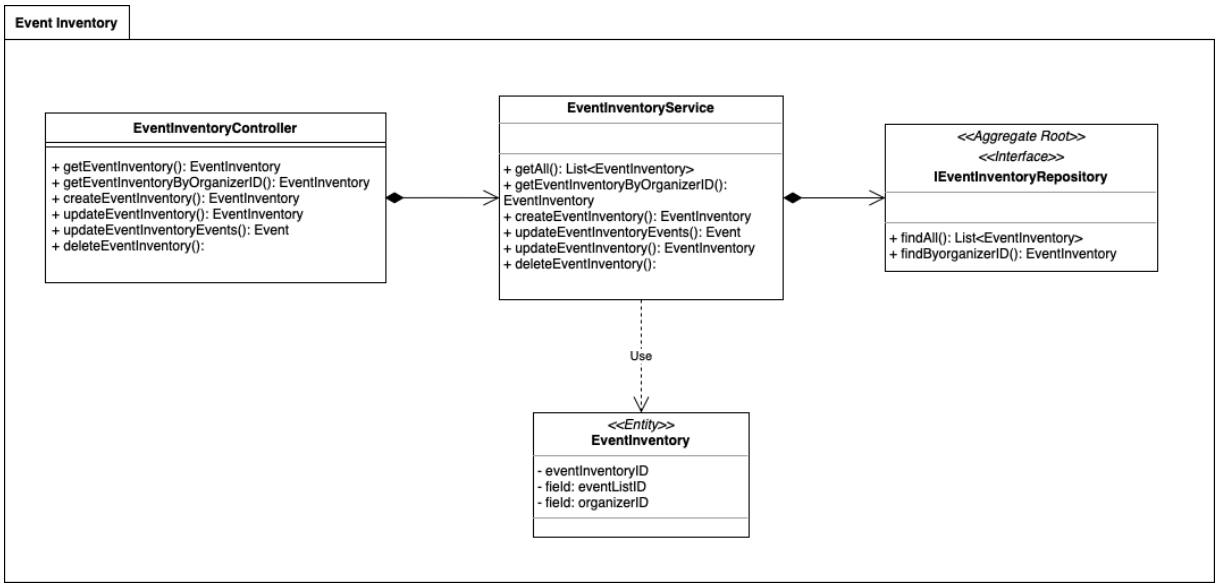
The Event Service is responsible for the Event CRUD. Events can be created, updated, read, and deleted. They are stored in the Event Entity and have a relation to the organizer by saving their ID. The types an event can have are predefined by an enum. Every other service that needs an event can get access through this domain via the Controller.

DDD-Building Blocks:

- Entity
- Value Object

- Aggregate Root

EventInventory Service



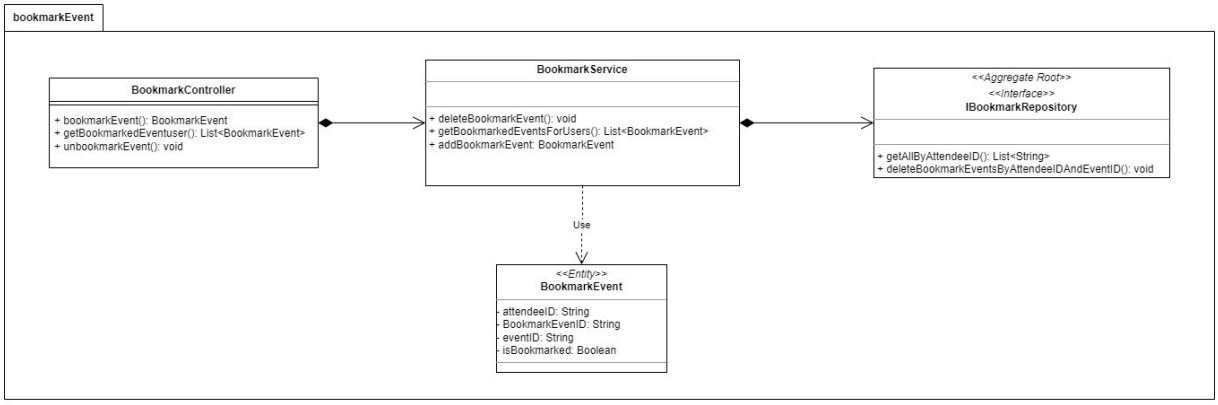
The Event Inventory Service is responsible for the Event Inventory CRUD. The Inventory can be created, updated, read, and deleted. It saves the id of the organizer to which it belongs and a List of Event IDs that are added by the organizer. The events themself can be accessed by the Event Domain.

DDD-Building Blocks:

- Entity
- Aggregate Root

Bookmark Service

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.



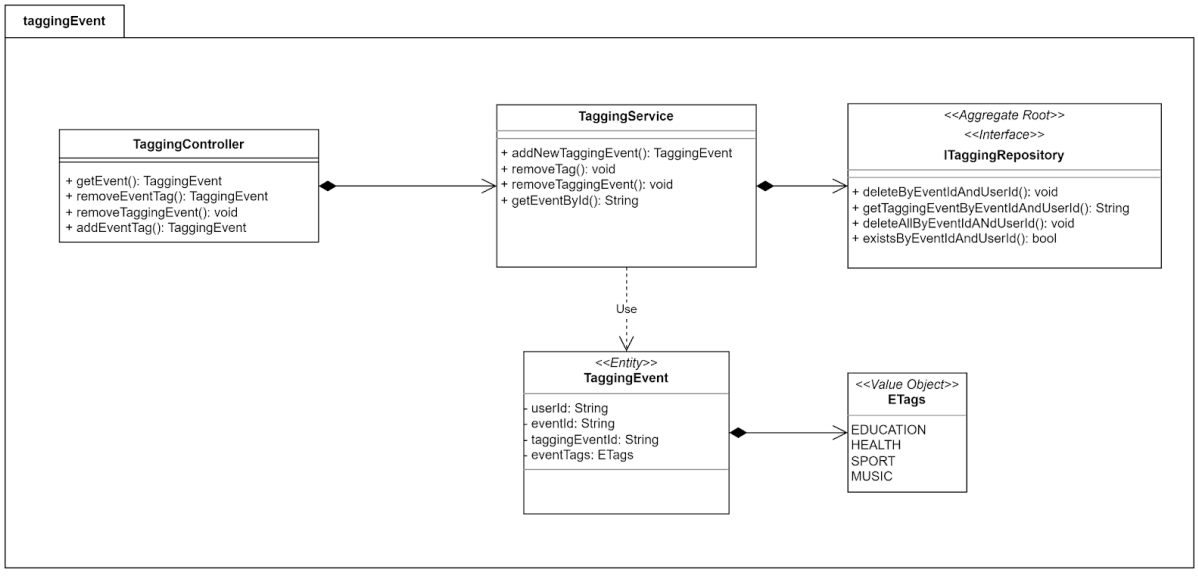
It's important to note that both the database and bookmarked events are currently identified using the corresponding id, rather than entities from other packages. Therefore if the bookmarking service is used, event data must be fetched from other sources based on the Id provided.

DDD-Building Blocks:

- Entity
- Aggregate Root

Tagging Event

The Tagging service is designed and implemented using the Model-View-Controller pattern.



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.

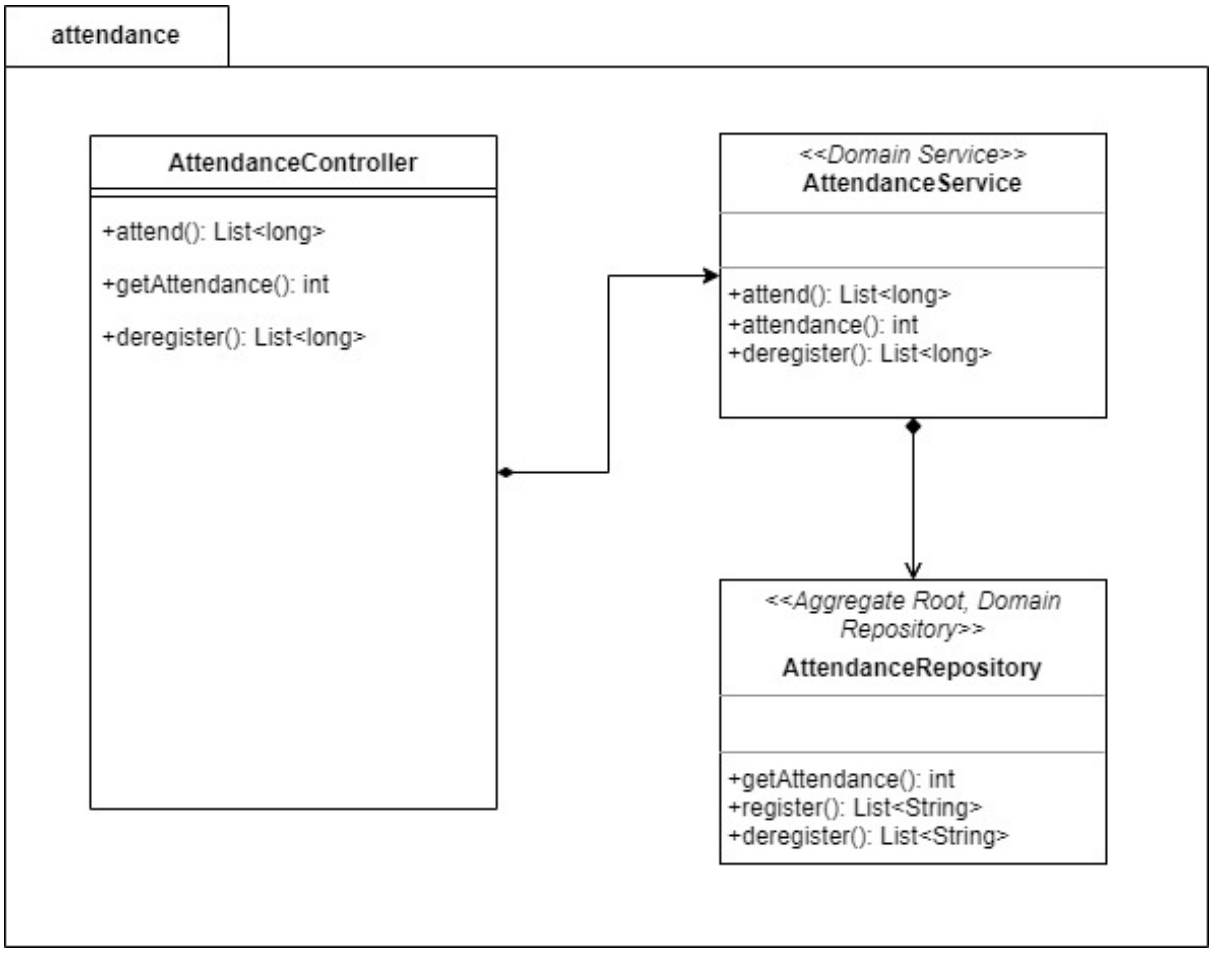
Furthermore, the database requests in the Repository are mostly based on the eventId and userId, which provide the necessary data.

It's important to note that both the database and tagged events are currently identified using the corresponding id, rather than entities from other packages. Therefore if the tagging service is used, event data must be fetched from other sources based on the Id provided.

DDD-Building Blocks:

- Entity
- Aggregate Root
- Value Object

Attendance Service



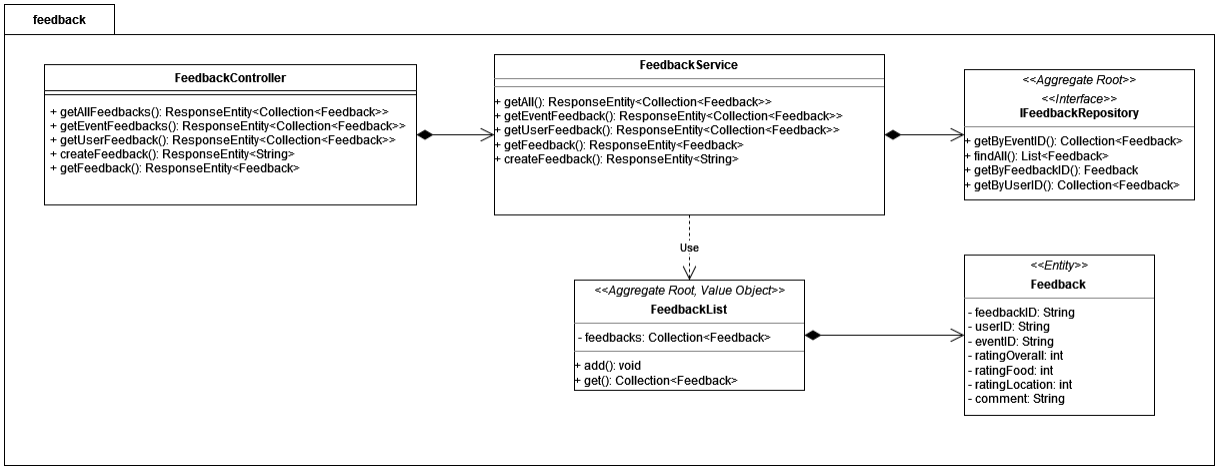
The attendance service itself is structured into the controller, the service and the repository. The attendance controller contains the endpoints for registering to an event with its eventID or to get the current attendance count of this event. Attendees

can also deregister from an event if they don't want to be part of it anymore. The attendance service receives commands from the controller, manages the repository and communicates with the event service to get the maximum capacity of the event that an attendee requests to attend

DDD-Building Blocks:

- Value object

Feedback Service

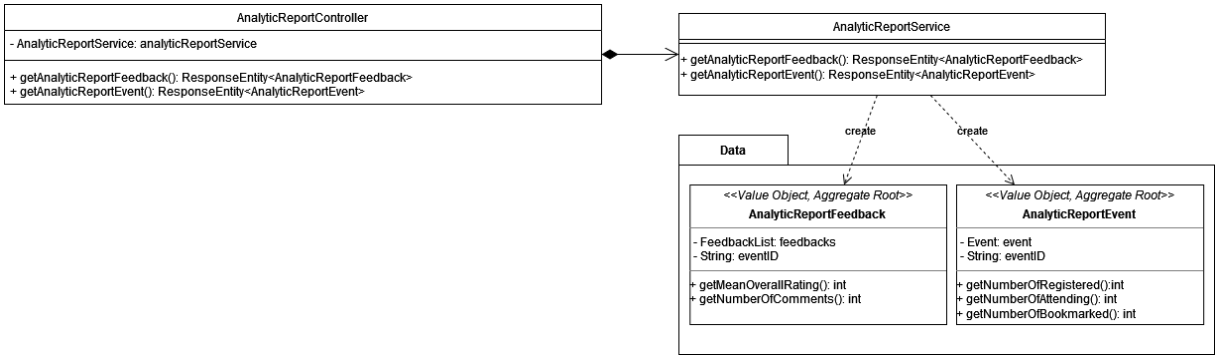


The feedback service is responsible for feedback CRUD. Feedbacks are stored and linked to the event by an eventID and to the user who created it by the userID. Important to note there is a separate class to store a list of feedbacks. This was necessary for the possibility to send entities instead of objects using Spring's http.ResponseEntity.

DDD building blocks:

- Entity
- Aggregate Root
- Value Object

Analytic Report Service



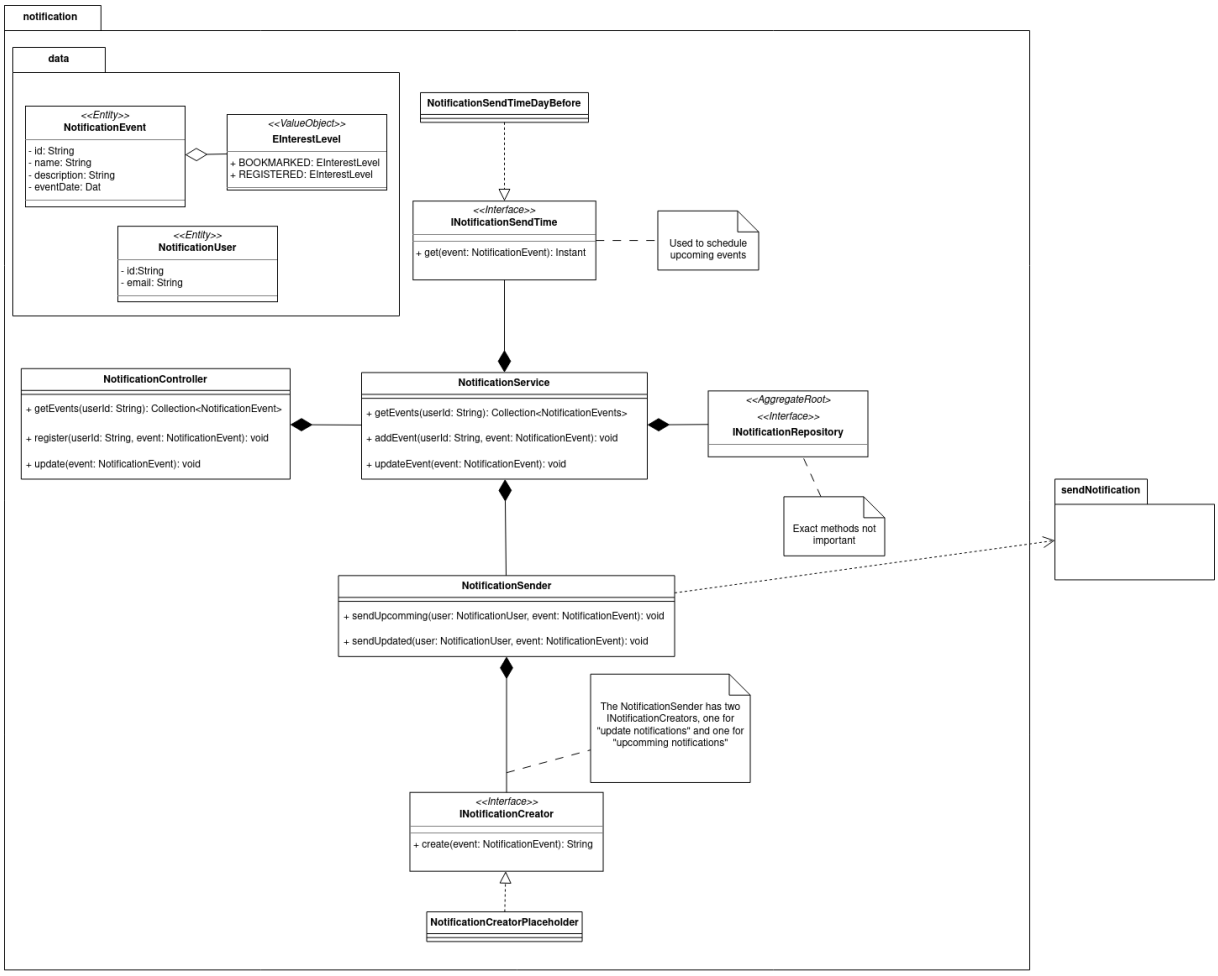
The Analytic Report service is responsible for fetching data about an event, or feedback given to an event, and processing the data to show to the user. There are

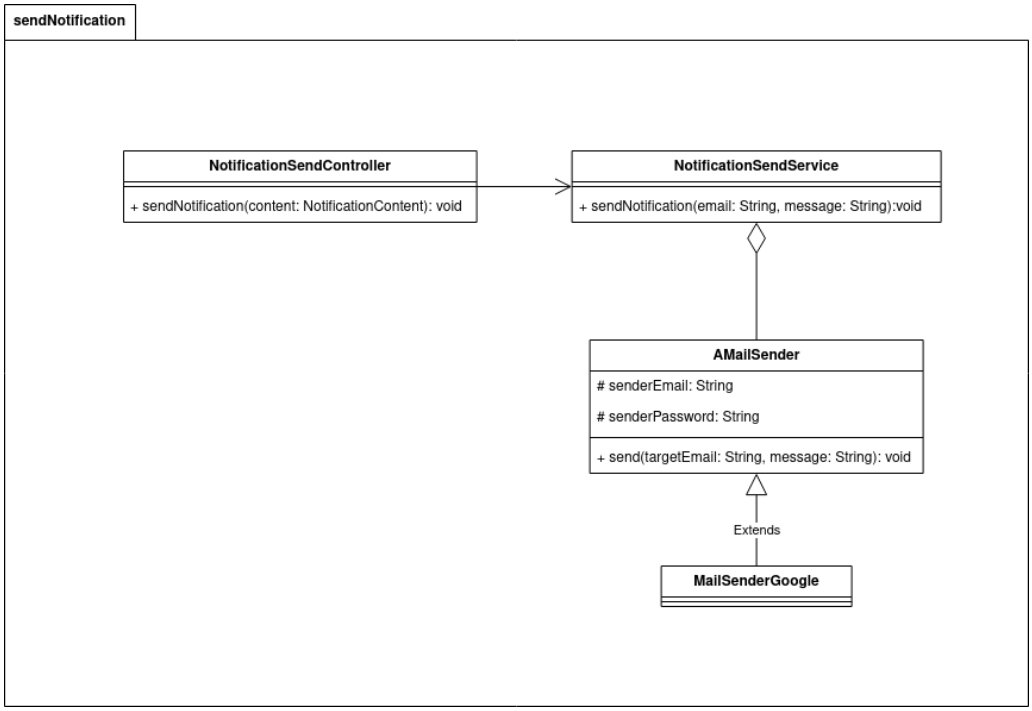
two different reports, one for the event, which only the organizer of said event can see, and one for the feedback for an event. The second report can be seen by organizers and attendees.

DDD building blocks:

- Aggregate root
- Value object

Notification service





We have decided to split the notification service into two main packages. The “notification” package is responsible for:

- Saving queued notification
- Generating notification content
- Updating queued notifications if a service update occurs

Whereas the sendNotification service is just responsible for

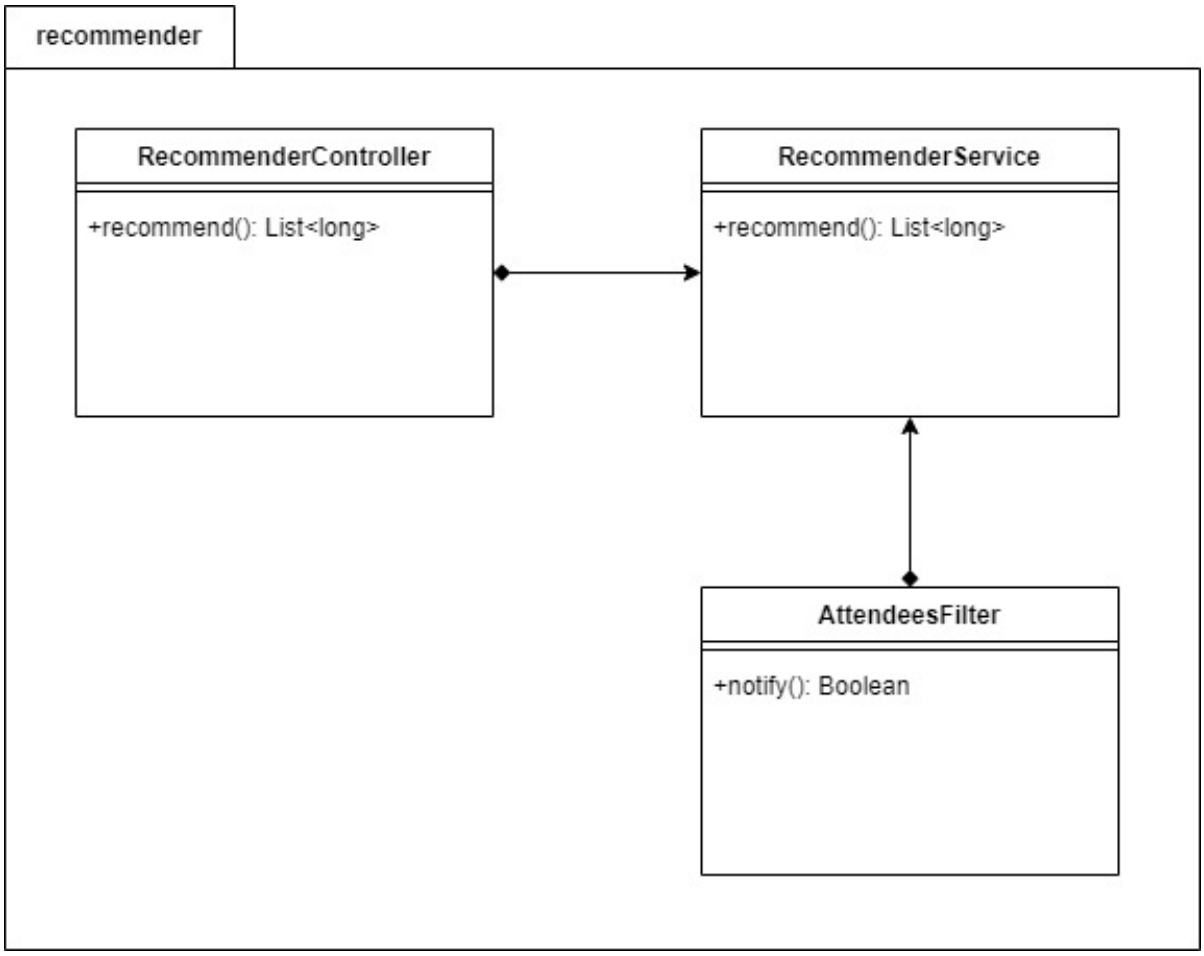
- Send a notification to a target email address with a given message

This split occurred due to other services apart from the notification service needing the ability to send notifications (see for example the recommender service). This abstraction should allow the functionality to be simply reusable.

Interesting architecture features of the notification service are the interfaces `INotificationSendTime`, responsible for the time when a notification reminder will be sent, and `INotificationCreator`, responsible for the contents (message) of the to-be-sent notification. These classes both model business logic and allow us to simply change the behavior of this service by defining new implementations of these interfaces.

Similarly in the `sendNotifiation` service, the abstract `AMailSender` class allows to define multiple email providers, through which an Email-notification might be sent.

Recommender Service

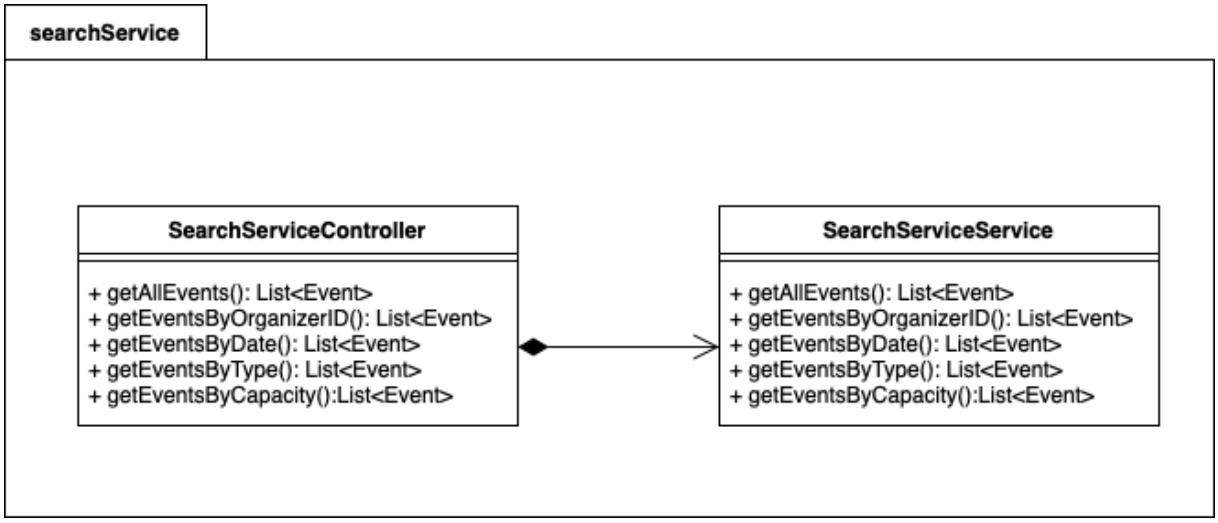


The recommender service consists of the controller, which manages the endpoint, a recommender service class, and an attendee's filter. When this service is requested by the "recommend" endpoint, it will communicate with the tagging service and with the event service. The recommender service will then filter attendees for possible events and return a List with eventIDs.

DDD-Building Blocks:

- Entity

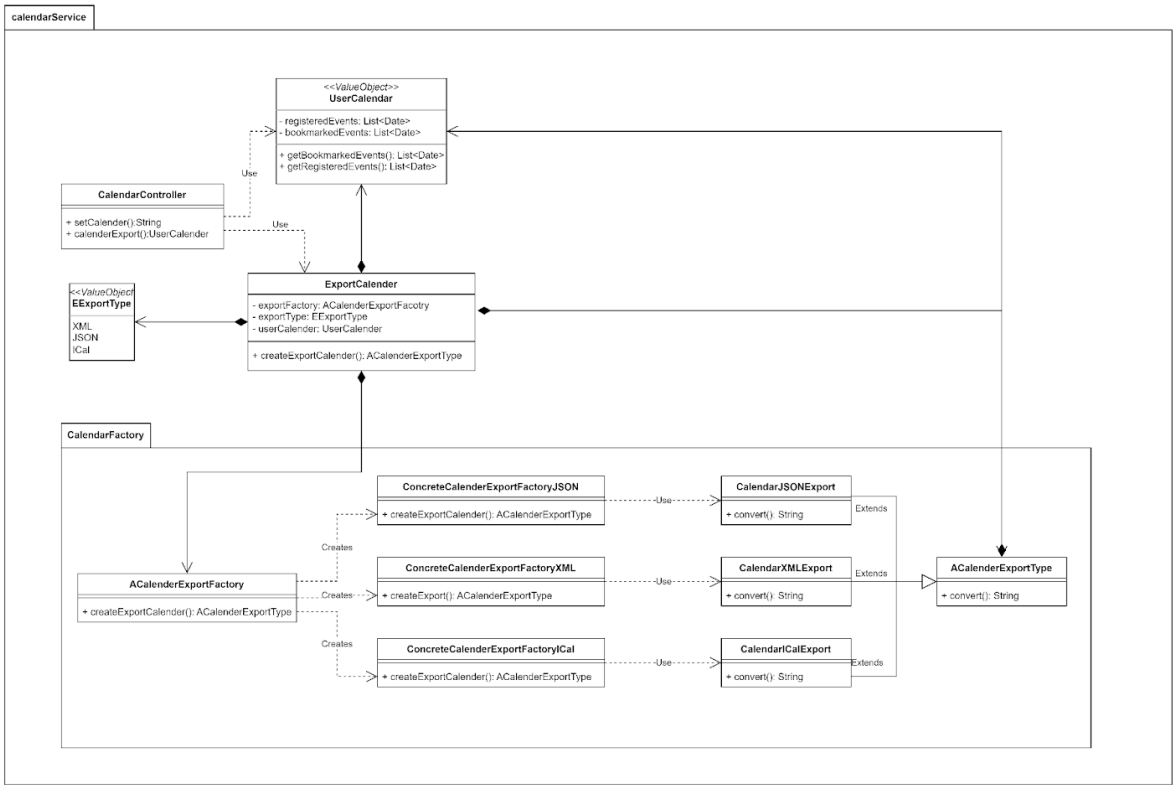
Search Service



The Search Service Domain is used to search for events by different criteria. The controller gets the search request with the criteria and the service forwards it to the Event endpoint to get the right events.

Calendar Service

It is important to note that our current model calendar service does not fully account for future interactions with other services or persistent data storage in the calendar. The specific connection between the calendar and export type, as well as how attendees will use these features, will be determined in further implementation.

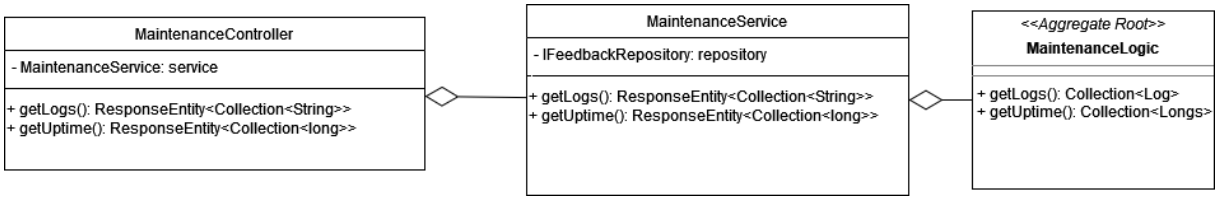


Overall, our model utilizes a factory pattern to generate a variety of export types for a user's calendar. These export types are defined as an enumeration and include JSON, XML, and ICAL formats. The user's saved events are represented within the userCalendar and therefore require data from other services or packages.

DDD-Building Blocks:

- Value Object.

Maintenance Service



The maintenance service is responsible for gathering data about the system and its services in terms of uptime and logs and shows it to an admin user.

DDD building blocks:

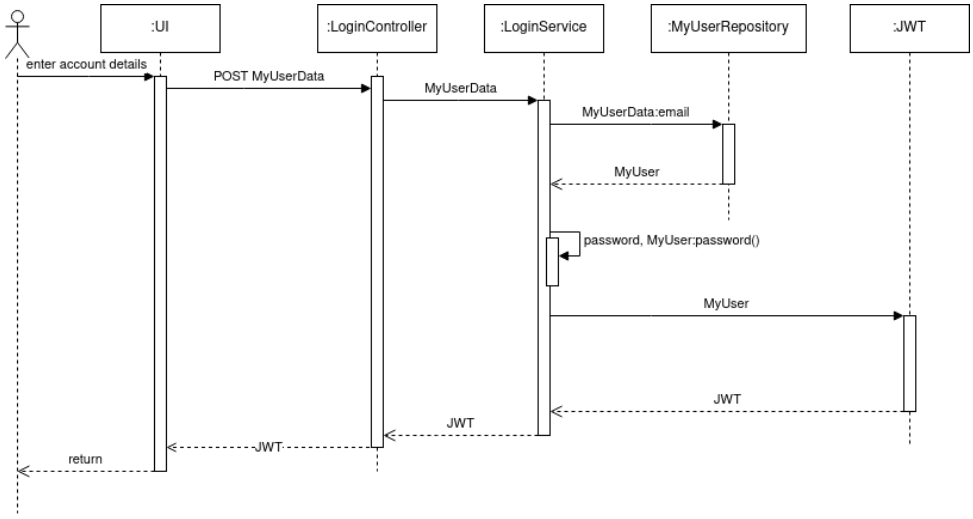
- Aggregate Root

3.3. Process View

Login Service

The following sections highlight the planned processes conducted by the login service. As already mentioned in the logical view, these are likely to be changed in the future and thus may not exactly reflect the used architecture.

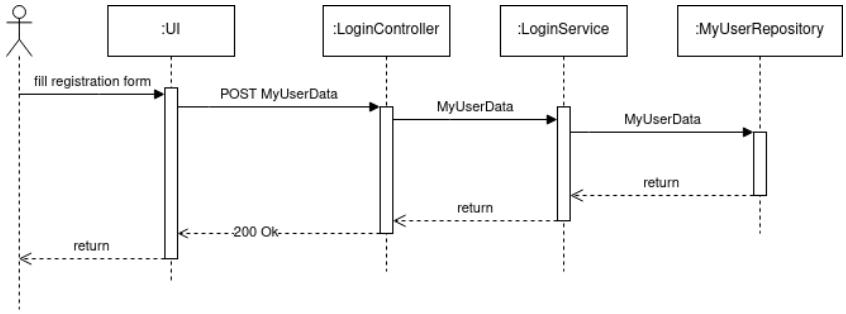
User authentication



The sequence diagram above highlights the process of a user authenticating with (in other words logging into) the system. The process is initiated by the user filling in his

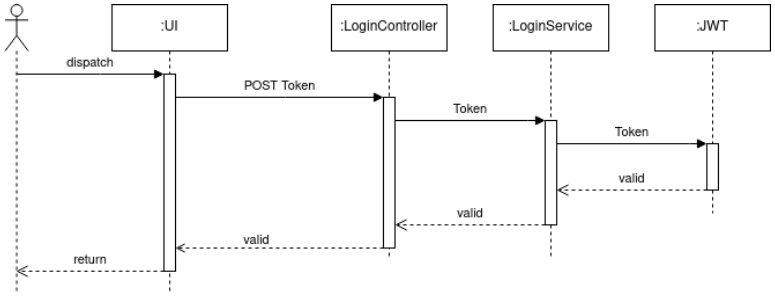
account details (email and passwords) and ends, in the happy scenario, with the UI receiving a JWT token, which can be further passed to other requests.

User registration



The sequence diagram above highlights the process of a user registering with the system. For this, the user has to provide their registration details, which will be further processed by the system components and ultimately saved in the user repository (in the happy scenario). After this process is finished, the user should be able to authenticate with the system.

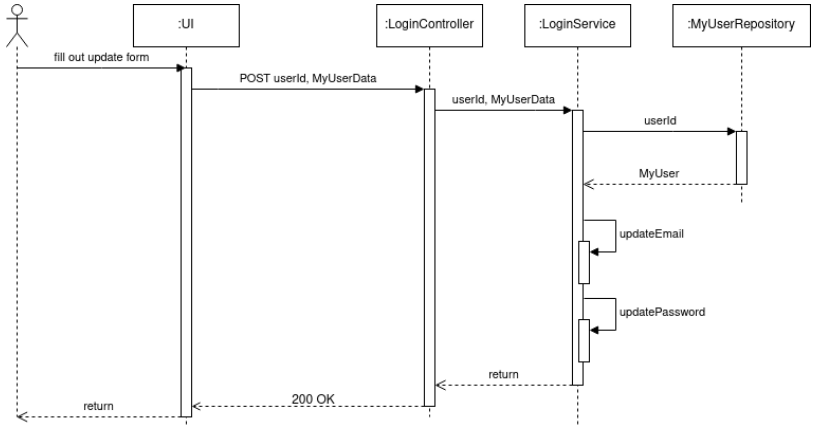
Token validation



The login service is further responsible for validating existing tokens. This process is initiated by the user conducting any operation, which requires authorization. The token is then simply passed through the system and validated.

Note: Whether such an endpoint is needed is dependent on the security architecture we decide to use. While to our understanding OAuth2 requires it, JWT can be used in a way where such an endpoint would be redundant. There, the token may only need to be checked against a public key, which could happen in the service requesting the authorization itself.

Update user

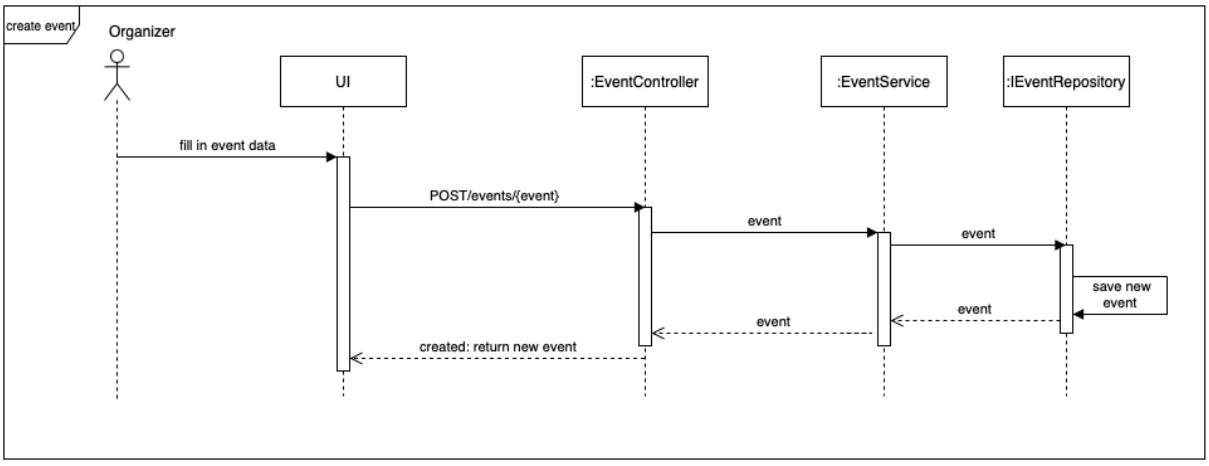


The login service further supports an update process. For this the user has to fill out an update form, which will be parsed by the system. From the user request the appropriate user is queried from the repository, for which the relevant information will be updated.

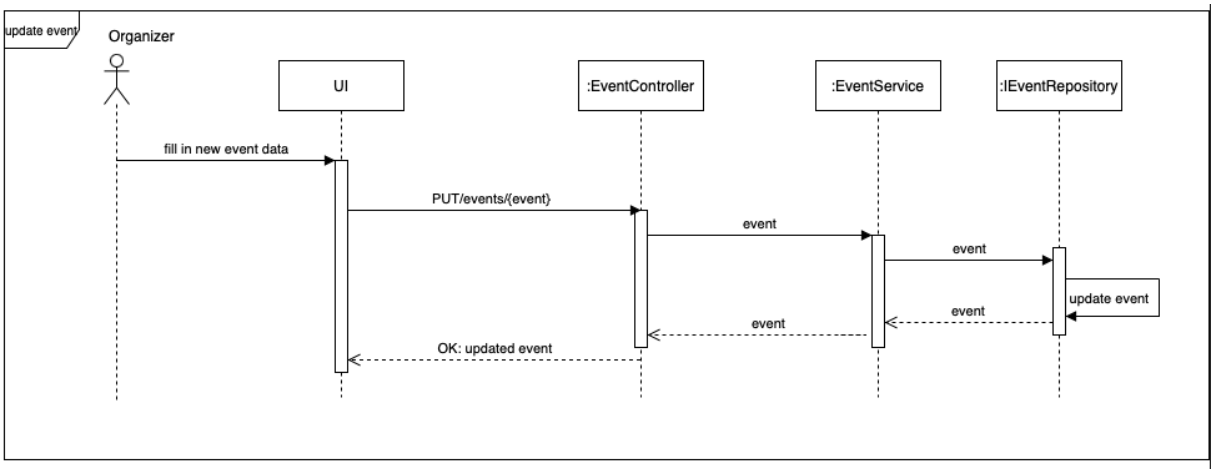
Event Service

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

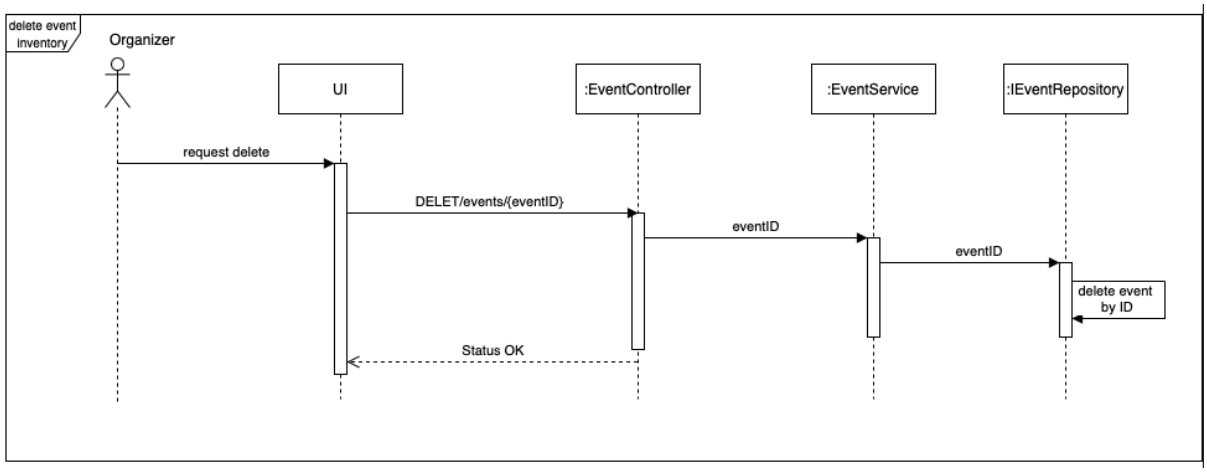
*... not mandatory field



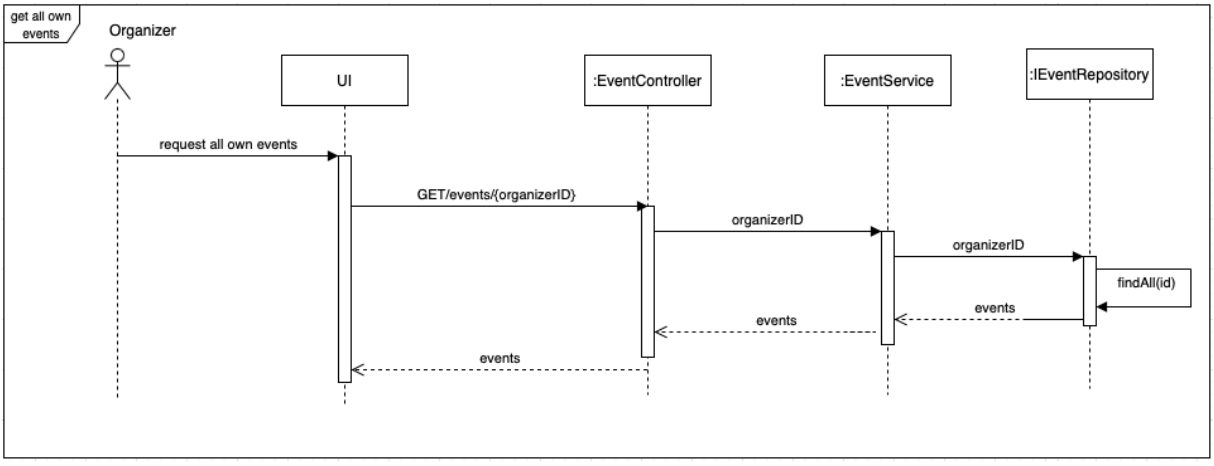
The sequence diagram above highlights the process of an event being created by an organizer. The process is initiated by the organizer filling in the event details (name, description*, capacity, date, type*) and ends, in the happy scenario, with the UI receiving a created and saved event.



The 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.

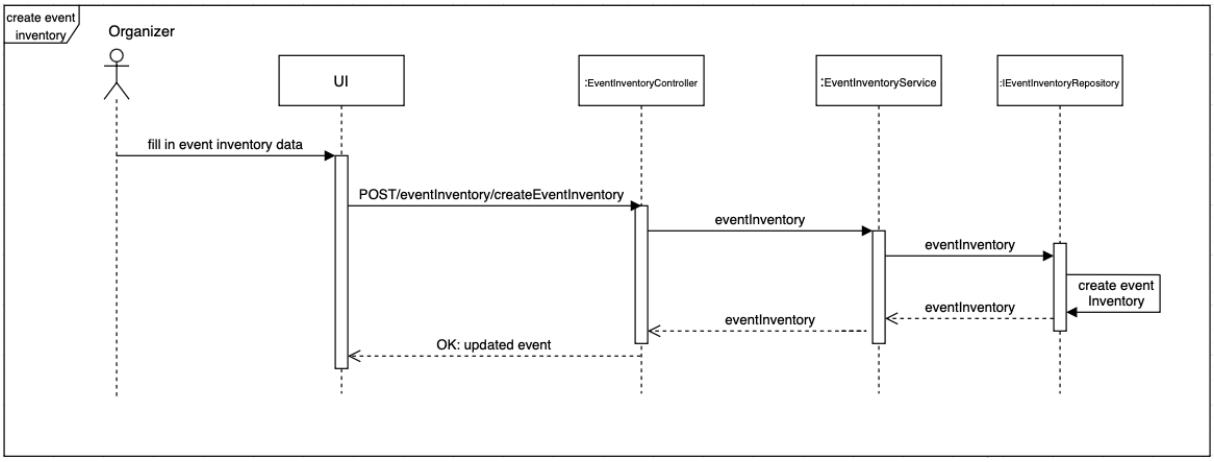


The sequence diagram above highlights the process of an event being deleted by an organizer. The process is initiated by the organizer requesting the deletion and ends, in the happy scenario, with a status update.

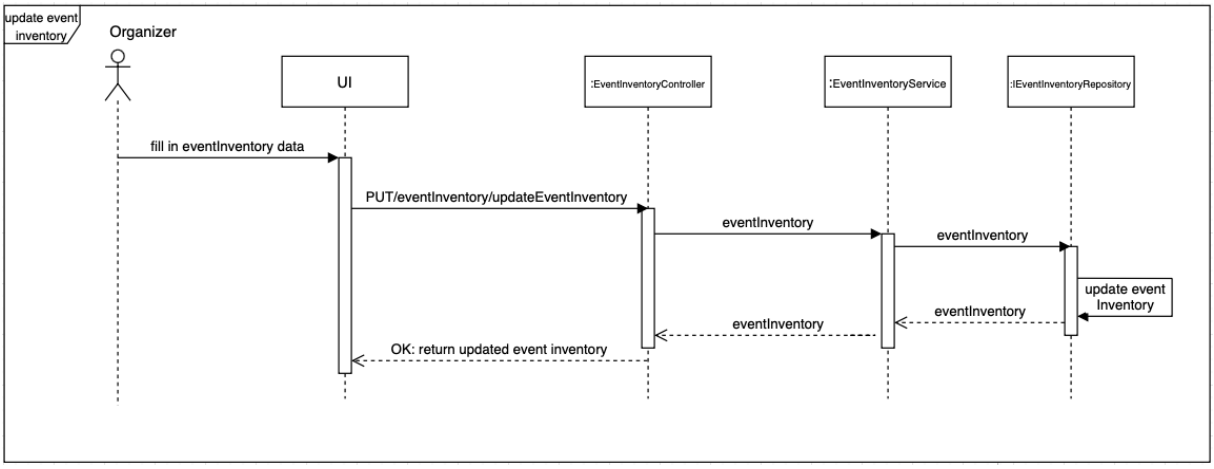


The sequence diagram above highlights the process of a request to get all events. The process is initiated by the organizer starting the request and ends, in the happy scenario, with the UI receiving all events which have this organizerID.

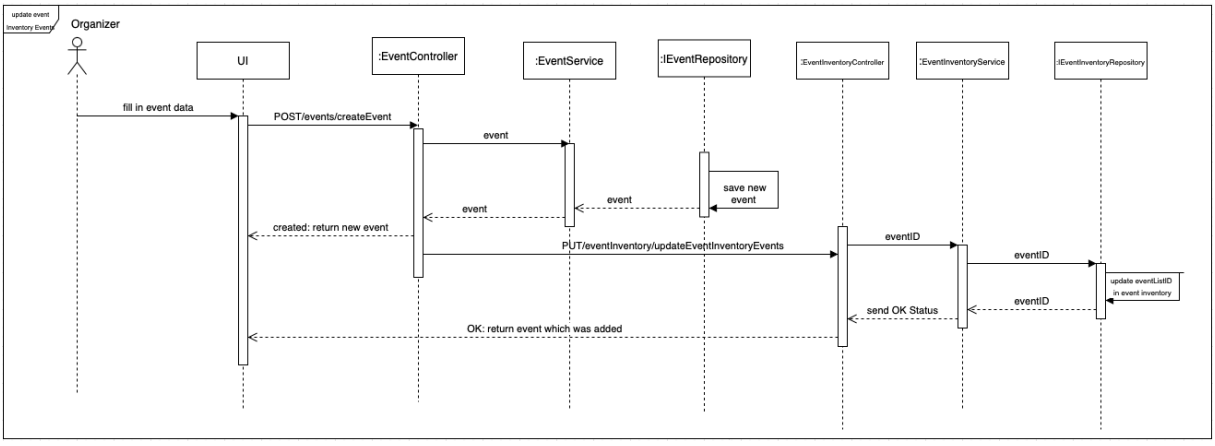
Event Inventory



The sequence diagram above highlights the process of an event inventory being created by an organizer. The process is initiated by the organizer creating an event and ends, in the happy scenario, with the UI receiving a created and saved event inventory.

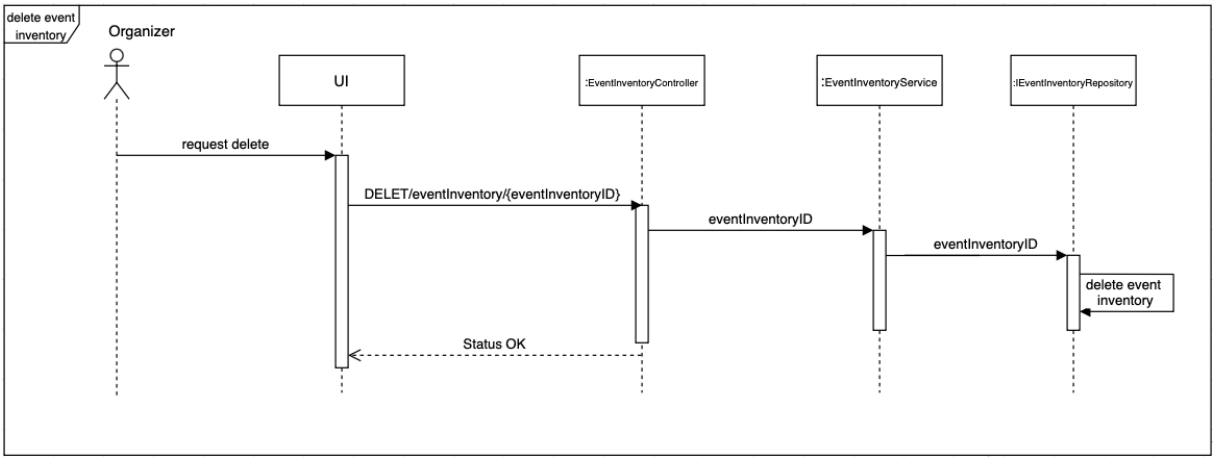


The sequence diagram above highlights the process of an event being created by an organizer. The process is initiated by the organizer filling in the event inventory details and ends, in the happy scenario, with the UI receiving an updated and saved event inventory.



The sequence diagram above highlights the process of an event being added to an existing event inventory. The process is initiated by the organizer filling in the new

event details and then it's checked if this organizer already has an event inventory. If so the process ends, in the happy scenario, with the UI receiving an created and saved event to the organizers event inventory.

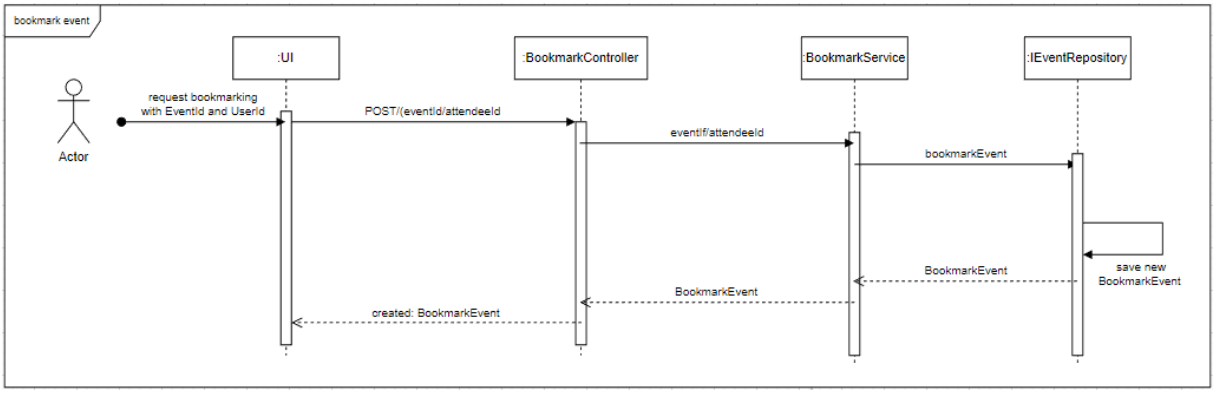


The sequence diagram above highlights the process of an event inventory being deleted by an organizer. The process is initiated by the organizer requesting the deletion and ends, in the happy scenario, with a status update.

Bookmark Event

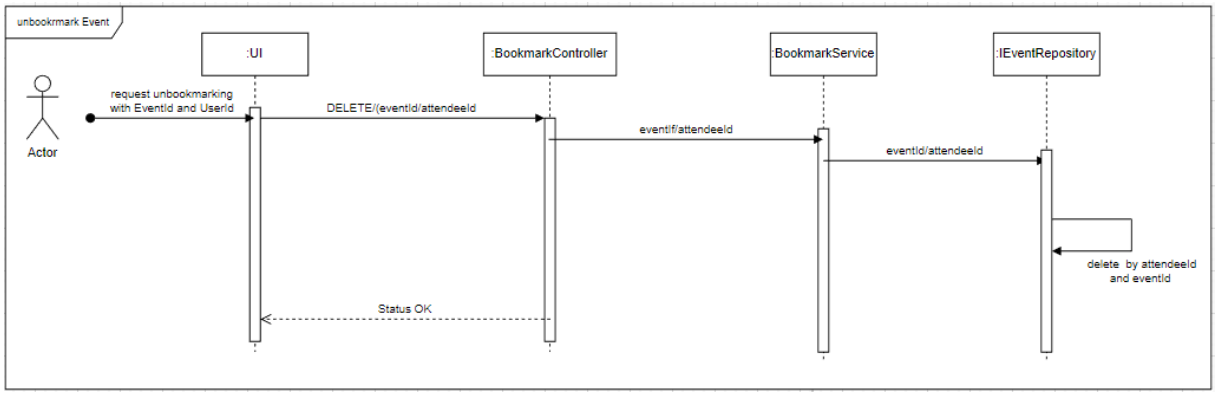
The following section describes the process of bookmarking and unbookmarking an event as well as the process of requesting bookmarked events per attendee.

Bookmark event



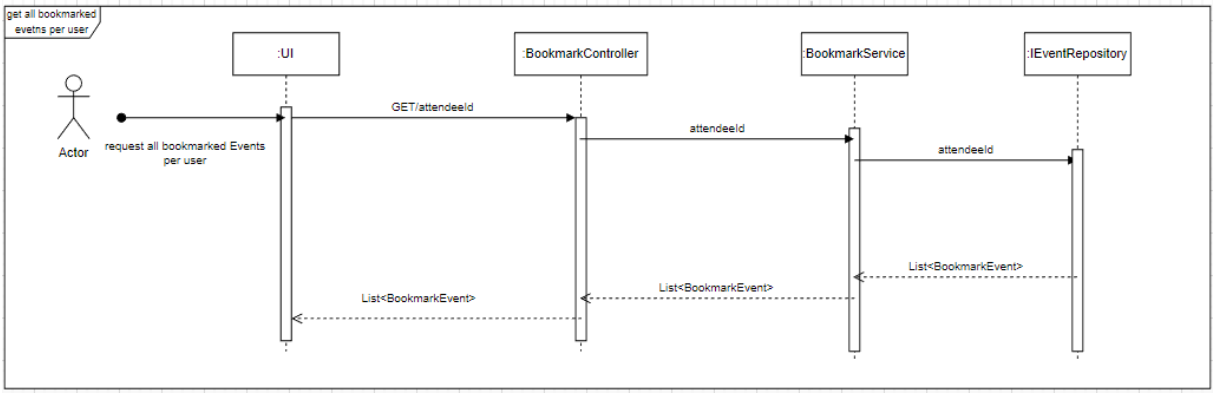
To bookmark an Event the User sends a POST request to the Controller and with the eventId and its attendeeId. This Triggers the Bookmark Service, which handles the creation of a new BookmarkEvent for the database. The Service also returns the successfully created BookmarkEvent within the process.

Unbookmark event



To unbookmark an event the BookmarkEvent will be deleted in the database. The eventId and the attendeeId will be propagated to the service and the BookmarkEvent will be deleted in the database.

Get all bookmarked events per user

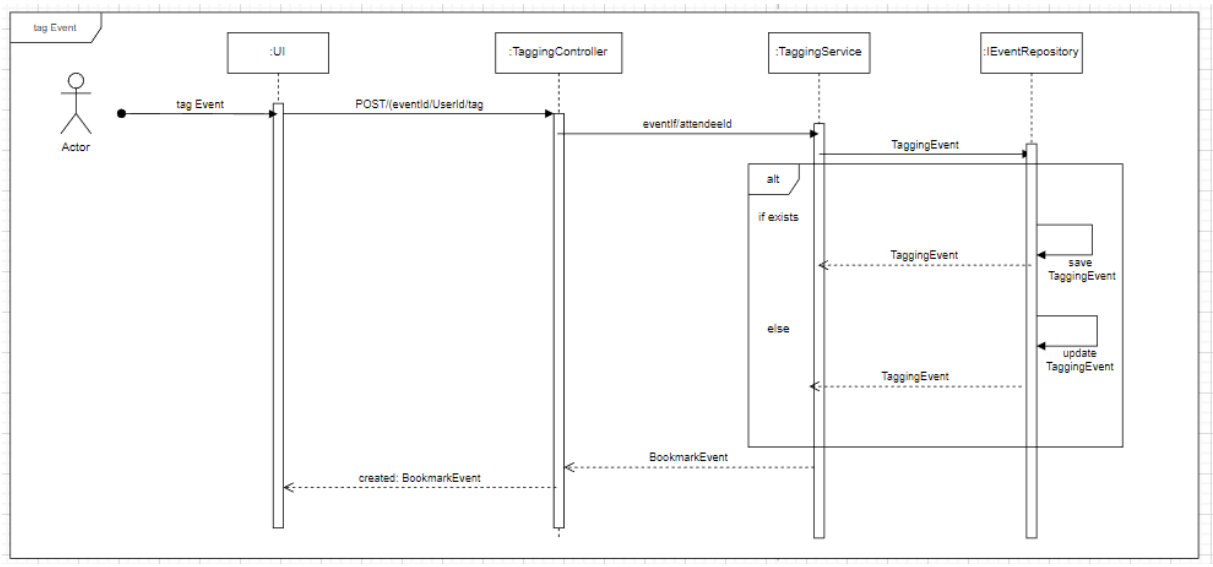


In order to facilitate additional processing within the services, it's necessary to obtain a list of all events bookmarked by each user. This can be accomplished with a GET request that uses the attendeeId as a parameter. The BookmarkService is responsible for handling this request and searching the database for bookmarked events associated with the specified attendeeId. Once the search is complete, a list of bookmarked events will be returned to the user.

Tagging Event

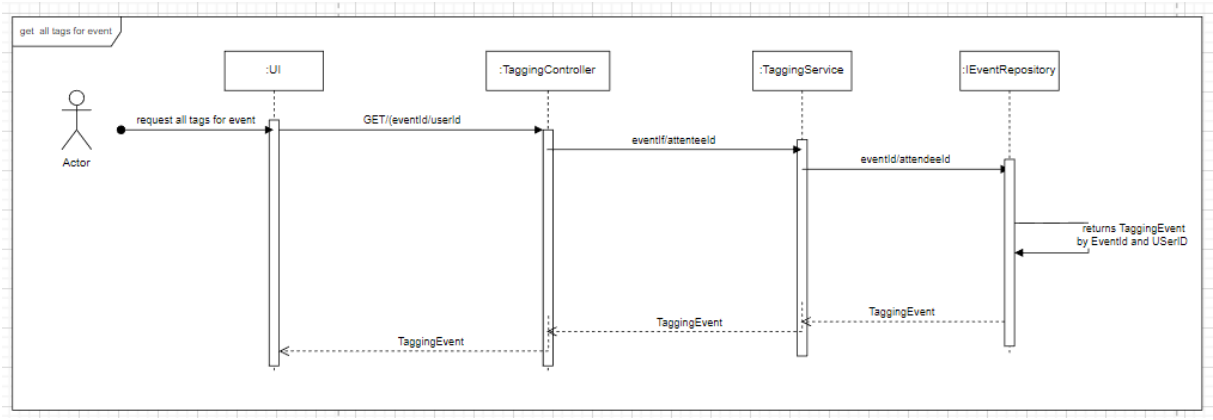
The following section describes the process of tagging and untagging an event as well as the process of requesting tags per user and event.

Tag event



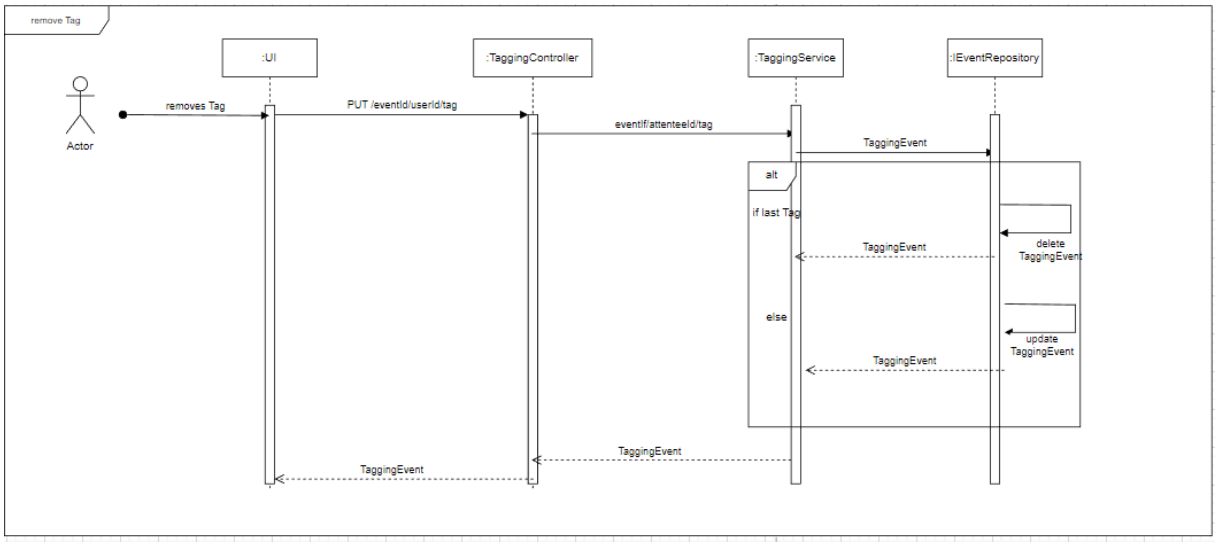
When a user wants to tag an event, they can propagate a POST request to the Controller Endpoint. The endpoint is responsible for creating or updating the TaggingEvent, depending on whether it already exists. If the event has not been tagged previously, a new TaggingEvent will be created. Otherwise, the existing TaggingEvent will be updated in the database. Regardless of whether a new event is created or an existing event is updated, the TaggingEvent will be returned to the user.

All tags for event



To retrieve all tags associated with an event, a GET request that includes both the userId and eventId is required. The request will be propagated to the service, which will use the Repository to search for TaggingEvents associated with the specified eventId and userId. The complete TaggingEvent objects will then be returned to the user.

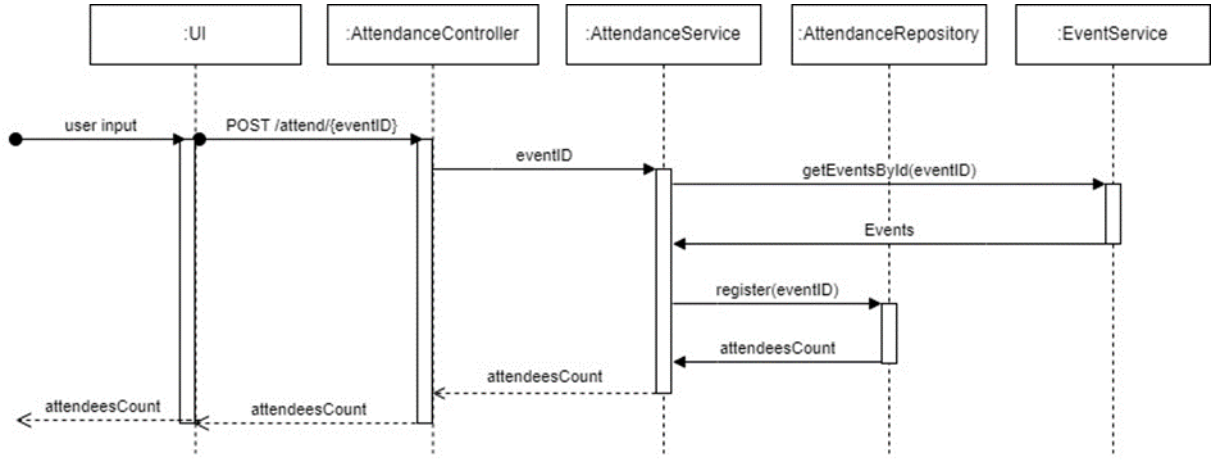
Remove tag



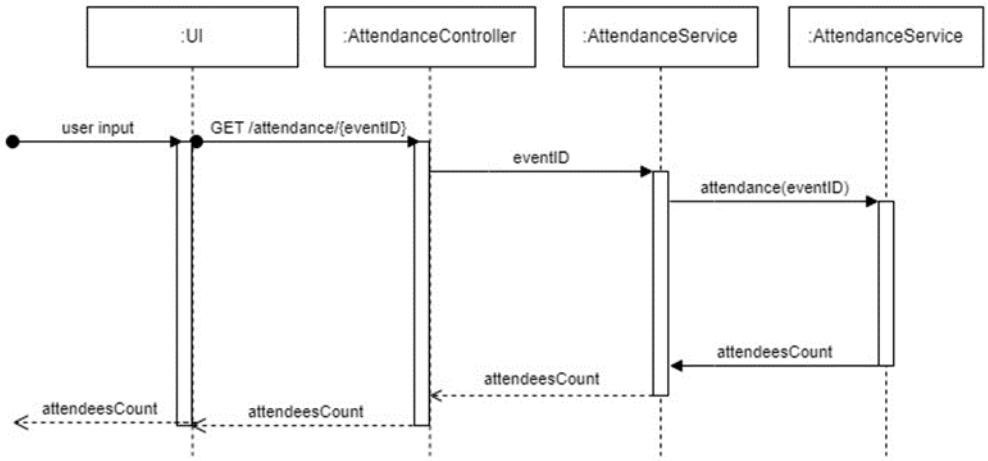
To remove a tag from a TaggingEvent, a PUT request is sent to the Controller and then propagated to the Service. The service checks whether the tag is the last one in the TaggingEvent. If it is, the entire TaggingEvent is removed from the database. The Repository returns a modified TaggingEvent object or null in case the event has been deleted.

Attendance Service

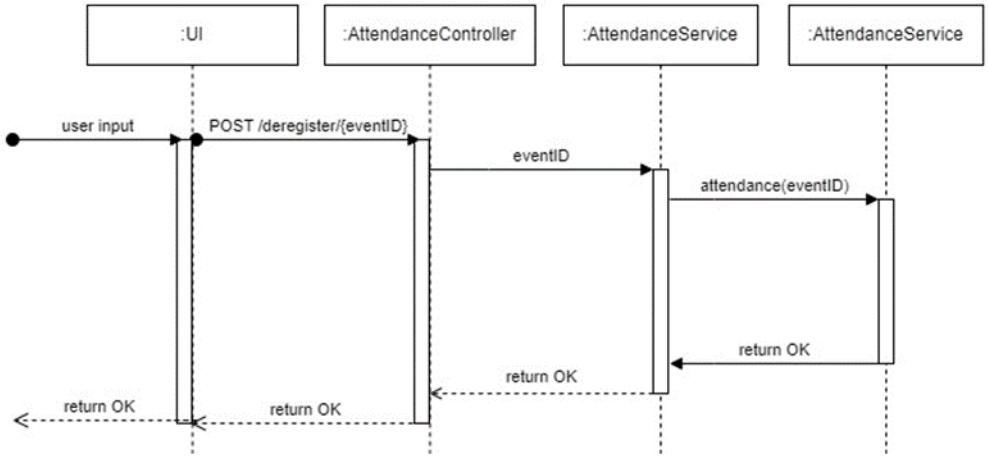
This sequence diagram illustrates the communication between the UI, the controller, the attendance service, and the event service for the three endpoints: Attend, attendance,and deregister.



When one user wants to attend an event, a POST request with the 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.



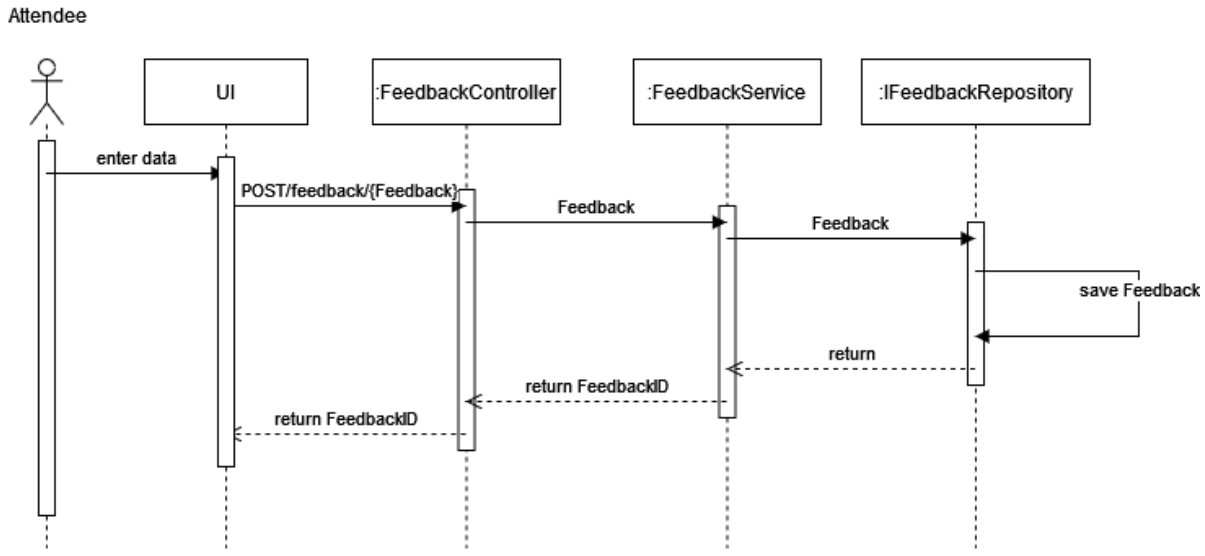
To get the current attendance count, a GET request with the corresponding eventID is sent to the controller. In the end, the attendance count gets returned.



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

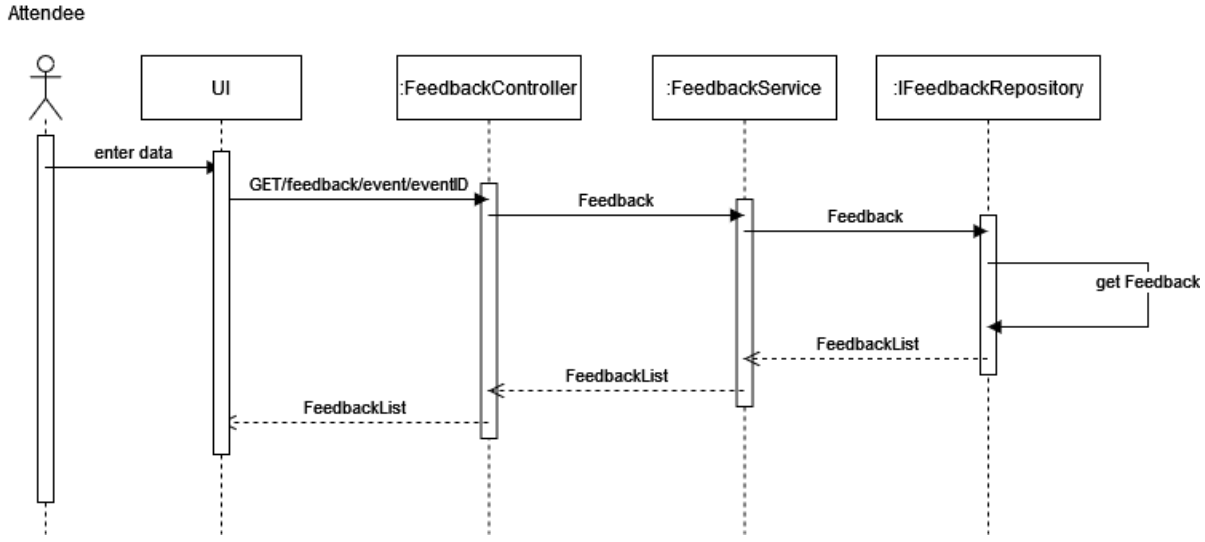
Feedback Service

Give Feedback



The above sequence diagram shows the process of feedback being created by a registered attendee. It starts with the user inputting the necessary data into the UI which sends the request to the controller with the feedback in the body of the message. This process returns the ID of the feedback back to the UI.

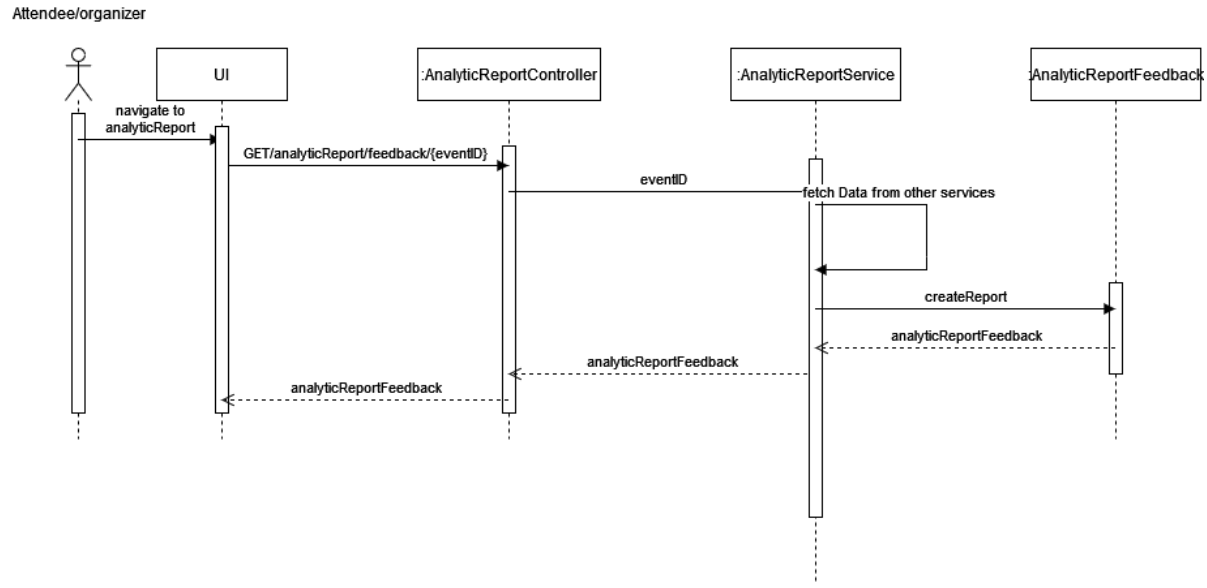
See Feedback



The above sequence diagram shows the process of the feedback being fetched for the UI after the user navigated to see the feedback for one specific event. The UI sends the according request to the controller. The id of the event the feedback is fetched for is passed as a path variable. This process returns a FeedbackList, an object that contains a list of feedbacks, to the UI.

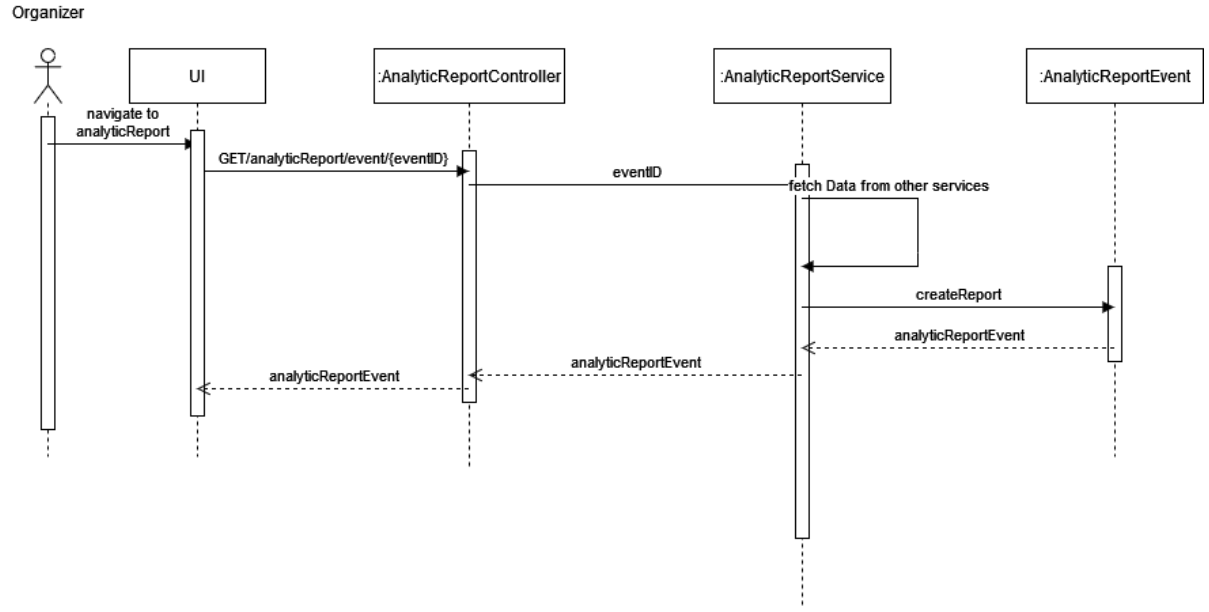
Analytic Report Service

See Analytic report for feedback



The above sequence diagram shows the process of the analytic report being created and passed to the UI. For the process to start the user navigates to see the analytic report about events. The UI sends the request to the controller, the eventID is passed as a path variable. The Service then fetches the necessary data from the feedback service endpoint and creates an analytic report which is passed back to the UI.

See analytic report for event



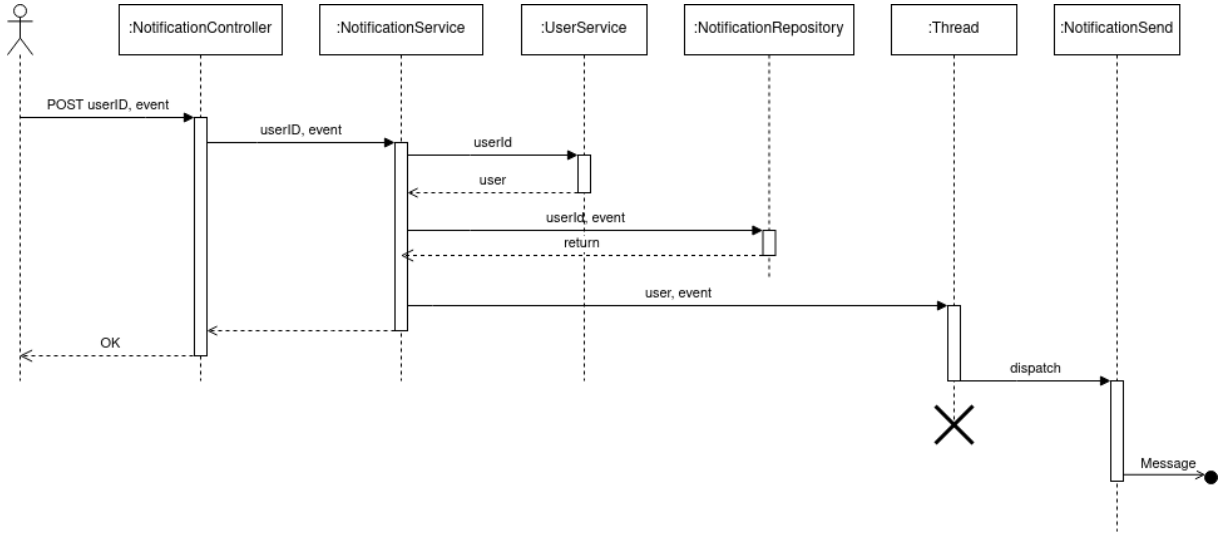
The above sequence diagram describes the process of the analytic report for an event being created and passed to the UI. The process is the same one as the one for the analytic report feedback. The only differences is that only the organizer can initiate this process and a different report, called analyticReportEvent is returned to the UI.

Notification Service

The following sections describe the main processes conducted by the notification view. For simplicity's sake, the entire `sendNotification` service has been abstracted away, as those steps would be the same for all identified processes and is not relevant to the important information processing structure.

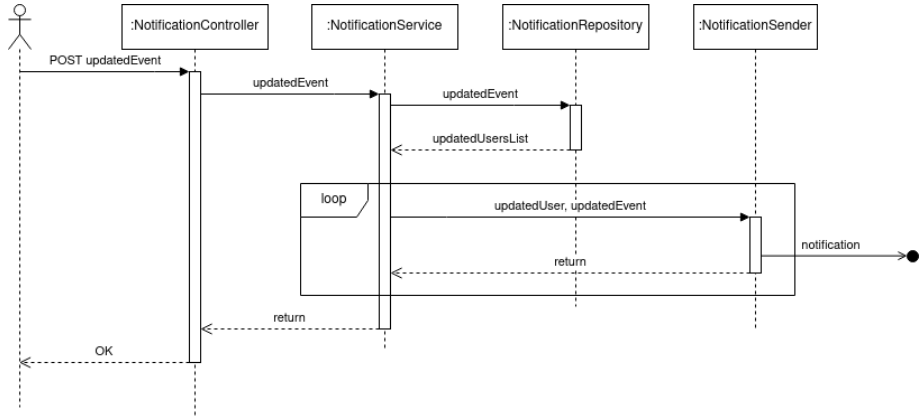
Further, even though the user is modeled as the triggering actor of the notification service, this is not entirely true. The notification service gets called as part of other actions the user performs, such as an organizer updating an event, and is not directly called by the user.

Notification register



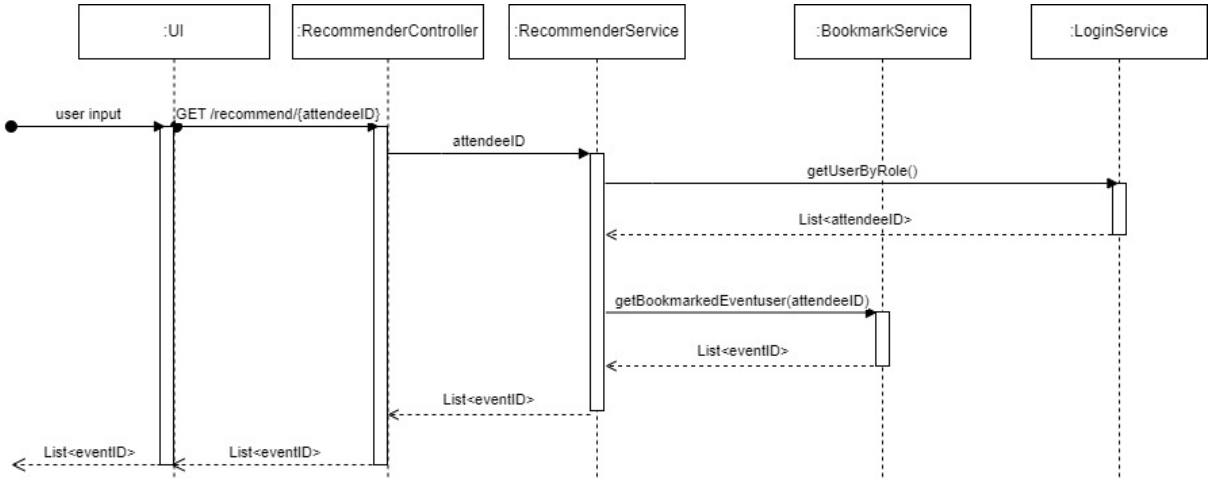
This process is triggered whenever a user bookmarks or registers for an event. This action schedules a notification in the notification repository, which eventually leads to a notification being sent to the target user.

Notification updated event



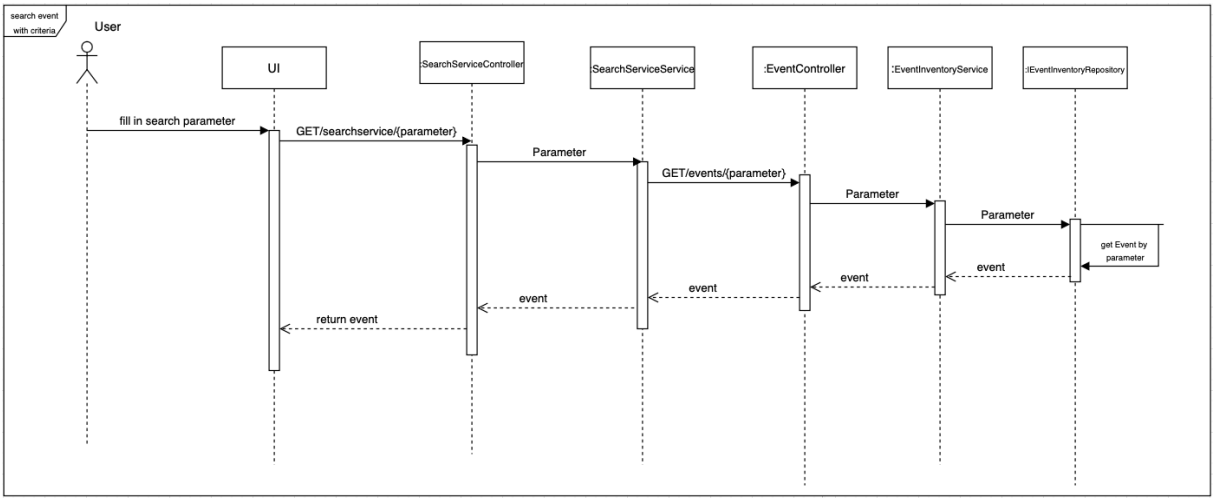
This process is triggered when the organizer updates an event. This triggers the notification service to search for all users, who are registered for a notification from this event and send them a notification about the change and update the old event.

Recommender Service



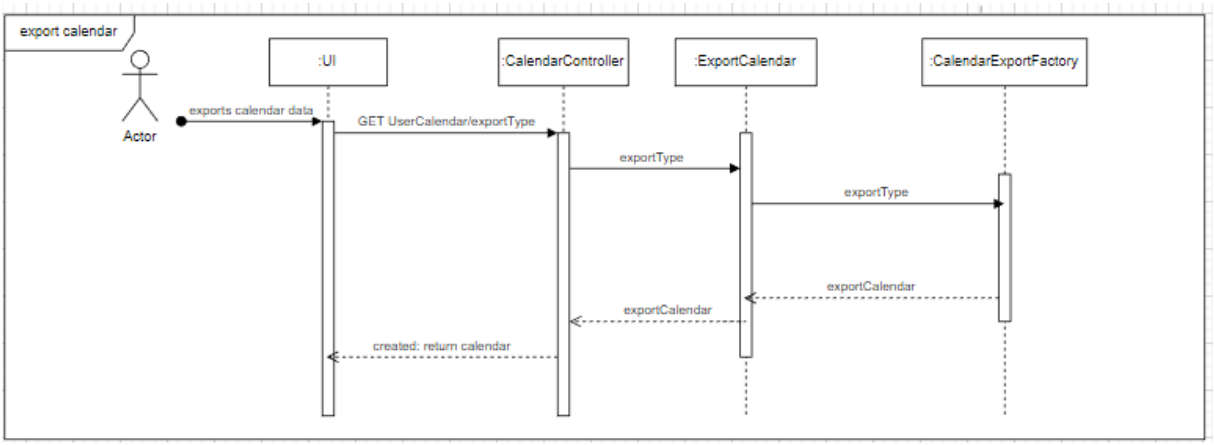
The sequence diagram of the recommender service shows the communication between the UI, the controller, the recommender service, the bookmarking service, and the login service. The recommender service sends a request to the login service to get a list of all attendees and sends another request to the bookmarking service to get a list of potential events. The internal logic of the recommender service generates a list of recommender events out of this information that can be used by the notification service.

Search Service



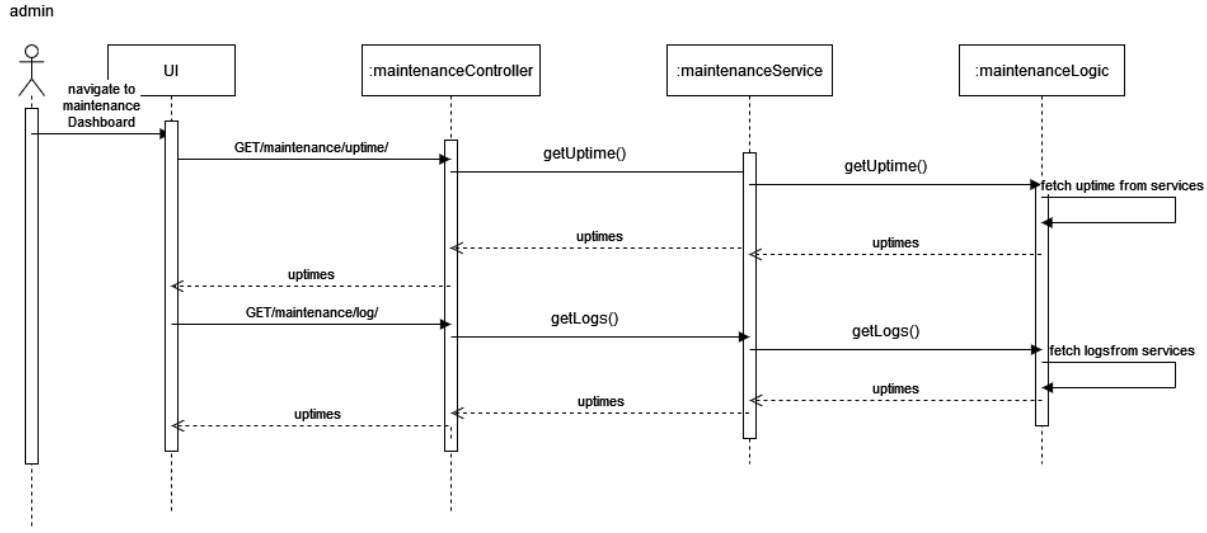
The sequence diagram above highlights the process of getting events based on different criteria. The process is initiated by the organizer starting the request and filling in the search parameter (which can only be one of the fields of the event) and ends, in the happy scenario, with the UI receiving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

Calendar Service



The calendar system processes user-provided calendar data, triggering the Controller endpoint and passing the export type to the Factory. The Factory then generates a new exportable calendar, which is returned to the end user.

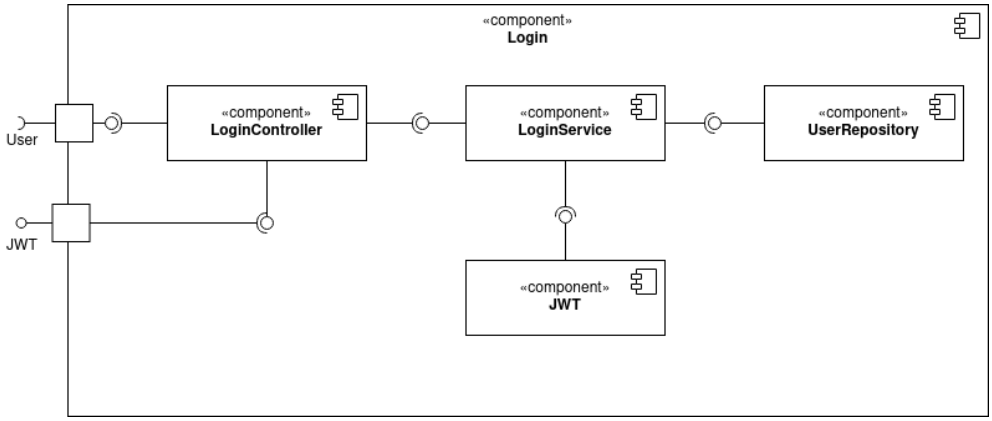
Maintenance Service



The above sequence diagram describes the process of the acquisition of the maintenance dashboard, after the user navigates the UI to display it. The way it is implemented right here and now is that the data has to be requested from every service. This will change once the architecture changes from the monolith to the microservice architecture, but we have not decided yet how it will be achieved in the end.

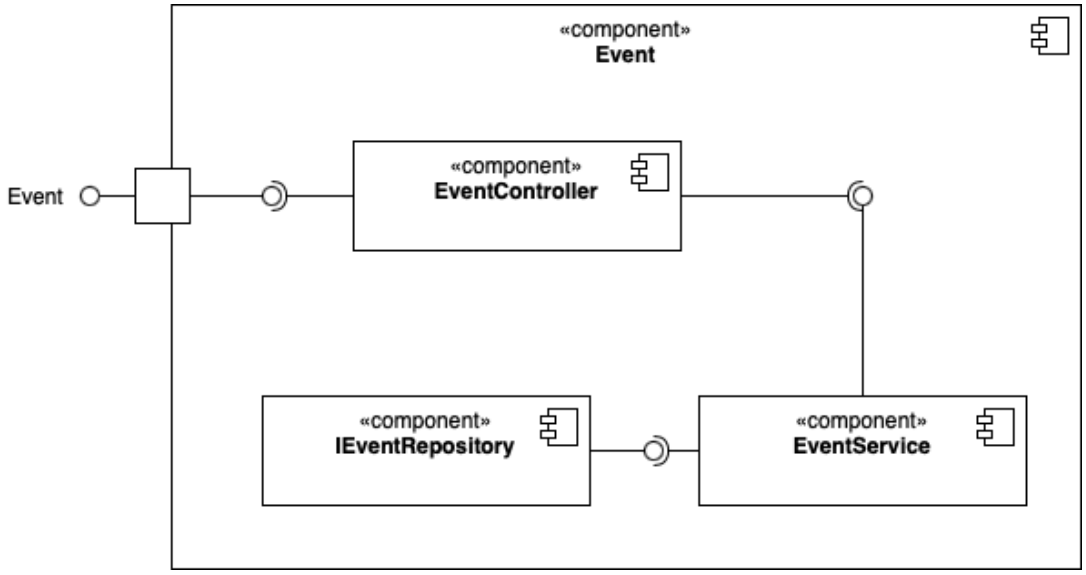
3.4. Development View

Login Domain



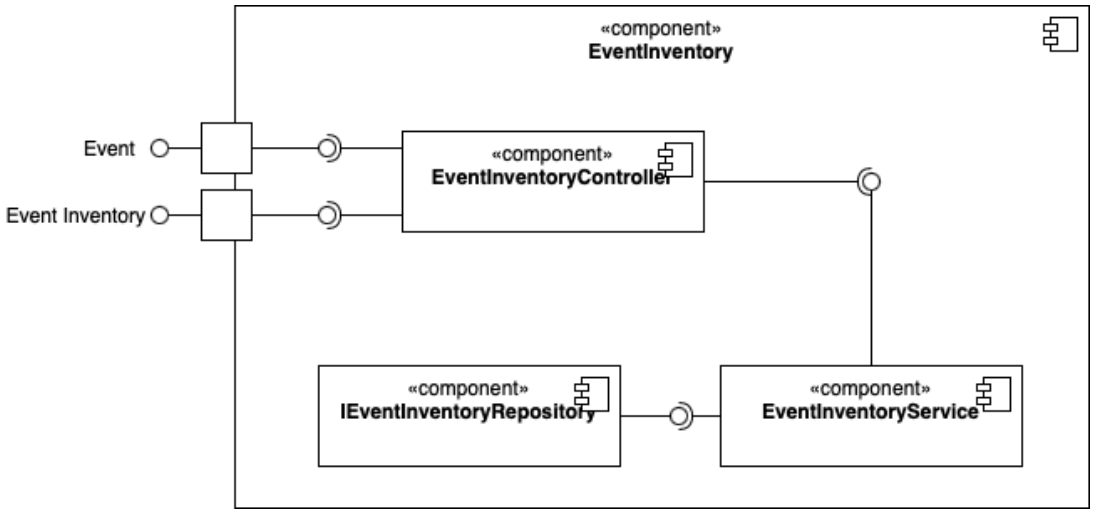
This components diagram shows the structure of the login domain. It needs to receive Users and issue JWT.

Event Domain



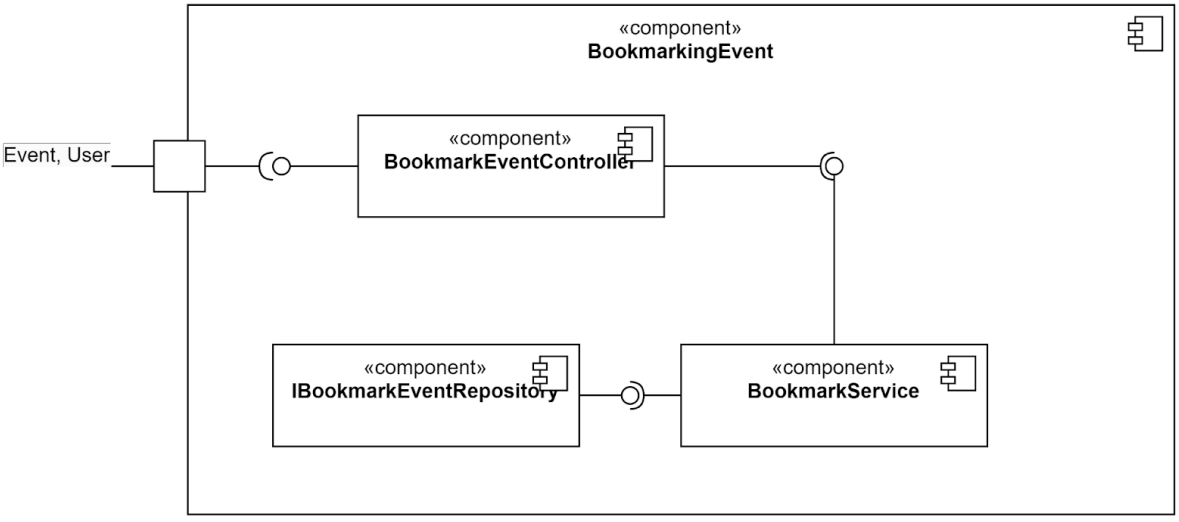
This component diagram shows the event domain. The controller gets an Event and returns either an Event or a List of Events. Several services have relations to the event domain, this is shown in the package diagram in 3.2.

Event Inventory Domain



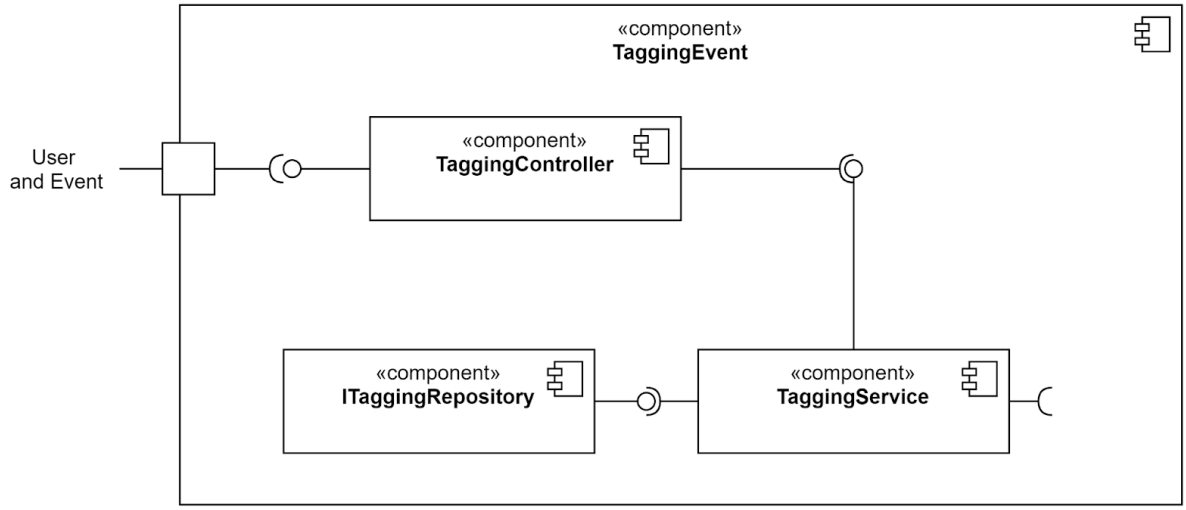
This component diagram shows the event inventory domain. The controller gets an Event Inventory or an Event and returns an Event Inventory or Event. The Event Inventory Domain has a relation to the event domain, as it saves the IDs of events.

Bookmark Event



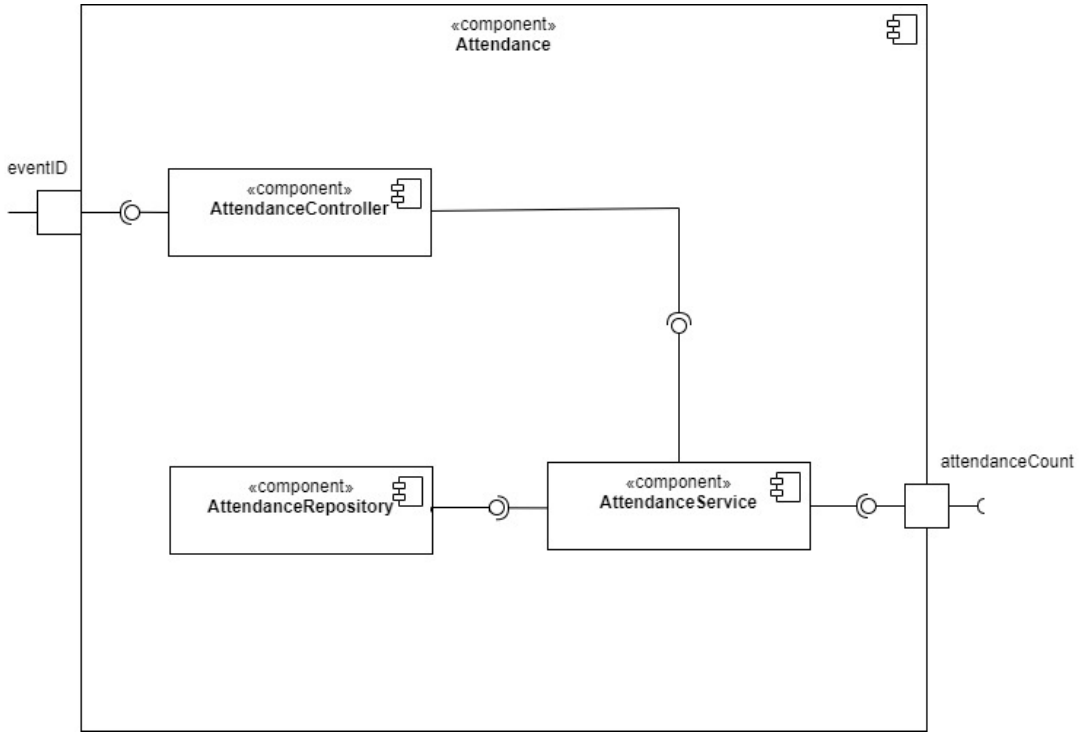
The component diagram shows the bookmarkingEvent, which gets user information and event information. It uses Event information and User information.

Tagging Event



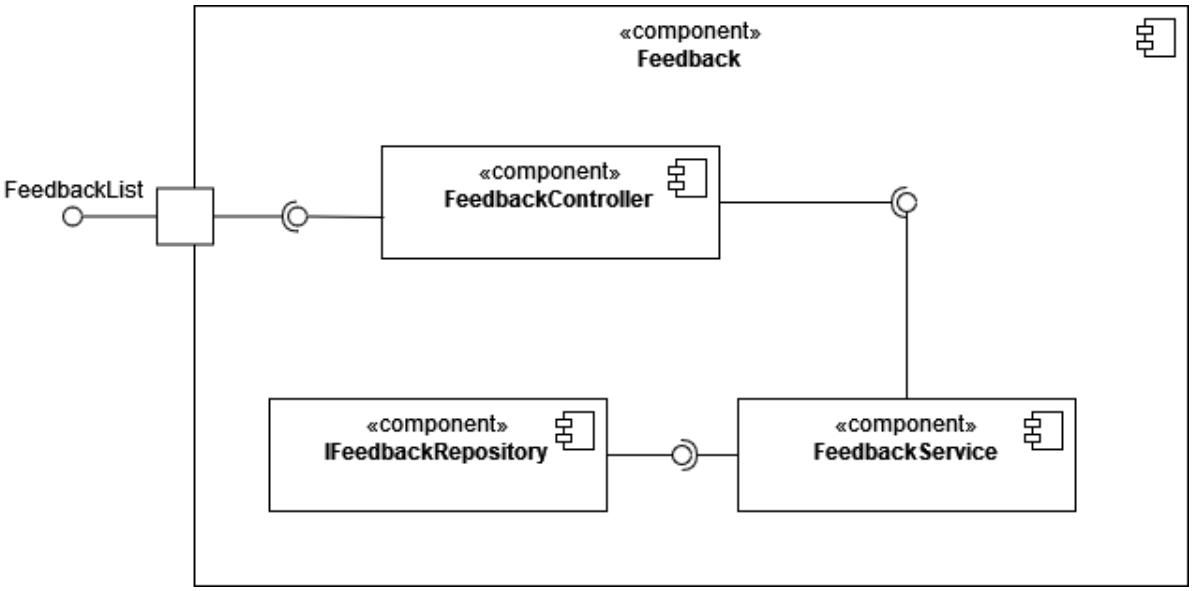
The TaggingEvent component has three components: a Controller, a Service, and a Repository, all of which are utilized for fetching and saving data from a data source. The component diagram illustrates that to create a TaggingEvent, both user and event information is required. The TaggingService processes this information using the Controller, which then returns the finalized TaggingEvent.

Attendance Service



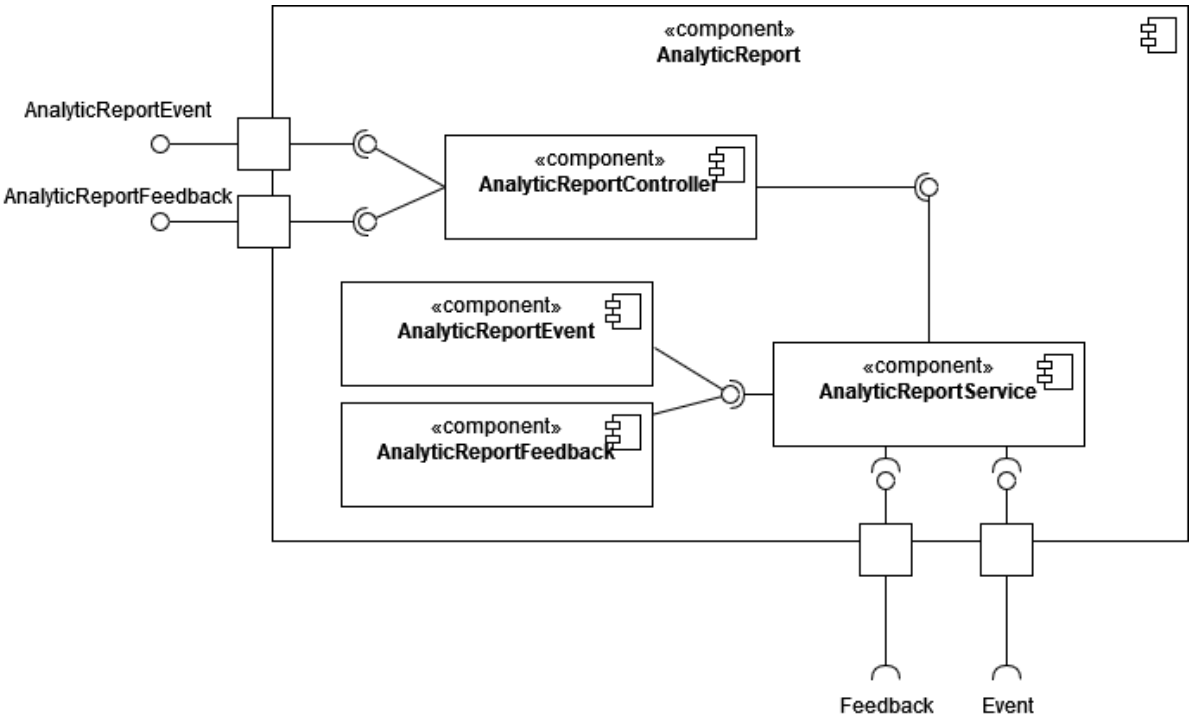
The component diagram of the attendance services shows the three main components controller, repository, and service as well as the input endpoint eventID and the output endpoint attendanceCount.

Feedback Domain



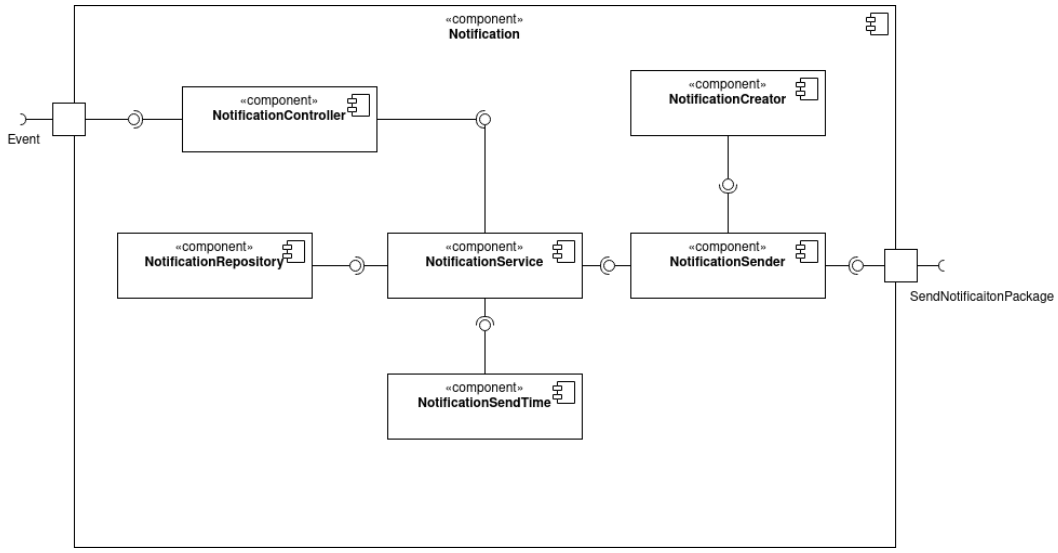
The component diagram of the feedback domain is shown.The controller provides interfaces for the crud operations and provides the results in form of a FeedbackList, or an id in case of the create.

Analytic Report Domain



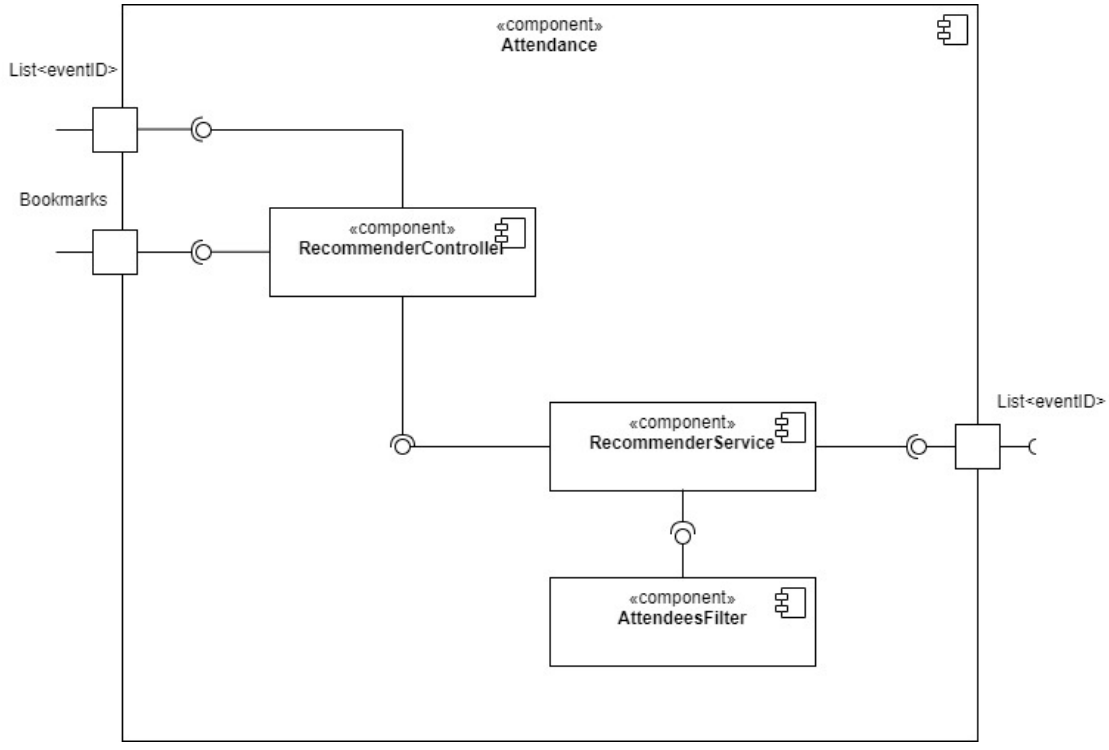
The component diagram of the analytic report domain is shown. The controller component provides two interfaces where an eventID can be handed over to receive an analytic report. Depending on the request a different report is returned. For the data acquisition, an interface between the feedback domain and the event domain is needed.

Notification Domain



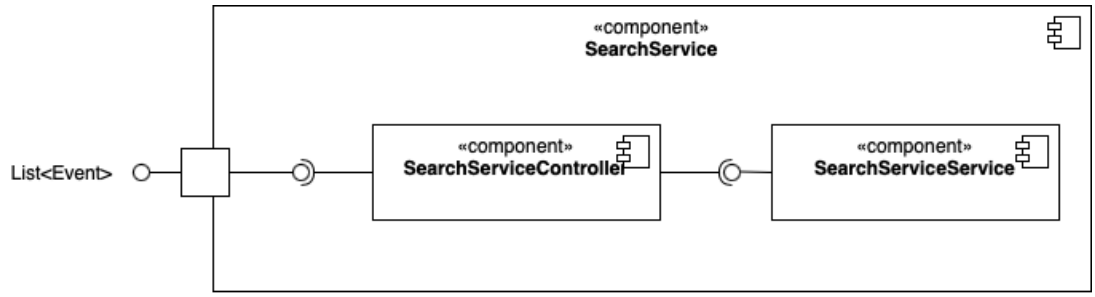
For the notification domain to function it needs an Event, about which will be notified, and a “sendNotificationPackage” (which is responsible for actually sending the notification). The internal connections of the components are also modeled.

Recommender Service



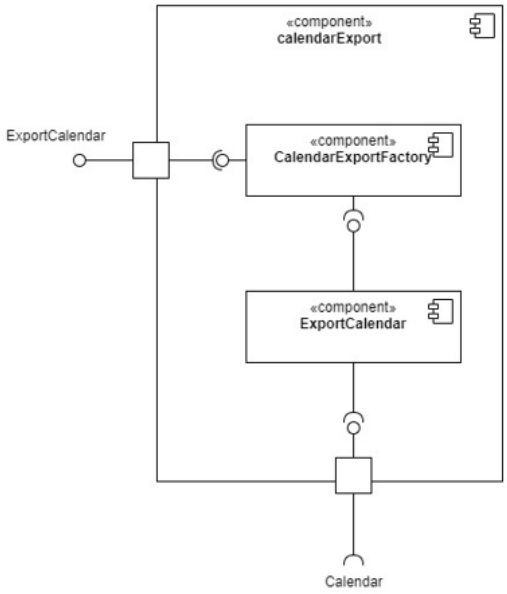
The component diagram of the attendance services shows the three main components controller, service, and filter. The whole attendance “blackbox” is connected by the input variables eventIDs and Bookmarks and the outputs a List of eventIDs

Search Service Domain



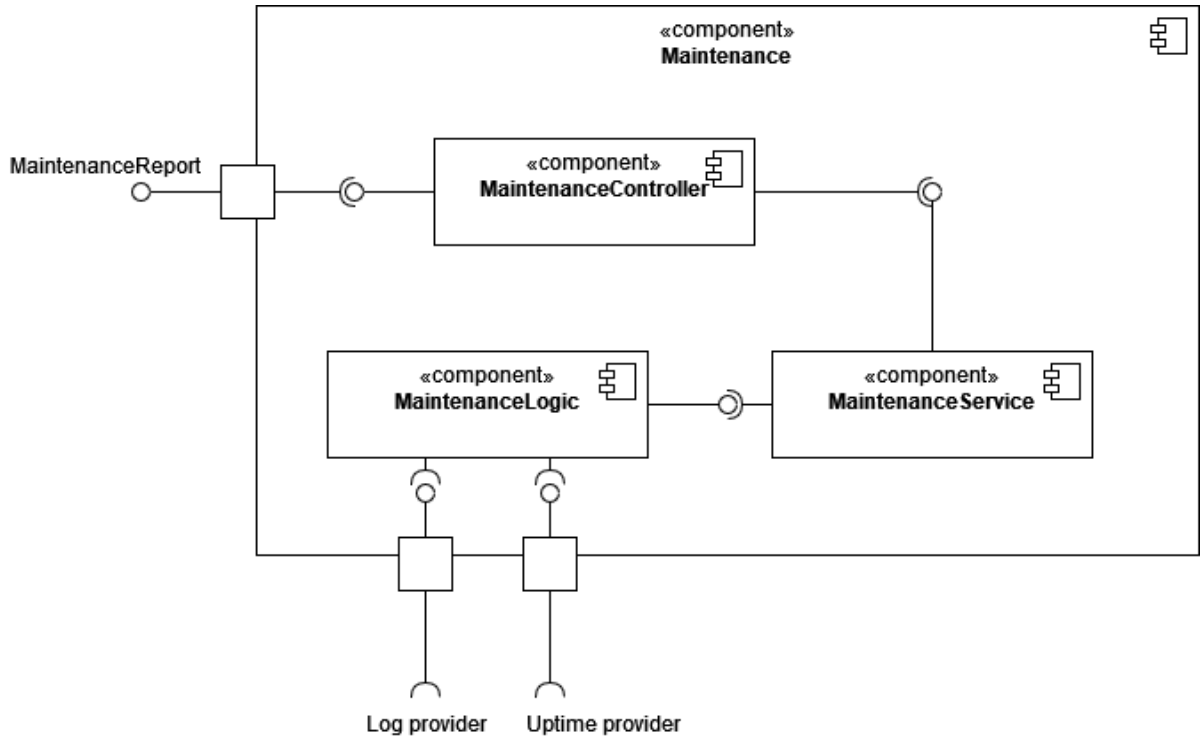
This component diagram shows the search service domain. The controller gets a parameter, which correlates to a field of an event, and returns a List of Events. It works with the Event Controller..

Calendar Service



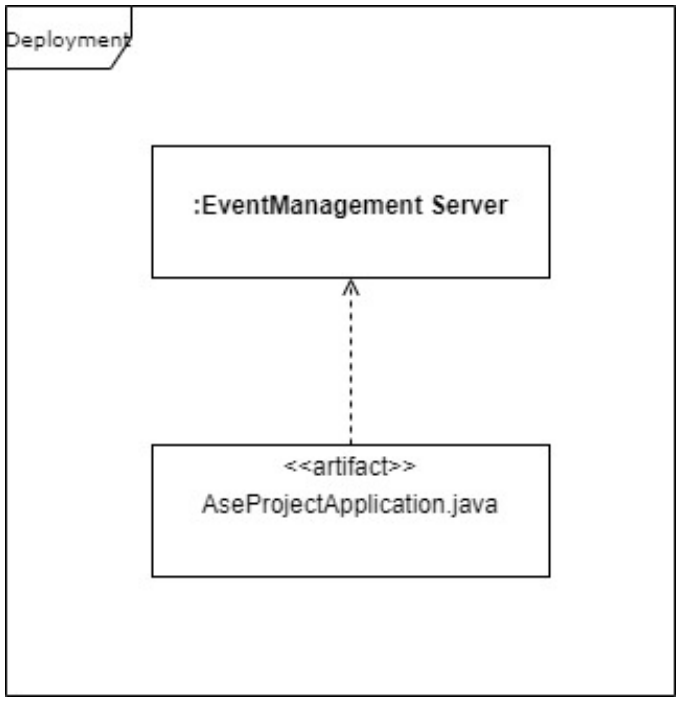
The component diagram shows the calendar service which gets calendar data and creates a calendar object in the exportable format. The creation of the exportable calendar with the factory is in this diagram abstracted in one component, CalendarExportFactory, which contains the components of a factory pattern.

Maintenance Domain



The component diagram of the maintenance domain is shown. The maintenance controller provides an interface where a maintenance report is returned. For this report, the Logic component fetches the logs and uptimes via an interface. How this interface looks differs from the monolith architecture to the microservice architecture and is not yet finally decided for the latter.

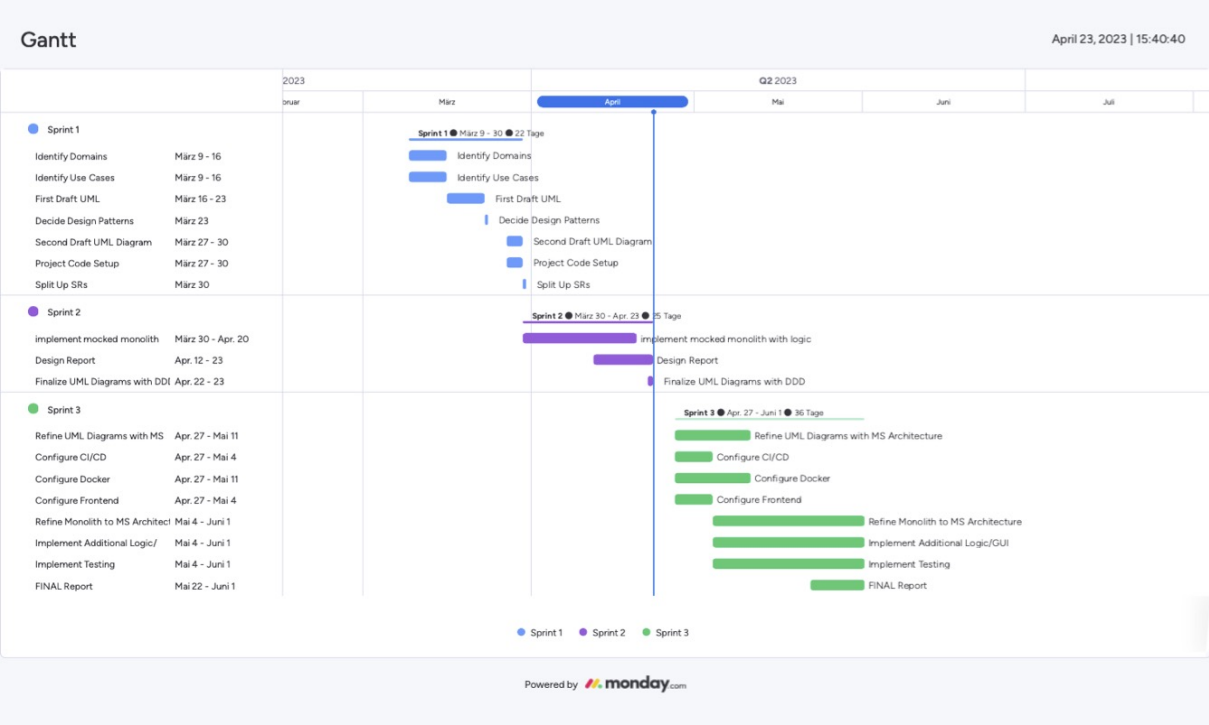
3.5. Physical View



This deployment diagram should illustrate the execution architecture of the current monolithic system. This system can be deployed on a single device by executing the “AseProjectApplication.java”, for example in the development environment IntelliJ. The device which executes the file acts as a server and will provide all endpoints under the local host. The server device is not specified further in the deployment diagram, because the focus of this project lies on the software side. The final product will later be separated into the following microservices, which can be executed independently: AnalyticReport, attendance, bookmarkEvent, calendarExport, Event, eventInventory, feedback, login, maintenance, notification, recommender, searchServiceEvents, sendNotification, and taggingEvent.

4. Team Contribution and Continuous development method

4.1. 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 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.

4.2. Continuous Integration, Delivery and Deployment Plan

In this project we have a master branch, which only contains the finished version for the deadlines and a development branch where we continuously upload our implementations. For releases, the development branch is merged into the master branch. After identifying each development issue, a new branch was created. After the necessary implementation, the branch for the issue was submitted for review, ensuring that at least one team member was assigned as a reviewer for this feature.

Additionally, developers with related issues should be assigned for further notice. By completing the review process and approval of the merge request is granted the branch could be merged into development and subsequently the implementation integrated into the further release environment. To ensure code quality, a standard test pipeline provided by Gitlab was utilized before merging into the development or master branch. However, we are committed to improving this process in the final assignment which specifically requires a CI/CD pipeline. One area of focus for improvement is the ease of integrating the pipeline within a Docker container for further development. This concern will be addressed as we continue to refine and enhance our development process and the further implementation of the business logic. Additionally, tests on the development branch should be verified after each merge of development issues as well as the master branch before the release.

4.3. Distribution of Work and Efforts

To effectively distribute the workload, we decided to define the project requirements based on the intended application and the corresponding system requirements outlined in the assignment. This allowed us to approach the design and implementation process as a distributed effort, with a focus on developing the project as a collection of microservices. Each member of the team was tasked with modeling two distinct parts of the project and contributing to the prototype for the first assignment. Given that the initial assignment prioritized modeling over business logic implementation, our primary focus was on ensuring a high-quality modeling process. Team members contributed equally to the modeling effort and maintained ongoing group meetings to discuss similar design approaches and combine our progress. Furthermore, we divided additional tasks such as Git management, research, and documentation setup equally among the team members based on their previous knowledge, skill set, and individual interests.

Contribution of Member 1:

- Development of Tagging/Bookmark Event service
- Development of Calendar Export
- Documentation of various topics
- issue management, defining issues related to domain and future services

Contribution of Member 2:

- Development and modeling of
    - Feedback Domain
    - Analytic Report Domain
    - Maintenance Domain

Contribution of Member 3:

- Attendance Domain
- Recommender Domain
- Deployment View Diagram

Contribution of Member 4:

- Event Domain
- Event Inventory Domain
- Search Service Domain
- Use Cases / Scenario View
- Writing of Report (general topics)
- Meeting documentation

Contribution of Member 5:

- Login Domain
- Notification Domain
- Research into OAuth2 and other security alternatives
- Resolving of issues with git branches
- Writing of report

5. How-to / mock-up documentation

As previously mentioned, we opted to use Postman for testing our application's endpoints and environment. To facilitate this process, we have provided a set of Postman scripts that include access to all endpoints.

Before running the application, it is necessary to set an environmental variable for the email service. The application can then be executed by using the main function in the AseProjectApplication class. The provided Postman script can be integrated into the locally installed Postman environment and can be executed either as a collection or for individual API testing. In the following we show the building and setup in the IntelliJ IDE which is highly recommended for setting up the project. Initial steps like cloning the git repository, open a project and/or the initial start of an appropriate technical device are not demonstrated and are assumed to be general knowledge.

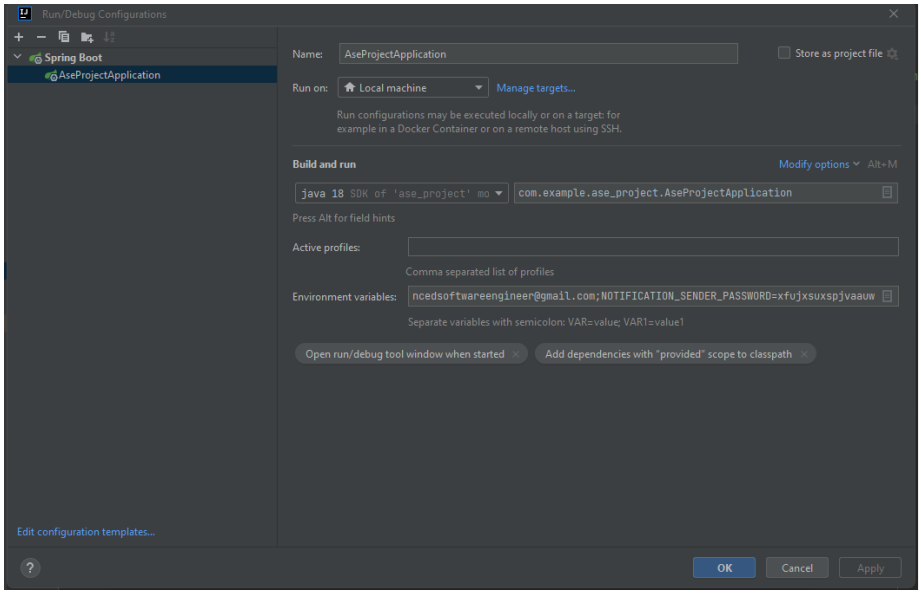
Step-by-Step Guide:

1) To run the application the environment variable must be set.

   For this click on "*Edit Configurations*" -> *"Modify Options"* -> "*Environment variables"* in the text field **Environmental variables** type the following parameter:

   ```
   NOTIFICATION_TARGET_EMAIL=<YOUR_NOTIFICATION_TARGET_EMAIL
   >;NOTIFICATION_SENDER_EMAIL=advancedsoftwareengineer@gmai
   l.com;NOTIFICATION_SENDER_PASSWORD=xfujxsuxspjvaauw
   ```

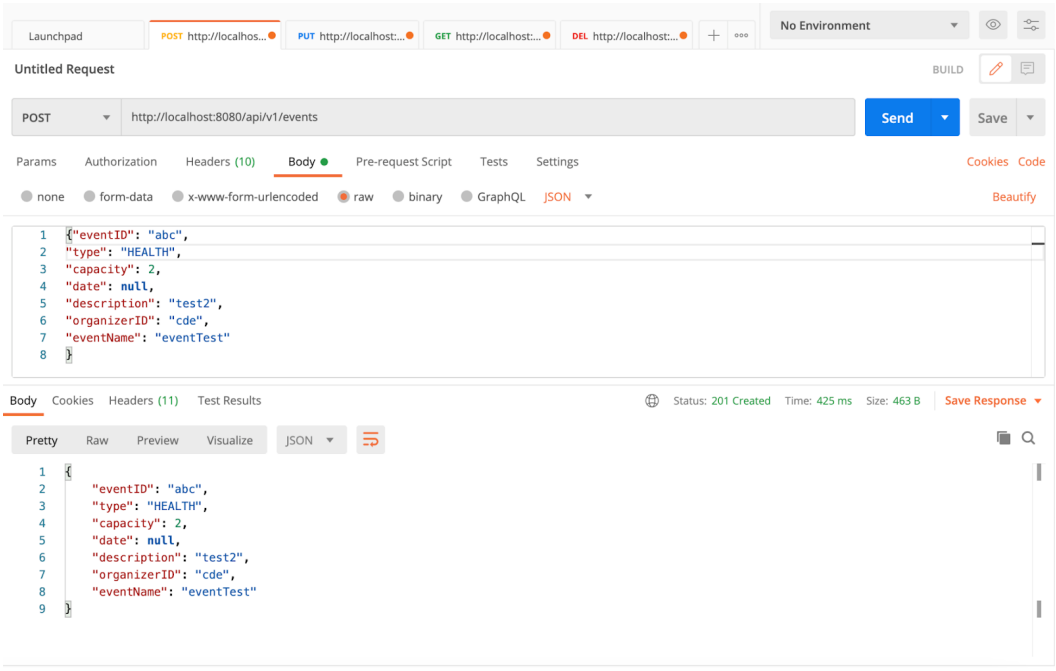
   Replace `<YOUR_NOTIFICATION_TARGET_EMAIL>` with your email, which should receive the reminder and update notifications. This is a temporary solution to mock the notification service. The remaining variables configure the notification send server.



2) Run the application by clicking on the “Run” or “Debug” button.
3) Integrate the endpoint tests in postman.

To integrate the provided test collections, postman must be installed. Then click on “Import” and upload the provided files via drag and drop or folder, if saved locally.

4) Demonstration of important features
    - Create Event



Use Postman as a UI to control the Event Endpoint. To create and save a new event, open a new POST request and paste the URL “http://localhost:8080/api/v1/events”. Then you have to check that the Header “content-type” has the value “application/json” so you can write a json into the Body. Open the tab “Body” and paste following JSON:

```
{"eventID": "abc",
"EEventTypes": "HEALTH",
"capacity": 2,
"date": null,
"description": "test2",
"organizerID": "cde",
"eventName": "eventTest"
}
```

Date is null for simpler usage of postman, but a correct date could be filled in. After that you can send the request and you will get the created Event back. (In the illustration the first part is the request and the second part is the response.)

In order to register for an event to attend it, simply make a GET request in Postman with the following URL “http://localhost:8080/api/v1/attend/{eventID}”

The evendID corresponds to the Event for which the logged in user should get registered. For example, the URL for the event with ID 101 would be following:

http://localhost:8080/api/v1/attend/101

If the request was successful, then it returns the current number of attendees which are overall registered for this specific event.

Reminder notification mock-up

The postman scripts allow you to also test the reminder notifications (i.e. notifications about upcoming events). For this:

1. Navigate to the “Notification” collection and select the POST register endpoint
2. View the body of the request
3. In the body JSON, edit eventDate. Keep in mind the reminder will be sent exactly one day before the event
    1. eventDate follows the format YYYY-MM-ddThh:mm:ss
    2. as the date insert tomorrow's date
    3. set the time to 1 minute from the current time. KEEP IN MIND THAT THIS SERVICE USES UTC+0 AS ITS TIME ZONE, SO DEPENDING ON YOUR LOCATION YOU MAY NEED TO MODIFY THIS NUMBER (for Vienna you need to enter a number 2 hours earlier than the current time)
4. you should receive a reminder notification exactly one day before the event date

If you don’t want to go through this trouble, the sending of notifications can also be tested using the update service. For this you just need to run the two notification postman scripts given without any modifications.