Advanced Software Engineering
FINAL REPORT

Team number:	0601

Team member 1	
Name:	David Reichert
Student ID:	00809424
E-mail address:	a00809424@unet.univie.ac.at

Team member 2	
Name:	Fritz Ziernhöld
Student ID:	11709105
E-mail address:	a11709105@unet.univie.ac.at

Team member 3	
Name:	Matthias Fritz
Student ID:	11910902
E-mail address:	a11910902@unet.univie.ac.at

Team member 4	
Name:	Jin Yu
Student ID:	11717460
E-mail address:	a11717460@unet.univie.ac.at

Remark: Due to the reduced team-size and thus missing SRs (SR4, SR5) the system does not contain SR5 (search-service) as it was not necessary to realize the user-case of other services. However, we decided to create a very basic implementation of SR4 (event-inventory service) as it was essential to enable the use-cases of most other SRs.

1. Final Design

1.1. Design Approach and Overview

1.1.1. Assumptions

SR3:

- Roles and role-mappings cannot be added dynamically. They are hardcoded in configuration files.
- User is not able to edit its account information
- No password salting/security is required as well as no email validation

SR6:

- It is assumed that 3 Tags (SPORT, FOOD and EDUCATION) can be added/removed to a posted event and could only be seen by the user themselves in the system. E.g. if I add a SPORT tag on an Event, my friend will not see that Tag on that Event in his interface, neither will the Organizer or the Admin, the tags are private to myself.

SR7:

- It is assumed that all Attendees have a valid e-mail address in the system to which newly newly added event suggestions can be delivered.
- It is assumed that location information of the user is available (country, city).

SR8:

- It is assumed that when an attendee wishes to provide feedback for an event, they also need to rate various criteria on a five-point scale
- It is assumed that it is possible to rate different criteria on an event without leaving a written feedback description
- It is assumed that submitted feedback can be updated by the user

SR9:

- It is assumed that the standard Calendar export will be in .ics file format.
- It is assumed that only the most important information of an Event will be exported, which includes the *event name*, *event location, event tags and event start date.*

SR10:

- It is assumed that a user can view analytics that are based on the ratings submitted by other users for an event
- It is assumed that feedback analytics show an average rating per feedback/rating criteria
- It is assumed that organizers can only create reports of their own events

SR11:

- It is assumed that the payment of booking attendance to events is not handled within the system.
- It is assumed that organizers can only send text-messages to attendees, and no attachments can be added.
- It is assumed that an Attendee can freely attend events, and no restrictions, such as minimum age, are present.

SR12:

- Bookmarked events and/or attendees of an event get the same email
- Only a single type of notification: event has changed information

1.1.2. System Design Decisions

Also see [Section 1.2 - Architectural Design Decisions] for additional context on how we arrived at the current architecture solution.

For the overall system, we decided to design it as a microservice architecture right away, utilizing the database-per-service pattern. This way, each service maintains its own view of the data - storing only what is relevant to achieve its use-case. The model is then updated through messages (events) based on which topics the microservices are subscribed to (publish-subscribe pattern). This communication is then built around a standardized data format, in our case JSON. The number of messages sent for each event was kept minimal by using DTOs (Data transfer object).

While this approach can lead to inconsistent views of the data at times (only achieving eventual consistency), it makes the microservices a lot more loosely coupled, as each service can work independent of others. In turn, this also improves the fault-tolerance of the system, as other services can continue to operate in case a single service experiences an outage.

Another decision we made was to keep anything related to authorization and role access-controls strictly to the Login-Service (SR3) and the API-Gateway (SR1) - which acts as an entrance barrier into the system. Thus, any current (and future) services do not have to be concerned with whether a user is authorized to access certain resources as all of this is transparent to them and handled behind their back. As a result of this centralized authentication, it simplifies the implementation for other services and allows them to focus on their specific business capability that they provide. We also decided to use the open source identity and access management service Keycloak. We used keycloak in a limited way in that we only used a keycloak server to generate access tokens and assist the login-service generating and retrieving access tokens for newly logged-in users.

1.1.3. Design Decisions of Individual Services

SR3:

As mentioned in the design report, the relation between the login-service and keycloak is that keycloak provides the login-service only with an access-token and the login-service handles all the rest (role mappings and verification). This turned out to be a great idea and hasn't changed for the final. I decided to move the login-service before the api-gateway as the first entry point to our application. So this service behaves a little bit differently than the others in that the client directly talks to it and not through the api-gateway. This is also not entirely true as the login-service, as instructed in the assignment, also contains the maintenance view. The maintenance view is only allowed to be viewed by the administrator role and therefore is accessible through the api-gateway. This is an obvious design flaw as the maintenance view is exposed and could be directly called through the login-service url without having an administrator role. Unfortunately this was noticed way too late and I did not want to include an ip-filter and therefore decided to mention it here. In my opinion the maintenance view should be its own service and lie behind the api-gateway so that we could have a nice and clean abstraction.

SR6:

Major design decision of the Marktag service is the model structure. Currently the Marktag service stores the Attendee as an Aggregated root and Event as an Entity. This structure successfully handles the parent and child relationship between Attendee and Event. The decision was made during the design phase by pure intuition. An Aggregate root was the most apparent solution and suitable for the requirement after analyzing at that time. However, during the end to end test as well as manual testing, I noticed that there is a lag after the user clicked the bookmark button on an event, the event will only appear in the bookmark UI section a little bit late.

Even though the time lag was less than a second and insignificant in my opinion, I did consider an alternative solution that could reduce the lag. The causes might be the complexity of the aggregated root. To solve this, Event and attendee information could be stored in one Entity together, that potentially requires less logic to add/update/delete existing data and maybe even faster access (fetch) time. However, due to time constraints, the alternative solution was not able to be executed and tested.

SR7:

The functional requirements of the recommender service were extended to also allow attendees to retrieve personal event recommendations (“Personal Picks”) based on the currently available events, instead of only notifying them when a newly added event could be interesting to them. In this regard, one of the main considerations was how to determine which events should be recommended to users, taking into account the information available of the user (previously attended events, hometown etc.).

To realize this behavior, the strategy pattern is utilized which either injects a very personalized recommendation strategy, or if no event-interests of the user are known, a generalized recommendation strategy which considers, e.g. the location or trending events. This approach also makes it easy to add additional strategies in the future, which can then simply be chosen at runtime.

Moreover, because incoming messages of other microservices were used in different parts of the service, the observer pattern is utilized for incoming consumer messages to reduce the coupling between components and make the communication more flexible as well as extensible for future use-cases.

Furthermore, all available endpoints were designed according to the RESTful architectural style and best practices to be self-descriptive. In this regard internal data is also converted to only expose relevant information to the user.

Lastly, the whole service was designed with the single-responsibility principle in mind in addition to using a layered-architecture and not allowing classes to bypass layers, i.e. a RestController is not concerned with the business logic at all, and delegates to a dedicated service class (see also UML-class diagram).

SR8:

After the process of separating the business logic from the infrastructure and application layer I designed the system according to the domain-driven design approach by structuring the parts into the DDD building blocks in a layered architecture. Since we decided on a database per service design there will be an own database container inside this microservice to make the service independent from others.

SR9:

A major design decision for Export service would be how this service receives the tag information for both bookmarked and attended events. The solution I have decided is that the Export service is only listening to Marktag service for bookmarked events, attending events and the tags with them. Export service is subscribing to Attendance service no longer, which was originally planned during the design phase. In this design, Marktag service will need to subscribe to the Attendance service and post the attendance topic again additionally with the tag information. Export service then needs to only subscribe to Marktag service and the Event inventory service.

Alternatively (which was planned during the design phase), the Export service would need to additionally subscribe to Attendance service as well and then get the Tag information from the Marktag service, thus more message traffic that could potentially slow down the export process. However, the downside of this design is the export service now also needs to keep track of tag information in its database that could require expensive storage, but on the other hand again, if the Marktag service is down, the tag along with events could still be exported.

SR10:

During the initial design phase, the original plan was to generate reports and feedback analyses on demand without storing them in a dedicated database. However, during the implementation process, considerations were made regarding performance concerns and efficient resource utilization. As a result, it was determined that storing the feedback analyses per event in their own database, and transitioning them from value objects to entities, would be more advantageous.

Since the creation of feedback occurs infrequently compared to the access of feedback analyses, it was deemed appropriate to generate new feedback analyses only when new feedback is added. Consequently, the database is accessed to retrieve the already created and still valid analyses.

Furthermore, during the design phase, there were discussions about shifting the responsibility of feedback analysis to the feedback service. Despite the potential concern regarding the separation of concerns, the decision was made to maintain the existing design, keeping the feedback analysis logic separate within the analytics and report service. This design choice facilitated the implementation of a stream publisher on the feedback microservice's side and a feedback consumer on the analytics and report service side, enabling efficient communication between the two services.

SR11:

SR11 was designed with the same design principles in mind as SR7, as such DTOs are used to facilitate the transfer of data to the outside and the observer pattern is used to improve maintainability and simplify future extensions by making observer/subject agnostic of each other, e.g. the StreamProducer is notified when new event-bookings are created and consequently published the information. Moreover, a layered architecture is also used as well as repositories to facilitate the data-access.

For SR11, additional requirements were added to provide functionality that is common in other systems. Namely, Attendees can cancel Event-Bookings up to 24h before the event starts. Furthermore, digital entrance tickets (QR-Codes) are created for each event they attend. For this, the factory pattern is utilized due to the rather complex construction of ticket objects which includes the ZXing library to encode the most important information in a QR-code. In this regard a HATEOAS-link to the

QR-code image was also included when retrieving all event-bookings of an attendee to quickly navigate to the related resource.

One design decision was to keep the mapping class of Event – Attendee (M:N - EventBooking) separate from the entry ticket. While these two are related, they still are quite different concepts and thus were kept separated. However, to avoid situations where an EventBooking is canceled and the ticket remains valid (or an eventbooking is created without ticket), a cascade constraint was added to maintain consistency.

SR12:

As this service has very minimal responsibility it was designed as simple as possible. The notification-service is connected to the email-service Send Grid, which allows it to send emails and create email-templates. The notification-service has no endpoints and only reacts to changes in event data and if a bookmark or attendance entry has been created.

1.1.4. Birds-eye view

To get an overview of the system, please refer to the diagrams provided in the 4+1 Views-Model. For individual SRs, individual class diagrams are provided under the logical view.

1.2. Architectural Design Decisions (ADDs)

ADDs were kept to decisions that we actively had to decide on, not those that were already given such as using a Microservice architecture or pub-sub; also we decided to use the more detailed decision template from the lecture instead of the minimalistic decision documentation.

AD shortname	AD-01
AD name	Choice of Database Architecture
Problem Statement	All services operate on different subsets of the data, thus it must be determined how it should be stored.
Decision Drivers	1. Independence and autonomy of microservices 2. Data consistency and synchronization 3. Scalability and performance 4. Development and Deployment flexibility 5. Flexibility in data model and database technology
Alternatives	1. Database per Service 2. Shared Database accessed by all Services
Recommendation	We recommend option 1): a database-per-service approach, where each microservice has its own dedicated database.
Decision Outcomes	

Status	Realized
Chosen Alternative	Option 1: Database per Service
Justification	Using a database per service makes the services a lot more loosely coupled and able to operate autonomously, thus they can be developed and deployed independently.
Consequences	**+** Services can be developed, deployed and scaled independently **+** Each service can optimize its data model and database technology to fit its needs**.** **-** More complex data management which can lead to inconsistent views of the data at times (only achieving eventual consistency)
Assumptions	Services must maintain and update their own data model to synchronize changes through pub-sub messages using a standardized data format (JSON).

AD shortname	AD-02
AD name	Service access with access-tokens
Problem Statement	A user should only be able to talk to the api-gateway, and therefore the service, if it has a valid access-token which should be retrieved at login.
Decision Drivers	1. System security 2. Consistency 3. Usability
Alternatives	1. The user first needs to login, i.e., retrieve the access token which then gets saved. With the saved access token the user can access resources behind the api-gateway. 2. The api-gateway integrates access token validation/creation and performs validation actions at each request.
Recommendation	We recommend 1) because it allows an easier implementation and a smoother and more flexible usability of the service.
Decision Outcomes	

Status	Realized
Chosen Alternative	1
Justification	Was the better solution for the scale of our system and allowed us to integrate role-based authentication in the early stages of development.
Consequences	The login-service endpoints are exposed and need to be properly secured.
Assumptions	A user keeps the same access token throughout its session and only receives a new one if it logs out and back in again.

1.3. Major Changes Compared to DESIGN

System Wide Changes:

Our initial design proved to work very well and therefore no major changes had to be made. In this regard we were very glad that we put a lot of effort in during DESIGN to “get it right” and have a very in-depth design and models to help us during the FINAL implementation.

One minor change that deviates from the design was to move the login service in front of the gateway, as such the user can now directly access the login service to register and consequently login to receive an access token. However, subsequent requests to other services must then always go through the API-Gateway as outlined in DESIGN.

Another minor change was the health check that was required for the Administrators. During the DESIGN phase, we considered the Eureka server as our first approach. We shortly abandoned it due to the different versions that caused inconsistent connection issues between the Eureka server and each individual service. Instead we integrated a less complicated Spring Boot Actuator, that also avoids the usage of external libraries. With Spring Boot Actuator, we achieve the functionality of health check by exposing a health endpoint that is directly accessed by the Administrators view.

Individual SRs:

SR3

As mentioned before, the only major change made in the login-service is that its relationship with the Keycloak service changed. Now Keycloak provides the login-service only with the access-token. These two services are closely coupled and the login-service has dedicated services that initialize connection to the Keycloak service and communicate with it.

SR6 & SR9

A minor change was the topic posting and subscribing for the attendance topic. Since the Export service (SR9) requires the tags, Marktag service is responsible to include the tags only whenever a tag is added to an event, but also whenever a user bookmarks or attends an event. E.g, when a user added some tags to an event, then bookmarked/attended the event, its tags along with other necessary details should be sent to the message broker together. This scenario was not considered during the first design phase, and was discovered during integration test and end to end test. From the perspective of Export service (as previously explained in Section 1.1.3 - SR9), it only subscribes to Event inventory service and Marktag service, excluding the Attendance service which was originally planned in the first DESIGN phase.

SR7 & SR11

The design proposed in DESIGN proved to work very well during the implementation, and thus no major changes had to be made which can also be seen when comparing the class and sequence diagrams from DESIGN and the updated ones from FINAL. They were merely refined, such adding more detailed method structures and database queries in the repository or deciding on the concrete library to use (i.e. using SendGrid[1] to send Emails).

SR8 & SR10

Regarding the design, there was only one significant modification regarding SR 10 and the feedback analytics. During the design phase, the intention was to implement and design a Feedback Analyze (SR 10) as a Value Object, generated upon access. However, due to the real-time nature of the environment, the frequency of feedback creation is relatively low compared to feedback analysis accessing. Consequently, it was determined that an event would encompass a feedback analyze entity, which would be stored in a database. This approach aims to minimize the workload, recalculating and restoring the analysis solely when a new feedback is added.

SR12

No major design changes were needed as this service has minimal requirements.

[1] https://sendgrid.com/

1.4. Development Stack and Technology Stack

For the used technologies, our first decision went towards using Java 17 (LTS) and Spring Boot. Most of us already had some experience working with the framework and moreover, it was also recommended in the course.

Additionally, it allowed us to utilize many Spring-native solutions, such as Spring Cloud Gateway, Spring Boot Actuator and Spring Cloud Stream, where the latter allows us to deploy event-driven communication independent of the underlying message broker while also eliminating a lot of boilerplate code. While the message broker can be freely exchanged, we decided to use Apache Kafka as it is a very well established and highly performant platform which fulfilled all of our needs.

For the database, we decided on a relational database, namely PostgreSQL, in combination with Hibernate ORM to simplify the development as well as making the code less dependent on the underlying database implementation. Lastly, because the databases can vary between microservices, we also explored other options individually, like using NoSQL databases such as MongoDB, but ultimately decided to use PostgreSQL for each microservice.

For generating the OAuth2 JWT access tokens, we decided to use a Keycloak server. There we can set up realm roles which are then appended to the access token which will be verified in the login service.

1.4.1. Development Stack

Build Management	Maven 3.9.1
Version Control	GitLab
Integrated Development Environment	IntelliJ
CI/CD Platform	GitLab CI/CD
API-Client (Development and Testing)	Postman

Maven:
We chose maven as most of us had experience with it, it integrates very well with Java and the other technologies we used.

Gitlab Version Control | CI/CD
Given by the assignment.

IntelliJ:
Most of us were familiar with this IDE and could make use of its features and tools well in this project.

Postman

Postman is a simple API-Client which works very well to quickly test the API of our services during development by not having to write tedious curl commands.

1.4.2. External Libraries

Library	Usage (SR)
com.google.zxing	Creation of QR-Codes for digital entrance tickets (SR11)
org.mnode.ical4j	Exporting as Calendar format (SR9) (https://www.ical4j.org/)
com.sendgrid	Email-delivery platform.

1.4.3. Technology Stack

Programming Language	Java 17
Application Framework	Spring Boot 3.0.5
API Gateway Framework	Spring Cloud Gateway 4.0.3
Messaging Framework	Spring Cloud Stream 4.0.2
Testing Framework	JUnit 5.9.2
Database	PostgreSQL 15.2
Message Broker	Apache Kafka 3.2.3
Distributed Coordination Service	Apache ZooKeeper 3.7.0
Containerization	Docker Engine 20.10.23
OAuth2 Provider	Keycloak 15.0.2
Front End	Vue js 3.2.13

Java 17: We chose Java 17 for its robustness, maturity, and wide range of libraries. As one of the most popular programming languages, it has a large community and extensive support, making problem-solving easier. Additionally, Java's backward compatibility ensures long-term viability for our applications. Additionally, it was also the recommended choice (+ Spring) for the project.

Spring Boot 3.0.5: Spring Boot simplifies the setup and development of Spring applications by providing defaults and auto-configuration for commonly used functionalities. It also integrates well with other Spring projects, which we have chosen in this tech stack, thereby making the development process smoother.

Spring Cloud Gateway 4.0.3: This was chosen for its seamless integration with the Spring ecosystem and its wide range of features. It provides an easy way to route API requests, apply rate limits, and handle concerns like security and monitoring.

Spring Cloud Stream 4.0.2: To handle data-streaming scenarios and microservice communication, Spring Cloud Stream offers an easy-to-use event-driven programming model that integrates well with our existing Spring-based applications. Its abstraction for message brokers ensures the possibility of changing the underlying broker if required.

JUnit 5.9.2: JUnit is a widely-used testing framework in Java, offering a broad variety of tools and options to carry out unit testing efficiently. It integrates seamlessly with popular IDEs and build tools, fostering a test-driven development approach and ensuring software quality.

PostgreSQL 15.2: PostgreSQL is a powerful, open-source object-relational database system. It is widely used and has a reputation for reliability, data integrity, and correctness.

Apache Kafka 3.2.3: Kafka is a distributed event-streaming platform designed to handle high volume real-time data feeds. It's scalable, fault-tolerant, and allows for real-time processing and data pipelines, making it a great choice for our messaging framework to realize pub-sub.

Apache ZooKeeper 3.7.0: ZooKeeper helps manage and coordinate distributed systems by providing services like synchronization, configuration management, and naming registry. It is required by Apache Kafka.

Docker Engine 20.10.23: Required by the assignment.

Keycloak 15.0.2: Keycloak is an open-source Identity and Access Management solution aimed at modern applications and services. It provides out-of-the-box authentication and authorization.

Vuejs 3.2.13: Vue.js for the front-end development significantly reduces the debugging time. It also provides a very strong reusability of different components and various external libraries that can be imported directly.

2. Updated Requirements

2.1 Main Functional Requirements

Identifier	FR03-01
Priority	High
Description	The system should support only 3 different roles: attendee, organizer and administrator
Affected use-cases	All
User roles	Attendee, Organizer, Administrator
Associated SR	SR1, SR3
Implications on Implementation	The login-service contains a configuration file that contains role mappings of each user role
Verification Method	Unit and Integration testing, Manual inspection
Verification Status	Implemented

Identifier	FR03-02
Priority	Medium
Description	The administrator is able to see the system health (maintenance) of each service.
Affected use-cases	UC03-01
User roles	Administrator
Associated SR	SR3
Implications on Implementation	Each service integrates a Spring Boot Actuator that exposes a health endpoint. This endpoint is used by the login-service to retrieve the health status of a service.
Verification Method	Unit and Integration testing, Manual inspection
Verification Status	Implemented

Identifier	FR03-03
Priority	High
Description	Role based access control should be granted using an access token
Affected use-cases	UC03-02
User roles	Attendee, Organizer, Administrator
Associated SR	SR3, SR1
Implications on Implementation	Each service endpoint is registered in the login-service and mapped to a specific role which will then have access to that endpoint.
Verification Method	Unit and Integration testing, Manual inspection
Verification Status	Implemented

Identifier	FR06-01
Priority	Medium
Description	As an Attendee, I should be able to bookmark my favorite events or the one that interests me.
Affected use-cases	UC06-01, UC06-02
User roles	Attendee
Associated SR	SR6
Implications on Implementation	The system needs to keep track of the bookmarked event for each attendee individually.
Verification Method	Manual inspection, Unit Test, Spring integration Test, End to End Test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR06-02
Priority	Medium
Description	As an Attendee, I should be able to bookmark or unbookmark every events that are available (displayed).
Affected use-cases	UC06-01
User roles	Attendee
Associated SR	SR6
Implications on Implementation	The system needs to be synchronized with the event that is being posted by the organizers.
Verification Method	Manual inspection, Unit test, Spring integration Test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR06-03
Priority	Medium
Description	As an Attendee, to find events quicker, but also provides an overview without looking into the details, I should be able to add and remove the following tag/s on event/s once (no duplications): #sports, #food, #education. (which also should be only visible to me and not other Attendees)
Affected use-cases	UC06-03, UC06-04
User roles	Attendee
Associated SR	SR6
Implications on Implementation	The system needs to keep track of the tags for each event individually for each attendee.
Verification Method	Manual inspection, Unit Test, Spring integration Test, End to End test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR07-01
Priority	Medium
Description	As an Attendee I should be notified via Email if a newly added Event matches my interest profile, taking into account my previously attended events or information such as hometown.
Affected use-cases	UC07-01
User roles	Attendee
Associated SR	SR7
Implications on Implementation	An Email-Client / external API is necessary to send the Emails, for which SendGrid is used.
Verification Method	Manual inspection, Unit Tests, Integration Test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR07-02
Priority	Low
Description	As an Attendee I should be able to opt out of receiving promotional Emails that suggest new events to me.
Affected use-cases	UC07-01
User roles	Attendee
Associated SR	SR7
Implications on Implementation	The system must keep track of the current state and a default one must be chosen on user registration. (= true).
Verification Method	Manual inspection, Unit Tests, Integration Test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR07-03
Priority	Medium
Description	As an Attendee I should be able to view “Personal Event Picks”, which shows me events that are recommended to me.
Affected use-cases	UC07-02
User roles	Attendee
Associated SR	SR7
Implications on Implementation	Attendee data (such as hometown, previously attended events, bookmarks) needs to be stored to make sensible recommendations.
Verification Method	Manual inspection, Unit Tests
Verification Status	Verified: Implementation meets Requirement

Identifier	FR08-01
Priority	Medium
Description	As an Attendee I should be able to leave a feedback comment for an event I attended, as well as a rating about the description of the event, the location and an overall rating.
Affected use-cases	UC08-01, UC08-02
User roles	Attendee
Associated SR	SR8
Implications on Implementation	The system needs to keep track of of the attendance entries for the events.
Verification Method	Manual inspection, Unit Test, Spring integration Test, End to End test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR09-01
Priority	Low
Description	As an Attendee, to make user's attending or bookmarked events more accessible, the system should allow user to be able download (export) them into an external file include the following 3 formats: .ics .json .xml
Affected use-cases	UC09-01, UC09-02
User roles	Attendee
Associated SR	SR9
Implications on Implementation	The system needs to keep track of the tags for each bookmarked and attending event individually for each attendee.
Verification Method	Manual inspection, Unit Test, Spring integration Test, End to End test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR10-01
Priority	Medium
Description	As an Organizer I should be able to generate a report for events I'm organizing
Affected use-cases	UC10-01
User roles	Organizer
Associated SR	SR10
Implications on Implementation	Only the event organizer with the matching ID is allowed to generate reports for the corresponding events.
Verification Method	Manual inspection, Unit Test, Spring integration Test, End to End test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR10-02
Priority	Medium
Description	As an Attendee I should be able to see analytics about given feedback of events. This analysis should include averages of different criteria.
Affected use-cases	UC10-02, UC08-01
User roles	Organizer
Associated SR	Attendee
Implications on Implementation	The system must keep track of given feedbacks.
Verification Method	Manual inspection, Unit Test, Spring integration Test, End to End test
Verification Status	Verified: Implementation meets Requirement

Identifier	FR11-01
Priority	High
Description	As an Attendee I should be able to select an event that I want to attend, which then creates a digital entrance ticket for me.
Affected use-cases	UC11-01
User roles	Attendee
Associated SR	SR11
Implications on Implementation	The service must keep track of the vacancies of an event and update them via transaction to avoid conflicts.
Verification Method	Unit Tests, Integration and End-to-End Tests
Verification Status	Verified: Implementation meets Requirement

Identifier	FR11-02
Priority	Medium
Description	As an Attendee I should be able to directly view all of my event-bookings and the corresponding entrance tickets under my attending-events section.
Affected use-cases	UC11-01, UC11-02
User roles	Attendee
Associated SR	SR11
Implications on Implementation	The service must make sure to invalidate canceled tickets and no longer show them to the Attendee (see FR11-03). Additionally, the ticket information is encoded in a QR-code which can be retrieved and downloaded.
Verification Method	Manual inspection
Verification Status	Verified: Implementation meets Requirement

Identifier	FR11-03
Priority	Medium
Description	As an Attendee I should be able to cancel a previously made Event-Booking up to 24h hours before the event begins.
Affected use-cases	UC11-02
User roles	Attendee
Associated SR	SR11
Implications on Implementation	The service must make sure to invalidate canceled tickets which is made sure through a delete cascade on event-bookings
Verification Method	Manual Inspection, Unit-Tests
Verification Status	Verified: Implementation meets Requirement

Identifier	FR11-04
Priority	Medium
Description	As an Organizer, I should be able to directly message all or a subset of the attendees of an event that I am organizing, which will then be forwarded to the E-Mail addresses of the Attendees.
Affected use-cases	UC11-03
User roles	Organizer
Associated SR	SR11
Implications on Implementation	An Email-Client / external API is necessary to send the Emails, for which SendGrid is used.
Verification Method	Manual Inspection
Verification Status	Verified: Implementation meets Requirement; *However via Frontend it is only possible to message all attendees*

Identifier	FR12-01
Priority	Medium
Description	Whenever event data changes, all attending and bookmarked users get notified via email about the changes.
Affected use-cases	-
User roles	Attendee, Organizer
Associated SR	SR12, SR4
Implications on Implementation	The notification-service reacts to each change in event-data and sends an email.
Verification Method	Manual inspection
Verification Status	Implemented

2.2 Non-Functional Requirements

Identifier	NFR1
Priority	High
Description	It must be ensured that Users are only allowed to access protected resources that their role (Attendee | Organizer | Administrator) has access to and thus require an access token to authorize their request.
Implications on Implementation	Access tokens must be created and returned to the user upon login (i.e. using OAuth).
Verification Method	Manual Inspection, E2E-tests
Category	Security

Identifier	NFR2
Priority	High
Description	The system should be able to stay performant and responsive even under higher traffic loads and the 95th percentile of requests should have a response time of under 1 second.
Implications on Implementation	Communication between services should happen asynchronously to improve performance.
Verification Method	Manual Inspection, Load-testing
Category	Performance & Scalability

Identifier	NFR3
Priority	Medium
Description	The system should ensure that data displayed to the user is synchronized and consistent across different components and cannot result in an invalid state.

Implications on Implementation	-Information changes must be propagated to other services using messages and consequently updated. -The system must re-request any outdated information to always return the newest state and return an error for requests that would result in an invalid state.
Verification Method	Manual Inspection, E2E-tests
Category	Consistency & Synchronization

Identifier	**NFR4**
Priority	Medium
Description	The system and its microservices must be designed in a modular and maintainable manner which should allow for easier updates and future extensions to the system.
Implications on Implementation	Microservices must be very loosely coupled and should work independently from others.
Verification Method	Manual Inspection, Code and Documentation Review
Category	Maintainability and Extensibility

Identifier	**NFR5**
Priority	Low
Description	The User Interface needs to be function focused
Implications on Implementation	UI design needs to be simple and self explaining enough that does not require any external training to be able to user
Verification Method	Manual Inspection
Category	Usability

3. Updated 4+1 Views Model

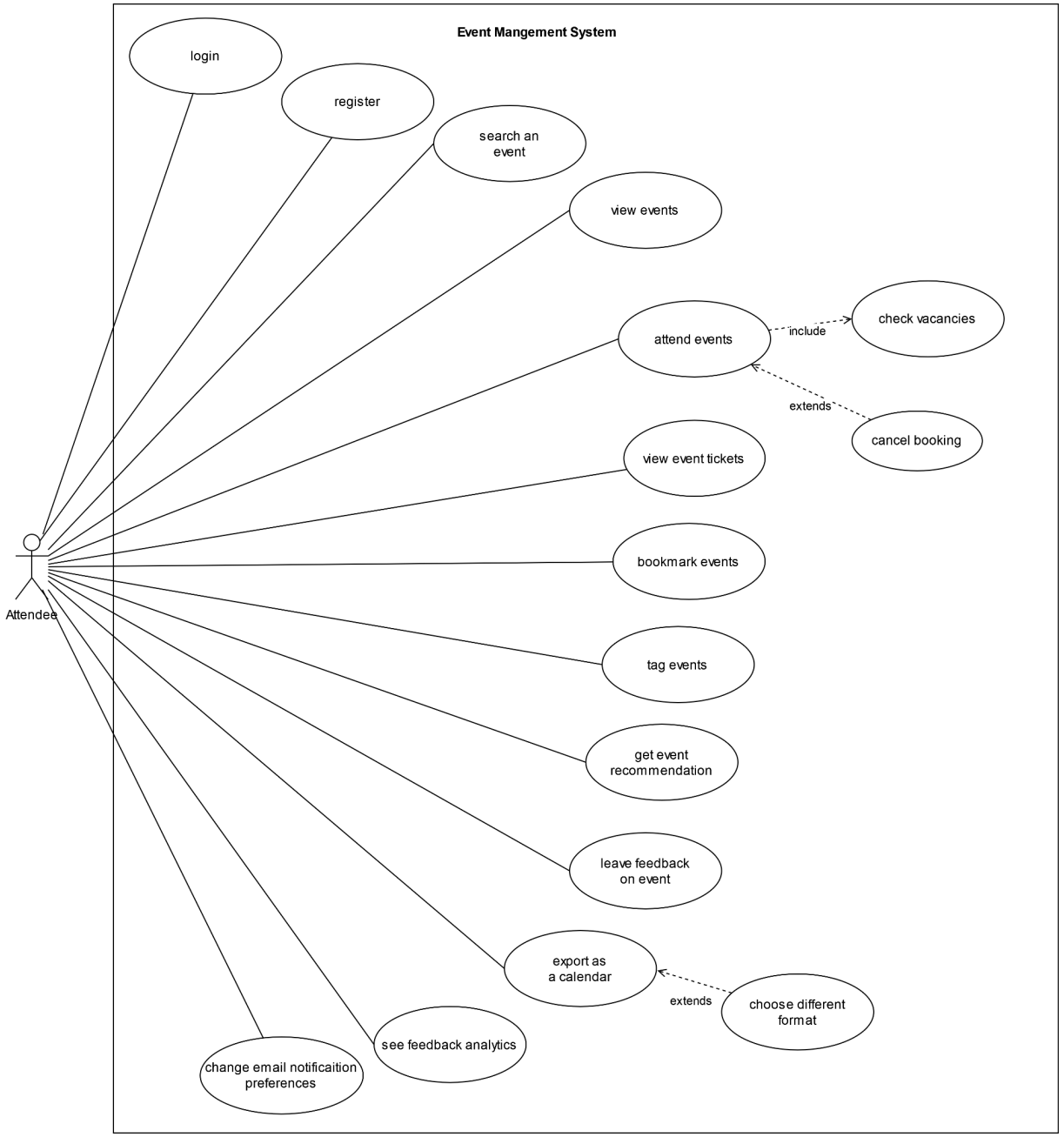
While often only minor changes were made, all artifacts in the 4+1 View model were updated to represent their state during FINAL, which incorporates the received feedback (i.e. explicitly showing layers for the SRs, split up use-case view, using shared kernel for common types etc.).

As previously described during DESIGN, we decided to utilize zoom factors for the Logical View, as such there is a more abstract view on the system-level that shows the core domain entities and in which contexts they reside in or are shared. For the bounded contexts, we kept the shared kernels strictly to the DTOs which are exchanged in order to reduce the coupling between the services on one hand. On the other hand, it made the most sense for each service to have its own view of the data, keeping only what is relevant to fulfill its use-case. The communication is then facilitated using these DTOs. This is also the reason why we decided to have each service in its own bounded context, with the exception of the event-management and search being combined into one.

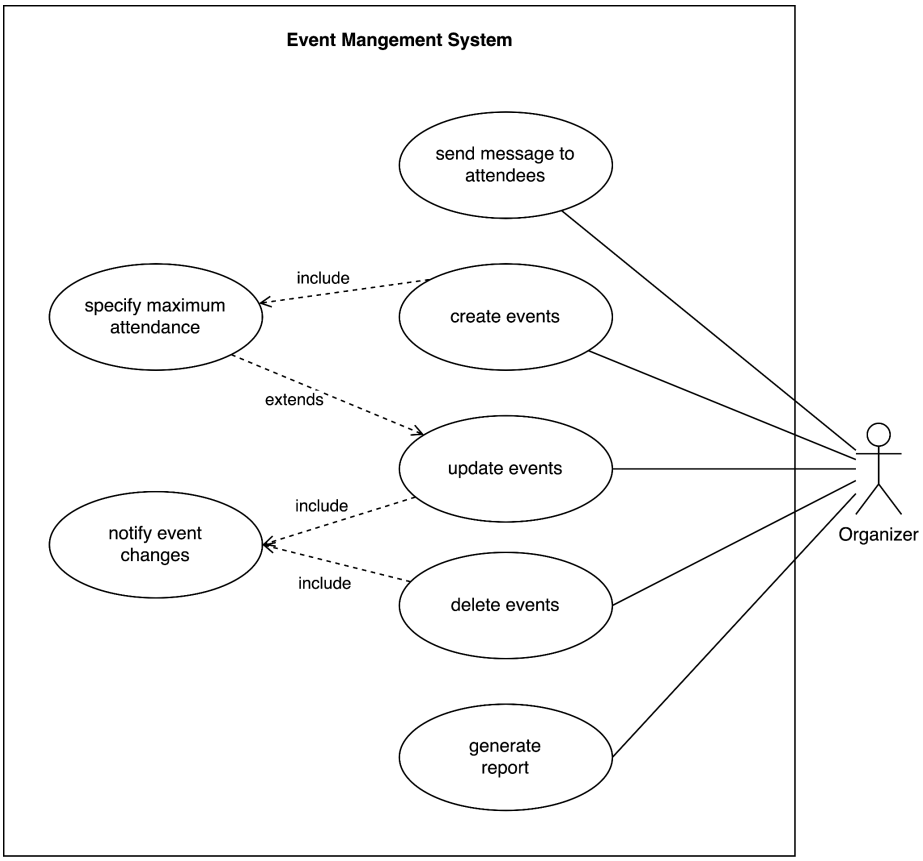
Consequently, more detailed views were created on the service-level which shows the structure of our microservices in more detail.

3.1. Scenarios / Use Case View

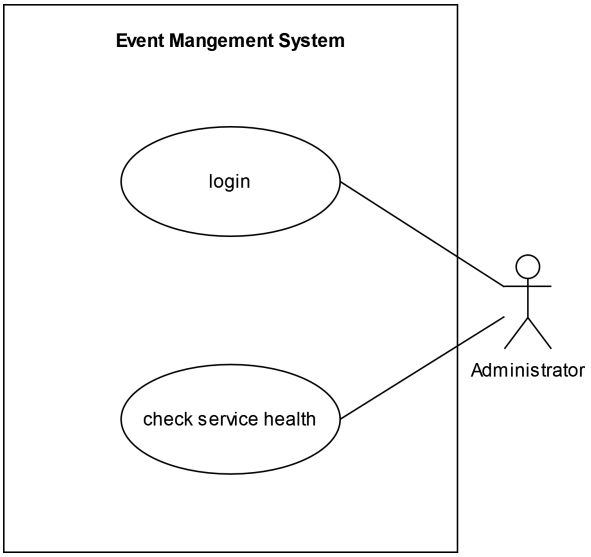
3.2. Attendee Use-Cases



3.3. Organizer Use-Cases



3.4. Administrator Use-Cases



3.4.1. Use Case Descriptions

SR 3

Use case:	View system maintenance
Actor(s):	Administrator
Use case ID:	UC03-01
Brief description:	An administrator wants to view the system maintenance by accessing the maintenance view.
Pre-conditions:	• A user of type administrator has to be logged into the system • User has to access the maintenance view
Post-conditions:	• User gets redirected to the maintenance view • User can see health of system
Main Success Scenario:	1. An administrator is using the system 2. Administrator accesses maintenance view 3. Administrator can view the health of the entire system
Priority:	Medium

Use case:	Authentication
Actor(s):	User
Use case ID:	UC03-02
Brief description:	A user logs into the system using their credentials and receives a role-based access token with which they have limited access to services depending on their role.
Pre-conditions:	The user must be registered to the system.
Post-conditions:	The user receives the access token and will be redirected to its role specific welcome page.
Main Success Scenario:	1. User inserts credentials into the login form 2. User clicks login 3. User receives access token and gets rerouted to the role specific welcome page
Priority:	High

Use case:	Register new user
Actor(s):	User
Use case ID:	UC03-03
Brief description:	As a user I can choose to create an account on the login page to have access to the system
Pre-conditions:	• User wants to create a new account • Newly created account does not exist in the system yet
Post-conditions:	The new account will be added persistently to the system
Main Success Scenario:	1. User chooses to create a new account 2. User fills in all necessary information to the account form 3. User presses the “create account” button 4. New account is created successfully
Priority:	Medium

SR 6

Use case:	**Bookmark an Event**
Use case ID:	UC06-01
Actor(s):	Attendee
Brief description:	An attendee marks an event via the User Interface then the Event will be updated as “bookmarked” on the UI.
Pre-conditions:	• The Attendee must have an account in the system. • There must be Event(s) that exist already (posted by organizers). • The Event should not have been “bookmarked” before.
Post-conditions:	• The Event will be updated as “marked” on the UI. • The Event should not be able to be “marked” again. • The “Mark” text/button will turn into “Unmark”
Main Success Scenario:	1. The Attendee selects an Event that interests him. 2. The Attendee clicks “Bookmark”. 3. “Mark” will turn into “Unmark” 4. The selected Event will be marked. 5. That Event will be updated as “marked” on the UI.
Extensions:	The Event could be “unmarked” afterward by clicking the same button.
Priority:	Medium
Performance Target:	All Events should be able to be marked
Issues:	Different ways of disabling “Mark” after the Event has been marked. (E.g. hide the button, disable the button, toggle, still allowing to click but it will not do anything.)

Use case:	Unbookmark an Event
Use case ID:	UC06-02
Actor(s):	Attendee
Brief description:	An attendee unbookmarks an event via the User Interface then the Event will be updated as “unmarked” on the UI.
Pre-conditions:	• The Attendee must have an account in the system. • There must be Event(s) that exist already (posted by organizers). • The Event should have been “bookmarked” before.
Post-conditions:	• The Event will be updated as “unmarked” on the UI. • The Event should not be able to be “unbookmarked” again. • The “Unmark” text/button will turn into “Bookmark”
Main Success Scenario:	1. The Attendee selects an Event that he is no longer interested in. 2. The Attendee clicks “Unmark” from the marked Event. 3. “Unmark” will turn into “Bookmark” 4. The selected Event will be unmarked. 5. That Event will be updated as “unmarked”.
Extensions:	The Event can be “bookmarked” afterwards by clicking the same button.
Priority:	Medium
Performance Target:	All marked Events should be able to be unmarked.
Issues:	Different ways of disabling “Unmark” after the Event has been unmarked. (E.g. hide the button, disable the button, toggle, still allowing to click but it will not do anything.)

Use case:	Add a Tag to an Event
Use case ID:	UC06-03
Actor(s):	Attendee
Brief description:	An attendee adds one of the Tag options (sports, food, education) on an event via the User Interface then the Tag will be displayed on that Event.
Pre-conditions:	• The Attendee must have an account in the system. • There must be Event(s) that exist already (posted by organizers). • The Event should not have the Tag that the attendee is going to add. (the same Tag should not be added twice.)
Post-conditions:	• The Tag will be added and displayed to the selected Event.
Main Success Scenario:	1. The Attendee selects an Event that he wants to add a Tag on. 2. The Attendee selects one of the 3 Tags (sports, food, education). 3. The Attendee clicks “Add Tag”. 4. The Tag is added to that Event.
Extensions:	The Tag could be removed (“untagged”) afterwards.
Priority:	Medium
Performance Target:	All Events should be able to add up to 3 Tags
Issues:	-

Use case:	**Remove a Tag from an Event**
Use case ID:	UC06-04
Actor(s):	Attendee
Brief description:	An attendee removes a Tag on an Event via the User Interface then the Tag will not be displayed anymore on that Event.
Pre-conditions:	• The Attendee must have an account in the system. • There must be Event(s) that exist already (posted by organizers). • The Event should have at least a Tag within.
Post-conditions:	• The Tag will be removed and will not be displayed to the selected Event.
Main Success Scenario:	1. The Attendee selects an Event. 2. The Attendee selects a Tag that he wants to remove. 3. The Attendee clicks “Remove Tag”. 4. The Tag is removed on that Event.
Extensions:	The removed Tag could be added (“tagged”) again afterwards.
Priority:	Medium
Performance Target:	All Tags on an Event could be removed
Issues:	-

SR 7

Use case:	Suggest new Events to Attendees
Use case ID:	UC07-01
Actor(s):	Attendee
Brief description:	The attendee is notified via Email if a newly added event is recommended to them, i.e., matches their interests
Pre-conditions:	The attendee must have a registered account in the system and not have explicitly opted-out of receiving promotional Emails.
Post-conditions:	The user receives an e-mail informing them that an event is recommended to them.
Main Success Scenario:	1. A new event is added to the system. 2. The system determines whether the event matches the interest-profile of a user. 3. An event-recommendation notification is sent to all attendees that are likely to be interested in the event.
Extensions:	The Attendee can opt-out of receiving these promotional Emails.
Performance Target:	The Emails must be delivered within a minute even for thousands of recipients.
Priority:	Medium
Issues:	-

Use case:	Create personalized Recommendations for Attendee
Use case ID:	UC07-02
Actor(s):	Attendee
Brief description:	The attendee can request personalized recommendations, which is a list of events which the system determines the attendee could be interested in, based on time, location and information (i.e. event-category) inferred from previously visited events.
Pre-conditions:	The user must have a registered account in the system.
Post-conditions:	-
Main Success Scenario	1. The user selects “View personal recommendations”. 2. The system generates event-recommendations based on the recorded interests of the user. 3. The list of the top 5 recommended events, each with a relevance score (max 100%) is shown to the user.
Extensions:	-
Performance Target:	The system must return the top 5 most relevant events within 1 second.
Priority:	Medium
Issues:	Fallback recommendation criteria are necessary and thus used if no interest profile has been established for a user, such as location or time (soon upcoming events).

SR 8

Use case:	Leave Event Feedback
Use case ID:	SR8-01
Actor(s):	Attendee
Brief description:	Users are able to give feedback on events. A feedback includes an optional comment and three five-point-scales for an overall, location and description rating of the event.
Pre-conditions:	• User must be authenticated • Event must have taken place • The attendee must have attended to the event
Post-conditions:	• Feedback is stored in the database
Main Success Scenario:	1. User chooses the event they want to leave a feedback 2. System displays feedback comment field and rating options 3. User enters text and uses five-point scale for different criteria 4. System displays confirmation message to the user
Extensions:	-
Priority:	• Medium
Issues:	• None

SR 9

Use case:	Export all bookmarked Events
Use case ID:	UC09-01
Actor(s):	Attendee
Brief description:	An Attendee exports all Events that he bookmarked into a Calendar, JSON or XML format based on Attendee's choice. If Attendee does not select any specific format, the Event will be exported as Calendar format.
Pre-conditions:	• The Attendee must have an account in the system. • There must be Event(s) that he bookmarked already.
Post-conditions:	• The bookmarked Event(s) will be exported as a Calendar, JSON or XML file based on attendee's choice, if there is no specific format is selected, the export will be in Calendar format.
Main Success Scenario:	1. The Attendee clicks "Export bookmarked Event". 2. The Attendee ticks "XML" or "JSON" (optional) 3. The Attendee clicks "Export" 4. All bookmarked events will be exported as an external file based on Attendee's choice.
Extensions:	The exported file could be opened externally.
Priority:	Low
Performance Target:	The file export should start within 5 seconds after user clicked "Export".
Issues:	If there are too many Events, the export might be slow?

Use case:	Export all attending Events
Use case ID:	UC09-02
Actor(s):	Attendee
Brief description:	An Attendee exports all Events that he is attending into a Calendar, JSON or XML format based on Attendee's choice. If Attendee does not select any specific format, the Event will be exported as Calendar format.
Pre-conditions:	• The Attendee must have an account in the system. • There must be Event(s) that he is attending already.
Post-conditions:	• The bookmarked Event(s) will be exported as a Calendar, JSON or XML file based on attendee's choice, if there is no specific format is selected, the export will be in Calendar format.
Main Success Scenario:	1. The Attendee clicks "Export Attending Event". 2. The Attendee ticks "XML" or "JSON" (optional) 3. The Attendee clicks "Export" 4. All events that he is attending will be exported as an external file based on Attendee's choice.
Extensions:	The exported file could be opened externally.
Priority:	Low
Performance Target:	The file export should start within 5 seconds after the user clicked "Export".
Issues:	If there are too many Events, the export might be slow?

SR 10

Use case:	Generate Event Report
Use case ID:	SR10-01
Actor(s):	Organizer
Brief description:	Users are able to generate reports about events they are authorized for. The report offers different insights and calculations about the corresponding event.
Pre-conditions:	• User must be logged in • User has to have proper access rights • Event must be created beforehand
Post-conditions:	• Report is removed
Main Success Scenario:	• User chooses the event they are interested in • System displays available report options • User selects report options of interest • System generates and displays report
Extensions / Additional Remarks:	The report can be accessed in browser or exported as a PDF file
Priority:	• Medium
Issues:	• None

Use case:	View Feedback Analytics
Use case ID:	SR10-02
Actor(s):	Attendee
Brief description:	Users have the ability to view analytics on events. These analytics are divided into categories such as average rating, average description rating and average location rating.
Pre-conditions:	• User must be registered and have proper access rights
Post-conditions:	• User sees feedback analytics
Main Success Scenario:	1. User chooses the event they are interested in and selects view analytics 2. The System displays all available feedback comments and the analysis regarding these feedbacks
Extensions:	-
Priority:	• Medium
Issues:	• None

SR 11

Use case:	Attend an Event
Use case ID:	UC11-01
Actor(s):	Attendee
Brief description:	Attendees can browse a catalog of events and select one they want to attend.
Pre-conditions:	The attendee must be logged in to the system.
Post-conditions:	The attendee is registered for the event and the entrance ticket can be downloaded under their 'attending events' section.
Main Success Scenario:	1. The user browses the event catalog and chooses an event by selecting "Attend Event". 2. The system verifies that vacancies are still available. 3. The system marks the spot as taken and creates an entrance ticket (QR-Code). 4. The booking confirmation is displayed to the user.
Extensions:	The booking can be optionally canceled by the attendee. (see UC11-02)
Priority:	High
Issues:	It must be ensured that events are not overbooked due to concurrent purchases.

Use case:	Cancel an Event-Booking
Use case ID:	UC11-02
Actor(s):	Attendee
Brief description:	Attendees can cancel previously booked events up to 24h before the event starts.
Pre-conditions:	The user must be logged in to the system and is registered to an event.
Post-conditions:	The Event-Booking and entrance ticket are no longer available to the attendee. The vacancies of the event are adjusted accordingly.
Main Success Scenario:	1. The attendee browses their booked events and selects “Request Cancellation”. 2. The system verifies whether the request is still within the cancellation window (24h before event start). 3. The Event-Booking and corresponding entrance ticket are deleted. 4. The cancellation confirmation is displayed to the user.
Extensions:	-
Priority:	Medium
Issues:	It must be ensured that the digital entrance ticket is no longer valid upon cancellation.

Use case:	Send message to Attendees
Use case ID:	UC11-03
Actor(s):	Organizer, Attendees
Brief description:	Organizers can send messages to the attendees of their event (e.g. to inform them of information/schedule changes).
Pre-conditions:	The organizer is logged in to the system and has an event registered with the system.
Post-conditions:	The created message is sent to all participants of the event via their registered e-mail.
Main Success Scenario:	1. The organizer chooses one of their events and selects “Notify Attendees”. 2. The organizer selects all or a subset of attendees as recipients 3. The organizer enters the message and selects “Submit”. 4. The system retrieves the e-mail address of the selected attendees and forwards the message to their stored e-mail address. 5. The organizer is notified that the messages have been successfully forwarded.
Extensions:	-
Priority:	Medium
Issues:	Verification that only the organizer of the specific event can send messages.

SR 12

Use case:	Notification of attending or bookmarked event changes
Use case ID:	UC12-01
Actor(s):	Attendee
Brief description:	As an attendee, when I mark an event as attending or bookmarked, I will receive a notification when any event information gets updated.
Pre-conditions:	• Attendee marks event as attending or bookmarked • Event data gets updated
Post-conditions:	• Attendee receives email notification for related change
Main Success Scenario:	• Attendee marks event as bookmarked or attending • Event data changes • Attendee get notified about the change via email
Priority:	Medium
Performance Target:	The notification should be sent ideally within one minute up to five minutes.

3.5. Logical View

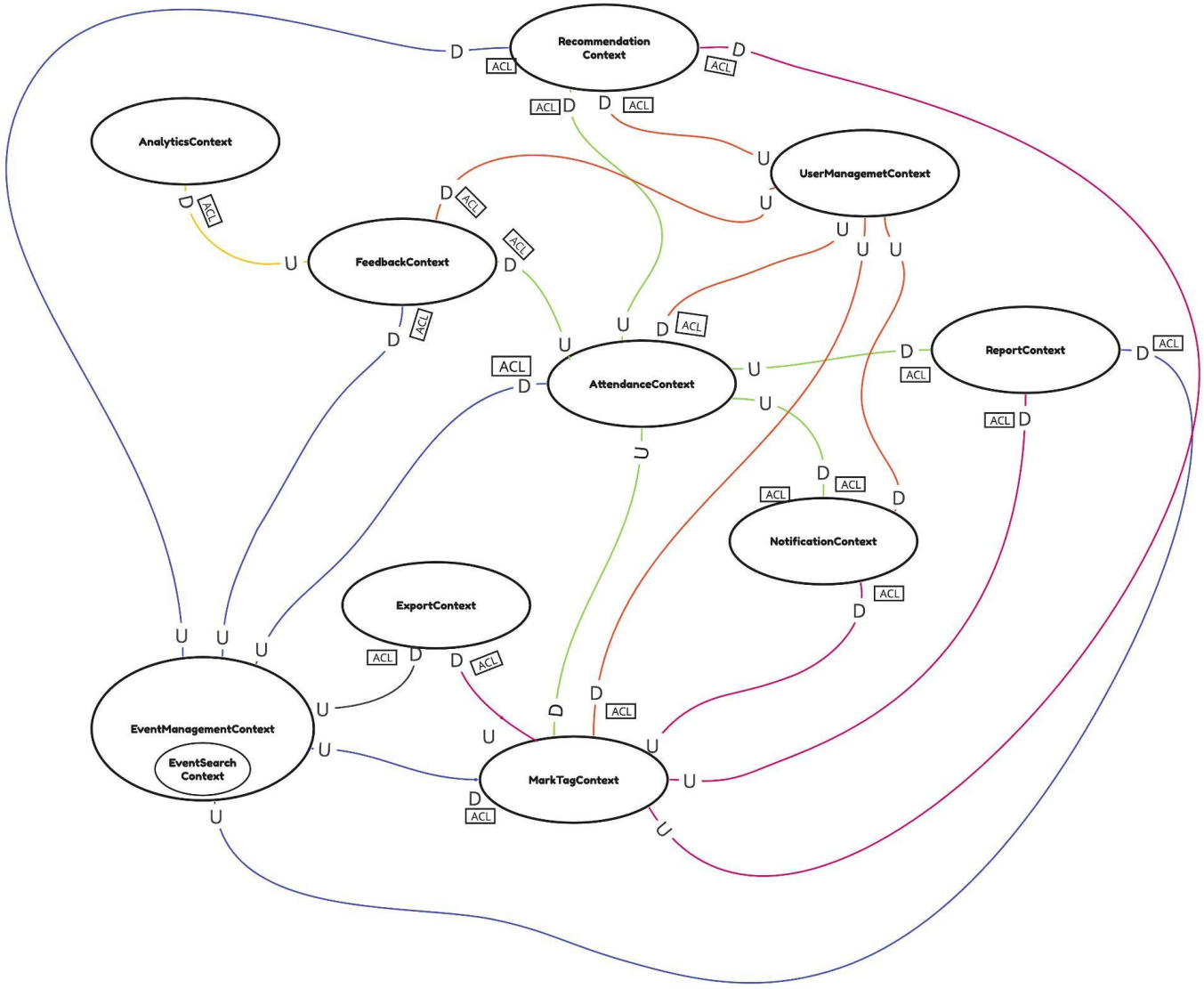


Figure 3.5.1: Context map for the whole system

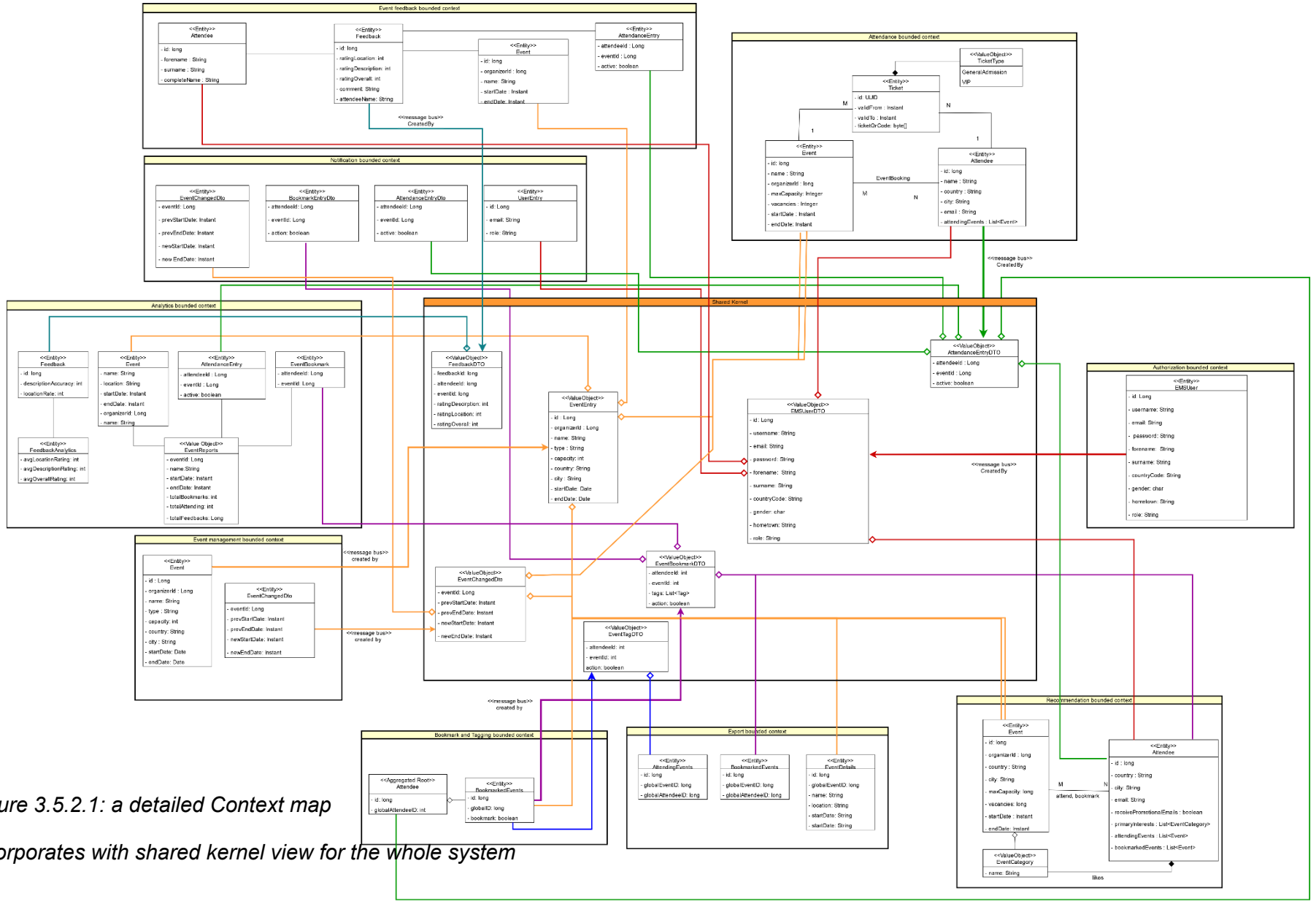
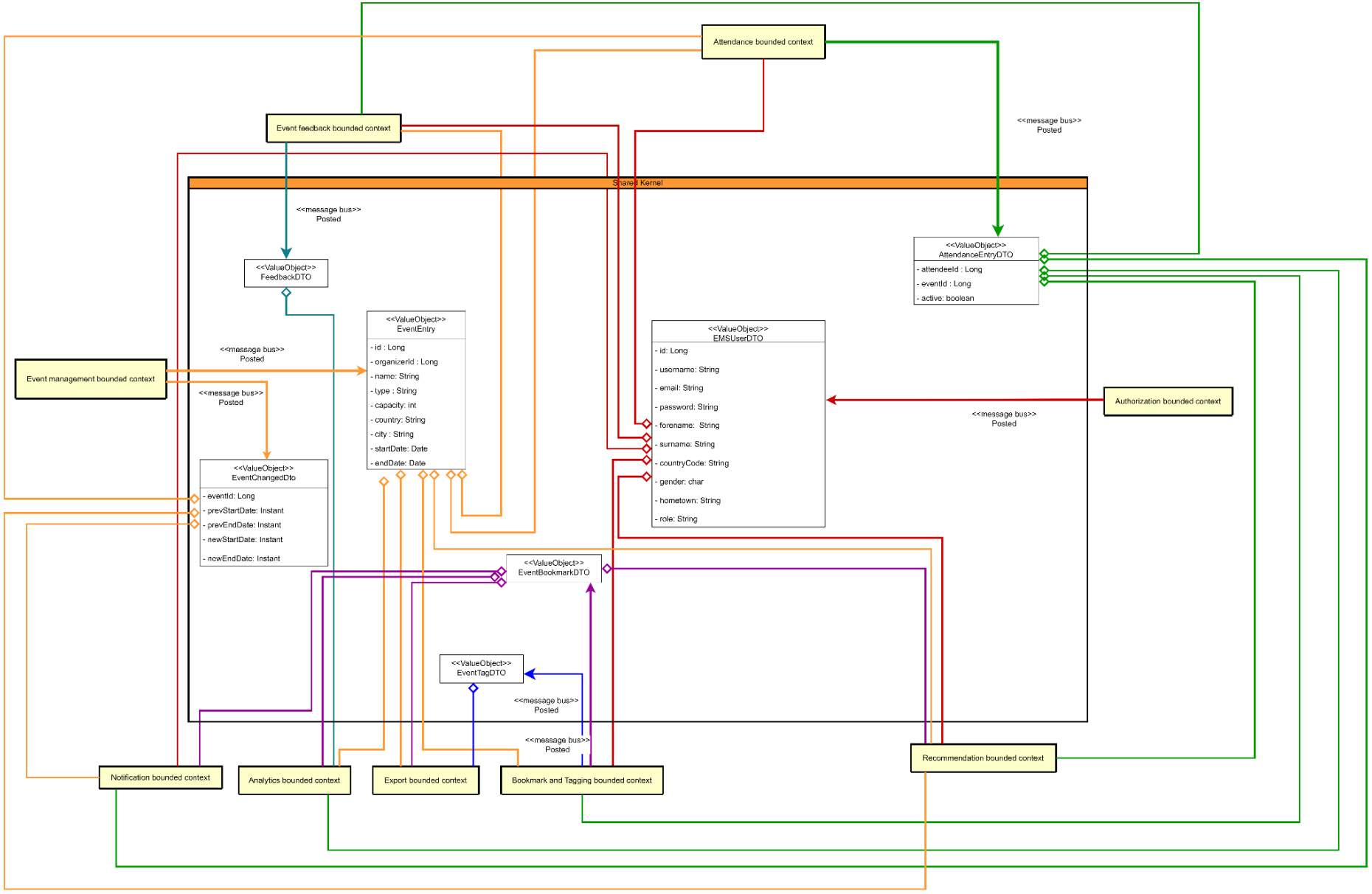


Figure 3.5.2.1: a detailed Context map incorporates with shared kernel view for the whole system

Figure 3.5.2.2: A more abstract and condense Shared Kernel view for the whole system



SR3

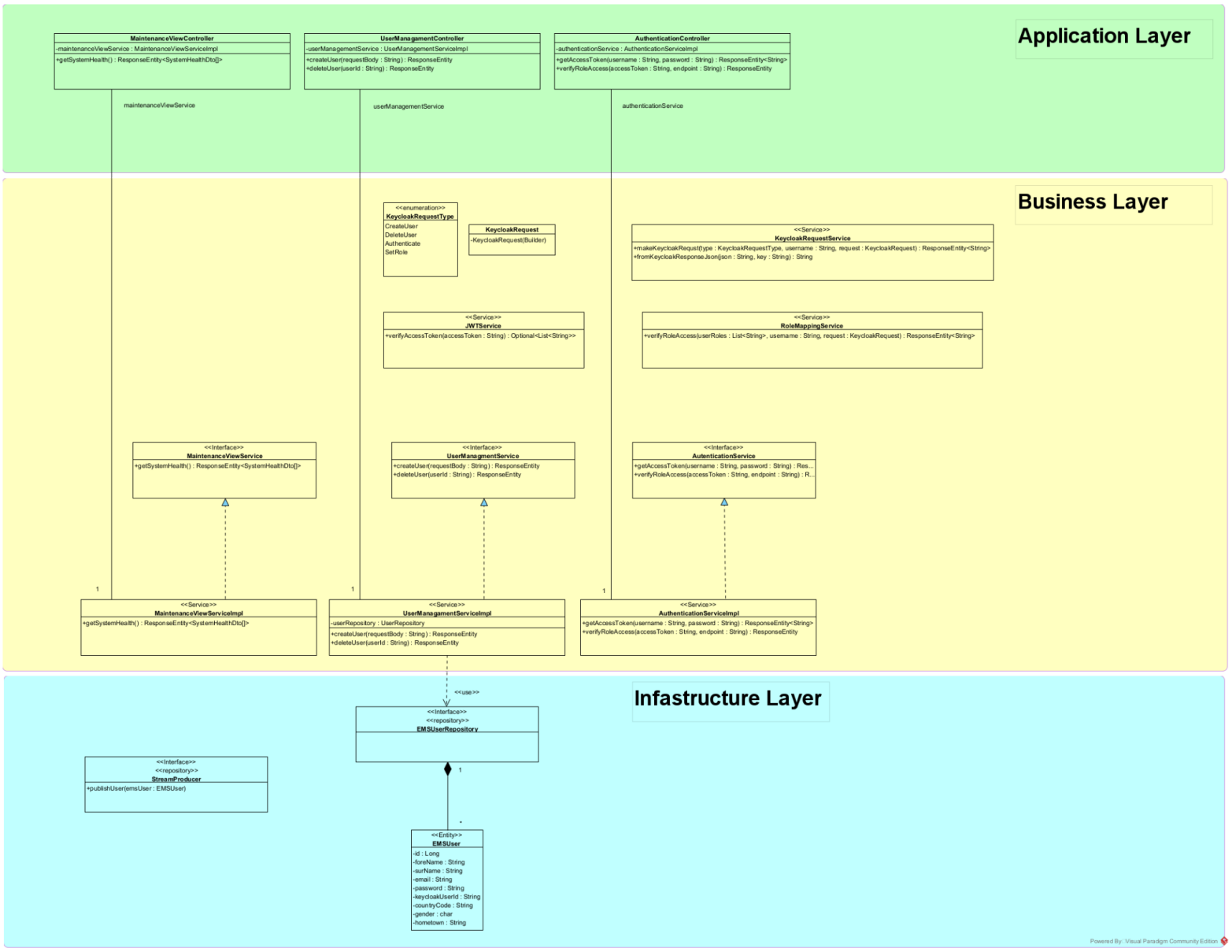


Figure 3.5.3: class diagram for Login service

SR6

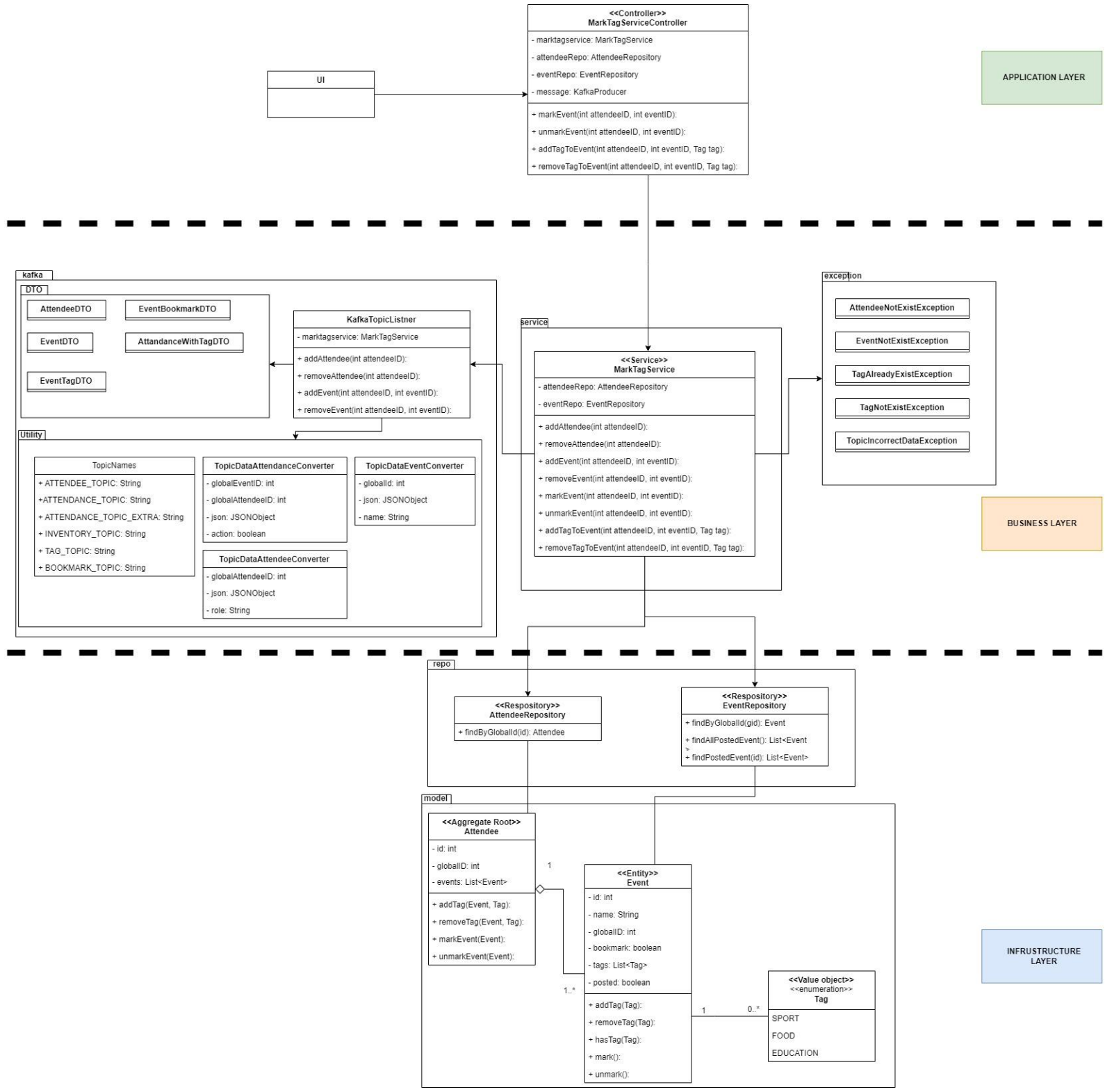


Figure 3.5.6: class diagram for Marktag service

SR7

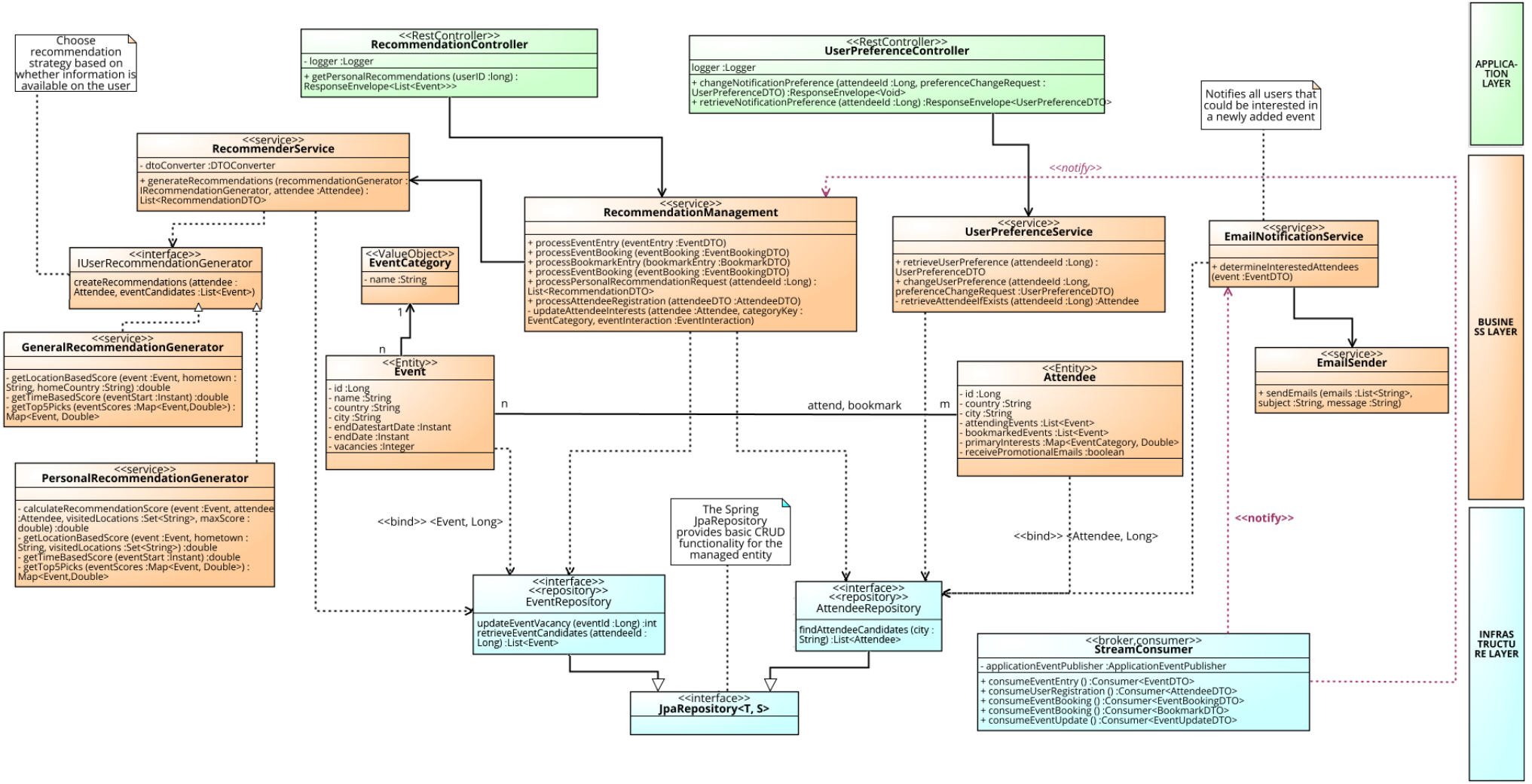


Figure 3.5.7: class diagram for Recommender service

SR8

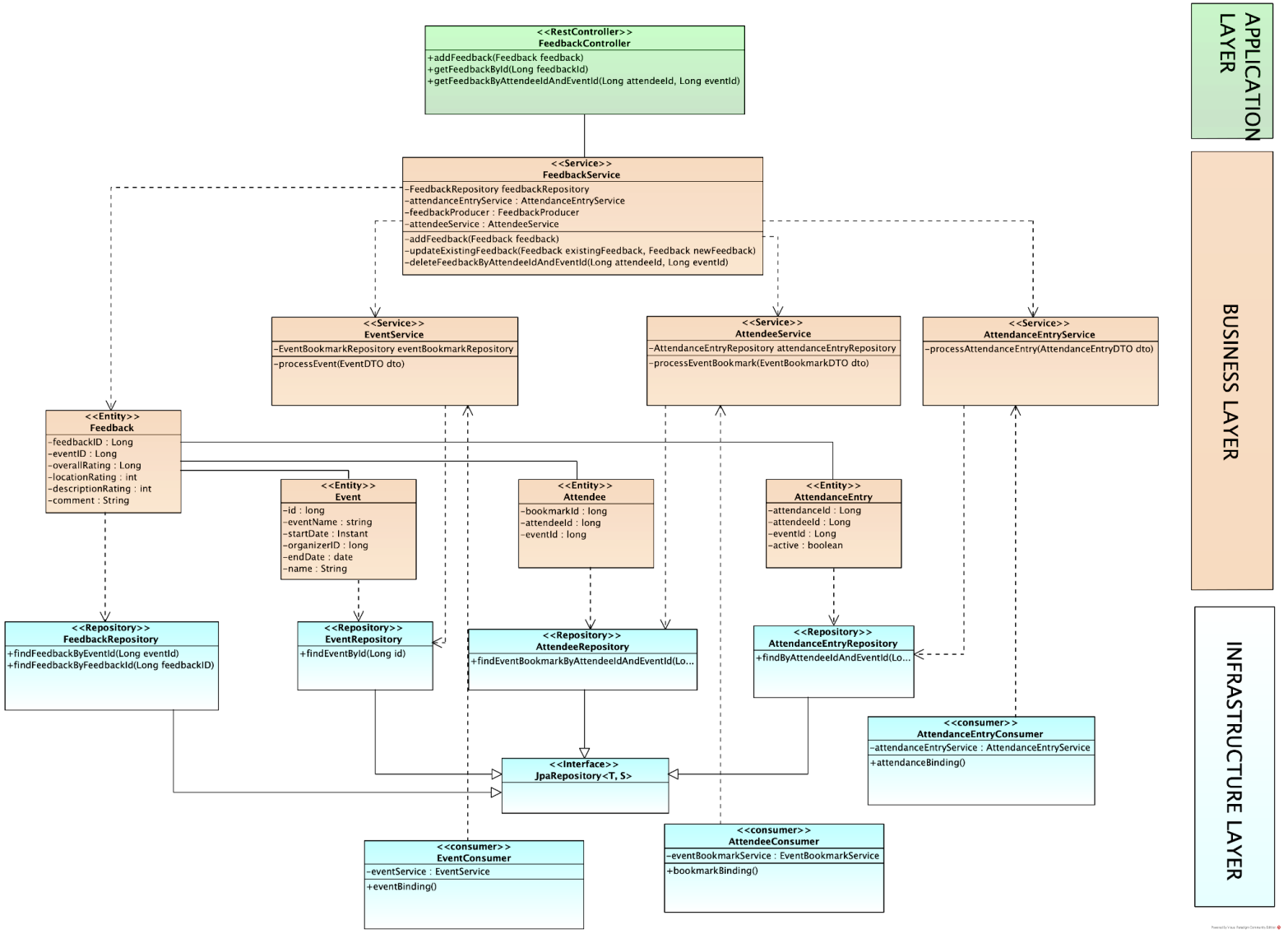
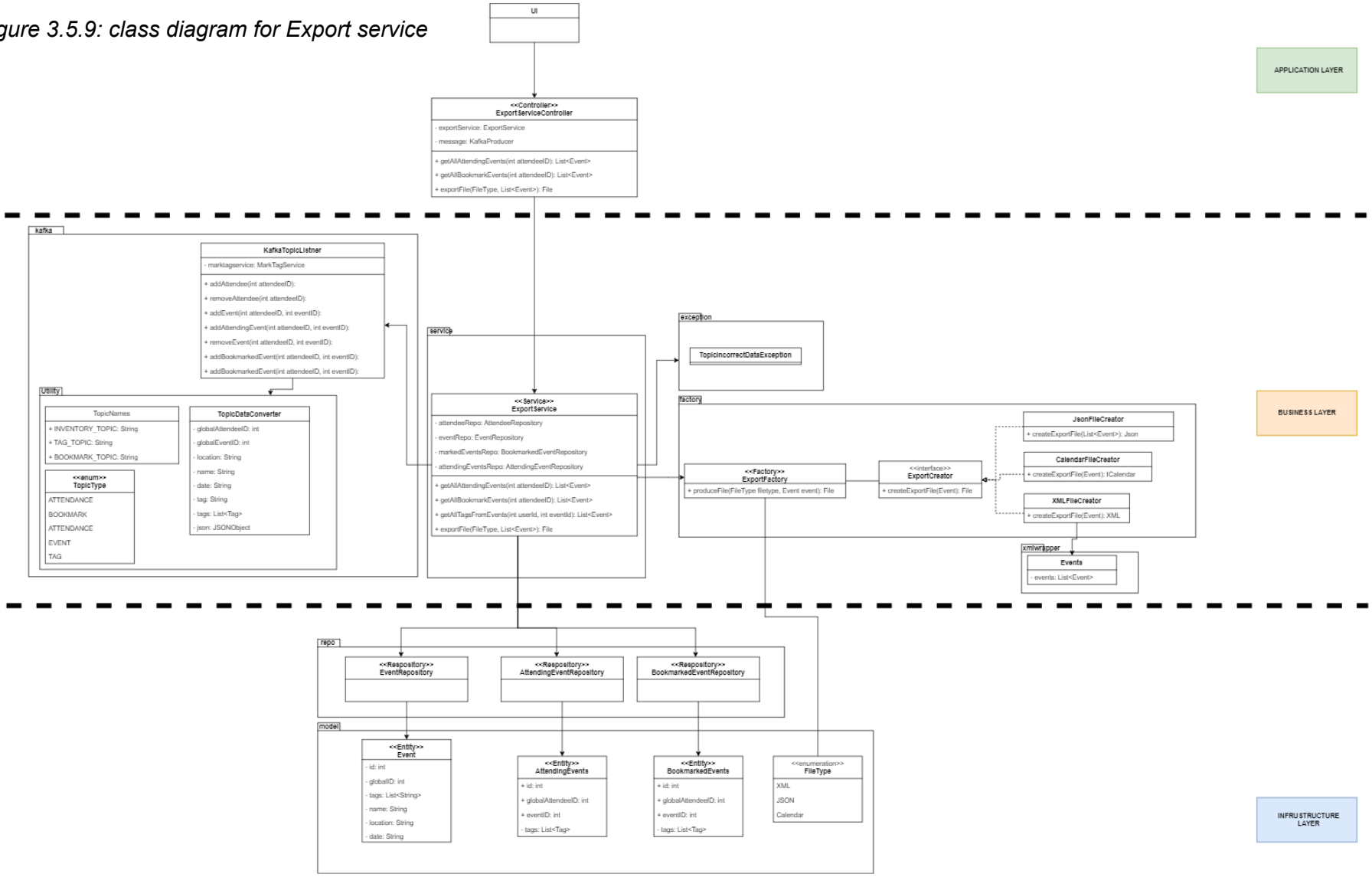


Figure 3.5.8: Class diagram for Feedback Analytics and Event Report Service

SR9

Figure 3.5.9: class diagram for Export service



SR10

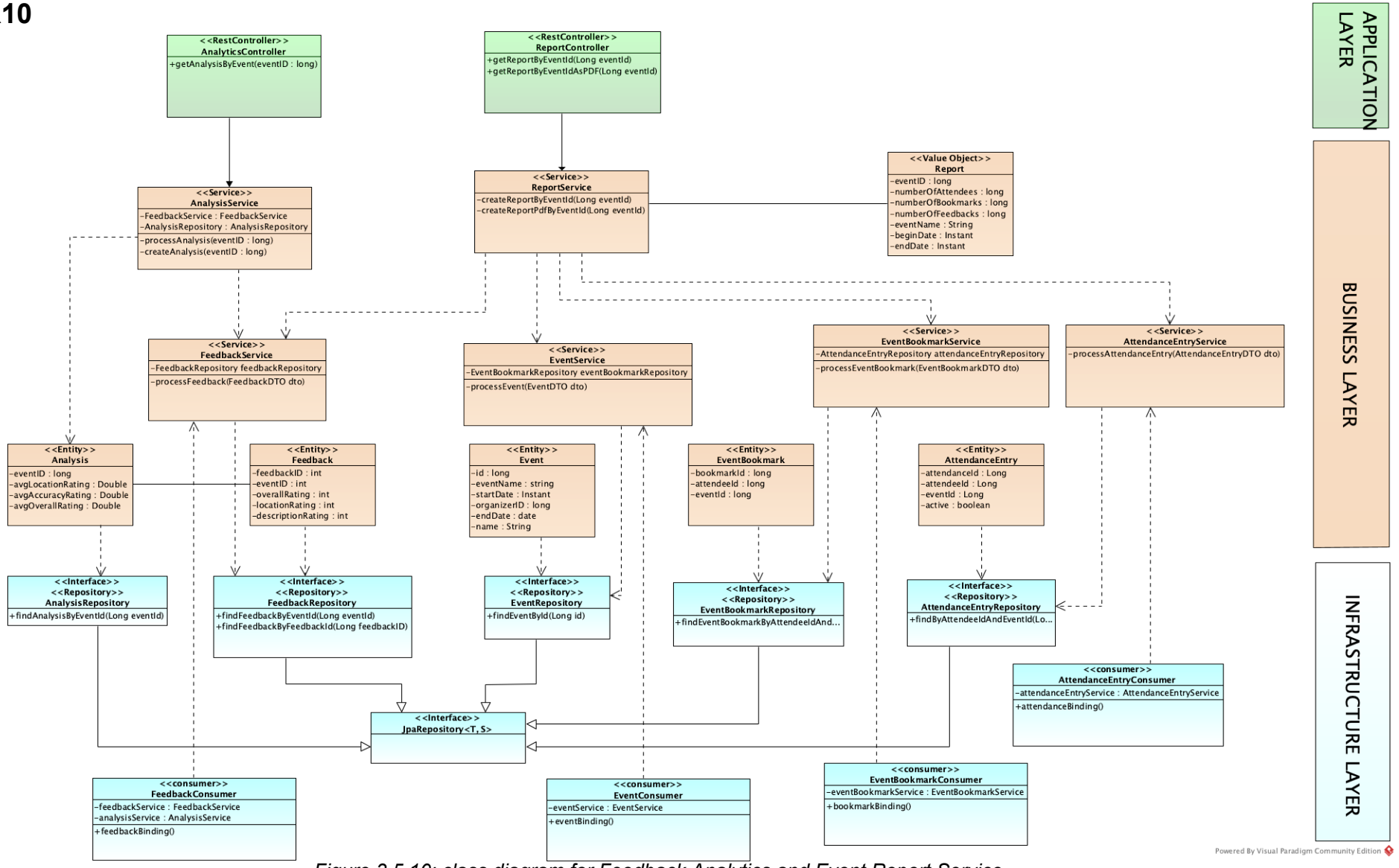


Figure 3.5.10: class diagram for Feedback Analytics and Event Report Service

SR11

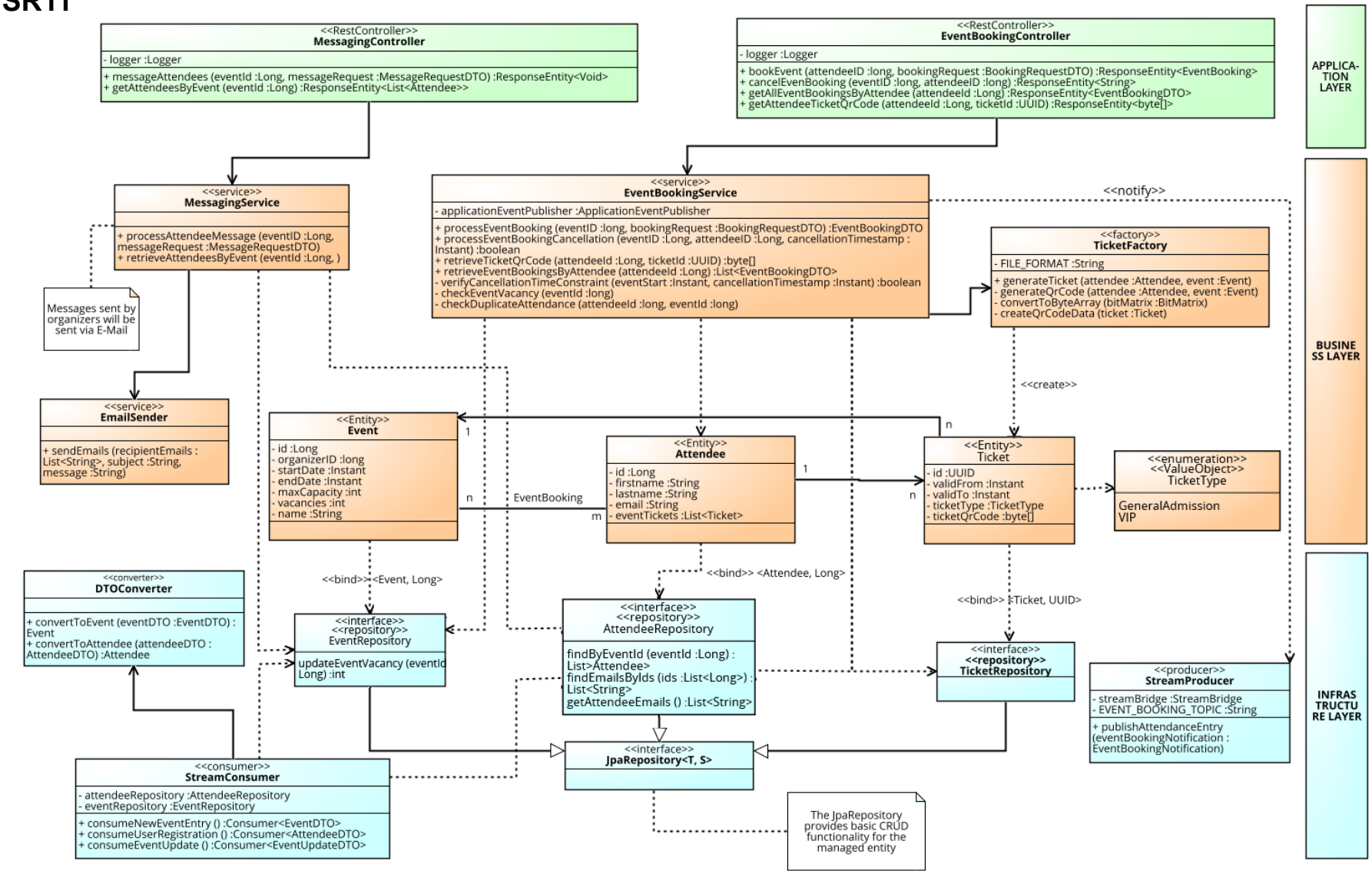


Figure 3.5.11: class diagram for Attendance service

SR12

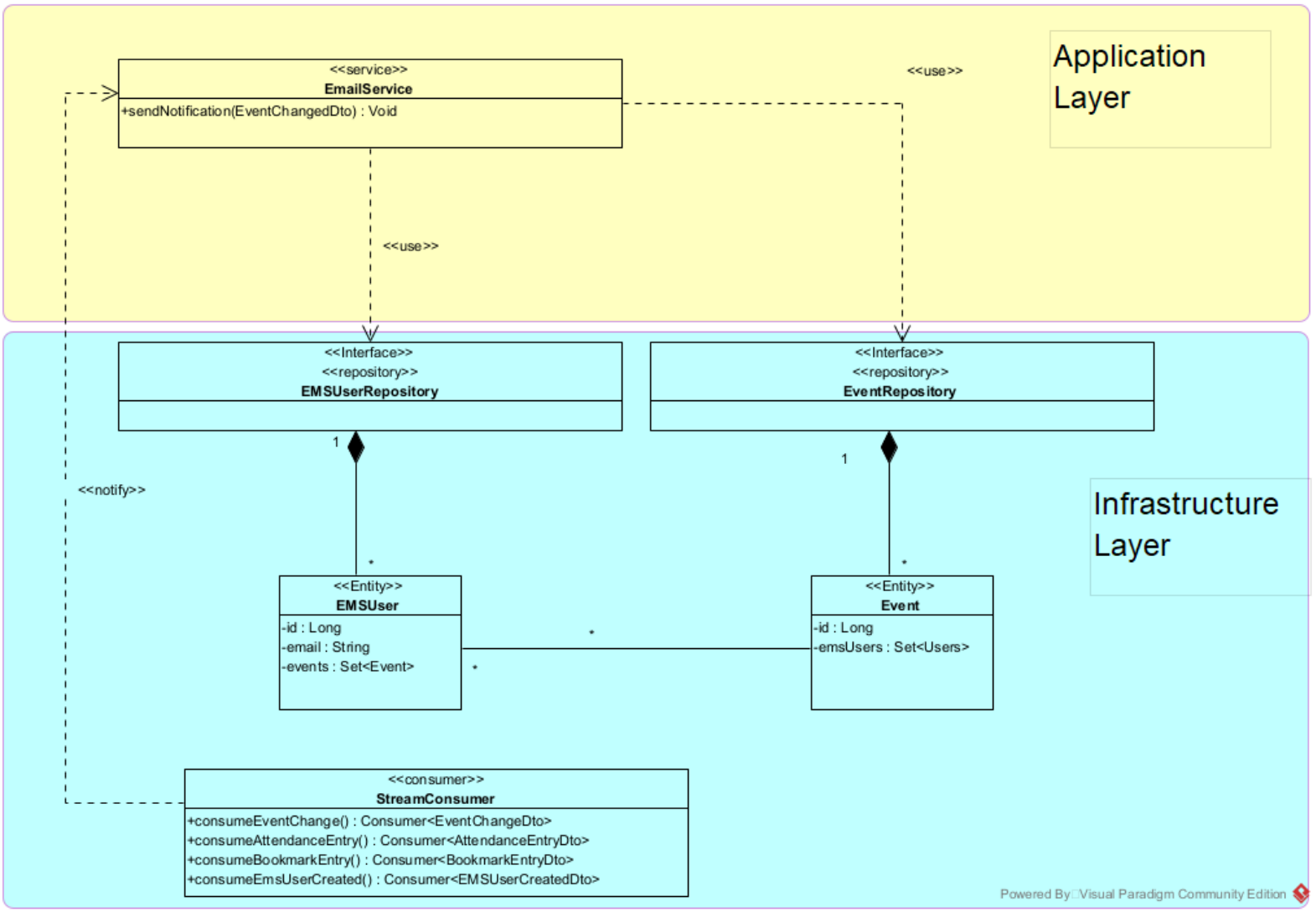


Figure 3.5.12: class diagram for Notification service

3.6. Process View

To highlight the system's behavior during the use case, we decided to use a Sequence Diagram as it provides a simple representation of the interactions between actors and objects.

SR3

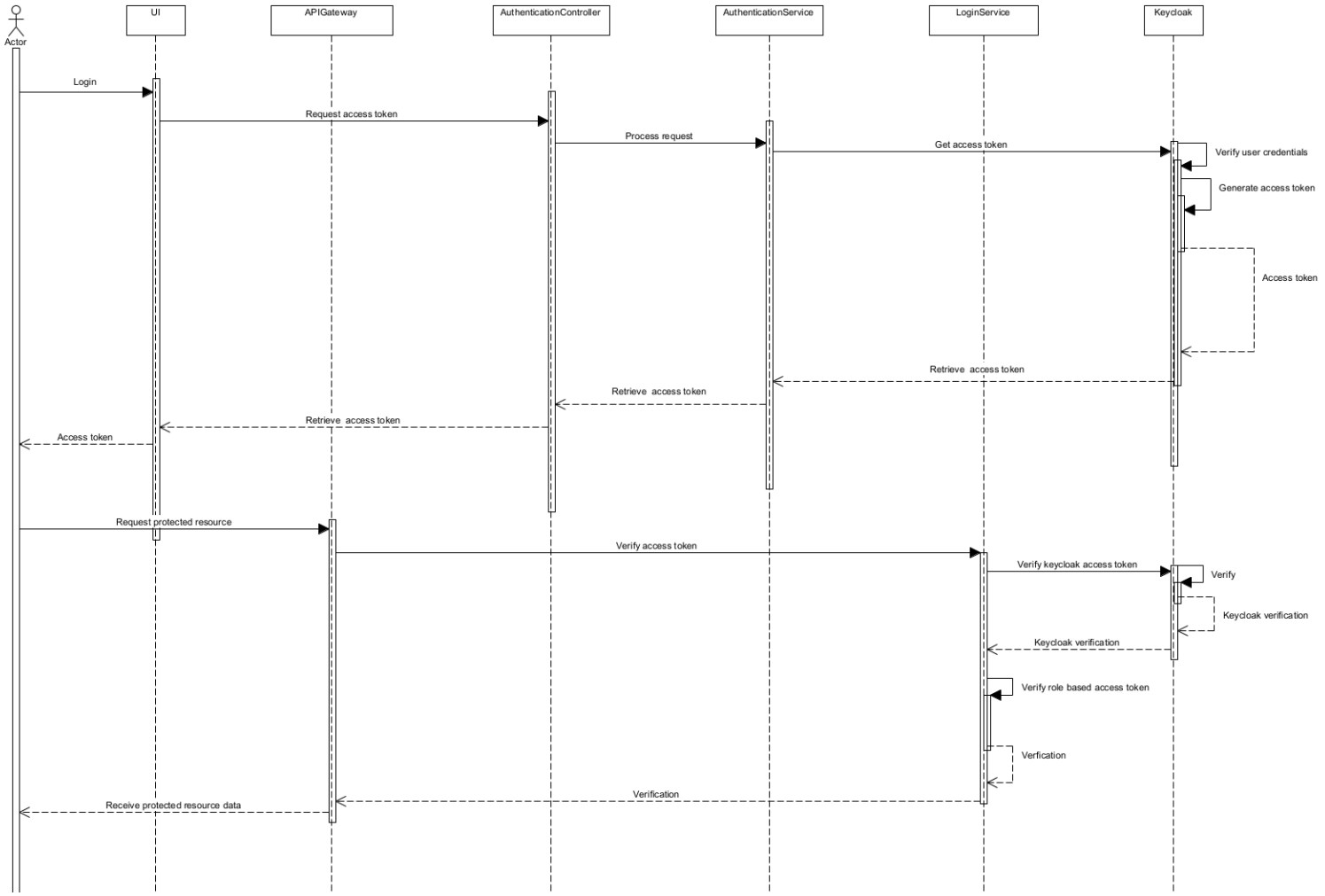


Figure 3.6.3.1: Sequence Diagram for Login service

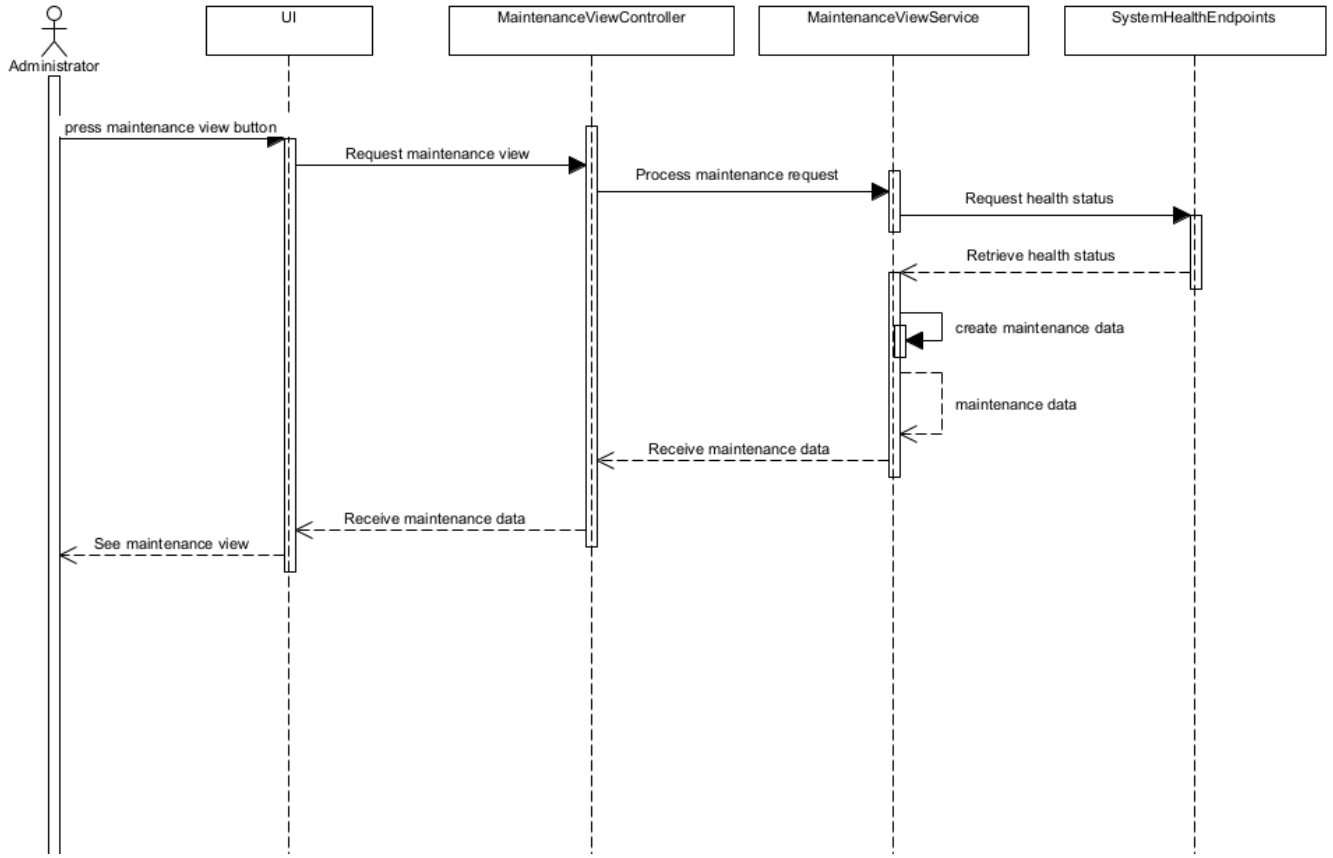


Figure 3.6.3.2: Sequence Diagram for Login service

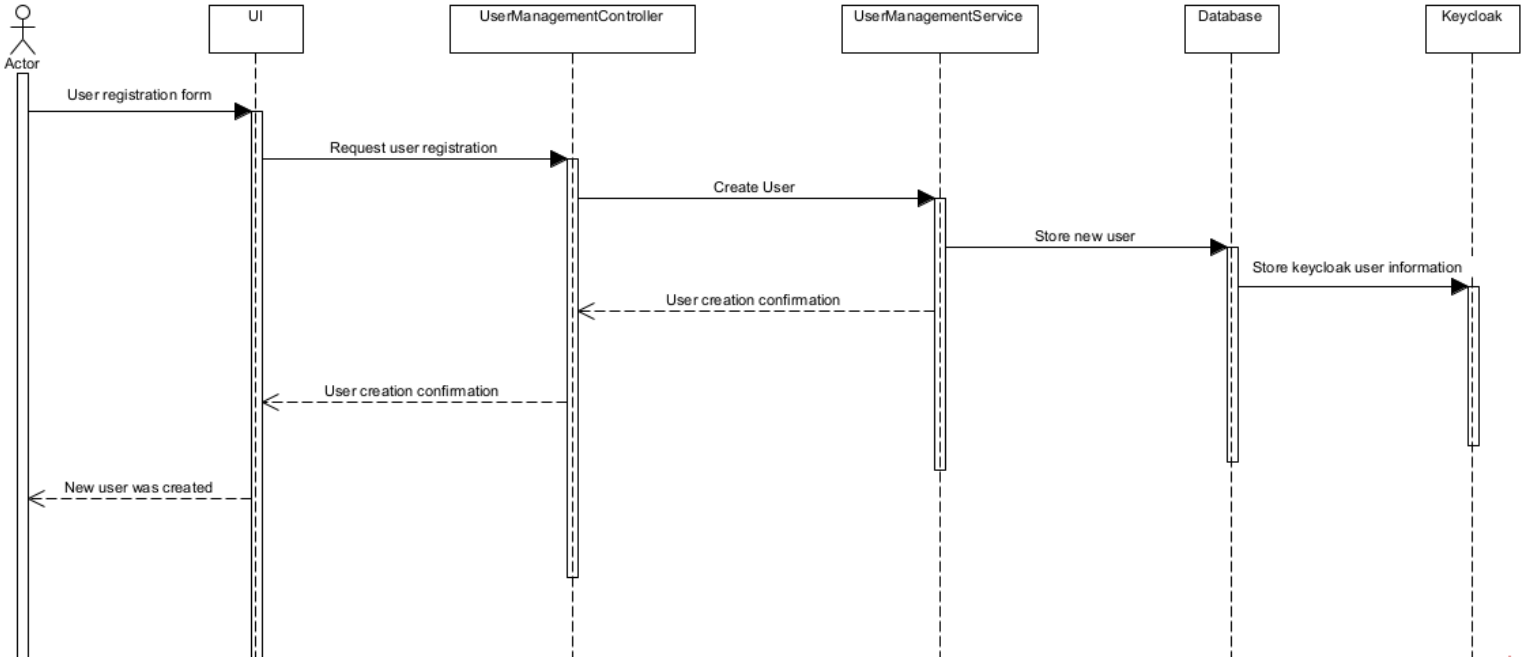


Figure 3.6.3.3: Sequence Diagram for Login service

SR6

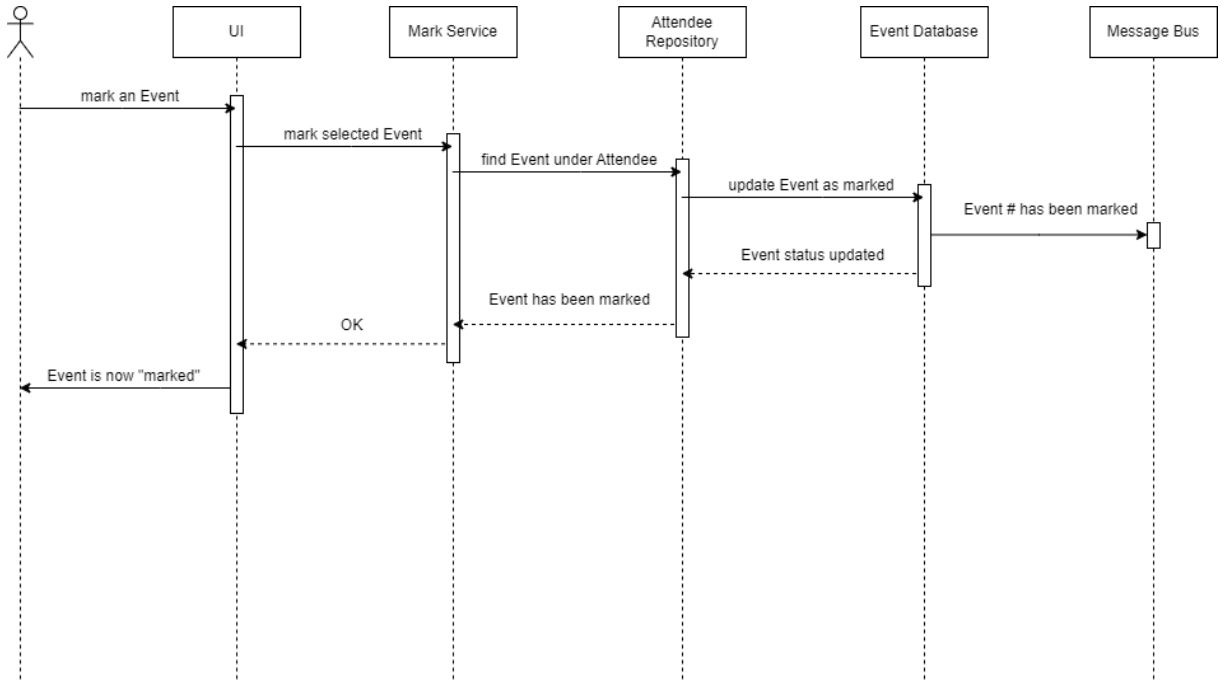


Figure 3.6.6.1: Sequence Diagram for bookmarking an Event

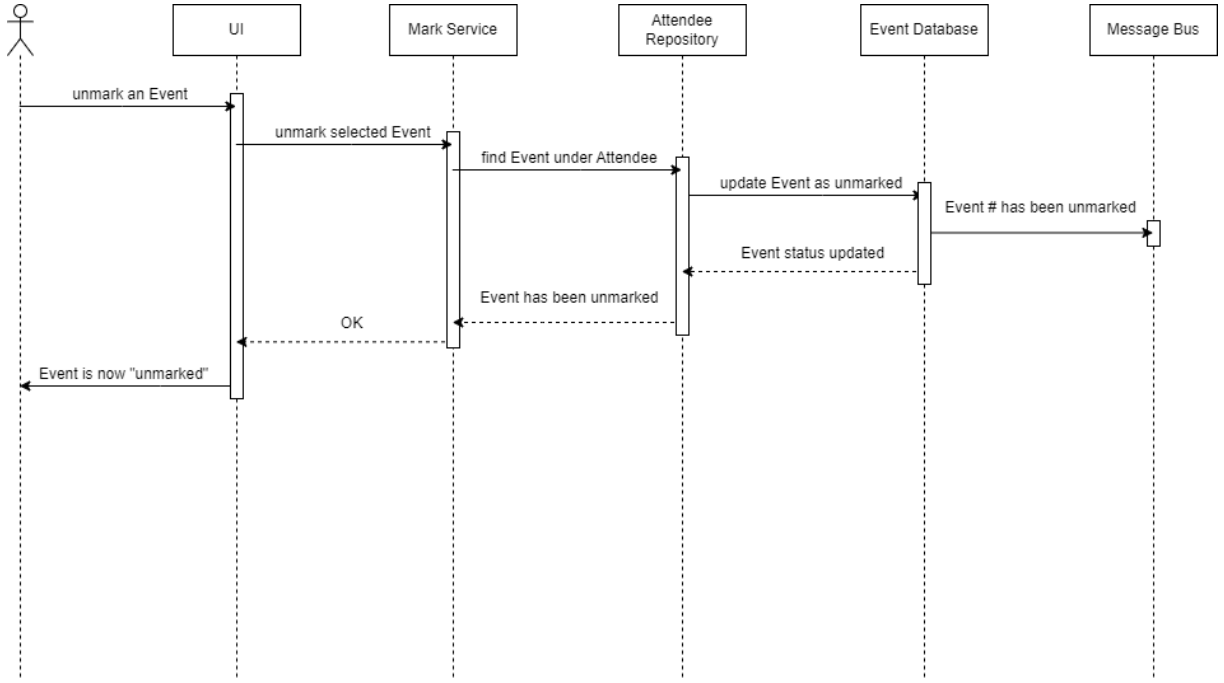


Figure 3.6.6.2: Sequence Diagram for unbookmark an Event

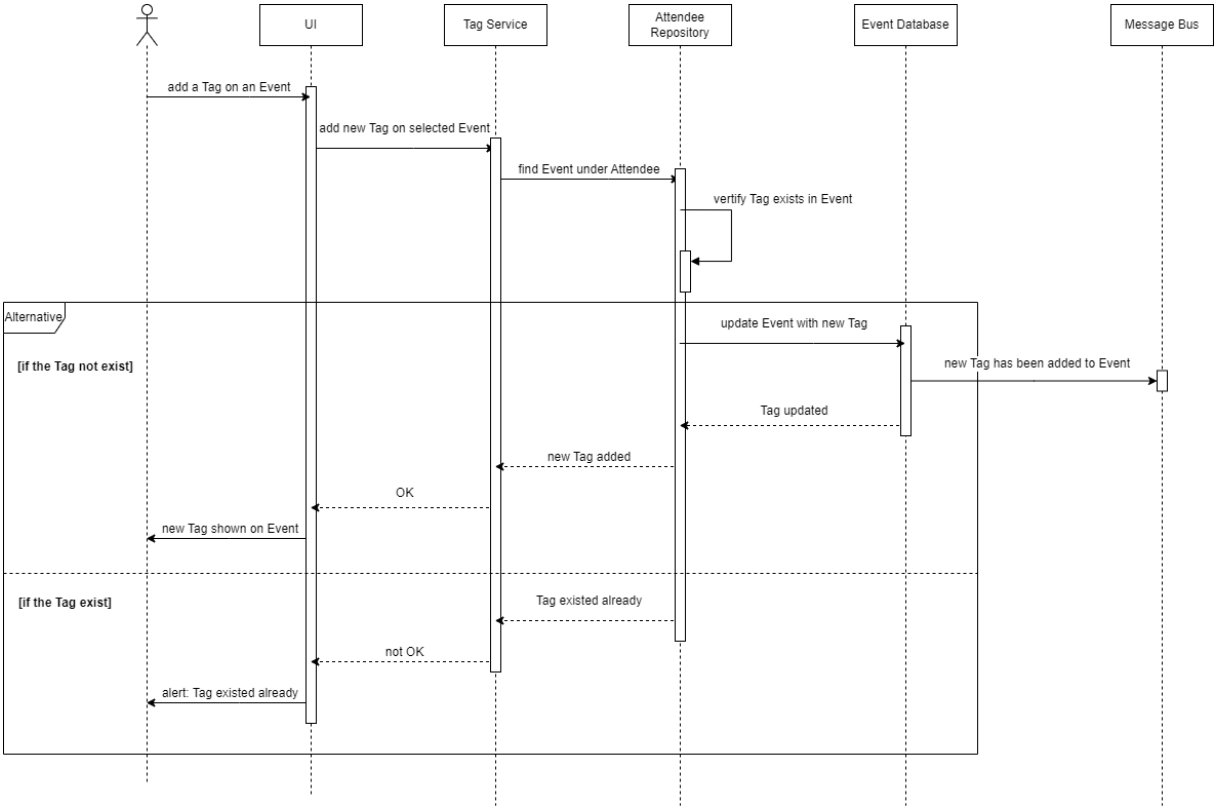


Figure 3.6.6.3: Sequence Diagram for adding a Tag to an Event

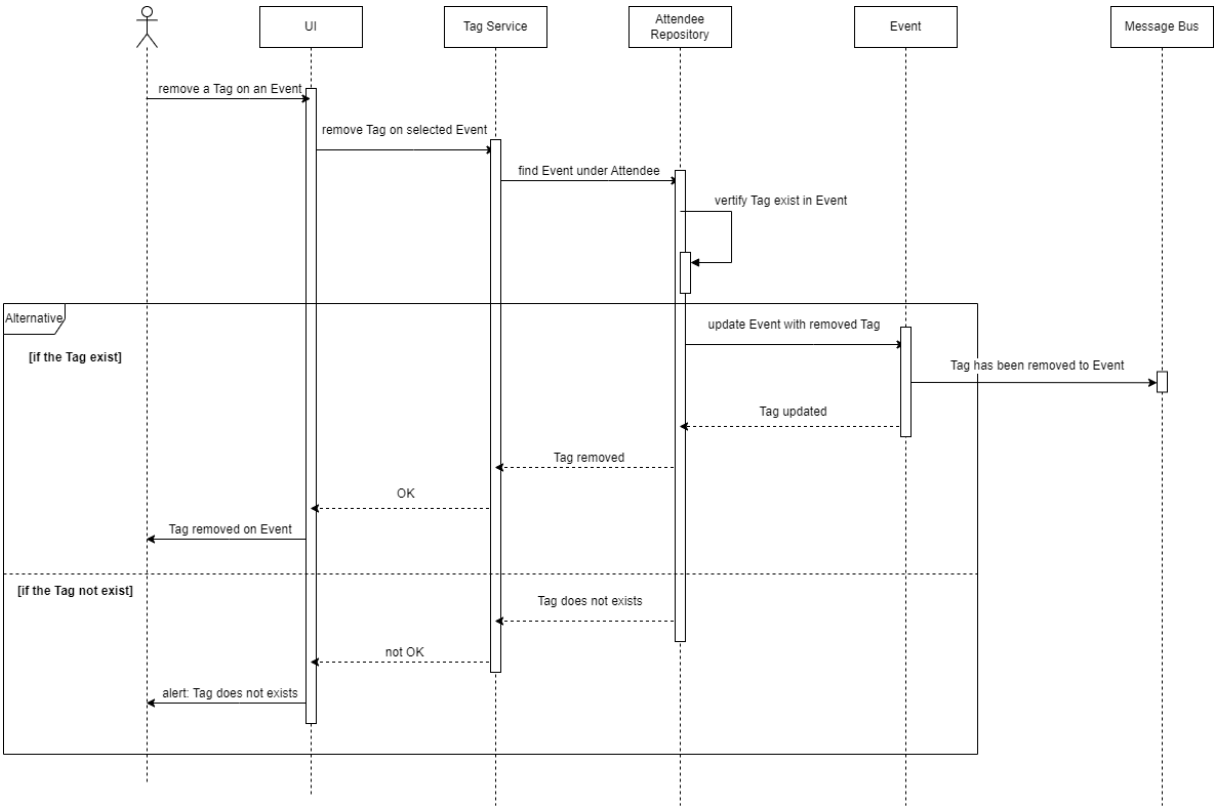


Figure 3.6.6.2: Sequence Diagram for removing a Tag to an Event

SR7

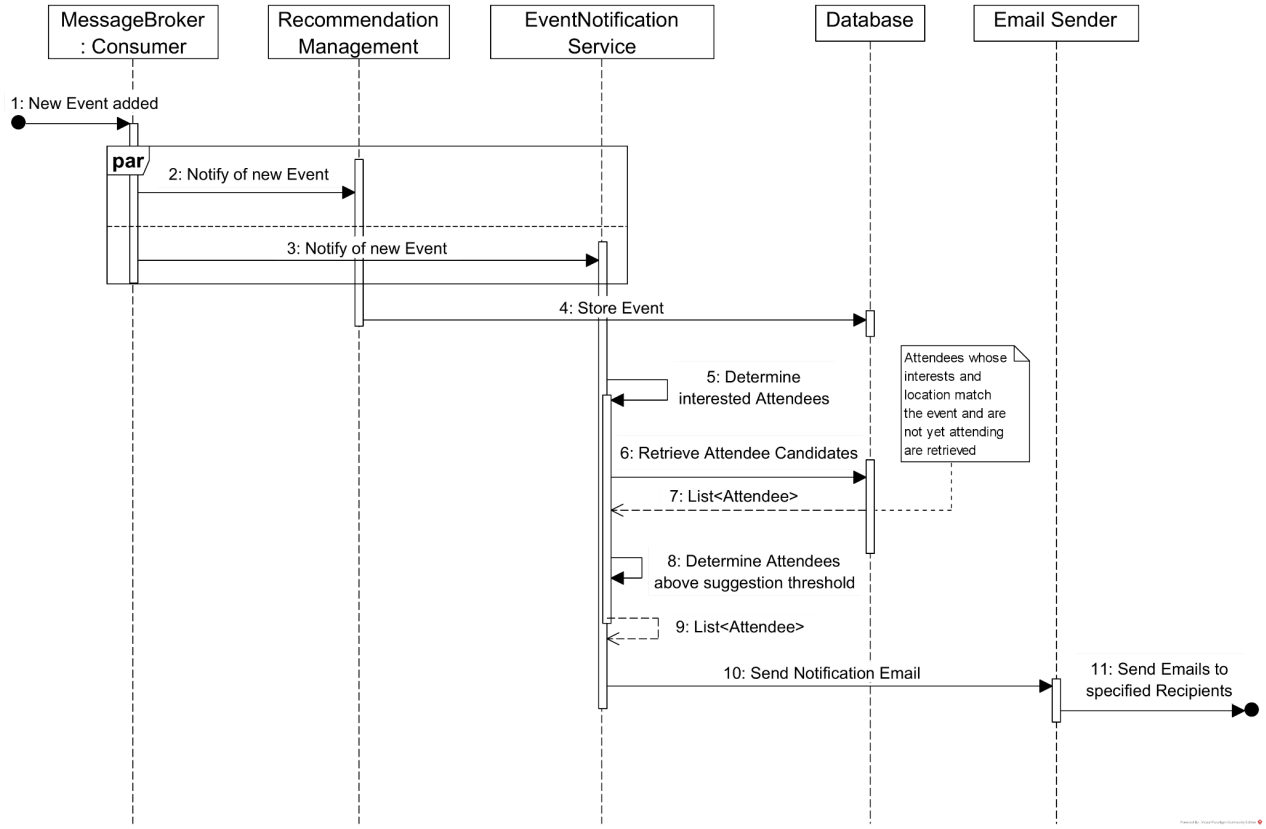


Figure 3.6.7.1: Sequence Diagram for suggesting newly added Events to Attendees (see UC07-01)

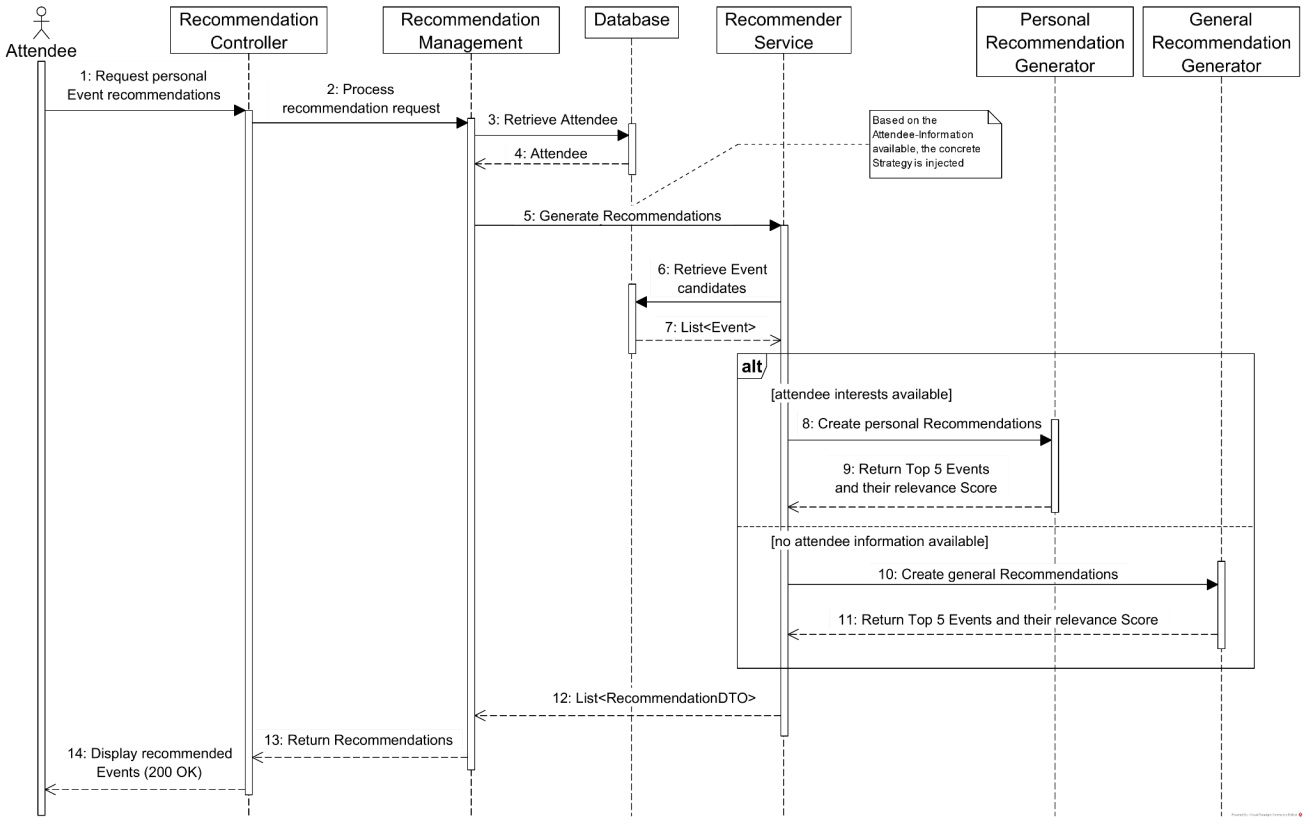


Figure 3.6.7.2: Sequence Diagram for request personal Event Recommendations (see UC07-02)

SR8

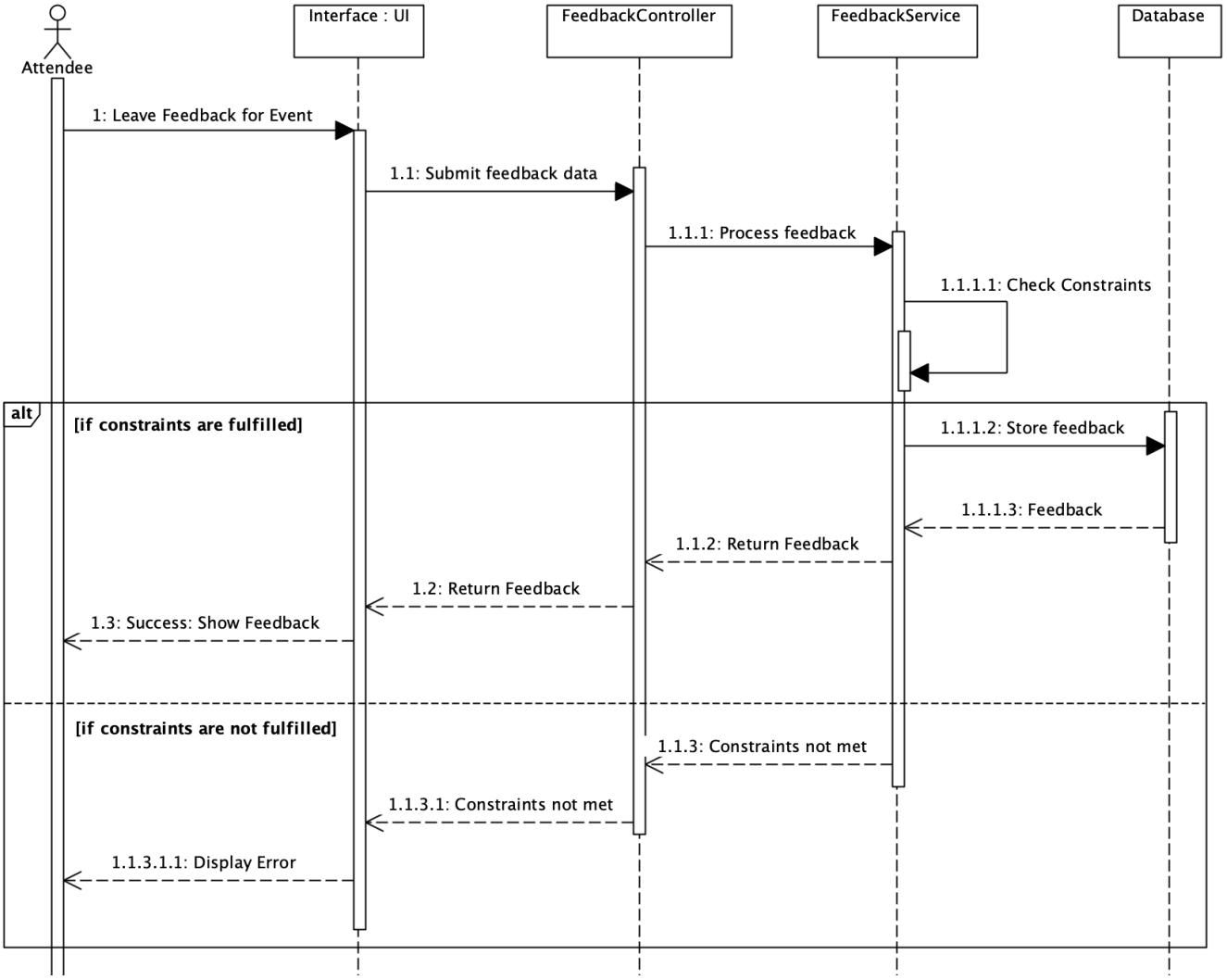


Figure 3.6.8: Sequence Diagram for Feedback service

SR9

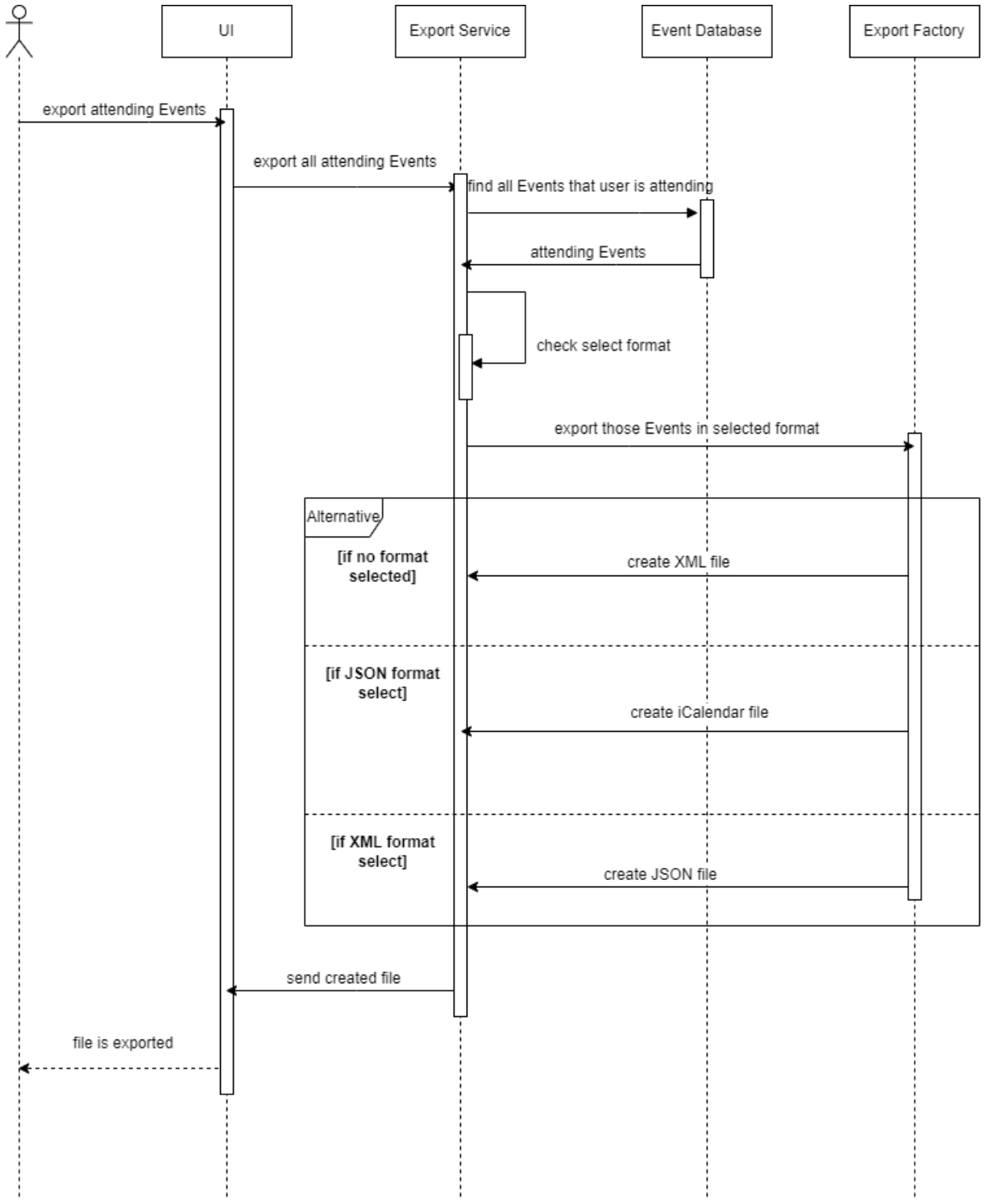


Figure 3.6.9.1: Sequence Diagram for exporting all attending Events

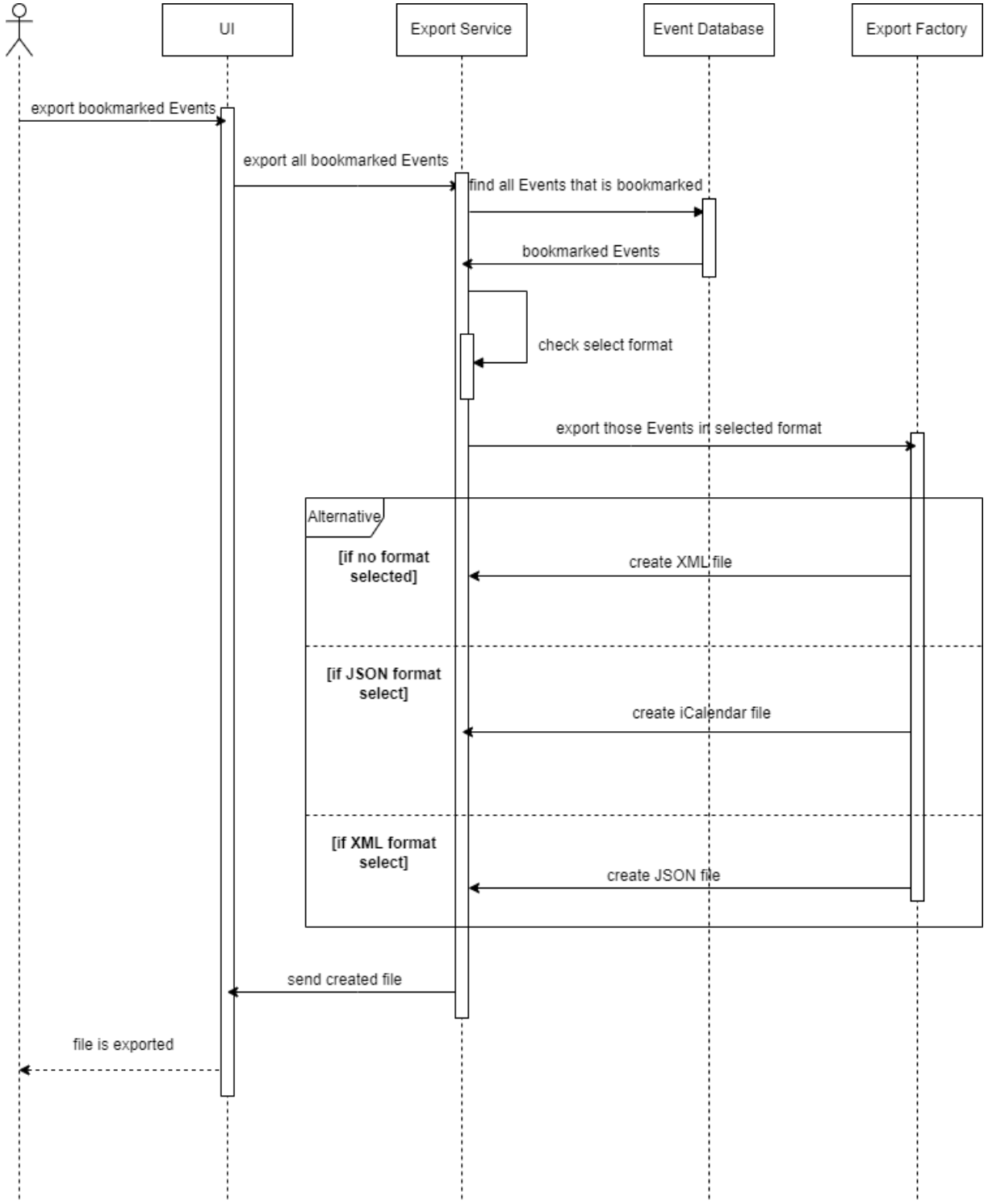


Figure 3.6.9.2: Sequence Diagram for exporting all bookmarked Events

SR10

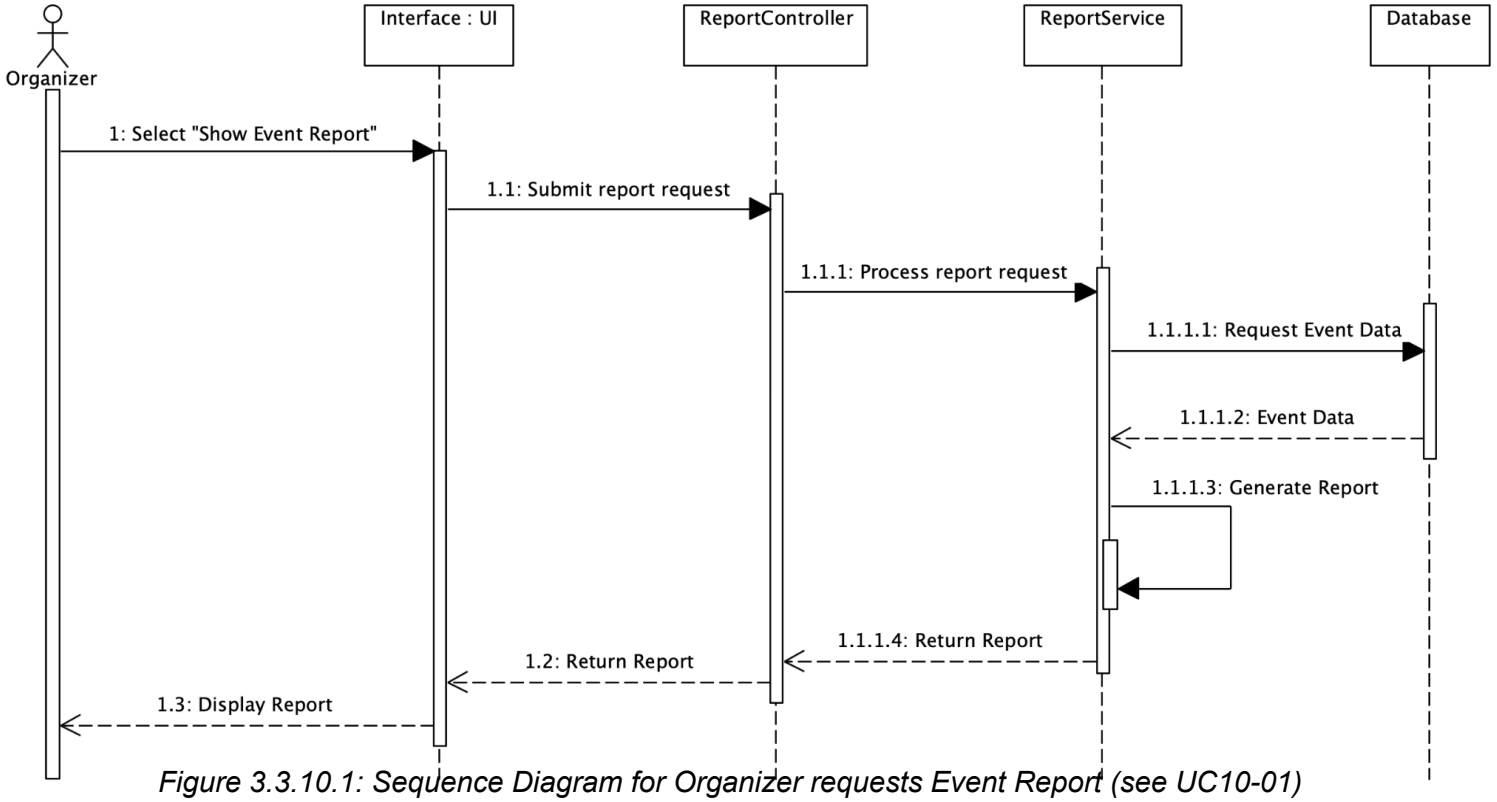


Figure 3.3.10.1: Sequence Diagram for Organizer requests Event Report (see UC10-01)

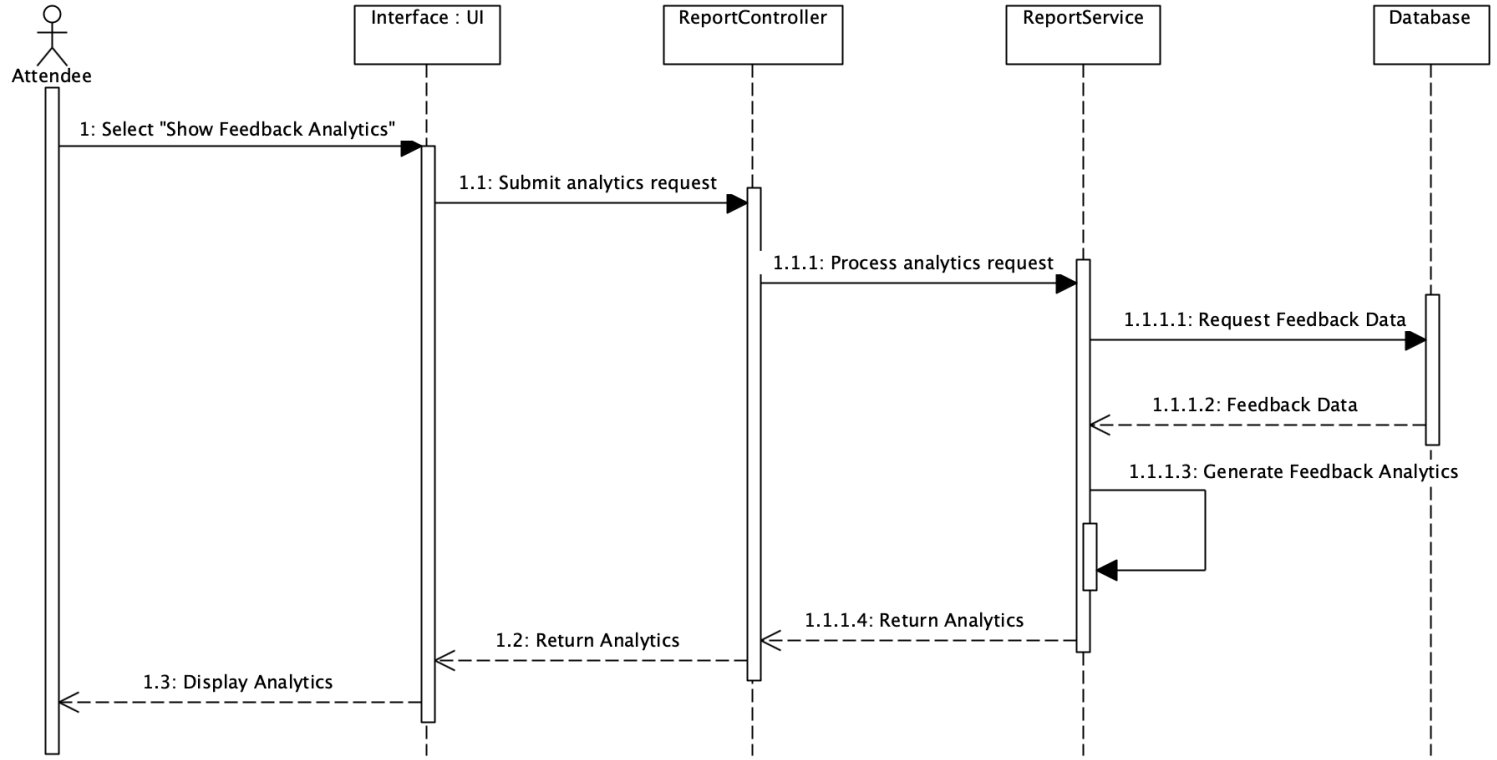


Figure 3.3.10.2: Sequence Diagram for Attendee requests Feedback Analytics (see UC10-02)

SR11

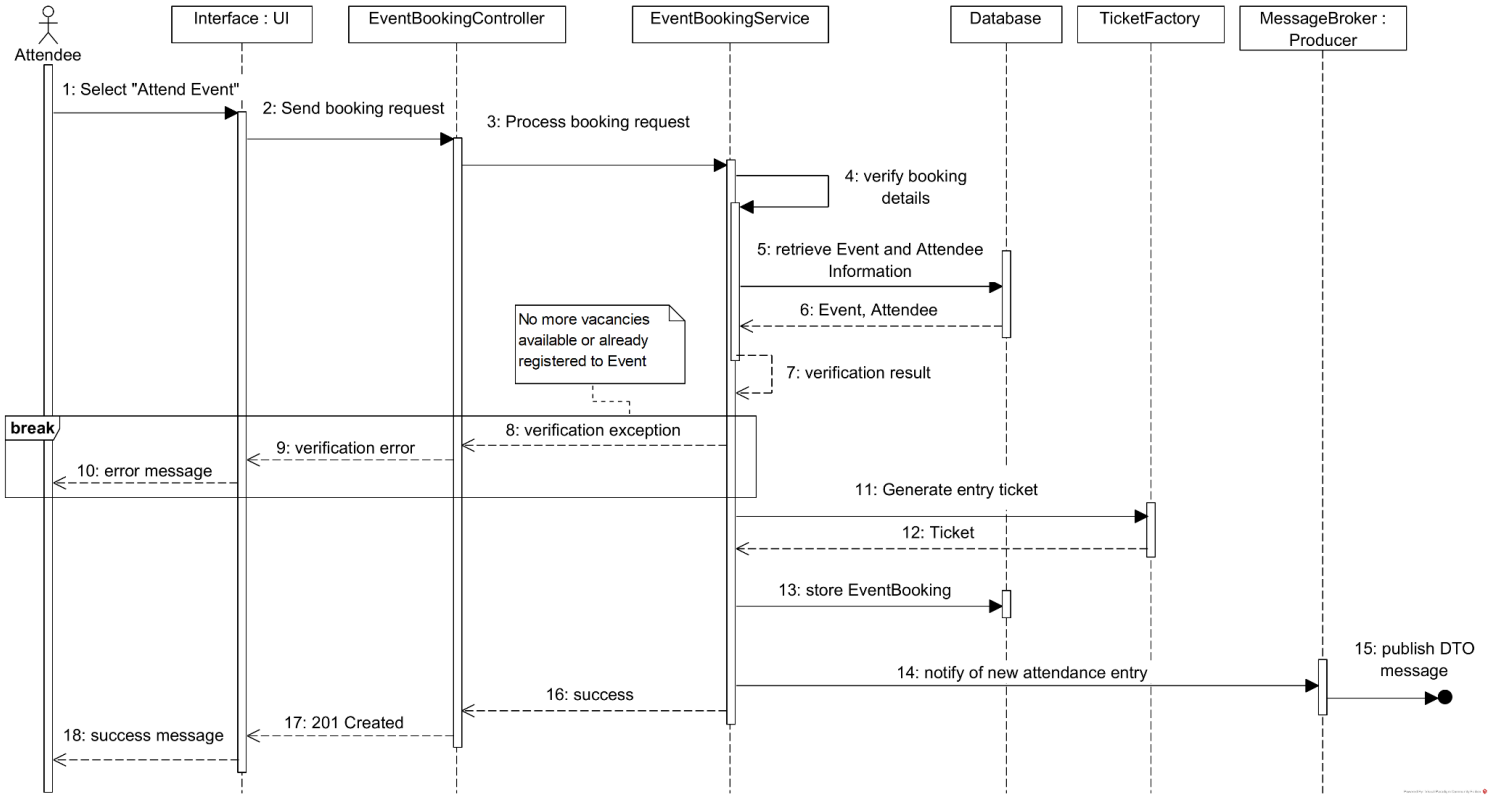


Figure 3.6.11.1: Sequence Diagram for Attendees attending (booking) Events (see UC11-01)

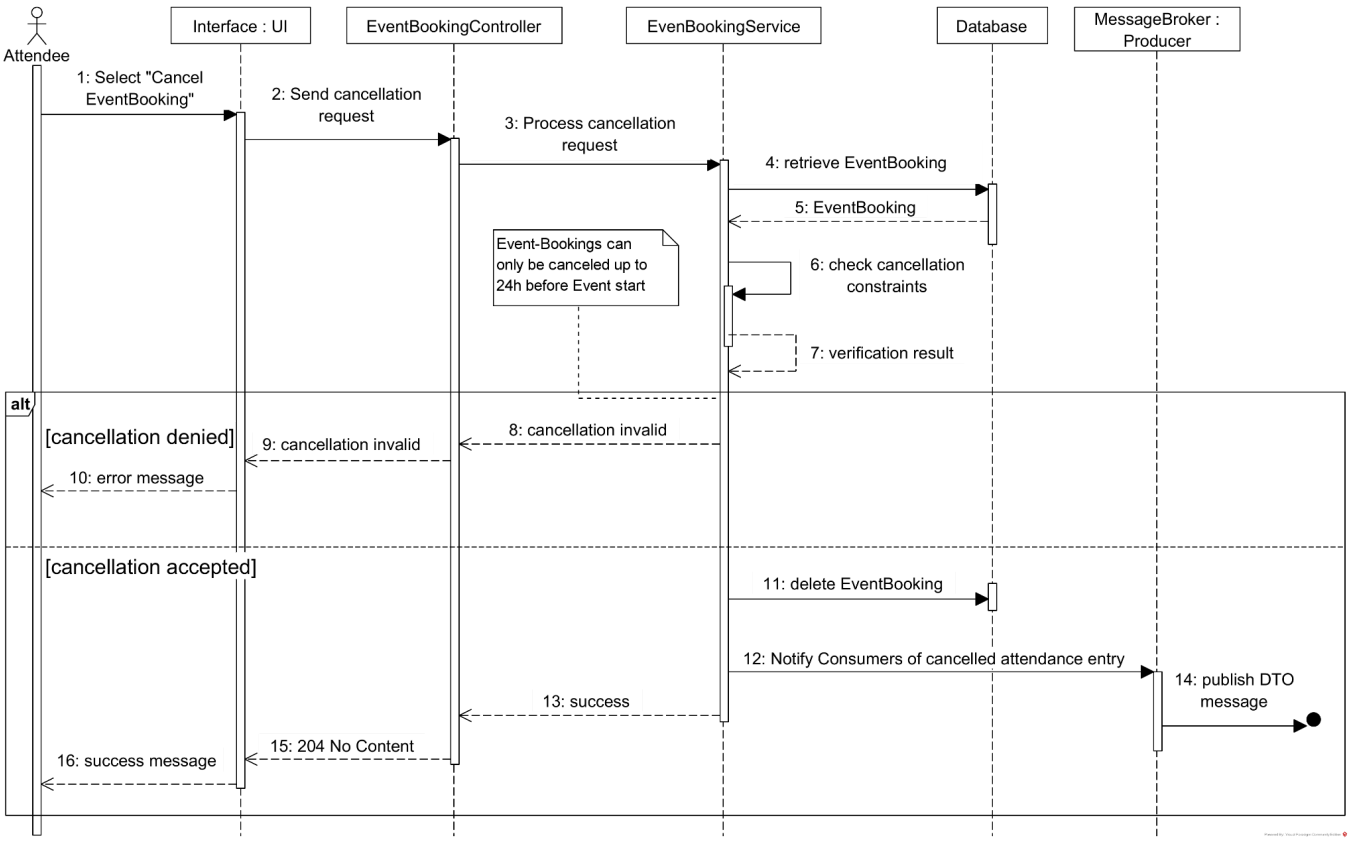


Figure 3.6.11.2: Sequence Diagram for Attendees canceling an EventBooking (see UC11-02)

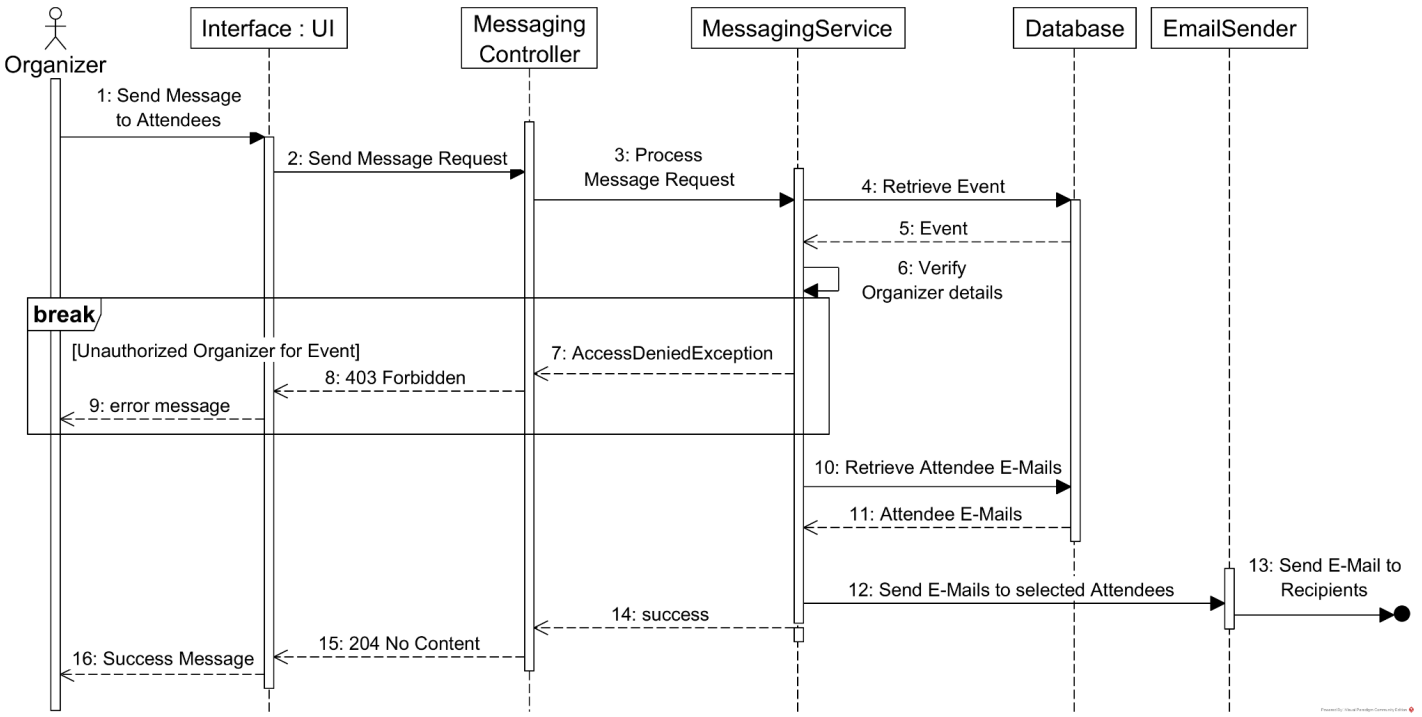


Figure 3.6.11.3: Sequence Diagram for Organizer sends message to their Attendees (see UC11-03)

SR12

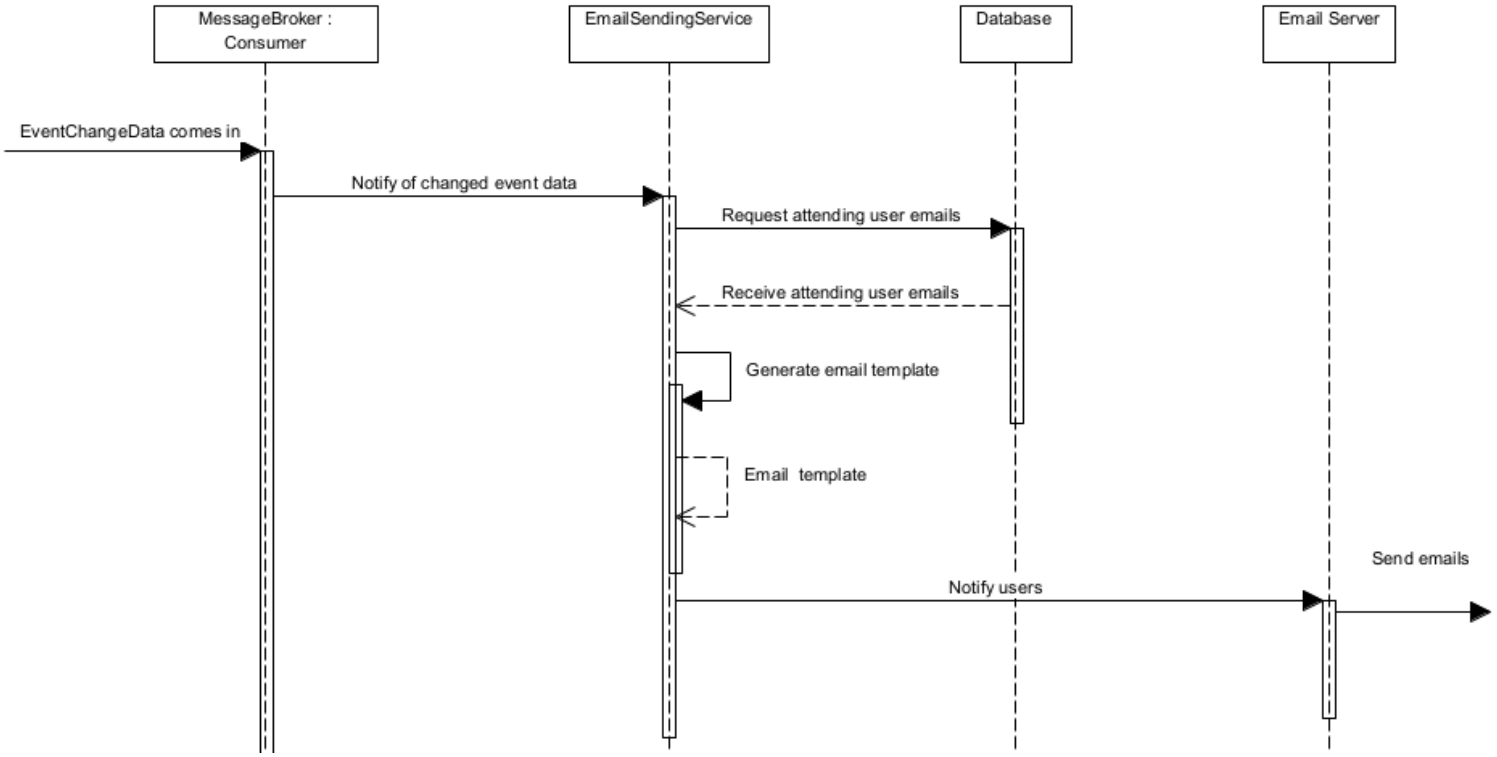


Figure 3.6.12: Sequence Diagram for Notification service

3.7. Development View

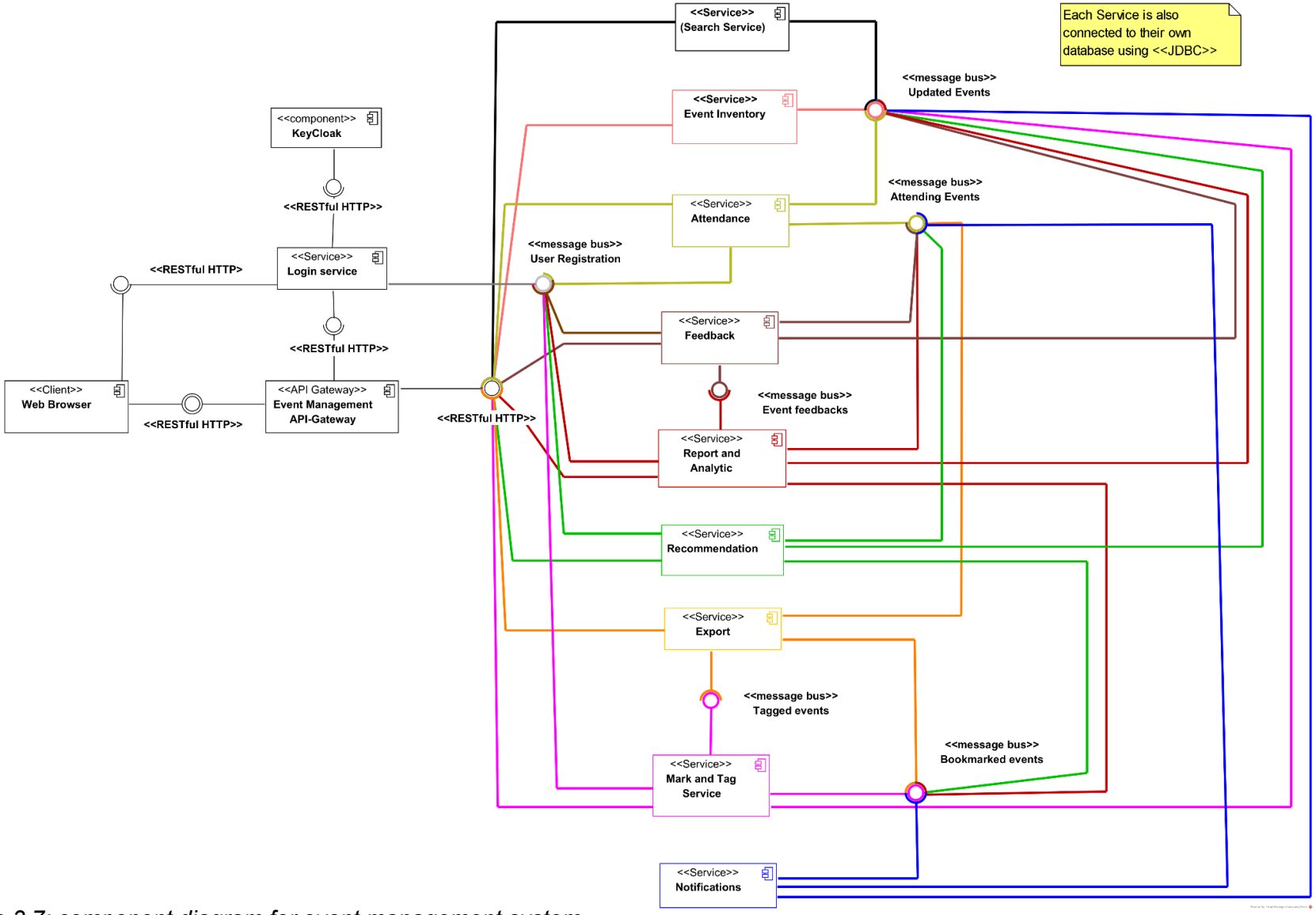


Figure 3.7: component diagram for event management system

3.8. Physical View

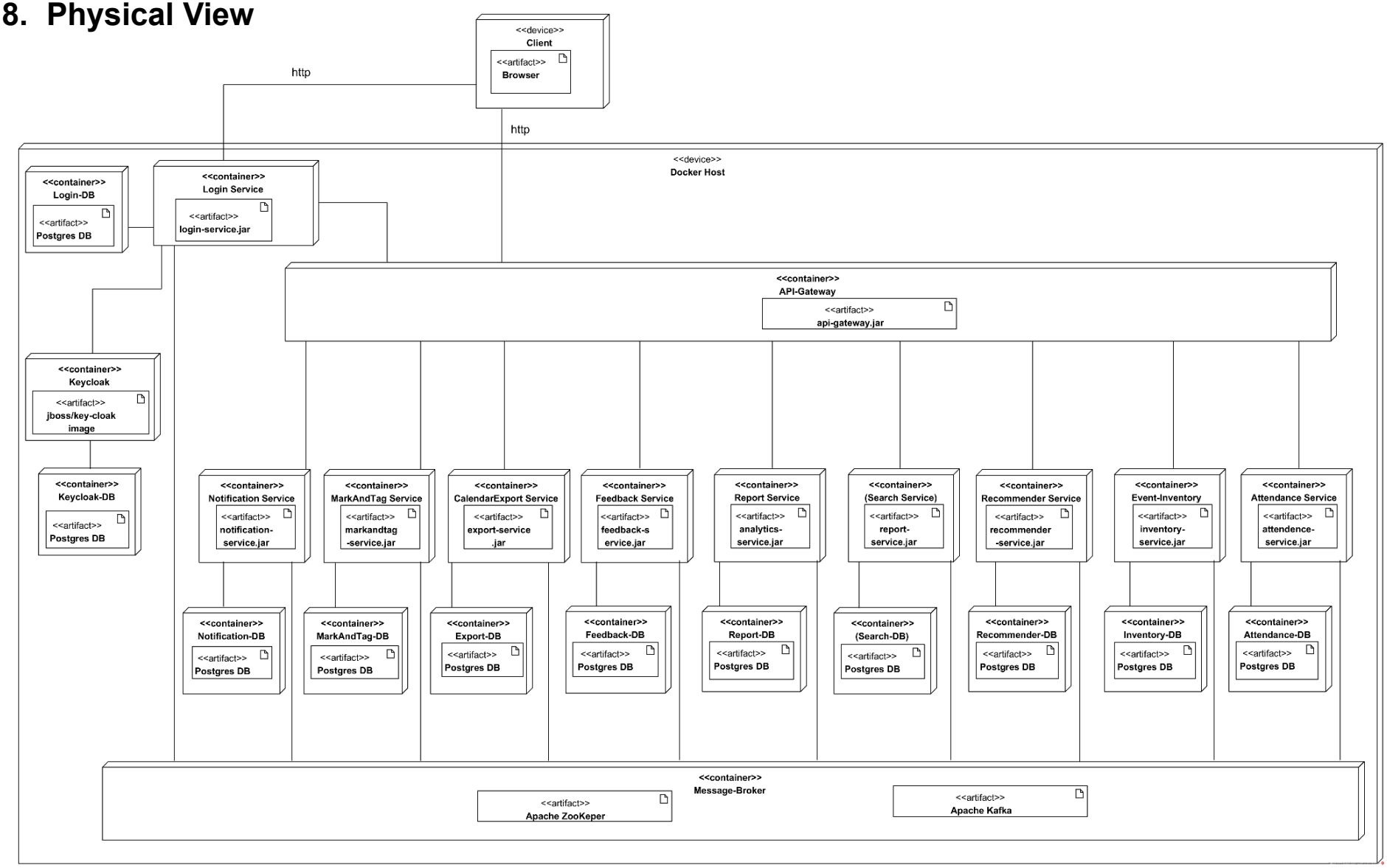


Figure 3.8: deployment diagram for event management system

4. Continuous Delivery

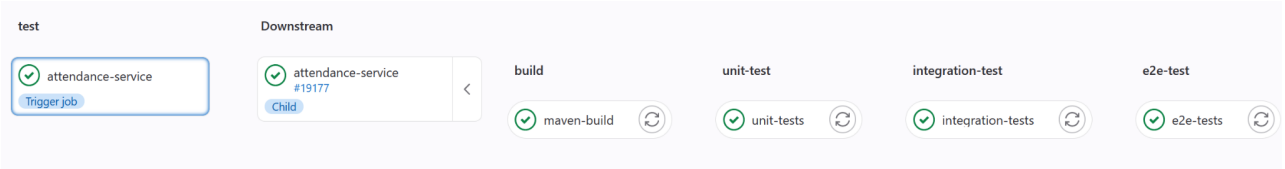
4.1 CI Pipeline & Testing Strategy

Our CI Pipeline is structured in the following way: we have one .gitlab-ci.yml in the root of the repository and then one pipeline configuration file for each service. The configuration in the root is first invoked when a commit is pushed and merely contains triggers which check whether the changes were made in a directory containing one of our service implementations. If this is the case, the pipeline of the affected services is then executed, which can be seen in the figures below.

Exemplary Trigger-Job to run the pipeline on changes in the directory:

```
attendance-service:
  trigger:
    include: implementation/attendance-service/.gitlab-ci.yml
    strategy: depend
  rules:
    - changes: [implementation/attendance-service/**/*]
```

The pipeline of the affected Service(s) being executed:



Our CI-Pipeline is structured into the following stages:

1. **Build:** This stage builds the jars of all services and provides them as artifacts which can be downloaded and used in later stages. Usually only the jar for which a pipeline is triggered is built, however as we require them later in the pipeline to run the e2e-tests, we decided to build all of them here.
2. **Unit-test:** Runs all unit-tests of the given service. As these are individual for each SR, see Section 6 for more documentation. If just one of them fails the pipeline fails and is aborted. Additionally, the one who committed is notified via email.
3. **Integration-Test**: Runs all integration-tests of the given service. As these are individual for each SR, see Section 6 for more documentation.
4. **End-to-End Tests:** In this stage, we test the whole system without mocking any components and execute the main use-case for each service to test that they work well together and there are no issues also concerning message broker integration or the databases.

For this we basically simulate the request flow that a user would typically make and then verify the results that come back through assertions. This also lets us verify whether information that is created is correctly propagated through the system using the message broker.

To run the end-to-end tests in the pipeline (for which we created a separate project /implementation/end-to-end) we run docker-compose up to start all components which utilizes the previously built jars provided as artifacts. Afterwards the docker image of the e2e-test project is built and then executed after a slight delay (using a rather inelegant solution to just sleep for a set amount of time) to make sure that all services are up and running. The results are then logged and if no assertion fails, the pipeline succeeds.

4.2 Lessons Learned

As outlined in the project schedule during DESIGN, setting up a CI-Pipeline was one of our first priorities. By doing this at the start, we can actually benefit greatly from it during the development as it allowed us to detect issues earlier and reduce the risk of pushing incorrect code to the repository. While setting up a CI-pipeline was new for all of us, and some struggles arose, we did not want to wait until the end to set it up as it would’ve been quite pointless after the implementation.

While there were some issues particularly with setting up the end-to-end tests and the artifacts not being deleted automatically after some time, it really helped to check whether individual changes to a service would cause issues during the integration with the rest of the system. As such, it would have been even more beneficial to implement the e2e tests early for earlier detection of issues.

Another concern that we are aware of, was the commits that were purely testing the pipeline environment pile up really quickly. The commits itself do not add much value, since the failing conditions were only caused by the environment setup, all our unit tests, integration tests, end to end tests were passed within our local environment, thus might be better for the future to test the CI-Pipeline in a separate branch. However, this was also an important part in the learning process and getting used to working with GitLab CI/CD. As we did quite a lot of test runs to set up the e2e-tests, we eventually then even managed to reach the 6 hour pull limit from docker.

5. Team Contribution and Continuous development method

5.1. Project Tasks and Schedule

Task	Start	Finish	Duration	Priority	Responsibility	13.03.23 11	20.03.23 12	27.03.23 13	03.04.23 14	10.04.23 15	17.04.23 16	24.04.23 17	01.05.23 18	08.05.23 19	15.05.23 20	22.05.23 21	29.05.23 22	05.06.23 23	12.06.23 24
Analysis																			
Requirements Analysis	10.03.23	13.03.23	3	1	ALL														
Techstack Determination	13.03.23	20.03.23	7	1	ALL														
Work Distribution	20.03.23	20.03.23	1	1	ALL														
Design																			
Indiviual Approach, Assumptions, Decisions	20.03.23	24.04.23	35		ALL														
Use Case Definition and Description	20.03.23	03.04.23	14		ALL														
Logical View	27.03.23	17.04.23	21		ALL														
Process View	10.04.23	17.04.23	7		ALL														
Development View	17.04.23	24.04.23	7		ALL														
Deployment View	17.04.23	24.04.23	7		ALL														
Documentation	17.04.23	24.04.23	7		ALL														
DESIGN Submission	24.04.23	24.04.23	1		ALL														
Development																			
Mock-Up of each service	27.03.23	24.04.23	28		ALL														
Containerization	17.04.23	05.05.23	18		ALL														
Implementation CI/CD Pipeline	03.05.23	12.05.23	9		ALL														
Implementation of each services	01.05.23	29.05.23	28		ALL														
Testing																			
Unit-Tests	08.05.23	15.05.23	7		ALL														
Integration-Tests	15.05.23	29.05.23	14		ALL														
End-to-end-Tests	22.05.23	05.06.23	14		ALL														
Final Delivery																			
Report	29.05.23	05.06.23	7		ALL														
Final Project Submission	08.06.23	08.06.23	1		ALL														
Project Presentation			1		ALL														

5.2. Distribution of Work and Efforts

Responsibility	Team Member
SR1 + SR2	ALL
SR3 + SR12	Fritz Ziernhöld
SR4 + SR5	LEFT THE TEAM
SR6 + SR9	Jin Yu
SR7 + SR11	Matthias Fritz
SR8 + SR10	David Reichert

Team Member: David Reichert	
FINAL	**Time spent**
Meetings	30h
Implementation	30h
Unit-Tests	15h
Integration-Tests	10h
End-to-End Tests	5h
CI/CD Pipeline	10h
FINAL Report	5h
Overall	130h

Team Member: Fritz Ziernhöld	
FINAL	**Time spent**
Meetings	30h
Implementation	30h
Unit-Tests	25h
Integration-Tests	10h
End-to-End Tests	25h
CI/CD Pipeline	25h
FINAL Report	5h
Overall	150h

Team Member: Matthias Fritz	
FINAL	**Time spent**
Meetings	30h
Implementation	25h
Unit-Tests	14h
Integration-Tests	10h
End-to-End Tests	12h
CI/CD Pipeline	20h
FINAL Report	12h
Overall	123h

Team Member: Jin Yu	
FINAL	**Time spent**
Meetings	30h
Implementation	24h
Unit-Tests	15h
Integration-Tests	15h
End-to-End Tests	7h
CI/CD Pipeline	13h
FINAL Report	8h
Overall	112h

6. How-to documentation

Frontend:

To use the system and verify its use-cases, a Vue.js frontend was created. When starting the whole system (**/implementation/deploy.sh**) it will also start the container necessary for the frontend (as shown in the video). Subsequently after waiting for all services to start, it can be accessed under localhost:4200. While the CORS-issue has been fixed, it showed that some ad-blockers can sometimes block requests to the analytics-and-report service, thus it is best to disable them beforehand.

However it was tested with Firefox and Chrome and all requests worked when disabling ad-blockers.

Alternatively the web-security can also be disabled to make sure that all requests go through:

Run in folder of chrome.exe

```
chrome.exe --disable-web-security
--user-data-dir="c:/ChromeDevSession"
```

SR3 & SR12

Containerization

1. Create the java jars by running `mvn package -Dmaven.test.skip` in the login-service and notification-service folder respectively
2. Navigate back to the implementation folder. Run `docker compose up -d login-service` to start the login-service and `docker compose up -d notification-service` to start the notification-service.

Mind that both services depend on other containers that will start up as well when running the docker-compose commands.

CI/CD pipeline

Any changes to the notification-service or the login-service will trigger the pipeline to run. The pipeline consists of 4 stages, build, unit test, integration test and end-to-end test. Both configurations can be found in the .gitlab-ci.yml file in the folder respectively.

Build:

For both services, SR3 and SR12 the build stage is equivalent. We create all jars from each-service and expose them as artifacts for the following tests.

```
mvn clean package -Dmaven.test.skip --projects
!edu.ems:frontend
```

Unit tests:

Integration tests require the actual database and other services to run so therefor to run the unit tests, each specific test class is listed in the maven test call.

SR3:

```
mvn test
-Dtest=EMSUserRepositoryTest,AuthenticationServiceImplTest,Maintena
nceViewServiceImplTest,UserManagementServiceImplTest
```

SR12:

```
mvn test -Dtest=EventRepositoryTest
```

Integration tests

For both services when running integration tests on the pipeline, all necessary containers and services are run in a docker environment before the tests start. These tests are trying to simulate real behavior and check the actual databases and brokers for behavior. For this to work properly, the integration tests are build in a Dockerimage and then deployed in the same network as the running services.

SR3:

- In the login-service folder, run `docker build --build-arg SKIP_TESTS_BUILD="true" -f Dockerfile.integration -t integration-tests .`
- In the implementation folder, run `docker-compose up --build -d login-service-db zookeeper broker keycloak-db keycloak`
- In the login-service folder, run `docker run -e SKIP_TESTS_RUN="false" --network implementation_default integration-tests`

SR12:

- In the notification-service folder, run `docker build --build-arg SKIP_TESTS_BUILD="true" -f Dockerfile.integration -t integration-tests .`
- In the implementation folder, run `docker-compose up --build -d notification-service-db zookeeper broker`
- In the notification-service folder, run `docker run -e SKIP_TESTS_RUN="false" --network implementation_default integration-tests`

End to End tests

In the implementation folder you will find a script called “e2e.sh”. This script simulates all the steps the pipeline does when running our end-to-end tests. Please run that script to test the end-to-end tests.

SR7 & SR11

Containerization

The main artifact for the containerization can be found under

- `/implementation/recommender-service/Dockerfile`
- `/implementation/attendance-service/Dockerfile`

First, a JDK17 image is pulled, then the working directory is set to /app and consequently the previously built jar is copied into the container. The entrypoint command then specifies the command that should be executed when the container is started, in this case executing the jar.

To build the JAR, execute the following command in the root folder of the service:

```
mvn package -Dmaven.test.skip
```

Start the Service(s):

For the application to start, its database must also be started as well as the message broker in order to communicate with other services. However, building and running each image separately would be very tedious, thus a docker-compose.yml is used. It also makes use of environment variables e.g. to set the same database credentials for both the database and service container as well as health-checks to make sure that the service starts after the database.

To only start containers relevant to this microservice, we can execute the following command in the /implementation folder:

- **SR7:** `docker compose up -d recommender-service`
- **SR11:** `docker compose up -d attendance-service`

This will then start all containers necessary to run the service.

Alternatively if only the image is required it can be built executing the following command in the root directory of the service:

- **SR7:** `docker build -t recommender-service .`
- **SR11:** `docker build -t attendance-service .`

CI Pipeline:

The CI Pipeline is triggered whenever a change for the respective service has been committed. To achieve this, a .gitlab-ci.yml configuration is placed in the project root which contains triggers which then include the service's pipeline configuration when a change in its directory was pushed.

The CI configurations can be found under:

- `implementation/recommender-service/.gitlab-ci.yml`
- `implementation/attendance-service/.gitlab-ci.yml`

The Pipeline for the two services then includes 4 stages: (*commands must be executed in the root of the respective service)*.

1. **build**: in this stage the jars of all services are built and then provided as artifacts which can then be downloaded or used in other stages. The reason why all jars are built instead of only the service itself is that they will later be necessary during the e2e-tests

2. **unit-test**: All unit-tests of this service are executed using a maven image. If one test fails, the pipeline fails and is abandoned. The unit-tests are very fine-grained and their scope is on the method-level and focus on verifying business logic in isolation of other components. They are executed via:

   ```
   mvn test -Dtest=\!integration.*Test
   ```

3. **integration-tests**: The integration tests are performed as the next stage in the pipeline. These now focus on the integration between modules in the services and thus include all 3 layers and verify their integration. Here, other external services are mocked, i.e., messages of other services are simulated by pre-filling an in-memory database. The tests then use Spring's MockMvc to simulate a client making requests to the service. It is then checked whether the correct response is returned, and the state of the data is also verified by checking the updates in the database. These can be executed via:

   ```
   mvn test -Dtest=integration.*Test
   ```

4. **e2e-tests:** After the previous stages have passed, End-to-End tests are executed which now no longer mock other components, but deploy the whole system and test the main use-cases to verify that the services and their used components (database, message broker, api-gateway) work together correctly. For this, the previously created JAR-artifacts will be used. As this was a joint-effort and is not specific to the SR, please refer to section [4. Continuous Delivery] for additional information.

Additionally, to build the JAR and container image, as well as run unit and integration-test, the provided script 'test.sh' can be executed (requires maven and docker):

```
1. chmod +x test.sh
2. ./test.sh
```

The E2E tests can be executed similarly using the e2e.sh script in the implementation folder.

SR6 & SR9

Containerization

Step 1: To containerize the Marktag service and Export service, the jar for both service need to be build first with command within their root folder:

```
mvn package -Dmaven.test.skip
```

Step 2: The docker container for each services could be started with the following commands (Note: all the dependent services e.g. marktag-service-db, broker, etc. will be started as well during the execution):

```
docker compose up -d marktag-service
```

```
docker compose up -d export-service
```

Alternatively, all services could be started in `docker-compose.yml` directly, located in the implementation directory, that executes the `Dockerfile` in each services directory.

CI/CD pipeline

The CI/CD pipelines for Marktag service and Export service are only triggered when changes happened within their service and pushed into the branch.

During the pipeline, a set of unit tests, integration tests and end to end test will be automatically executed (there are no explicit steps that needs to be performed during the execution).

The detailed configuration is defined in the .gitlab-ci.yml file inside each services.

Build:

the build stage is fundamental for creating artifacts and environment. Since the end to end test examine the behavior of other services too, all servies jars are mandatory to be build (except front end) with the command as stated in .gitlab-ci.yml file below:

```
clean package -Dmaven.test.skip --projects !edu.ems:frontend
```

Unit tests:

The unit test include the topic-process-tests. As the stated, all of tests are executed under the environment of `maven:3.8.1-openjdk-17`, and the command that are being execute as stated in both .gitlab-ci.yml file are:

SR6:

```
mvn test -Dtest=MarktagServiceModelTest,MarkTagServiceTest
```

```
mvn test -Dtest=MarkTagServiceTopicProcessingTest
```

SR9:

```
mvn test -Dtest=ExportServiceModelTest,ExportServiceTest

mvn test -Dtest=ExportServiceTopicProcessingTest
```

Integration tests

The integration tests test the interaction between different layers, mostly the domain layer and the infrastructure layer, that requires an actual database to start. (Note: if the tests are wish to start locally, the database related to each services need to build and started locally as well)

SR6:

```
mvn test -DPSQL_IP=postgres -Dtest=AttendeeRepoTest,
EventRepoTest,ParentChildrenRelationshipTest
```

SR9:

```
mvn test -DPSQL_IP=postgres -Dtest=AttendingRepoTest,
BookmarkedRepoTest,EventRepoTest
```

End to End tests

The end to end tests execute a sequence a tests that examines the behavior in a “real world setting” that suppose to simulate a real endpoint request. Since all the jars are build during build stage, they are being deployed at this stage with the concept of docker in docker under CICD pipeline environment, and will be pruned (including cache) after the tests are finished. The detailed command can be seen in each .gitlab-ci.yml file.

SR8 & SR10

Containerization

0. Assuming user is in the root folder of the corresponding service
1. Before the containerization can start it is necessary to build the JAR which will then be used and copied into the container according to the upcoming steps. This can be done with this command:
   - `mvn package -Dmaven.test.skip`
2. There is a Dockerfile for each of the services which is used to dockerize the service. All necessary commands are inside of that file:
   - /implementation/analyticsandreport-service/Dockerfile
   - /implementation/attendance-service/Dockerfile
3. After that the services can be started with docker-compose:
   - `docker compose up -d analyticsandreport-service`
   - `docker compose up -d feedback-service`

It is also possible and recommended to use the docker-compose.yml inside of the main implementation folder.

CI/CD

Both services are configured to run in a CI/CD environment. The necessary configuration files are inside of the implementation folder of the corresponding service:

- `implementation/analyticsandreport-service/.gitlab-ci.yml`
- `implementation/feedback-service/.gitlab-ci.yml`

For each service there are four phases to get passed:

- build
  - In this phase, the artifacts of the service are built, typically by utilizing the Maven build command. This process involves compiling the source code, resolving dependencies, and packaging the application into a deployable format.
- unit-test
  - Each service is equipped with unit tests that need to be passed. These tests focus on testing individual components and functions of the service in isolation, ensuring their correctness and expected behavior.
  - Analyticsandreport-Service: AnalysisServiceTest, AttendanceEntryServiceTest, EventServiceTest, ReportService Test

    - Feedback-Service: FeedbackIntegrationTest

- integration-test

    - The integration test phase involves testing the interaction and integration between various components of the service. It verifies that different modules work together seamlessly, checking for proper communication and data flow.

    - Analyticsandreport-Service: AnalyticsIntegrationTest, ReportsIntegrationTest

    - Feedback-Service: AttendanceEntryServiceTest, EventServiceTest, FeedbackServiceTest

- end-2-end test

    - The end-to-end test phase examines the entire service as a whole, simulating real-world scenarios and user interactions. It tests the functionality and performance of the service from start to finish, ensuring that all components work harmoniously and meet the desired requirements.