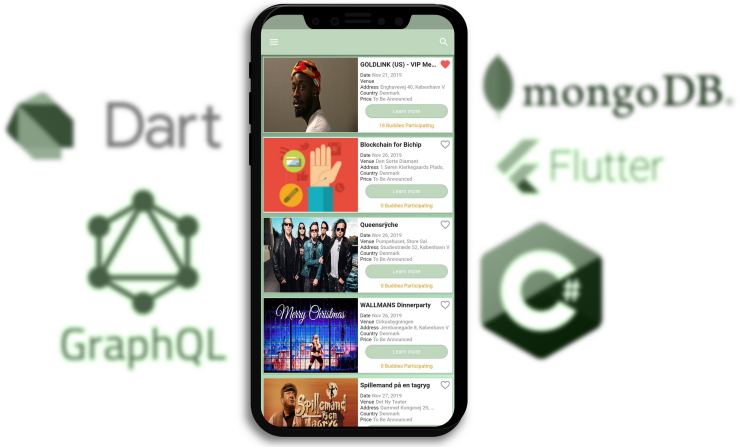
Distributed Event-Management Platform with Modern Technologies

Thomas Mascagni & Daniel Larsen



January 12, 2020

Kongens Lyngby

DTU

IT MINDS

DTU Compute
Department of Applied Mathematics and Computer Science

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk

Abstract

Denne afhandling omhandler udviklingen af en begivenheds-håndterings platform som har til formål at indsamle begivenhedsinformationer fra flere forskellige kilder såsom TicketMaster, Billetlugen, SeatGeek samt personlige begivenheder fra Facebook, Google mm. Dette skal synkroniseres op mod brugerens kalender, så man i et og samme sted kan orientere sig om der er mulighed for at deltage i den pågældende begivenhed. Alt dette vil blive præsenteret i én enkelt brugervenlig applikation ved navn *EventLink*.

Formålet med projektet er at designe og implementere det ovenstående distribuerede system via moderne udviklingsprincipper og teknologier. Mere specifikt betyder dette, at der vil blive benyttet iterative udviklingsprocessor såsom agil udvikling for at opnå det bedst mulige resultat.

Det endelige færdigbyggede system vil give slutbrugeren en nem og hurtig oplevelse når de skal søge efter begivenheder, i stedet for at skulle besøge flere forskellige hjemmesider eller applikationer med forskellige brugeroplevelser og design.

Summary

This dissertation addresses the development of an event-management platform, with the purpose of gathering event information from a diverse set of sources, such as TicketMaster, BilletLugen, SeatGeek as well as personal events from Facebook, Google, etc. This will be synchronized with the users calendar, such that they can decide whether they can attend the event in question. This will be presented in a single user-friendly application called *EventLink*.

The purpose of this thesis is to design and implement the above distributed system via modern development principles and technologies. More specifically, this means that iterative development processes such as agile development will be used to achieve the best possible result.

The final system will give the end user an easy and quick experience when searching for events, rather than having to visit several different websites or applications with different user experiences and designs.

Preface

This thesis was prepared at DTU Compute in fulfillment of the requirements for acquiring the B.Eng degree.
The thesis deals with the analysis, design, implementation and test of a software system.

Lyngby, January 12-2020

Thomas Mascagni

Daniel Larsen

Acknowledgements

We would like to thank our DTU advisor Daniel Kolditz Rubin-Grøn, for agreeing to have numerous meetings with us and coming with constructive feedback. Additionally, we would like to thank our external advisor from IT Minds, Martin Johannesson, who have been a great help with answering all of our questions. Finally, we would like to thank Thomas's cat Gubben, who have been a big emotional support during the whole project.

Contents

Chapter 1

Introduction

Since the smartphone was introduced in the 2000s, it has since been a common gadget for most people. Due to the vast variety of information from these, a persons attention span has decreased, gone from 12 seconds to approximately 8 seconds in 2018. [Wor18]

A study in the usage of applications shows that a general smartphone user uses approximately 9 applications a day and approximately uses 30 different applications a month. There is currently 2.47 million applications to choose between on the various app stores. [Per17] [Cle19]

As the lives of individuals gets busier and filled up with more errands, these problems will increase rapidly in magnitude. It is not a problem that a person uses many applications per say, but the fact that every application will help decrease the attention span as well as reduce time for other activities is a problem. For some people, this might have an affect on their well-being in day to day life, as it could potentially increase their level of stress.

In this thesis, a software system and a user interface will be developed, together called EventLink. EventLink is a system for managing events with the user in mind, such that they will use minimal time on their devices. This will be accomplished through a well-defined user experience as well as careful choice of technology. The aim of EventLink is to give a responsive, simple and intuitive

user experience. This way, EventLink will be an application that helps solve those problems.

In collaboration with IT Minds, the idea behind EventLink was defined as a single place to find and attend events. The idea is to assemble data from a variety of sources like Ticketmaster, Eventful etc., such that a diverse set of events will be available for users. In a perfect scenario, EventLink would be the only place a person would need to look for any type of event.

The goal is to make EventLink a realization satisfying the ideas above, as a step to move user experience in a different direction, such that those issues will be of a smaller concern in the future. EventLink should be an example of an application that does not have to be used frequently, but yet gives a higher sense of satisfaction than other applications. It should not be necessary for users to use the application for longer periods of time, due to its intuitive nature.

The thesis starts out with an introduction of IT Minds, and afterwards an in-depth development analysis that will include research of the system. Next the design phase will be described, which will refine the knowledge gained from the analysis phase. Finally, the implementation phase will be described as the system is built in practice. Then, a chapter regarding test of the system will take place and at last the project as a whole will be concluded upon with a result and conclusion chapter.

Chapter 2

IT Minds

IT Minds is a software solutions consultancy based in Copenhagen, Denmark. IT Minds offers a wide range of solutions, including web and mobile applications, machine learning, artificial intelligence, internet of things and much more. IT Minds has worked with and developed projects for many substantial Danish companies, including Jyske Bank, Oticon, Vestas and more. The image of IT Minds is seen as a young and fresh consultancy company, which thrives to be the best in their business.

IT Minds is known to be willing to help university students with their projects if they have an idea they want to attempt to develop further. For this reason, as well as a seemingly good culture fit, IT Minds was asked if they would be interested in helping developing and forming EventLink. Luckily, IT Minds were interested in the idea and a partnership was then formed.

IT Minds will provide an employee called Martin Johannesson as the external advisor for this project. Martin is a senior software developer at IT Minds and has been with the company for close to 4 years. Additionally, Martin holds a degree in software development from the IT University of Copenhagen.

Martin will primarily be used for guidance during the analysis, design and implementation phases. [Min19]

Chapter 3

Development Analysis

This chapter will describe the analytical aspects and mindset behind EventLink. The analysis will give insight into the development methodologies used, requirements engineering as well as use case and technology research. The domain of the system will modeled using a variety of UML diagrams. Finally, there will be created a MVP to define crucial system functionality together with an initial user interface prototype.

3.1 Methodology

For the development of EventLink, a lean and concise development methodology is desired. For the sole purpose of simplicity, the methodology should only include relevant artifacts, since it is desired to use time on development and the quality of software, instead of unnecessary artifacts.
With the above in mind, Unified process, Extreme programming and the Waterfall model have been considered. It was chosen to carefully select relevant artifacts from those, such that only artifacts that are relevant to this project are to be used.

The following is a list of the chosen artifacts that will assemble the development methodology.

Iterative development A key point in Unified process is to work in an iterative manner. Iterative development will be used to work in small sprints with a specific goal in mind to achieve.

Pair programming In Extreme programming, a common artifact is pair programming. Since only two developers will be working on EventLink, pair programming will be used to help produce better communication and higher quality software.

Code review As another artifact from Extreme programming, code reviews will be used to ensure mutual agreement of developed functionality.

Testing To make sure that requirements are satisfied, extensive testing will be done.

Iteration planning An artifact to help keep a sane overview over the development is iteration planning. Together with iterative development, iteration planning will be used to manage milestones and deadlines.

Sequential development To ensure that the project will not end up consisting of an infinite amount of iterations, sequential development will be used in some form to freeze development of milestones.

Taking the artifacts into account, the overall development methodology can be described vaguely as an iterative approach to the waterfall model. The general idea is to make an iteration plan with milestones, and when a milestone is completed, it will be frozen such that it will only be changed if absolutely necessary. During the iterations pair programming, code reviews and testing will be used extensively. A very early version of the milestone and iteration plan can be seen on Appendix: B.1.

3.2 Use cases

Use cases will be used to represent user and system interaction with EventLink. The actor descriptions will describe the type of users that will interact with the system, and user stories will characterize desired user functionality. A use case diagram will show core use cases and those will be described as well.

3.2.1 Actor descriptions

EventLink can be used by many different kinds of individuals, but in practice only two distinct actors will interact with the system. The primary actor is a regular person that interacts with the user interface and uses its functionality. Secondary actors will be external systems that are required to communicate with EventLink. An example of a primary actor and secondary actor follows.

Primary actor An adventurous individual who enjoys the thrill of participating in concerts and other kinds of events.

Secondary actor Systems that provide event information.

In the following user stories and use cases, the primary actor will be known as *User* and the secondary actor will be known as *Event Provider*.

3.2.2 User stories

The user stories have partially been created by asking potential users what functionality they saw fitting for the system, and the other part have been found during brainstorming sessions. Their primary goal is to gather information regarding desired user functionality. The presented stories are only a subset of all the user stories created and have been deemed to be the most relevant to carry on with. The following template will be used to write the stories:

As a <user> I want to <do something> so that <I can accomplish goal>.

The user stories can be seen on Table: 3.1.

Id	User stories
01	As a user I want to be able to sign up so that I have my information stored the next time I am using the system.
02	As a user I want to be able to find events so that I can go together with my friends.
03	As a user I want to able to see which of my friends participate in an event that I like, so we can go together.
04	As a user I want to be able to purchase tickets through the system so that I don't have to exit it.
05	As a user I want to be able to get recommendations on events so that I can experience more of what I like and try something new as well.
06	As a user I want the system to be an intuitive experience, so that no misunderstandings occur.
07	As a user I want the system to present relevant information, so that I spend as little time as possible using it.
08	As a user I want to be able to follow other users, so that I can see which events they are participating in.
09	As a user I want to be able to follow an event, so that I get notified if there are any changes to it.
10	As a user I want to be able to opt out of notifications, so that I do not get notified anymore.
11	As a user I want to be able to have my user information deleted, so that my data is not stored if I won't use the system again.

Table 3.1: User stories

3.2.3 Use case diagram

The use case diagram will be crafted from the actor descriptions and user stories. The diagram will be made such that it represents a few core use cases that will be used throughout the thesis. These are shown on Figure: 3.1.

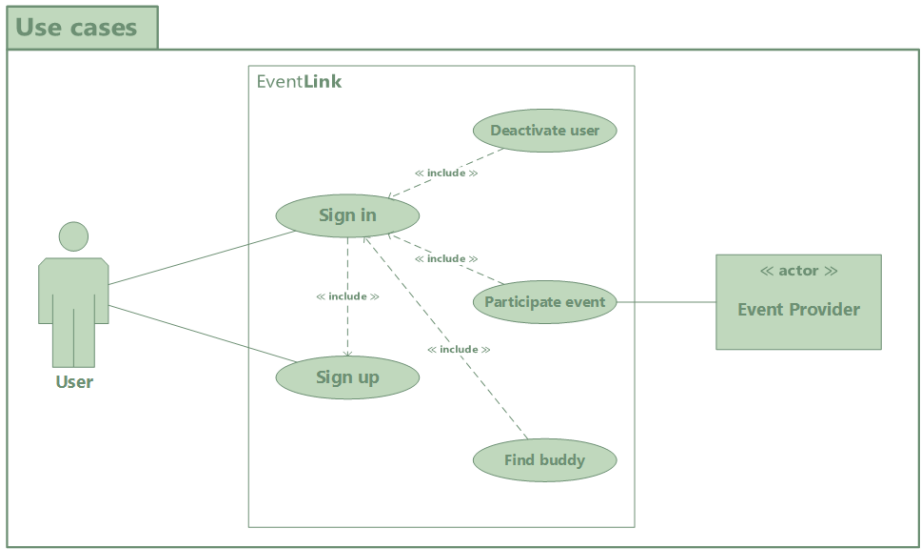


Figure 3.1: Use case diagram

The use case diagram presents the two actors along with five use cases, *Sign in*, *Sign up*, *Deactivate user*, *Participate event* and *Find buddy*.

Each use case is a specific scenario that the user can perform. For example, the user can both execute *Sign in* and *Participate event*. Use case dependencies are shown with include association arrows. For example, *Participate event* requires that *Sign in* has been performed. This means that the user can only participate in an event if the user is signed in.
As for the two actors, *User* is shown to have association with *Sign in* and *Sign up*. This is because the user either signs in or signs up (implicitly signing in) in order to gain access to the other use cases. The secondary actor is also shown to have an association with *Participate event*. This is because they provide the information for the events, meaning that the user whom is participating in an event is able to do this because of the data given by *Event Provider*.
These use cases are deemed to be core functionality. *Deactivate user* is seen as core functionality, due to the peace of mind for an user, that they can always

deactivate their account if need be.

3.2.4 Use case descriptions

Two types of use cases will be made, fully dressed and brief use cases. It has been decided to create a single fully dressed use case and leave the rest as brief use cases. The *Sign up* use case has been made fully dressed because this is the first interaction that every user must go through at least once in order to be able to use EventLink. The rest of the use cases can be found in Appendix B.1.2.

The description for the *Sign up* use case can be seen on Table: 3.2. This use case is a user goal because signing up is something that the user wants to achieve. Then the stakeholders are described, being the user and the company. Furthermore, the use case also describes the preconditions for the use case to be able to be performed. In this case, the user has to have access to EventLink in order to be able to sign up in the system. The post-conditions are that the user will be signed up and signed in after the sign up procedure is done. This has been decided for the sake of user experience, such that the user does not have to type in their credentials twice in a row. Additionally, the main success scenario explains the steps that the user will have to go through in the sign up process. The alternative flow explains the alternative decisions that the user will be able to take in a given step. For example, the user can decide to sign in instead of sign up at the second step in the main success scenario.

The brief use cases do not contain main success scenarios or alternative flows. It is also important to mention, that eventual errors in the alternative flows are not considered in the descriptions.

Id	UC01
Use case name	Sign up
Scope	EventLink
Level	User goal
Primary actor	User
Description	A user signs up in the system.
Stakeholders and interests	**User:** Wants an intuitive sign up method, so they can begin to use the system for finding events. **Company:** Wants the users to sign up, and thereby, hopefully get a frequent usage of the system. Also wants to provide a smooth and seamless user experience, so that users stay.
Preconditions	The user has access to the system and is ready to sign up.
Postconditions	The user has been signed up in the system, is now signed in as well and ready to use all of the functionality available.
Main success scenario	1. EventLink is opened by the user. 2. User chooses to sign up. 3. EventLink navigates to sign up screen. 4. User enters valid sign up information and signs up. 5. EventLink stores the users sign up information. 6. EventLink shows a message to the user that they are signed up. 7. EventLink navigates the user to the main screen of the system.
Alternative flow	2a. User chooses to sign in. 4a. User enters invalid information. 4b. System writes error message to user, and prompts them to try again.

Table 3.2: Fully dressed use case - Sign up

3.3 Requirements specification

The requirements specification will be made with the foundation of the actors, user stories and uses cases from Section: 3.2. FURPS+ is a model for categorizing and organizing requirements and will be used to classify the requirements. Two templates will be used to create the requirements; "*The system must...*" and "*A user should be able to...*". These templates will primarily be used to describe non-interactive and interactive functionality respectively. On Table: 3.3 the requirements specification is shown.

The id of a requirement indicates which category of FURPS+ the requirement belongs to. *F* is functionality, *U* is usability, *R* is reliability, *P* is performance, *S* is supportability, *I* is implementation and *L* is Legal. [Lar04]

Considering the requirement with id **F01**, it is a functional requirement, that states that the system has to be able to create a user, not necessarily with the interaction from an actual user of the system. This is categorized as a functional requirement since the act of creating a user is a function of the system.
The requirement with id **U03**, which is a usability requirement, states that the system should be intuitive to interact with for the user. This is a usability requirement since it has to do with the usability of the system.
Finally, the requirement with id **L01** states the system must comply with the regulations of GDPR. This is a legal requirement as it has to do with legal responsibilities.

Id	Description
F01	The system must be able to create a user.
F02	The system must be able to reset a users password.
F03	The system must be able to sign in a user.
F04	The system must be able to sign out a user.
F05	The system must be able to deactivate a user.
F06	The system must be able to store a users information.
F07	The system must collect event data from various event providers.
F08	The system must be able to present events to the user.
F09	The system must be able to show user profiles.
F10	A user should be able to search for specific events.
F11	A user should be able to participate in various events.
F12	A user should be able to remove participation from an event.
F13	A user should be able to favorite various events.
F14	A user should be able to unfavorite an event.
F15	A user should be able to search for specific users.
F16	A user should be able to see whom they are friends with.

Id	Description
F17	A user should be able add another user as a friend.
F18	A user should be able to remove a friend.
F19	A user should be able to purchase tickets for an event.
F20	A user should be able to see their own order history.
F21	A user should be able to see which events another user participates in.
F22	A user should be able to see which friends participates in an event.
F23	A user should be able to update their personal information.
F24	A user should be able to update their picture.
F25	A user should be able to change their e-mail.
F26	A user should be able to change their password.
U01	The system must have a responsive design.
U02	The system must show convenient error messages to the user.
U03	The system must be intuitive for the user to operate.
U04	A user should be able contact system support.
R01	The system must have an up-time of at least 98% of a week.
P01	The system must avoid using unnecessary resources.
P02	The system must not stall for more than 2 seconds at any given moment.
S01	The system must support multiple signed in users.
I01	The system must be able to operate on multiple platforms.
L01	The system must comply to GDPR.

Table 3.3: Requirements specification

3.4 System architecture

During this section, a potential system architecture is discovered. This will be done through the creation of a draft system model, which will describe system interaction. Additionally, a domain model will be made. Then system sequence diagrams will made to model interaction flows. Together, these elements will make a basis for an analysis of the system architecture.

3.4.1 System draft diagram

Since a solid foundation of knowledge regarding desired user functionality has been established, it is now possible to start thinking about what a potential system architecture could look like. A system draft diagram has been made for

this purpose and can be seen on Figure: 3.2. It is important to note that this diagram is not an official UML diagram, but instead custom made with some elements from UML.

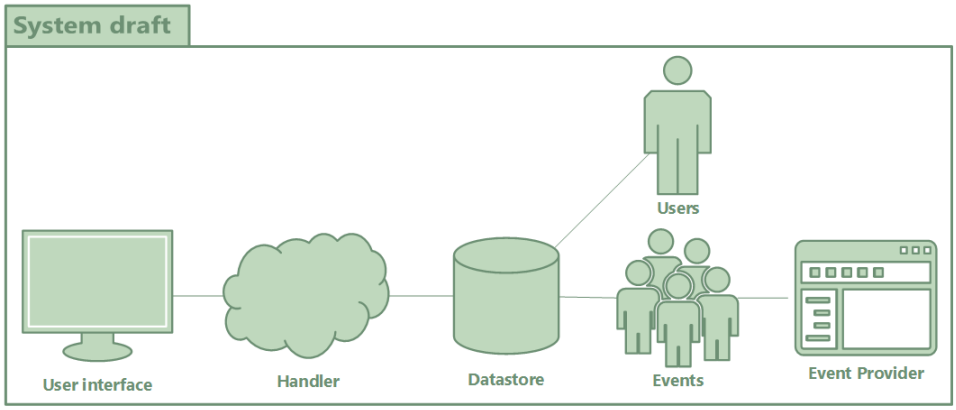


Figure 3.2: System draft diagram

The system consists of three main parts. The user interface, or GUI, that the users can interact with the system through. A handler or backend system that can handle communication to and from the user interface. A datastore that persistently stores user and event data. Given this system architecture, a system can be made to fulfill all of the requirements and wanted functionality. Keep in mind that this is a rough draft of the system and might change.

3.4.2 Domain model

The domain model will be inspired by the previous analysis in this chapter. The domain model can be seen on Figure: 3.3.

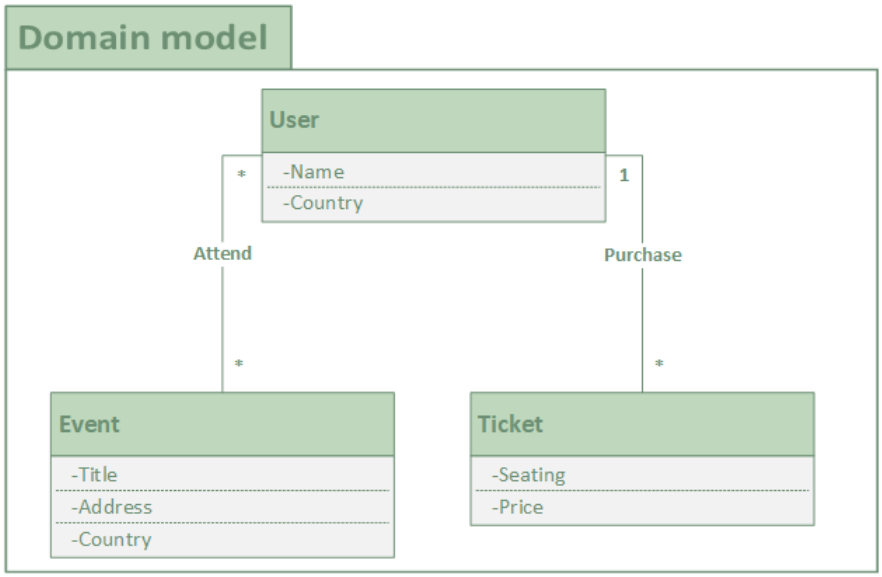


Figure 3.3: Domain model

The model domain has been narrowed down to contain three models. The *User*, *Event* and *Ticket*. Figure: 3.3 presents the models with some key attributes and relations. It is shown that a user have a name and a country. These have been deemed to be important knowledge for the system, so it can refer correctly to the person. If a person lives in the United Kingdom, they do not necessarily want to see events from Denmark. The user can attend many events, and an event can contain many users, a many to many relationship.
The necessary data for an event would be the title, this often tells which artist that performs, the address where the event takes place and the country.
A ticket is bound to a single user, but a user can purchase many tickets. It is important for a user to know their placement at an event if they bought a ticket, therefore the ticket contains seating information. The price is also a key factor and has therefore been added to the ticket model as well. All three models will be of significant importance when the design and implementation phases are reached.

3.4.3 System sequence diagram

System sequences will be used to characterize success scenarios of the use cases described in Section: 3.2.3 by modeling their communication flow. The sequences will be designed to accommodate for the problems described in the introduction. It is important to note that the messages that are sent between the user and the system are at this point regular request-response messages, as it is not known how the messages will react in practice. This section will contain two sequence diagrams for the *Sign in* and *Participate event* use cases. The rest can be found in Appendix: B.1.3.

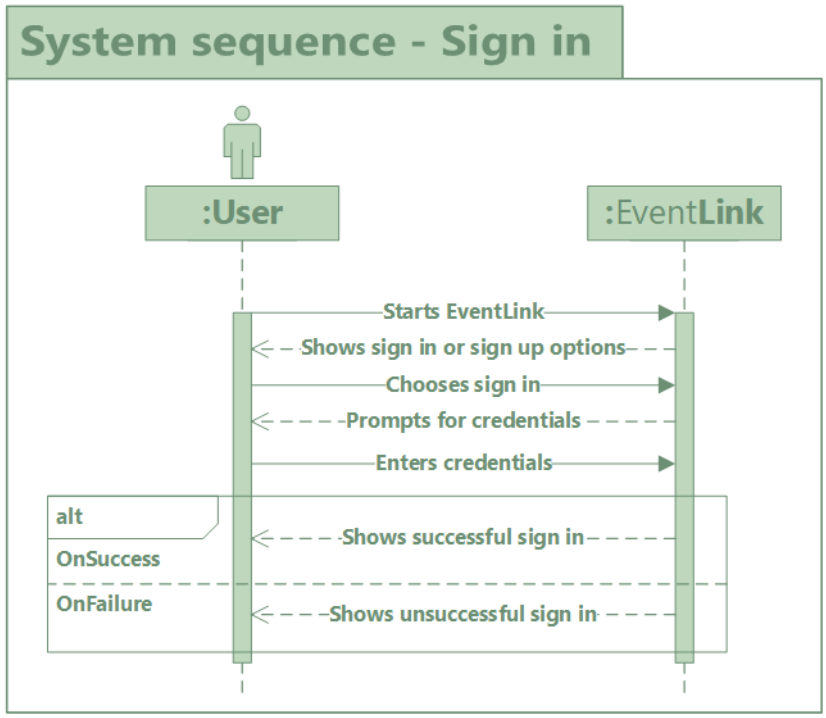


Figure 3.4: System sequence diagram - Sign in

The sequence diagram on Figure: 3.4 describes a flow for signing in to EventLink. This scenario can only be performed if the user is signed up in the system as shown previously. A user is prompted with sign in and sign up options at system startup. Then the user chooses sign in and enters the necessary credentials, after they are being prompted with a sign in form. If the user has entered valid credentials, then user is allowed to access other use cases that includes this one.

If the user fails to provide valid credentials, then the system should respond with an appropriate message.

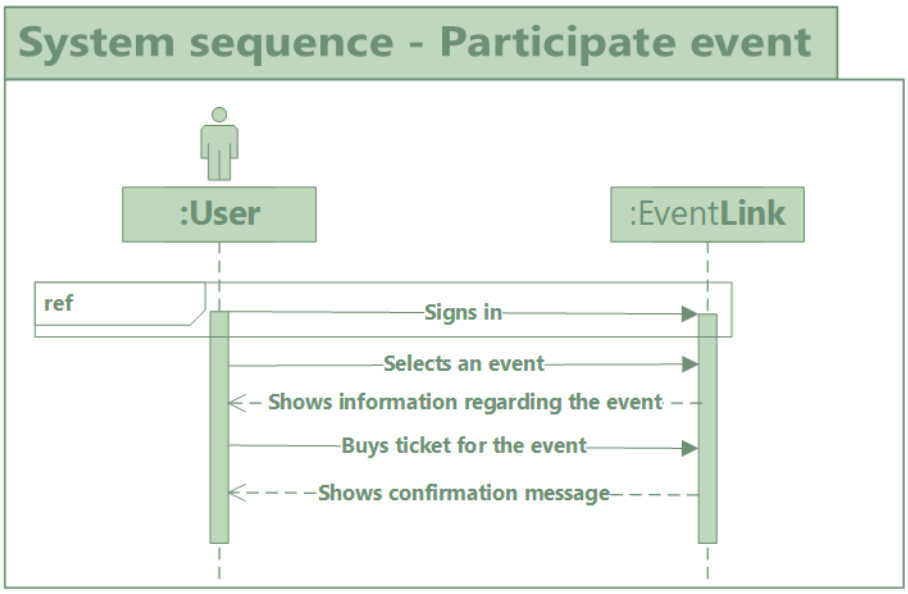


Figure 3.5: System sequence diagram - Participate event

The sequence diagram seen on Figure: 3.5 shows how a user participates in an event. This sequence has a reference block to the *Sign in* use case, meaning that before the user can participate in an event, they have to be signed in to the system. It is given, that the referenced sequence is successful for the user to be able to continue.
If the user successfully signs in, the user is prompted with a main screen where they are allowed to search for or select an event. Once they find an event, they can buy a ticket for the event and are thereby participating in it.
The *Participate event* use case is essential, as participating in events is the whole idea behind EventLink.

3.5 Technology

This section will dive down into the research behind which technologies that together would provide the best solution for EventLink. The section will discuss which kind of datastore, data collector, data communication and access control

is the most viable. This will include various comparisons between an array of technologies, such that every technology has been selected deliberately and with good reasoning.

3.5.1 Research

It is important to consider if any major constraints have been introduced. In the requirements specification on Table: 3.3, a requirement states that the system has to be able to operate on multiple platforms. This has to be taken into consideration when deciding which programming languages and frameworks to use, such that cross-platform support is achieved.
Taking in consideration that the system has to be cross-platform and responsive, the .NET Core Framework was quickly seen as a potential candidate to build the system upon. .NET Core contains a variety of libraries to create APIs and other useful software components. Similar frameworks were considered briefly, but in the end it was chosen to go with .NET Core for the backend system.

It was heavily considered to build the system using functional programming. Functional programming has numerous advantages over more regular programming paradigms such as OOP. This includes easier debugging, less source code, more readable code and much more. Since it was decided to use .NET Core, this leaves two options for programming languages.

F# A popular functional programming language maintained by Microsoft is F#. F# is closely related to C# in the sense that both languages have integration's for Microsoft's .NET Frameworks. Therefore, if functional programming was to be used, F# would be the ideal choice.

C# Knowing that .NET Core is going to be used, C# would be an obvious choice to use as a programming language. C# is the primary programming language to use with the .NET eco-system and it uses the OOP paradigm.

Due to limited knowledge and experience with functional programming, it was decided that going with a more familiar language like C# would be the better choice. Using functional programming and F# would be possible, but it would also introduce a bottleneck in the development, as substantial amounts of time would be required to get familiar with the language and functional programming. Therefore, it was decided that the system will be based on .NET Core and C#.

3.5.2 Datastore

To find a suitable datastore for the system, a few potential methods of storing data will be investigated and evaluated. The datastore candidates have been trimmed down to three different database systems, MSSQL, MySQL and MongoDB. These have been chosen mainly due to personal experience, performance and reliability.

MSSQL A popular database system is MSSQL, also called Microsoft SQL Server. MSSQL is a relational database that uses the T-SQL query language, which is an extension to SQL. SQL is inflexible, meaning that there is a need to pre-define the data types that will be stored. Additionally, SQL databases scale vertically, which means that you would usually upgrade its hardware components to accommodate for larger server load. Finally, MSSQL is very well integrated within .NET.

MySQL Another database system that uses SQL is MySQL. MySQL is a relational database created by Oracle, and is very similar to MSSQL. MySQL does not have as good of a integration with C# as MSSQL does, but it is entirely possible for the two to work together. Since MySQL uses SQL, the previous comments regarding scalability and pre-defined data types are applicable here as well.

MongoDB A newer and divergent way of storing data is using MongoDB. MongoDB is a document-oriented database system that uses a non-relational approach to store data. MongoDB uses NoSQL, where the user creates collections and documents instead of tables. NoSQL is more flexible compared to SQL due to the fact that there is no need to pre-define what kind of data resides in a document. Additionally, NoSQL is horizontally scalable, which means that it is designed to run on multiple servers, which in general is less expensive than vertically scaling solutions.

In the end, it was chosen to go with MongoDB as the datastore. This was mainly due its flexibility of not having to pre-define data types, as data sources will be multiple and are still unknown. This could potentially be a problem using relational databases. Since the data is not pre-defined, it can't be validated by the database either. This should however only be a minor issue. Adding to that, the horizontal scalability will be an economical advantage. [Mon18]

3.5.3 Data collection

It is at this moment not yet known which event providers are going to be used, so knowledge regarding potential data extraction is limited. It is desired to have a generic data collection method, such that minimum maintenance is required when wanting to support a new event provider. Achieving a generic way of collecting data from arbitrary data sources is going to be a difficult task, so compromises must be made along the way. The two considered methods for data collection are Scrapy and a custom solution.

Scrapy A widely used framework for web scraping and crawling is Scrapy. Scrapy uses something called spiders. These spiders are self contained crawlers, that attaches to a web page to crawl for data. Even though Scrapy is originally designed for web crawling, it also allows the extraction of data through APIs. [Scr19]

Custom solution Another consideration was to not use a framework for data extraction, but to implement a system from scratch. This solution could be fitted to perfectly match data collection from arbitrary event providers. A major disadvantage for this is that it requires more implementation and maintenance, but offers more flexibility regarding the way data is collected and parsed.

The custom solution ended up being chosen. This is because of the flexibility a custom system provides. Additionally, for starting out, it was decided to use event providers whom provide an API for data extraction, primarily for ease of implementation.

3.5.4 Data communication

The data communication component will be created as an API. It is decided that the potential candidates are SOAP, REST and GraphQL.

SOAP A very common way to develop APIs is using SOAP. SOAP is a messaging protocol designed to make it possible for different systems to communicate through HTTP. These messages are defined in high level XML.

REST Another way to implement APIs is using REST. REST is not a software framework, but an architectural style of communicating through the web. REST can be seen as a more flexible version of SOAP, as it can be

implemented in many different ways. Like SOAP, REST uses HTTP as its communication protocol.

GraphQL A third and more modern approach to API development is GraphQL. GraphQL is a data and manipulation language for APIs, made by Facebook. Like REST, GraphQL is not a framework, but instead an architectural style for data communication. GraphQL is different from SOAP and REST in the sense that it works much like a SQL database. In GraphQL, you query the data you want directly through the API resource. The main building blocks of a GraphQL API are queries and mutations, and in addition to that, JSON is used for its query syntax. [Fou19]

In the end, it was chosen to use GraphQL for the data communication. SOAP uses XML as its data transportation format, so it would be required to parse it into JSON before being able to use it in the MongoDB datastore. This would add extra overhead in the system. REST is a good alternative to GraphQL, but due to the fact that GraphQL contains more features, there was no reason to choose REST.

3.5.5 Access control

It was decided upon to use a GraphQL API for data communication. This API has to be secured such that only users of EventLink will be able to query and mutate data.
Due to limited knowledge of access control technologies, IT Minds was consulted regarding this topic. The external advisor Martin Johannsson, advised to use the .NET Identity Framework. After some research and better look through this framework, it was concluded that it would not be possible to use, due to the fact that it does not support MongoDB. Therefore, further research was required.
A few ideas as to what could help accomplish this kind of access control include the usage of Identity server and JWT. Additionally, OAuth2 is also considered due to its single sign on functions. It should be mentioned, to avoid confusion, that Identity server and the .NET Identity Framework are two separate entities.

Identity server A package solution for access control is Identity server. Identity server is a identity and access control solution made specifically for the .NET Frameworks. It can handle different services such as single sign on, identity management and API security. [Ide19]

JWT A simple way of authorization can be achieved by using JWT. JWT is an open standard defined in RFC 7519, which defines a compact and self-

contained way for securely transmitting information between parties as a JSON object. JWT primarily provide means of authorization. [Aut19]

OAuth2 Another and slightly different method for authorization is OAuth2. OAuth2 is a open protocol which allows secure authorization for mobile, desktops and web systems. It is a industry-standard protocol for authorization and is mostly known for its single sign on functionality for Google and Facebook etc. OAuth2 only provides authorization functionality. [Par19]

As for the final decision of what technology to use, both JWT and OAuth2 has been chosen. This is because JWT will provide authorization that is simple to implement for API security, together with OAuth2 being able to provide single sign on for social media accounts. This would make it possible to get event information from those accounts and use them in the system. Identity server was not used as it would take a substantial amount of time to get experienced with the solution.

3.5.6 User interface

It has been decided that the interactivity to the system should happen through a mobile application. This makes it conveniently accessible since most people have their smartphone on them throughout the day. Additionally, it will allow for the creation of a simple user experience on a small display.
As one of the ideas behind EventLink is to reach as many users as possible, it would make sense to maximize the amount of supported mobile platforms. It is important to note, that this requirement is different from **I01** stated in Table: 3.3. **I01** is regarding the implementation of the backend system, whereas this requirement is regarding the mobile application. It was decided to choose upon two different frameworks, React native and Flutter.

React native One way of developing cross-platform mobile application is using React native. React native is a framework made by Facebook created specifically for cross-platform development. It is used to create applications for Android, iOS, web and more. React native is a written with JavaScript, since it uses the Reactjs library.

Flutter A second tool for developing cross-platform applications would be Flutter. Flutter is Google's proposal for a toolkit to create cross-platform applications for a wide variety of platforms. Flutter was launched in 2017, so it is still a relatively new framework. It is primarily being developed at

Google's headquarters in Denmark. Flutter uses a programming language called Dart, which is also created by Google. In many ways, Flutter is very similar to React native.

Considering the scope of the project, choosing between React native and Flutter is not of significant concern. Both frameworks would be able to satisfy the needs of a client application as nothing other than cross-platform support is a must. Since there is limited experience with both JavaScript and Reactjs, it was decided to go with Flutter. Dart is more similar to languages that the developers have experience with. Additionally, Flutter is a relatively new framework that is in constant development, so minor issues might occur. Regardless, whether the application would be created in React native or Flutter, it would be able to contain the exact same functionality, and more importantly, run on the same platforms.

3.5.7 Result

As the final point in the technology section, all of the results will be summarized briefly. In Table: 3.4, an overview of the chosen technologies is shown.

Scope	Technology
Programming language	C#
Datastore	MongoDB
Data collection	Custom solution
Data communication	GraphQL
Access control	JWT, OAuth2
User interface	Flutter

Table 3.4: Technology results

Taking a look at the table, it is now possible to say that the backend system will be created using C#. Specifically, a GraphQL API and an integration layer to MongoDB will be developed, as well as functionality to support both JWT and OAuth2. Finally, the user interface of EventLink will be a mobile application created with Flutter that will communicate with the GraphQL API.

3.6 Risk analysis

A risk analysis will be made to give an overview of the specific risks that are likely (and perhaps also unlikely) to occur during this project. To state how significant a given risk is, each risk will be given a score, such that the higher the score is, the worse the impact would be on the project. The score will be evaluated by a risk matrix, which calculates the risk out from these criteria. The risk matrix can be found on Appendix: B.5. In general, the risks that are in the green area are not very significant, whereas the ones in the red area are deemed to be significant.

Id	Description	Score
R01	Illness	3
R02	Jobs	5
R03	Not reaching deadlines	6
R04	Research	4
R05	Lack of documentation	5
R06	Insufficient or poor planning	6
R07	Personal activities	3
R08	Exams	5
R09	Other educational responsibilities	4
R10	Internal communication	2

Table 3.5: Risk analysis

To further describe the reasoning behind the risks and their score, the first three risks will be assessed. The risks should largely be self-explanatory, but for completeness sake these will be further described.

Illness A common risk is illness. Illness is an obvious choice for a risk, since it is very common and can occur with a varying level of impact. Since there is always a chance that either one of the developers gets ill or injured, this risk has been give a score of 3. There is a larger chance that a person will get the flu rather than getting hospitalized, so the general impact is not seen to be high. The score of 3 is evaluated from a 26-50% chance of occurrence with a moderate impact.

Jobs The amount of time spent on working on the project is critical. Therefore other important activities, such as jobs, are a risk to its quality. One of the developers will be working part time during most of the project and thesis, which means that less time will be spent on actually working on the

project than what could have been possible. The jobs risk has been given a score of 5, since it has always been known that one of the developers would be working part time. The score was given by a 76-100% chance of happening with a moderate impact level.

Not reaching deadlines Not being able to reach deadlines in time is an important risk to consider as well. As explained previously in this chapter, milestones and iterations will be set for various goals. If those are not reached by their deadline, it will extend the time of other activities and deadlines as well, which in the end will result in a domino effect. Since there will be worked with a lot of unknown technologies, further research might be needed for those. Therefore this risk has been given a score of 6. This is due to a 51-75% chance of happening together with a critical impact.

3.7 Minimum viable product

Since the various requirements, user desired functions and technologies are now known, it is possible to create a MVP. The MVP is created to make sure that the work put in to the project, is put in at the right places. The MVP is similar to the requirements specification in a sense, but is more detailed and specific for each component. It is also important to mention, that the functionality described in the MVPs is not strictly user functionality, but also internal system functionality. The five key components of the system are considered.

Datastore The component that will communicate with the MongoDB database.

Data collector The data collection component.

Data communication The data communication GraphQL API.

Access control The access control component created with JWT and OAuth2.

User interface The mobile application created with Flutter.

There has been created a separate MVP for each of these components. These will be presented below, accompanied by a ranking. The lower the rank, the more important the given functionality is for the system to work. All ranks marked with a diamond ($\diamond$) must be implemented in order for the system to be able to function. All others without the diamond is key functionality, but not a strict requirement for the system to work. For example, on Table: 3.6 it can

be seen that all four elements are required for the system to be able to function and seen as important functionality.

Rank	Description
◇ **1**	The system must contain a collection for events.
◇ **1**	The system must contain a collection for users.
◇ **1**	The system must contain a collection for payments.
◇ **1**	The system must contain CRUD operations for the above collections.

Table 3.6: Minimum viable product - Datastore

Rank	Description
◇ **1**	The system must be able to fetch data from an API.
◇ **1**	The system must be able to parse the fetched data.
◇ **1**	The system must be able to communicate with the datastore.
◇ **1**	The system must be able to run with a given time interval.

Table 3.8: Minimum viable product - Data collector

Rank	Description
◇ **1**	The system must contain CRUD operations for events.
◇ **1**	The system must contain CRUD operations for users.
2	The system must be able to let a user participate in an event.
2	The system must be able to let a user remove participation in an event.
2	The system must be able to let a user add another user as a friend.
2	The system must be able to let a user remove another user as a friend.
2	The system must be able to let a user search for other users.
2	The system must be able to let a user search for events.
3	The system must be able to let a user purchase a ticket for an event.
3	The system must be able to let a user subscribe to an event.
3	The system must be able to let a user unsubscribe to an event.

Table 3.9: Minimum viable product - Data communication

Rank	Description
◇ **1**	The system must use JWT to secure the data communication API.
◇ **1**	The system must be able to handle e-mail sign in.

Rank	Description
◇ **1**	The system must be able to reset an users password.
2	The system must be able to handle OAuth2 Facebook sign in.
2	The system must be able to handle OAuth2 Google sign in.
3	The system must be able to handle OAuth2 Apple sign in.

Table 3.10: Minimum viable product - Access control

Rank	Description
◇ **1**	The system must contain a main screen displaying events.
◇ **1**	The system must contain a login screen displaying login options.
◇ **1**	The system must contain a user screen displaying user information.
◇ **1**	The system must contain a button for signing in using e-mail.
◇ **1**	The system must contain a button for signing up using e-mail.
◇ **1**	The system must contain a button for signing out.
2	The system must contain a search bar to search for events.
2	The system must contain a toggle for participating in events.
2	The system must contain a toggle for subscribing to events.
2	The system must contain a search bar to search for users.
2	The system must contain a button for adding a user as friend.
2	The system must contain a button for removing a user as friend.
2	The system must contain a button for signing in using Facebook OAuth2.
2	The system must contain a button for signing in using Google OAuth2.
2	The system must contain a button for deactivating the user.
3	The system must contain a button for buying a ticket to an event.
3	The system must contain a button for signing in using Apple OAuth2.

Table 3.11: Minimum viable product - User interface

The MVPs will be administered in GitKraken Glo, where more information can be found in Section: 7.1. Some of the MVP elements might seem vague, but that is intentional. At this point in time knowledge regarding the actual design of the system is very limited, so defining the MVPs slightly vague will make it easier to change. As mentioned earlier, the MVPs will primarily be used during the implementation phase.

3.8 User interface prototype

The idea with the user interface prototype is to be able to have a discussion with IT Minds regarding the potential user experience the user would have with it, as some kind of early user test. It is important to note that the prototype will not be made as a full-fledged prototype, but more as a graphical representation designed with the MVP in mind.
On Figure: 3.6, Figure: 3.7 and Figure: 3.8 prototypes for the login screen, main screen and user screen are shown respectively.

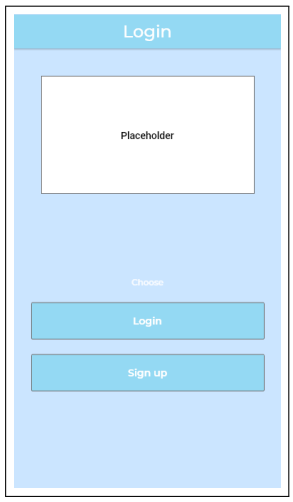


Figure 3.6: Prototype 1 Login screen

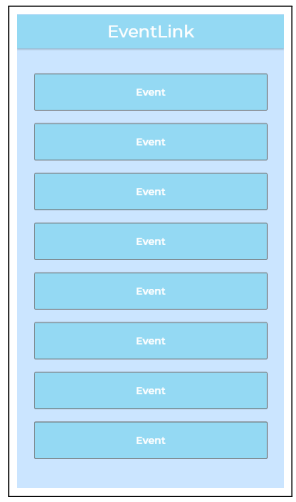


Figure 3.7: Prototype 1 Main screen

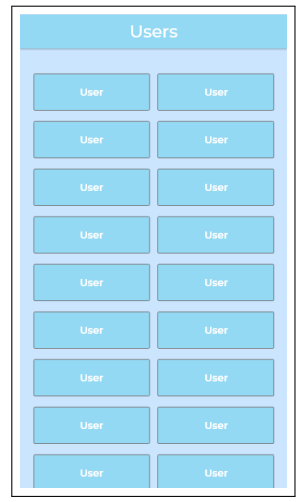


Figure 3.8: Prototype 1 User screen

Login screen The login screen is the initial screen the user will see when the application is opened. This screen will show a logo of EventLink in the placeholder section. Additionally, login and sign up buttons are available as required by the MVP. These buttons will most likely lead to their own separate screens.

Main screen The main screen of the application is thought to be a list displaying events. Each event will be a card structure showing title, address, time, etc. Pressing on an event will lead to a screen with more information.

User screen The user screen is similar to the event screen, in the sense that is a list structure that shows users. The structure could be a profile picture or a card structure like in the case of event, showing relevant data. Pressing an user element will lead to a user profile screen.

3.9 Summary

In this chapter, a development methodology was settled upon, which included iterative development, pair programming among a few more, to establish an iterative yet sequential development method. Two actor descriptions were created, the user of the system and the event provider. These have been used to create a total of 11 user stories, where some of these have been wrapped up in a use case diagram. A fully dressed use case was made, together with a few brief ones.
Then a requirements specification was made with foundation in the use cases. A system draft was then made with the requirements in mind, in order to start mapping out the outlines of EventLink, and a variety of different diagrams were made to understand a possible system architecture.
In order to be able to provide the right solution, a thorough research was undergone in order to pick the technologies that should be used in the system. It was chosen to go with MongoDB, GraphQL, JWT and OAuth2, Flutter and a custom solution for gathering event data. Additionally, it was chosen to use .NET Core and C# to create the backend system.
A risk analysis was made in order to understand the risks that are likely to occur during the project. With the previous knowledge gained during the analysis phase, a MVP could be created along with a initial prototype for the Flutter application.

The analysis phase as whole, has been deemed to be successful, and the design phase can begin to expand further on the information gained in this phase.

Chapter 4

Design

This chapter will build upon the knowledge and experience gained during the previous development analysis chapter. More specifically, this chapter will focus on defining a more concrete architecture for the system, as well as further specifying how the components of the system will function. This will be done through an array of UML diagrams as well as additional research where necessary.
First, the system architecture will be further defined by redefining the system draft, together with a few other diagrams. Then, relevant design patterns will be discussed together with the application servers that the system will run on. After that required design aspects of the datastore, data collection, data communication, etc. will be examined.

4.1 System architecture

A key factor behind designing the system was to design it as modular as possible, therefore this will be the primary idea by the design of the components. In the sections to come, a redefined system architecture of EventLink will be presented. An analysis class diagram will be provided for each of the components as well as a solution for how each component could communicate with each other. Additionally, supporting diagrams will be made.

4.1.1 Modeling

During the analysis, a system draft diagram was made in Section: 3.4.1. This was the first look of how a potential architecture of the whole system could look. A refined version of this diagram has been made on Figure: 4.1.

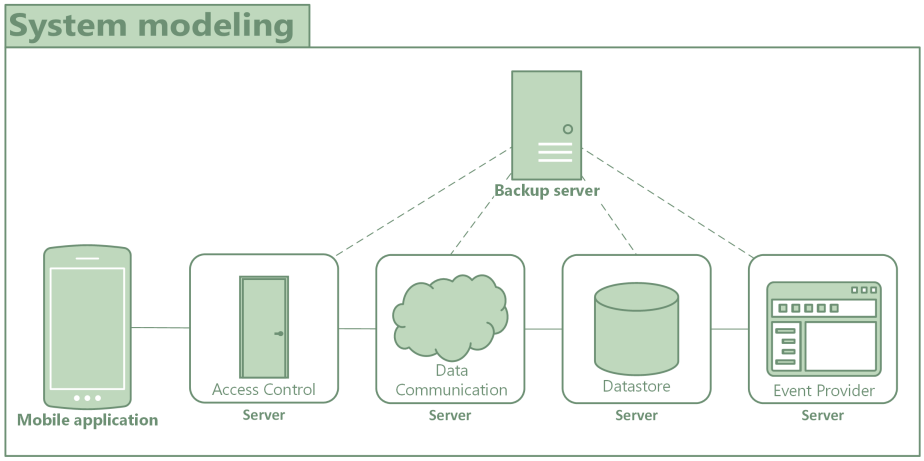


Figure 4.1: System modeling diagram

The components have been spread out on five distinct servers. This has been done primarily for load distribution concerns. Additionally, it will help avoid hard-dependencies between the components as they are on different servers. This will help with maintaining high cohesion and low coupling in the implementation as well. The previous handler is now the data communication component, and an additional server has been introduced.
It was discussed internally whether a backup server would be a good idea to

include. It was decided that it could be useful, for potentially creating some kind of hot-reload system, where, for example, if the access control server is unavailable, the backup server would boot the access control software and act as a temporary access control server. This is a potential use case, but not a final choice at this point in time.

4.1.2 Analysis class diagrams

A diagram presenting a high level overview over the system architecture has been made on Figure: 4.2. It is similar to the system modeling diagram, but adds additional information regarding component communication. Additionally, an analysis class diagram has been made for each of the components.
An important note regarding this section, is that both UML association and composition arrows have been used for the diagrams. Since this is still early in the design phase of the system, it is not yet quite known how strong of a bond the various classes will have. Therefore association are used for communication and unknown bonds, and composition is used to signify a dependency.

A new component introduced in Figure: 4.2 is the artifact of logging. It was discussed whether logging would be an useful feature to introduce into the system, and it was highly agreed upon that it would be a smart idea to include due to the ease of troubleshooting bugs and errors on the go. This will be explained more in depth later in the thesis.

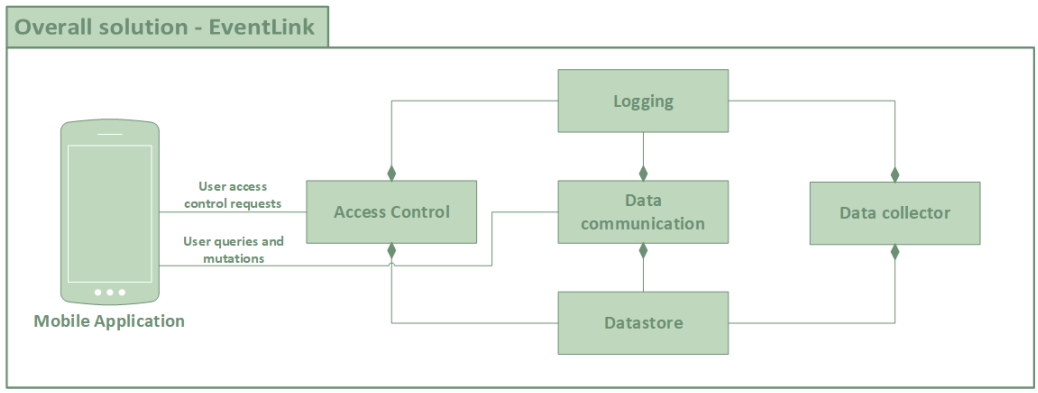


Figure 4.2: Overall solution diagram

On Figure: 4.2 the communication between the components is shown. For example, it is shown that a user can authorize themselves by communicating with

the access control component, and they can also send queries and mutations to the data communication API. The flow in this case goes from the mobile application to the data communication API, then from here to the datastore which will return some data. This will then be sent back to the mobile application where it can be displayed.
The access control component depends on the datastore in order to check whether the credentials entered by the user are valid, and the data collector depends on the datastore in order to update and insert event data. The design of the overall solution is based on natural component dependencies.

The first class diagram that will be introduced is for the datastore component and can be seen on Figure: 4.3. It is here important to note, that the diagram will show the business logic design for the datastore software component, and not the MongoDB database.

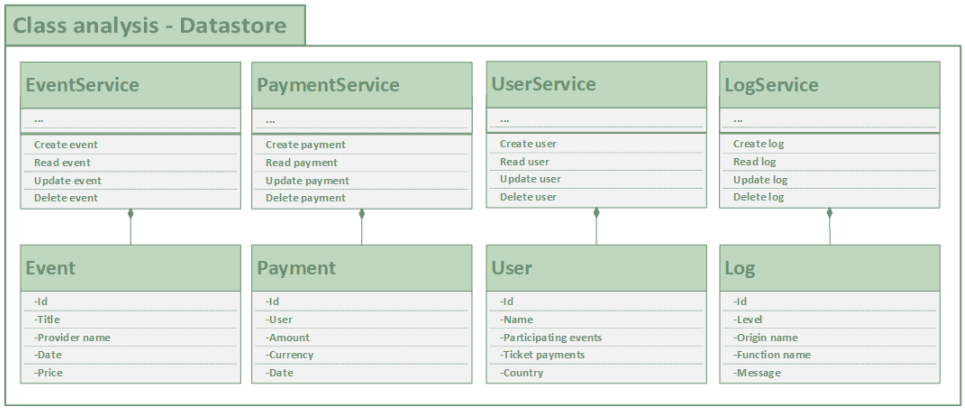


Figure 4.3: Analysis class diagram - Datastore

The idea behind the datastore is to implement it as a component that gives access to a variety of services. These services will be able to handle CRUD operations for each of the data models in the system. The reasoning behind this service architecture is explained in Section: 4.2. The data models are *Event*, *Payment*, *User* and *Log*. The *Payment* data model is a replacement for the previously mentioned Ticket in Section: 3.4.2. It was decided for the logging to be done through the datastore, which means that MongoDB will include an additional collection to store logs.

Considering the event data model, it can be seen that an event must have an unique id in order to make it uniquely identifiable. The event must also have a title, a name of the event provider it is gathered from, a date for the event and

the pricing of the tickets for the events. The other data models are similar, so these will not be explained.

The next diagram for the data collector can seen on Figure: 4.4. The data collector is a simple component with two functions, collecting data and parsing data.

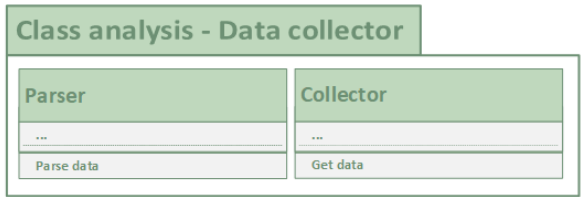


Figure 4.4: Analysis class diagram - Data collector

The reasoning behind this, is that event data will be collected from different event providers and these providers will have a variation in their data models and quality. Therefore, the collector has to be able to collect data from various providers, and the parser has to support parsing the respective data models. The event data will be parsed into a single unified data model that will be described later. The parser also has to ensure minimal data quality, for instance an arbitrary event object must contain a unique identifier, otherwise it is not valid.

The next diagram can be seen on Figure: 4.5. This is a simple diagram for the access control component, as currently only functionality for authorization has been designed. Authentication is believed to the handled by the datastore implementation.

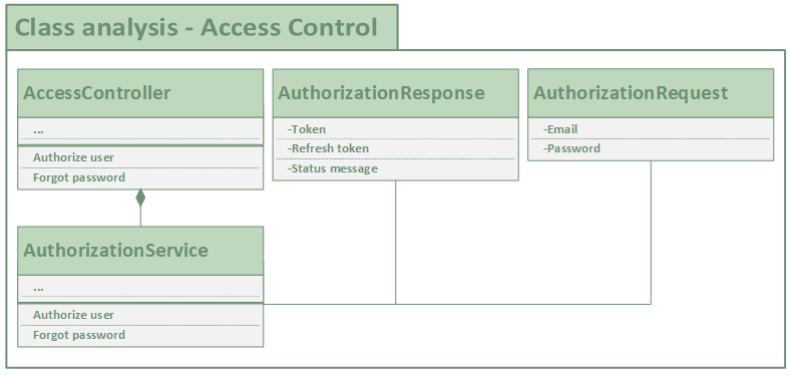


Figure 4.5: Analysis class diagram - Access control

The access control component has been designed with focus on JWT authorization. This is because it is still not yet known where OAuth2 is going to be implemented precisely. This component shares the same design principles as with the datastore regarding the usage of services. Additionally, two data models exist, *AuthorizationResponse* and *AuthorizationRequest*. These are used to encapsulate the data sent to and from the access control API. For example, the response, which will be sent to a user, contains a JWT token as well as a refresh token and a status message. The request contains a users email and a password. The *AuthorizationService* will make extensive use of the two models in all its methods, hence the associations. *AuthorizationService* is a hard-dependency on the *AccessController* and therefore composition is used.

Finally, is the class diagram for the data communication API. This diagram will be the most complex, due to design requirements by GraphQL. GraphQL APIs contain a variety of elements such as the schema, query and mutation. The diagram for the data communication API can be seen on Figure: 4.6.

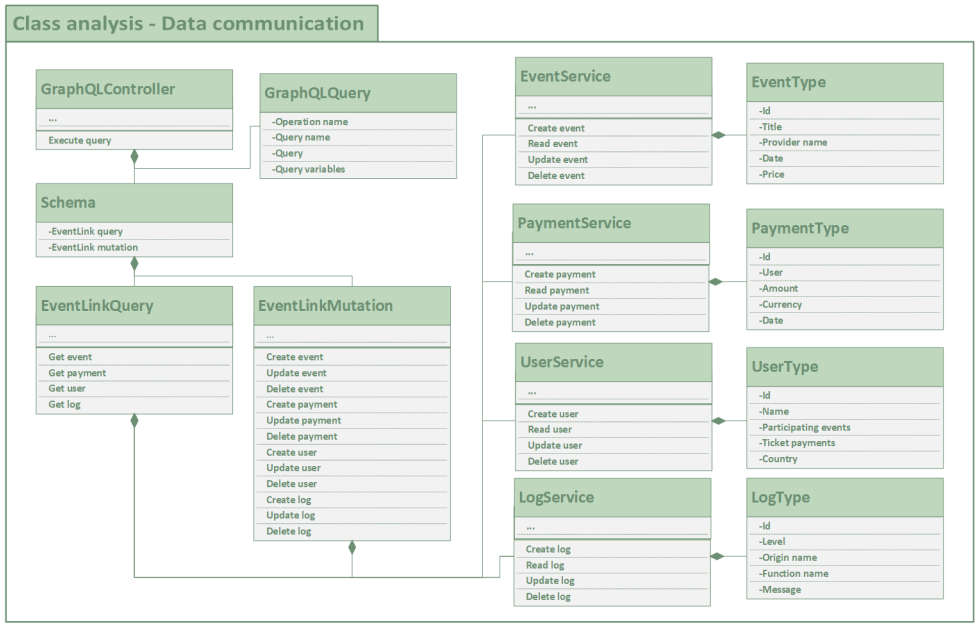


Figure 4.6: Analysis class diagram - Data communication

It is noticeable that the data models and services correspond to the same ones as in the datastore. This is done with a modular design in mind, such that business logic is encapsulated in the services, and *GraphQLQuery* and *GraphQLMutation*

then makes use of those. It is not possible to use the data models defined for the datastore, as GraphQL requires its own datatypes to work with, hence why *EventType*, *PaymentType* and *LogType* has to exist.
EventLinkQuery and *EventLinkMutation* define the operations that the API can do. Both the query and mutation class is used by the GraphQL Schema class, which is the building block of the whole API. Additionally, a *GraphQLController* and *GraphQLQuery* exist. The *GraphQLQuery* class should not be confused with the *EventLinkQuery* class, as the former is a generic query object, that could either be a GraphQL query, mutation or subscription. The query class includes four required fields, the operation name, query name, the query itself and query variables. These are strict requirements for the definition of a GraphQL query. Together, the *GraphQLController* uses both the *GraphQLQuery* and the *Schema* to be able to serve as an API.

4.1.3 Service layer diagram

To show an overview of the service layer architecture, Figure: 4.7 was made. This is yet another custom diagram that does not follow the direct guidelines of UML. As mentioned in the previous section, the components of the system are designed based on service layers.

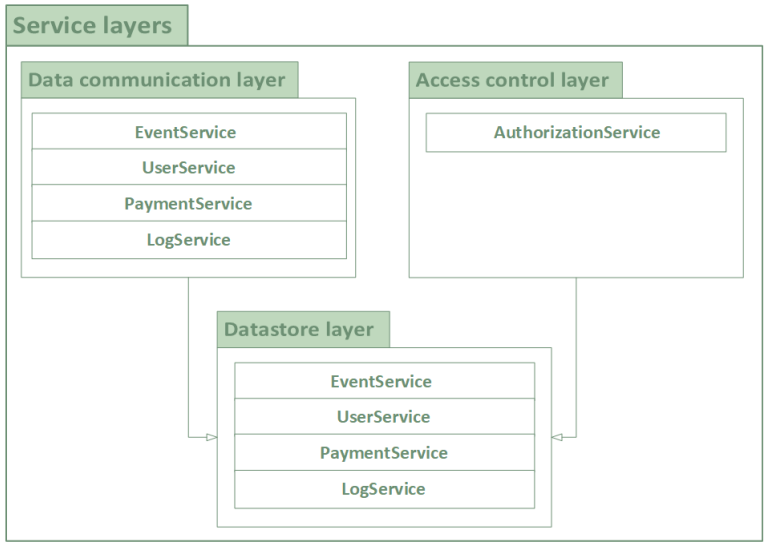


Figure 4.7: Service layer diagram

The diagram is made from the previous analysis class diagrams, in the sense that it displays the same services. The idea behind a service layer architecture is to make the system more modular as a whole, and thereby have less hard-dependencies between the components. It can be seen that the datastore layer provides services for events, users, payments and logs and that the access control layer provides a service for authorization. Additionally, the data communication layer provides services for the same models as the datastore. It is important to mention that the respective services in the data communication layer makes use of the corresponding services from the datastore. The datastore layer is a hard-dependency of both the data communication layer as well as the access control layer, due to the need of accessibility for the database. The services in the data communication layer will then be used in the GraphQL API implementation.

4.1.4 Deployment diagram

The diagram strives to describe how each component and server could communicate with each other as well as which communication protocol that is to be used. Later in the thesis, a concrete deployment diagram of how the system has been deployed will be shown. The generic deployment diagram can be seen on Figure: 4.8.

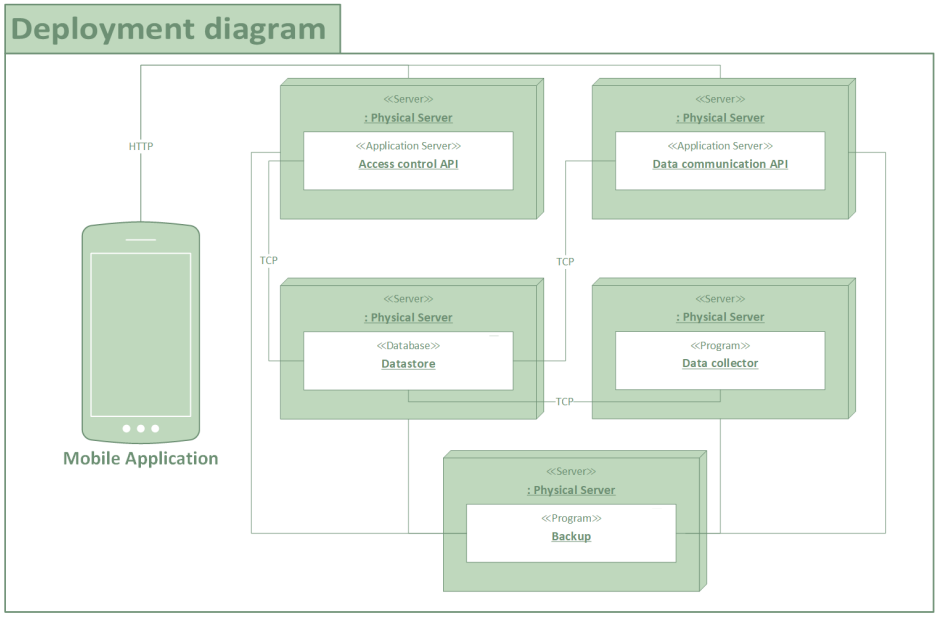


Figure 4.8: Generic deployment diagram

Each server seen in the diagram is representing a stand-alone physical server. In practice, this could as easily be a single physical machine running five virtual machines, each hosting their own component. Additionally, each component could in theory be deployed on the same server, but that would potentially lead to extensive load on the server. The operating system is not of high significance, as .NET Core is cross-platform and support the most widely used operating systems.

The mobile application communicates via GraphQL to the data communication API, which is a HTTP request with the GraphQL query as JSON payload. The access control API does not have a need to be a GraphQL API, so it will follow REST principles instead and therefore communicate through HTTP as well. The servers will most likely communicate through TCP as their communication will be implemented through the datastore business logic layer. It is not yet known how the backup server will communicate with the others, but the idea is for it to be some kind of ping, so it will most likely be TCP or UDP. Additionally, due to the scalability of NoSQL, it could be a good idea to think about if it would be possible for the MongoDB database to the distributed out to all five servers if needed.

4.2 Design patterns

This section will study the design patterns that are seemed to be fit for the system. In the previous sections, a service oriented architecture for all the components of the systems was mentioned, which will be explained further here. Taking into account both the modeling diagram as well as the analysis class diagrams, relevant design patterns are considered and discussed.

Service layer pattern The service layer pattern will contribute with having small, isolated pieces of business logic that is as independent as possible. This pattern is primarily used due to it making the architecture of the system simple, as well as more modular and maintainable. It makes the system more maintainable in the way that if the datastore would be changed from MongoDB to another database system, then only the services would have to be changed and nothing else. This way, the required maintenance is encapsulated to a well defined region in the system.

Singleton pattern The Singleton design pattern is a very simple pattern, but one that is very useful and deemed to go well together with a service layer pattern. Since the system will consist of a lot of services, which purpose in life is to provide functionality, it makes a lot of sense for these services

to be Singleton objects. The objects will not contain any mutable data, so making all of the service classes Singleton is an obvious choice. It does not make a difference, logic wise, whether two objects or a single exist of the same service. It does, however, make a difference in the resources consumed on the machine, as less objects will be allocated in memory.

Observer pattern The Observer pattern is a slightly more complex pattern, but useful when used in the right context. In the case of EventLink, the observer pattern would fit well for a kind of notification or subscription system. The idea here is, for example, if a user has subscribed to an event, they would get a notification if that given event changes in some way. The observer pattern would then be able to notify all of the subscribed users on that specific event. It is also important to keep in mind that similar functionality is built into GraphQL, so it might not be needed to implement directly.

GRASP A group of very important object-oriented design patterns is GRASP. During the general development of the system, a lot of principles from GRASP will be taken into consideration. Patterns such as high cohesion, low coupling, controller pattern, information expert and polymorphism will be used whenever fit. An example could be the encapsulation of data members of the data models, event functionality being restricted to the event service as well as the various controllers in the system. [Lar04]

4.3 Application servers

As mentioned previously in this chapter, five servers will be used to deploy the system upon. For this reason, a cloud provider is required. Some potential ideas could be AWS or DigitalOcean, however during the studies at DTU, there have been several occasions where DTUs own servers have been used. Since there is no requirement for any specific tools other than administrative server access, it was decided that using DTUs servers would be the best choice. This is due to the reason, that using servers from DTU will be free of charge, where either AWS or DigitalOcean would not.

A request was sent to Svend Mortensen from DTU to set up five servers with the hardware specifications seen below on Table: 4.1, Table: 4.2 and Table: 4.3. To avoid confusion, Memory is referring to RAM, CPU cores the amount of physical CPU cores on the machine and disk space to persistent storage space.

Hardware	**Minimum**
CPU cores	4

Hardware	Minimum
Memory	8GB
Disk space	50GB

Table 4.1: Hardware specification - Datastore server

Hardware	Minimum
CPU cores	4
Memory	8GB
Disk space	30GB

Table 4.2: Hardware specification - Data collection server, data communication server

Hardware	Minimum
CPU cores	2
Memory	8GB
Disk space	30GB

Table 4.3: Hardware specification - Access control server, backup server

Considering the first table, Table: 4.1, it is shown that a server with minimum four CPU cores, a minimum of 8GB of memory and 50GB of disk space is requested for the database server. EventLink is not going to be a resource intensive system in the start, so the specifications requested are fairly minimal. The same goes for the data collection and data communication servers which can be seen in Table: 4.2. It is the same specifications as the database server, but with less disk space. The last servers are the access control and backup servers from Table: 4.3. These servers will most likely perform less work compared to the others, so two CPU cores in this case will suffice. A bigger quantity of servers with more powerful hardware would be needed to scale the system if a lot of users would use it simultaneously. The machines rented from DTU is in fact not five different physical machines, but a single machine running five virtual Ubuntu Linux 16.04 instances. Running on virtual machines is fine for now, as the system will not be under substantial load.

4.4 Datastore

During the analysis, it was found that MongoDB and NoSQL were to be used. In this section, the usage of these two technologies will be looked further upon, especially on design details of the business logic layer for the datastore, as well as what kind of data the database should contain. Additionally, GDPR will be discussed as well, as it could have an important role in the future of EventLink.

4.4.1 MongoDB

The MongoDB business logic layer does not require a tremendous amount of design, as it is a fairly simple component. This layer will act as the interface between the other components and the datastore. Even though most of the required design was done in the analysis class diagram on Figure: 4.3, some relevant design questions are still present.

How will queries be designed? Since C# is used, the queries will be designed and executed using LINQ. LINQ is an extremely simple and powerful tool for querying data structures. The representation of collections will be an array-like data structure, which works very well with LINQ. The queries will not be created in a textual representation, but represented in source code instead. Consider the CRUD operation of reading an event with a given id. This would be done in C# with LINQ like this:

```
var event = collection.Find(e => e.Id == id).FirstOrDefault();
```

What this code snippet does, is that it uses LINQ to find the first object in the event collection with the exact same id as in the `id` variable. Other viable options for creating queries would be JSON queries, however due to the simplicity of LINQ, it will be used instead.

How is communication between MongoDB and C# obtained? An important element that has not been discussed yet is the communication between the business logic layer and the MongoDB database. The idea here is to use a Singleton database context class with the sole purpose in life of maintaining a connection to the database. This class would then exist inside all of the service classes, providing access to their respective collections. Luckily, MongoDB provides a library called MongoDB C#/.NET Driver for to maintain connection to a database. The database context class will be seen in the implementation chapter. [Mon19]

4.4.2 Data models

It is currently only vaguely known what kind of fields the data models should contain, looking at the point of view of Figure: 4.3. It is important to remember that the models will be used with JSON, so avoiding nested data structures is a plus. Each of the data models will be designed further below.

Event The data model for *Event* is without a doubt going to be the most complex data model of all. As event providers will contain their own event model, an idea could be to take a look at those to get some inspiration and then design the event model for EventLink. Therefore, a model for *Event* will not be discussed in this section, but will be evaluated in Section: 4.5.1 when the initial event providers are known.

User The data fields for *User* is going to be designed as any other ordinary user object. This means that the user model should contain information like first name, last name, birthdate, country etc. Additionally, some information specific to EventLink is required as well, such as which events the user is participating in, which events the user has favorited, as well as which users they are friends with. Taking all this in consideration, the following data model has been designed on Table: 4.4.

Datatype	Name	Description
string	Id	Unique identifier for this user.
string	FirstName	First name of the user.
string	MiddleName	Middle name of the user.
string	LastName	Last name of the user.
string	Email	The users e-mail address.
string	Address	The users home address.
Datetime	Birthdate	The users date of birth.
string	PhoneNumber	The users phone number.
string	Country	The users current country.
string	PasswordHash	The users hashed password.
List<string>	ParticipatingEvents	List of event id's the user is participating in.
List<string>	FavoriteEvents	List of event id's the user has favorited.
List<string>	PastEvents	List of event id's the user has participated in.
List<string>	Buddies	List of user id's this user is friends with.
List<string>	Payments	List of payment id's that belongs to this user.
Boolean	IsActive	States whether this user is deactivated or not.

Table 4.4: User data model

Log When the idea of logging was initially considered, the idea was to use it as a tool for debugging. Therefore, the data model for *Log* will be designed with this in mind. The data model can be seen on Table: 4.5.

Datatype	Name	Description
string	Id	Unique identifier for this log.
string	ParentName	The name of the class where this log occurred.
string	FunctionName	Name of function where this log occurred.
string	Message	The logged message, i.e. debug information.
LogLevel	LogLevel	Severity of the log.

Table 4.5: Log data model

Payment The *Payment* model should contain information regarding a users purchase or participation of an event. This includes price, currency, etc., such that a user can have access to a order history. The model can be seen on Table: 4.6.

Datatype	Name	Description
string	Id	Unique identifier for this payment.
string	UserId	The user who this payment belongs to.
string	EventId	The event that has been purchased.
Datetime	PaymentDate	The date of purchase.
string	Amount	The amount of money for the payment.
string	Currency	The currency used in the payment.
Boolean	IsCharged	Has the user been charged yet?

Table 4.6: Payment data model

4.4.3 General Data Protection Regulation

The General Data Protection Regulation (GDPR) is a EU regulation that was created to ensure data privacy and protection for each citizen within the EU. Therefore a lot of companies were to change their logic of what and how data should be stored and for how long. GDPR can be split up into two subcategories, personal data and sensitive personal data. [ApS19]

Personal data This is any information relating to an identified or identifiable natural person. This could be a social security number (CPR), phone

number, a full name or any direct personal data.

Sensitive personal data Sensitive personal data is the special cases of personal data, which describes the persons ethnic origin, political opinions, religion etc.

Specific data compositions could also lead to the identification of a singular person. For example, a person over the age of 90 living in a village of 200 people could be an indicator for a single person. Therefore it is needed to take caution for which data is stored within EventLink. In order to do this there will be made some precautions listed below.

Terms of Condition In order to make sure that the user of EventLink understands that their data will be stored, there could be created a terms of condition page, where they need to agree to specific criteria in order to use the application.

Anonymization Since a person is entitled to have their data removed from the application at all times, there has to be some way to clear a person from the database. Therefore there should exist some functionality in order to go through the database and remove the persons entries.

Pseudonymisation Another way to go about anonymization is pseudonymisation. The purpose of pseudonymisation is to replace a persons data with pseudo data. This could help to anonymize a person without losing the entries in the database.

As shown in Section: 4.4.2, some of the data that is going to be stored within EventLink is personal data. Therefore there should be implemented some functionality for a user to be anonymized or pseudonymized. As of now, there has been taken a decision that a persons data will be stored for 2 years after the subject has been deactivated. This should appear in the terms of condition. It should be noted that all of these criteria are extremely important, if EventLink were to turn into an actual product. [Con19]

4.5 Data collection

The custom data collection solution should collect data through event provider APIs. In order to do this there need to be conducted some research on which event providers can provide a sufficient amount of data to EventLink. To offer

as many events as possible, it is initially chosen to go with two substantial event providers. Once the providers have been determined, a data model for events will be designed.

4.5.1 Event providers

In order to settle on specific event providers, research upon these have been done. The criteria for this is the API must be free of charge to use and should be able to provide a decent amount of event data. A few event providers have been handpicked and will be further described in this section.

Ticketmaster Between the two developers of EventLink, the most well known event provider is Ticketmaster. Ticketmaster provides a number of different APIs. These allows to pull data from venues, events and payments. To be able to use the Ticketmaster API, a developer profile must be created. Since the developer profile is free, there is a limit to how many requests there can be made. The free profile is limited to 5000 requests a day, with a maximum of 5 requests a second. This means that event data can updated 5000 times a day, which would be done to ensure up to date data.

SeatGeek Another potential event provider is SeatGeek. SeatGeek offers a variety of different events. From sports to theater tickets. SeatGeek offers four different API endpoints. One for events, one for venues, one for performers and one for taxonomies. Like the Ticketmaster API there is a need for a developer profile. The major difference between the two is that there is no limitation on how many requests a user can create a day for SeatGeek. A major downside of SeatGeek is that it is made mostly for North America. This means that the events within the SeatGeek are located in either the US or Canada.

Eventful A third online ticket retailer is Eventful. Eventful provides an API that gives access to the worlds largest collection of events, taking place all over the world. In order to use the API a developer profile must be created. Eventful provides seven different endpoints, these being events, venues, users, images, performers, demand and categories. Like with Ticketmaster, it is also possible to only retrieve event data from a specific country using Eventfuls API. [Eve19b]

Ticketmaster and Eventful has been chosen as the initial event providers. This is mainly because both of these provide events in Scandinavia, whereas SeatGeek does not.

4.5.2 Event data model

Now that the initial event providers are found, their data models can be analyzed to get a better understanding of what information is commonly associated with events. The data model for Ticketmaster can be seen on Appendix: B.5. Unfortunately, it was not possible to acquire a diagram for the data model of Eventful, it is however described on their API documentation site. [Tic19] [Eve19a]

Considering Ticketmasters data model, the model that is being looked upon is *EVENTS*. Looking back at Figure: 4.3, most of the fields are already contained in the model. Investigating the data, it includes a lot of relevant fields like venues and social media links. The data model of Eventful contains many of the same elements as Ticketmaster but with a different structural layout. Eventful contains less nested data structures and in general less information than Ticketmaster.
Both the models for Ticketmaster and Eventful contain many data fields. To reason behind and explain why each of these would be a fit for EventLink would be too substantial. Instead, a table will be made with the decision of the initial event model. The data models of Ticketmaster and Eventful will be inspected thoroughly, and data fields that seem relevant and useful will be picked to be in the model. The final model can be see on Table: 4.7. This data model is mostly designed with Ticketmasters model in mind. The nested data structures seen in the table can be found in Appendix: B.2.2.

Datatype	Name	Description
string	Id	Unique identifier for this event.
string	ProviderEventId	Unique identifier given by the provider.
string	Title	The title of the event.
string	Type	The type of this object.
string	Url	URL to the event.
string	Locale	The locale for where the event is hosted.
string	Description	Textual description of the event.
Sales	Sales	Contains sales related info.
Dates	Dates	Contains date related info.
List<Classification>	Classifications	Contains genre, sub-genre, etc.
Promoter	Promoter	Contains event promoter related info.
List<PriceRange>	PriceRanges	Contains price related info.
List<Venue>	Venues	Contains venue related info.
List<Attraction>	Attractions	Contains relevant social media info.
List<Image>	Images	Contains images related to the event.
Boolean	IsActive	States whether the event is active.

Table 4.7: Data model - Event

4.6 Data communication

This section will dive deeper into the design behind the data communication component, more specifically GraphQL. The GraphQL API will be designed further by improving on the queries and mutations. Additionally, a GraphQL framework for .NET will be investigated.

4.6.1 GraphQL

Considering the analysis class diagram for the data communication component on Figure: 4.6. CRUD operations are fine as a basis, but as seen in the requirements, the user would like to be able to perform operations such as favoring an event or adding a user as a friend. For this reason, the queries and mutations will be redesigned.

Queries and mutations The queries and mutations will be redesigned such that they correspond better to the requirements of the system. An important thing to note here, is that this is not strictly necessary. All of the requirements could be accomplished by using CRUD operations together, which is what the actual business logic in the GraphQL API will do. Encapsulating all of this logic to the business logic in the GraphQL API will make the API a lot more developer friendly. On Table: 4.8 the redesigned queries and mutations can be seen.

Id	Function	Parameters
01	AddBuddy	UserId, BuddyId
02	RemoveBuddy	UserId, BuddyId
03	AddParticipatingEvent	UserId, EventId
04	RemoveParticipatingEvent	UserId, EventId
05	AddFavoriteEvent	UserId, EventId
06	RemoveFavoriteEvent	UserId, EventId
07	UploadProfilePicture	UserId, Picture
08	GetBuddiesParticipatingInEvent	UserId, EventId

Table 4.8: GraphQL queries and mutations

It is immediately noticeable, that the functions above are almost solely mutations. This makes sense, as regular data queries are simple and do not need to be changed. To take an example of the functions above and

what they should do, consider *GetBuddiesParticipatingInEvent*, which is not a mutation, but a slightly complex query. This query should be used when a user opens the mobile application and checks which of their friends are participating in a specific event. To be able to do this, the *UserId* must be given as a parameter in order for GraphQL to be able to know the context of the users friends. The *EventId* must be given to GraphQL as well, for it to understand the context of which event that the users friends should be participating in.

GraphQL .NET Framework There exists a official GraphQL framework for .NET, which is called GraphQL for .NET. This framework will be able to provide the infrastructure needed in order to be able to build a GraphQL API. The framework provides built in tools for GraphQL types, mutations, queries, subscriptions and schemas, which is precisely what is needed. [.NE19]

4.7 Access control

In this section, the OAuth2 integration will be discussed. It will be decided where the OAuth2 functionality will be placed. Secondly, security concerns for JWT token authorization will be examined as well.

4.7.1 OAuth

It was found out that it would be possible to sign in with an EventLink user through OAuth2. This would however require significant implementation in the GraphQL API which seems unnecessary. Therefore, it was decided not to use OAuth2 for sign in with an EventLink user, and to only keep it for third-party accounts. Because of this, it was also decided to push the OAuth2 business logic to the mobile application, such that no backend implementation would be necessary. This means that EventLink will have to rely on third-party libraries for the OAuth2 integrations. It could be argued that it would be smarter to implement it directly in the GraphQL API for the long run, but it is simply a choice to keep it out of the backend for now. The usage of OAuth2 will be explained further in Section: 4.8.1.

The idea behind the OAuth2 flow is to have a user sign in with a third-party account, using information from that account to create a EventLink user, based on a randomly generated password. Due to this constraint, the sign in functionality will be made such that once a user signs in with a specific sign in type, for

example Facebook, they will only be able to sign in through Facebook, and not use the same account to log in with their EventLink user.

4.7.2 JSON web tokens

Most of the necessary design for the access control API has been explained in Section: 4.1.2 and on Figure: 4.3. However, a few design elements for JWT are still present.

An access token expires after a set amount of time. Obviously, the less time a token is valid for, the more secure the authorization will be. However, when the access token expires, the user will have to request a new token. As explained previously, this will be done with a refresh token. The expiration time on an access token will be set for 20 minutes, as it is believed that this is a well balanced time-span.
The GraphQL API will be verifying a few things when provided an access token. To avoid confusion, it is important to remember that the access control API is issuing the access tokens by authorizing a valid EventLink user. It is the GraphQL API that receives the token and has to verify that it is legitimate. The GraphQL API will verify the issuer of the token (EventLink), the audience (GraphQL API), the lifetime (expiry), as well as its digital signature. This is thought to secure the validity of the token enough.

4.8 User interface

This section will design the mobile application by examining its widget architecture. Additionally, there will be searched for necessary libraries for OAuth2 and GraphQL. Lastly, a new design prototype of the application will be built upon the one created in the previous chapter.

4.8.1 Flutter

An analysis class diagram was not created for the mobile application, as it was not known how widgets and Flutter architecture work in practice. This has since then been understood better and a widget architecture can be examined.

The general idea for the widget design is to create as many as possible. Con-

sidering the user interface prototype on Figure: 3.7, each event structure would be a custom widget, containing relevant widgets itself. This would make it easy to create the list structure that includes multiple widgets. The same reasoning can be said for Table: 3.8, where the user would be a custom widget as well. A widget can either be *stateless* or *stateful*. The application will mostly be designed using *stateful* widgets as a lot of state changes will be needed. This will make for a more smooth user experience.

As single sign on with Facebook, Google and Apple have been deemed a MVP, libraries containing this functionality is required. Different libraries has been found which supports OAuth2 and can be seen on Table: 4.9. Additionally, a client library for GraphQL has been found as well.

Library	Description
oauth2 1.5.0	This library provides the basic OAuth2 functionality.
graphql_flutter 2.0.0	This library provides GraphQL functionality to flutter.
flutter_facebook_login 1.2.0	This library allows a user to sign in with facebook via the flutter application.
google_sign_in 4.0.11	This library allows a user to sign in with google via the flutter application.
apple_sign_in 0.1.0	This library allows a user to sign in with apple via the flutter application.

Table 4.9: Libraries for authentication within Flutter

It is seen on Table: 4.9, that Apple sign has a low version number. This is because signing in with Apple has been released very recently, and for this reason the Apple sign in has the lowest priority to be implemented.

4.8.2 Design draft

The prototype from Section: 3.8 will be redesigned into a more realistic view of the application. The sign in and sign up screens of the new prototype will be shown at first. These screens are only a subset of the whole prototype, which can be found on the following link in Appendix: [1]. The screens can be seen on Figure: 4.9 and Figure: 4.10 respectively.

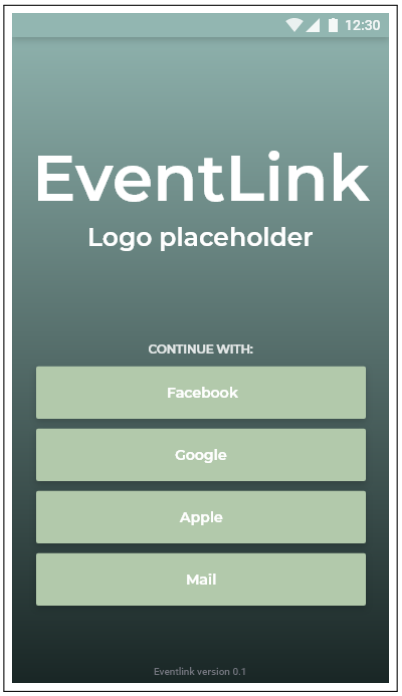


Figure 4.9: Prototype 2 Sign in screen

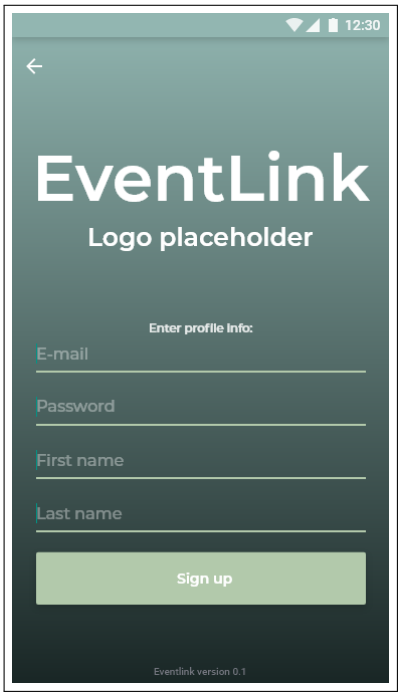


Figure 4.10: Prototype 2 Sign up screen

The OAuth2 sign ons are now present on the sign in screen and the colors have been changed to tone down the visual appearance. The user will be navigated to the sign up screen if the Mail button is pressed, and it will then be possible for the user to sign in or sign up. Once sign up is pressed, the user will automatically get signed in. Additionally, three more screens are presented, the main screen, event details screen and the application drawer seen on 4.11, Figure: 4.12 and Figure: 4.13 respectively.

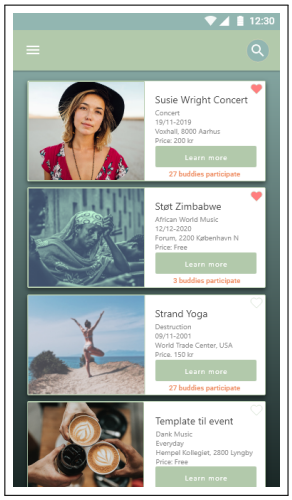


Figure 4.11: Prototype 2 Main screen

Figure 4.12: Prototype 2 Event details screen

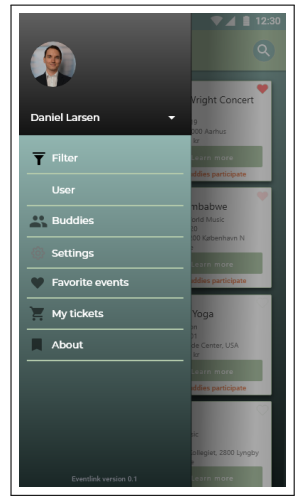


Figure 4.13: Prototype 2 Drawer screen

The main screen displays a list of events, where the user can favor an event by clicking on the heart. They can also search for an event by clicking on the magnifier at the top of the screen. When clicking on "learn more" the user will be navigated to the event details screen. This screen shows more details about the event, participating friends and the possibility to buy tickets for the event. If the user clicks on the hamburger button in the top-left corner on the main screen, they will be navigated to the drawer screen. The drawer allows the user to access shortcuts such as buddies, settings and ticket history. The screens are designed in such a way that they are believed to help reach the goals presented in the introduction.

4.9 Logging

Logging was vaguely introduced previously in this chapter, as a tool for debugging the source code. The logging system has to be designed in such a way, that it can use the data model seen on Table: 4.5 to log various kinds of useful information into the datastore. This will be done through a Singleton class that has access to the datastore, so that it can be used in the other components for logging. The logging system will use the *LogService* provided by the datastore

to insert its logs.

The logging system will be used such that if a user encounters an error, the users id will be logged together with the information described in the data model. This narrows down the location of an eventual error in the system. However, by doing this, there could be some challenges regarding the anonymization/pseudonymisation of a user, as the id is considered personal information.

4.9.1 Log levels

The last element that has to be designed for the logging system is the log severity levels. The logs should be classified in a way that makes important logs stand out from other less important logs. There will be six log categories. Each of these categories will be used to describe either debugging information, regular information or errors etc. The category list can be seen below.

Debug Used to log information that could be necessary to diagnose problems.

Info Used to log general system events.

Warning Used to log system events that may be an indication of a problem.

Error Would typically be logged in try-catch block, usually including the exception and contextual data.

Fatal A critical error that results in the termination of the system.

Trace Used to log the entry and exit of functions, for purposes of performance profiling.

There will be created additional collections in the datastore to better manage the logs. For example, a collection that only stores logs with a log level of *Error* and *Fatal*, and another collection that stores *Debug* and *Info* logs.

4.10 Summary

During the design phase, there has been chosen a system architecture which creates the foundation for the implementation and there has been made analysis class diagrams for each component. Additionally there has been created a

custom service layer diagram, in order to help give an overview of the chosen service-oriented architecture.
After this, there has been chosen three main design patterns, on which the system should be built upon. The chosen design patterns were *GRASP*, *Observer pattern* and *Singleton pattern* in addition to the *Service layer pattern*.
Once the initial work was done, servers were chosen in order to be able to deploy the system. It was chosen to ask DTU for five servers for economic reasons. There was handed over one physical server, consisting of five virtual machines. The MongoDB database has got an initial data model designed for the events, users, etc. Also GDPR was necessary to mention in regards to the storing of data. Two event providers were chosen for the data collector to work with, Ticketmaster and Eventful. Additionally, the data model provided by Ticketmaster, was chosen to be used as the basis for the event data model. The GraphQL API was designed, then redesigned to support more relevant API functionality.
The usage of OAuth2 and JSON web tokens was also explored in this chapter. In order to make debugging easier, logging was introduced. In the end, there was created a version 2.0 of the user interface prototype from the analysis chapter that is ready for the implementation phase.

The design phase was somewhat different from what is expected of a normal design phase. The analysis class diagrams helped tremendously with the design, but other elements such as how to design GraphQL queries and mutations, data models, OAuth2 and more had to be considered as well. The design phase in general describes what was needed in order to start the implementation and it is regrettable that the architecture was not elaborated further.
Seen in hindsight there should have been spent longer time on the design for the mobile application and the widget system of Flutter should have been researched more. Even though the design phase was not the best, the information required to initiate the implementation phase was still gained.

Chapter 5

Implementation

This chapter will use the knowledge gained in the analysis and design chapters to implement EventLink in practice. For each of the components of the system, a package diagram and design class diagram will show the desired implementation details. Then, specific implementation technicalities for interesting functionality in the components will be explained. Additionally, the challenges encountered during implementation as well as their solutions will be discussed for each component.

5.1 Deployment diagram

In the previous chapter a generic deployment diagram was seen on Figure: 4.8. The deployment diagram in this chapter is a explicit diagram of how the system will be deployed in practice and can be seen on Figure: 5.1.

As mentioned previously, the system will run on five virtual machines, running Ubuntu Linux 16.04. Each of the virtual machines has their own domain name for the sake of ease. The top domain name is *eventlink.ml*, however each server has their own sub-domain. The domain names are provided by Freenom as explained in Section: 7.6.
The diagram looks similar to the previous deployment diagram, however a few important elements are worthy to be noted. For example, the access control server (*auth.eventlink.ml*) and the data communication server (*api.eventlink.ml*) are both running Apache web servers. This is because the APIs have to be hosted, so running a web server on these machines is necessary.
Additionally, it can be seen that the backup server (*bak.eventlink.ml*) is running a VSFTP server as well as a python3 HTTP server. This is because there was a need to host the users profile pictures somewhere, so the backup server was given another purpose than previously thought. The pictures are uploaded through VSFTP and accessible through the HTTP server. An Apache server was not required for this purpose, as a simple Python HTTP server would work just fine.

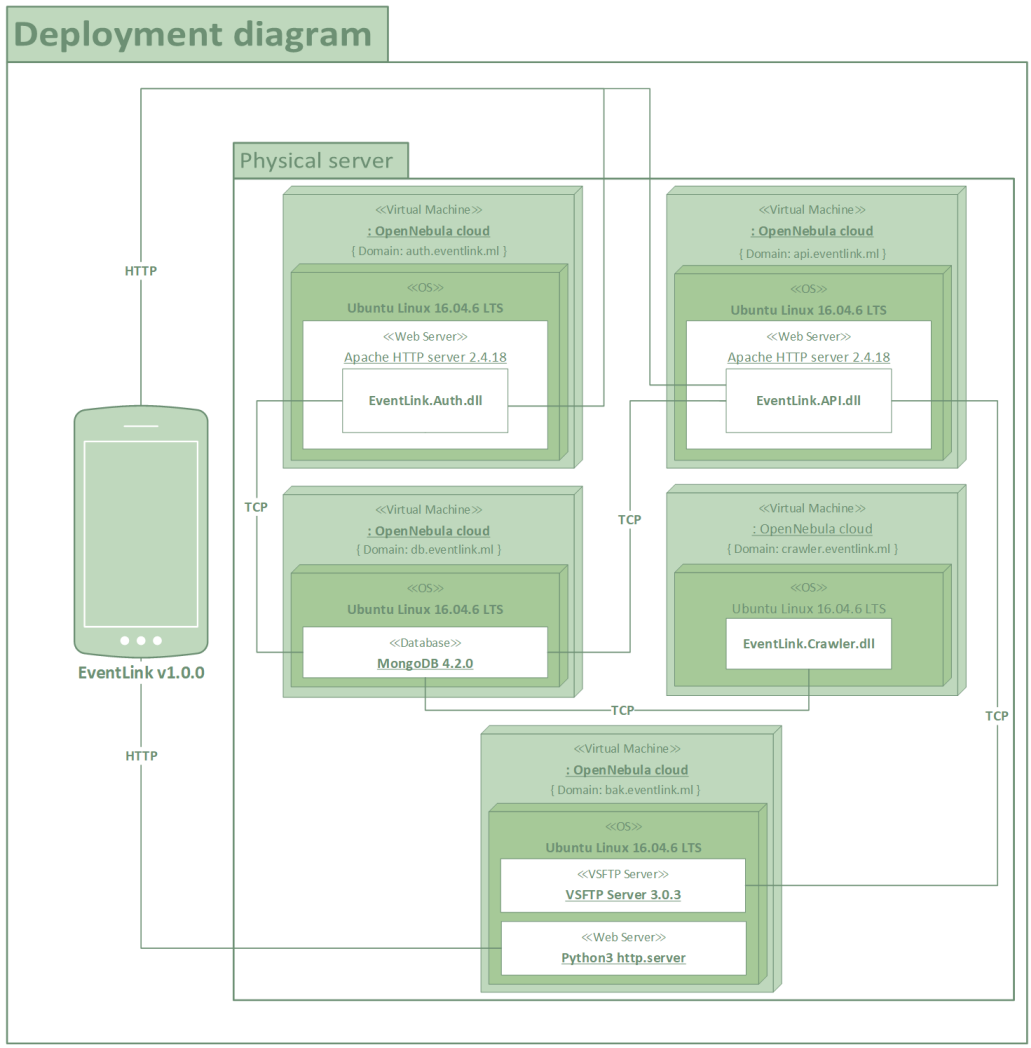


Figure 5.1: Explicit deployment diagram

5.2 MongoDB database

This section will show the implementation details of the software layer for MongoDB via a package and design class diagram. Then, the implementation of the event service will be explained and finally challenges regarding unique identifiers will be described.

5.2.1 Package diagram

As this is the first package diagram, a few important things has to noted. In the package diagrams, dependencies are shown with a dashed arrow. For example, on Figure: 5.2 you can see that all of the services in the *Services* package depend on *DbContext* and *DAException*. Notice that the arrow is going out from the package, and not a class.

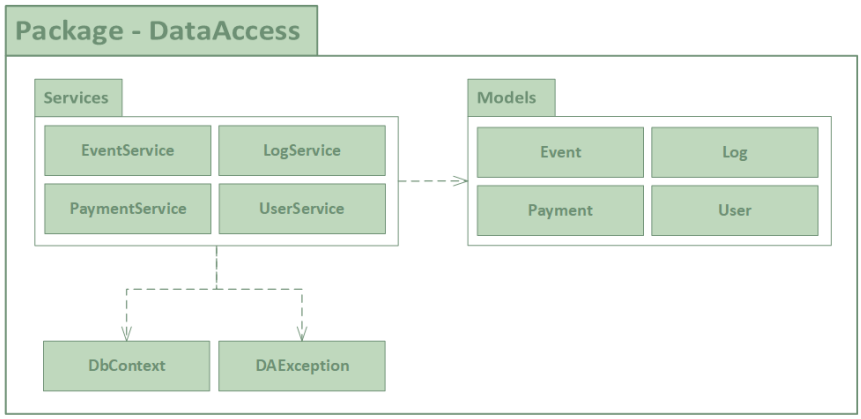


Figure 5.2: Package diagram - MongoDB software component

The implementation of the database component has not changed from the design phase. As mentioned, the *DbContext* class would be introduced. Additionally, a custom exception class has been implemented, due to the usefulness of having component specific exceptions. Otherwise, everything is the same as in the design phase. A package called *Util* containing utility classes also exists in the component, but was decided not to be shown in this diagram for simplicity reasons, as it exists in the other components as well and will be explained then.

5.2.2 Design class diagram

The design class diagram for the MongoDB software component can be seen on Figure: 5.3. UML composition and interface realization are used to signify dependencies and interface implementation respectively. Additionally, the generalization arrow is used for inheritance. Design class diagram have been made for the event data model as well. However, due to the severe extensiveness of the model, it will not be shown in the thesis or explained any further, as it has been explained a fair amount already. It can be seen on Appendix: B.7.

Considering the figure from top to bottom, interfaces for each service has been implemented, such that developing new services would be a simple task if need be. Taking a look at the interface methods, it can be seen that they are still largely CRUD operations with a few auxiliary methods. The service classes have been implemented as Singleton, as can be seen due to the static *Instance* objects. In addition to that, the data models have the same fields as described in the design phase, perhaps with an exception or two. The *DbContext* class is implemented as Singleton as well, and it uses the MongoDB C#/.NET Driver for interfacing with the actual MongoDB database. There has been created a few extra enumerations to help with things such as sign in constraints, log sorting etc. It is important to state that it seems on the diagram that *DAException* is the only exception being used by the services. This is not the case in reality, but it was decided to show it this way since all the other exceptions inherits from *DAException* and it makes the diagram simpler.

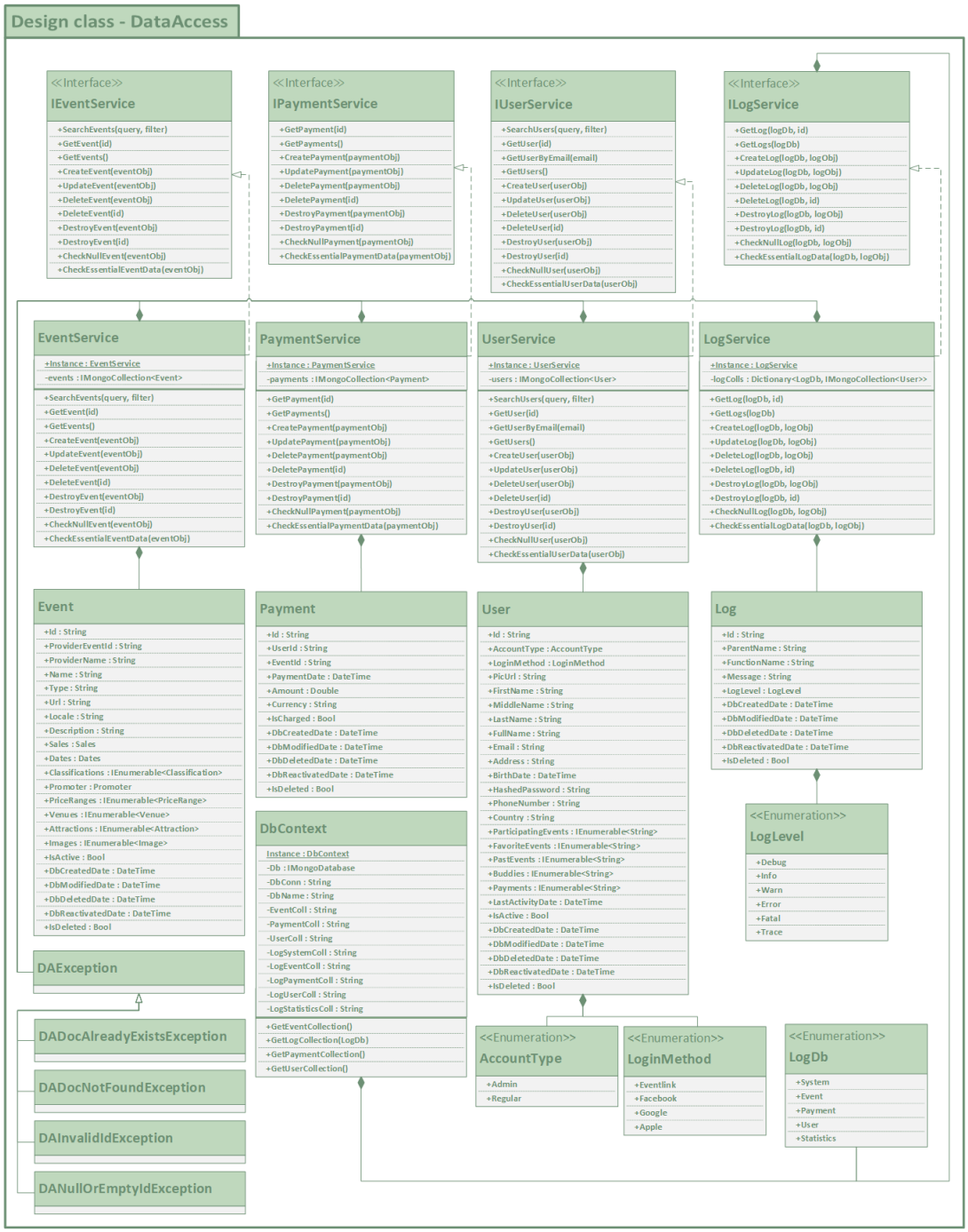


Figure 5.3: Design class diagram - MongoDB software component

5.2.3 Implementing the event database service

In order for the event service to function, it needs access to the database. As explained previously, this will be done by an auxiliary class called *DbContext*, of which a code snippet can be seen on Figure: 5.4.

```
private readonly string _dbConn = ConfigurationManager.AppSetting["ConnectionStrings:DbServerConnection"];
private readonly string _dbName = ConfigurationManager.AppSetting["DbNames:EventLink"];

private static IMongoDatabase _db;

1 reference | 0 exceptions
private DbContext()
{
    IMongoClient dbClient = new MongoClient(_dbConn);
    _db = dbClient.GetDatabase(_dbName);
}

1 reference | 0 exceptions
public IMongoCollection<Event> GetEventCollection()
{
    return _db.GetCollection<Event>(_eventColl);
}
```

Figure 5.4: Code snippet - DbContext

The figure shows an example of everything that is worth knowing regarding the *DbContext* class. It connects to the MongoDB database through a connection string and gets the specified database by name. The collections can then accessed through the database by name, and can then be used to access the data elements. For example, the *GetEventCollection* method is used in the *EventService* class, such that it has access to the event collection. A code snippet for the *EventService* class can be seen on Figure: 5.5.

```
20 references | 0 exceptions
public Event GetEvent(string id)
{
    Event eventDoc;

    /* If id is null or empty */
    if (string.IsNullOrEmpty(id))
    {
        throw new DANullOrEmptyIdException();
    }

    /* Tries to find the first object with the given id */
    try
    {
        eventDoc = _events.Find(e => e.Id == id).FirstOrDefault();
    }
    catch (FormatException)
    {
        throw new DAInvalidIdException("Id (" + id + ") has an invalid format!");
    }
    catch (MongoException e)
    {
        /* If exception is thrown from DB, pass the exception up */
        throw new DAException(e.Message, e);
    }

    /*
     * If the Event document is null, then we assume that an
     * Event with the given, id does not exist in the database.
     */
    if (eventDoc == null)
    {
        throw new DADocNotFoundException("Event with Id (" + id + ") was not found!");
    }

    return eventDoc;
}
```

Figure 5.5: Code snippet - Event Service

The code on the figure shows the read operation for the event data model. Since every event has its own unique identifier, this is used as the field to get the specific event. After some initial error checking, the method uses LINQ together with the event collection to search for an event with the specified identifier. Additional error checking exists, checking whether the identifier is correctly formatted, if the event exists, etc. All of the other CRUD operations look similar to this.

5.2.4 Challenges

During the implementation of the database component, a few challenges were encountered. Since the crawler and the database component work closely together and many of these issues are related to the event data model, some of the challenges will overlap. These will be explained in Section: 5.3.

Two unique identifiers The biggest challenge encountered was a problem related to using two ids. The MongoDB database has its own unique id (*Id* in the data model), and the event provider where the event is gathered from as well (*ProviderEventId*). For the event to be a valid event, both of these has to be present. This caused a lot of errors during early implementation, especially when updating events. This problem was solved by using *Id* as the main unique identifier, and then implementing the update method such that it directly supports all four possible scenarios (both ids are present, *Id* is present, *EventProviderId* is present, none are present). This problem was caught by unit tests.

5.3 Event crawler

This section will describe how the event crawler has been implemented through a package and design class diagram. After this, the implementation of event parsing will be explained. Once this is done, challenges regarding parsing and data quality will be described. The event crawler was previously called data collector, but was renamed during implementation.

5.3.1 Package diagram

The package diagram for the event crawler can be seen on Figure: 5.6.

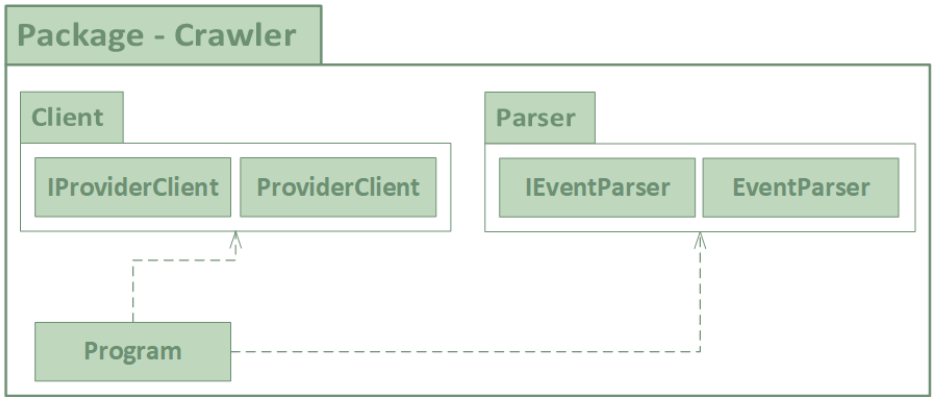


Figure 5.6: Package diagram - Crawler

The package diagram for the crawler is fairly simple, and has not been changed much from its design. A *Program* class is present in this diagram, as the crawler is an actual executable program, which for example, the database software component is not. *Program* is using the client and parser packages, as *Program* is where the actual collection of data and parsing happens. The *client* package is the previous *Collector* class, where the collection of events happen. Each of these packages contain an interface as well as a business logic class. These four classes will be explained in the design class diagram.

5.3.2 Design class diagram

The design class diagram for the crawler can be found on Figure: 5.7. Starting from the top left corner, the *IEventParser* is shown. This contains a single method that parses the event data. This is a semi-generic method, such that it is given an event provider (*Provider* enumeration) and some data as JSON. Since the provider is given, it then knows how to parse it correctly. The *ParseEventData* method is used by the *EventParser*, which contains a bunch of parsing methods.
Program uses logging, in order to make sure that the process of gathering events is logged within the datastore. It provides some statistics on how many new events were gathered, how many were updated as well as errors. This class also

uses the *EventProviderClient* to collect events.
At the end there is the *ConfigurationManager* left. The purpose of this class is to store and share the sensitive data necessary, this is for the example the event provider API keys. Additionally, custom exceptions for this component has been implemented as well.

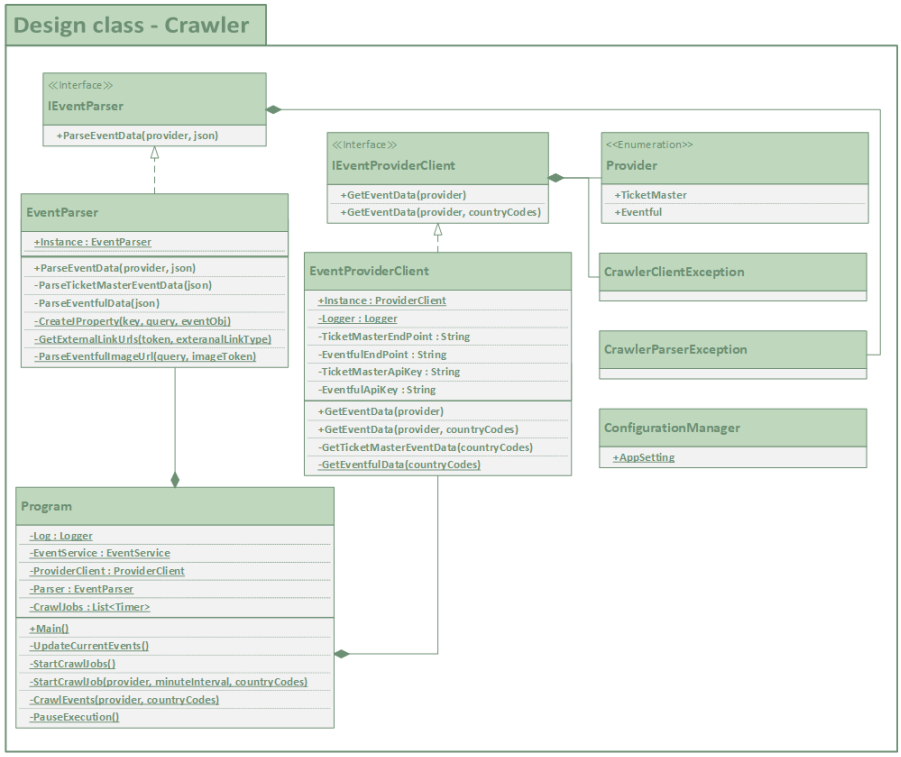


Figure 5.7: Design class diagram - Crawler

5.3.3 Implementing event parsing

The data within the MongoDB database is stored in JSON, so all the data received from providers is parsed to the event data model shown in Appendix: B.7. Therefore, two methods for parsing either Ticketmaster or Eventful can be seen on Figure: 5.8.

```
10 references | 0 exceptions
public IEnumerable<Event> ParseEventData(Provider provider, JObject json)
{
    switch (provider)
    {
        case Provider.TicketMaster:
            return ParseTicketMasterEventData(json);
        case Provider.Eventful:
            return ParseEventfulData(json);
        default:
            throw new CrawlerParserException("Invalid provider.");
    }
}
```

Figure 5.8: Code snippet - Parse event data

This method simply decides on how to parse the given JSON object by checking the value of the provider enumeration. Since the parsing of Ticketmaster and Eventful data is implemented very similarly, a code snippet of only Ticketmaster parsing will be shown on Figure: 5.9.

```
/* For loop to iterate through events from TicketMaster one at a time. */
foreach (var tmEvent in tmEventList)
{
    var customEvent = new JObject
    {
        /* Data from TicketMaster - Events. */
        CreateJProperty("providerEventId", "id", tmEvent),
        new JProperty("providerName", "TicketMaster"),
        CreateJProperty("name", "name", tmEvent),
        CreateJProperty("type", "type", tmEvent),
        CreateJProperty("url", "url", tmEvent),
        CreateJProperty("locale", "locale", tmEvent),

        /* Creates custom EventLink input. */
        new JProperty("description", ""),
        new JProperty("isActive", true),
        new JProperty("dbCreatedDate", new DateTime(1990, 1, 1)),
        new JProperty("dbModifiedDate", new DateTime(1990, 1, 1)),
        new JProperty("dbDeactivatedDate", new DateTime(1990, 1, 1)),
        new JProperty("dbActivatedDate", new DateTime(1990, 1, 1)),
        new JProperty("IsDeleted", false)
    };

    /* Creates custom object to store nested info from TicketMaster - Sales. */
    var sales = new JObject()
    {
        CreateJProperty("startDateTime", "sales.public.startDateTime", tmEvent),
        CreateJProperty("startTBD", "sales.public.startTBD", tmEvent),
        CreateJProperty("endDateTime", "sales.public.endDateTime", tmEvent)
    };
    customEvent["sales"] = sales;
```

Figure 5.9: Code snippet - TicketMaster event parsing

The parsing starts out with creating a new JSON object (JObject) with the name of *customEvent*. A method called *CreateJProperty* is an auxiliary method made to query a JObject for data. The initial data from the providers are encapsulated in a JObject. So for example, the first invocation of *CreateJProperty* parses the *Id* field to *ProviderEventId* in the *customEvent* object. Additionally, some custom fields such as *dbCreatedDate*, *dbModifiedDate* are initialized as well. At the bottom the nested JObject *sales* is created and put into *customEvent*.

In addition to the event parsing, it should also be noted that the crawler is implemented such that it will collect events from the providers, then it will update already existing events in the database, and insert new ones. The interval is specified in a variable loaded on runtime, such that it can easily be changed.

5.3.4 Challenges

Three primary challenges were encountered during crawler implementation related to *DateTime* parsing, data quality and provider restrictions.

DateTime parsing Throughout the implementation it has been a challenge to use the *DateTime* class. The purpose of this class is to store date and time information in various formats. On Figure: 5.9, it is seen that all of the functions where *DateTime* is used has the value of 1/1/1990. The idea here is that these values would then be changed to more realistic values when the given object was inserted in the database. However, it has been encountered that, especially the *dbCreatedDate DateTime* object never changes, even though functionality has been implemented correctly. Additionally, in early implementation, the way that both Ticketmaster and Eventful stored their date and time information in their JSON objects made was difficult to parse. This was caught by unit tests and has since then been fixed.

Data quality Another significant problem have been the data quality supplied by the event providers. Since the currently used data model is heavily based on Ticketmasters data model, it means that when a Eventful event is parsed, many data fields are potentially empty. This is not due to Eventful not providing the right data, but it simply does not contain as much data as Ticketmaster. It was a decision to not improve on this problem. Additionally, data quality issues with the data provided by Eventful is present. Sometimes, whole data fields would be missing and the provided data would be missing elementary things, such as `http://` in front of an URL.

Provider restrictions To ensure that the event data stored in the database is up to date, the events should be updated as much as possible. Ticketmaster has a restriction on the amount of API requests that can be made to their API with a free developer profile, which is 5000 requests a day. Now, evening out 5000 requests a day will be enough, but in regards to future event providers that might have more strict restrictions, it could potentially be a problem. Another thing is, if EventLink were to go into actual production, a free developer accout would not be used. This has not been an issue per say, but important to mention nevertheless. Additionally, it could be discussed whether this is the right approach to gathering up to date event information.

5.4 GraphQL API

This section will show implementation details regarding the GraphQL API. A package diagram as well as a design class diagram will be shown, and the implementation technicalities of GraphQL queries and mutations will be presented. Finally, challenges regarding the GraphQL execution engine as well as data model inconsistencies will be discussed.

5.4.1 Package diagram

The package diagram for the GraphQL API can be seen on Figure: 5.10. The Schema package is blank due to simplicity purposes, a stand alone package diagram for Schema can be seen on Appendix: B.6.

This is another component that has not changed architecturally since the design phase. Starting from the top, a package is containing the *GraphQLController* and *GraphQLQuery* classes, as these are relevant classes for API control. Next, the *Util* package is found here as well, with an additional *FTPClient* class used for uploading user profile pictures. The services package is present as well, containing the services used by the query and mutation classes inside the Schema package. Since this is an executable program that hosts an API, the *Program* class is present here as well. The *Startup* class is required boilerplate code for the API, where various configurations can be done. Both *Program* and *Startup* do not use any of the other classes in a functional way, so it was therefore decided to omit the dependencies.

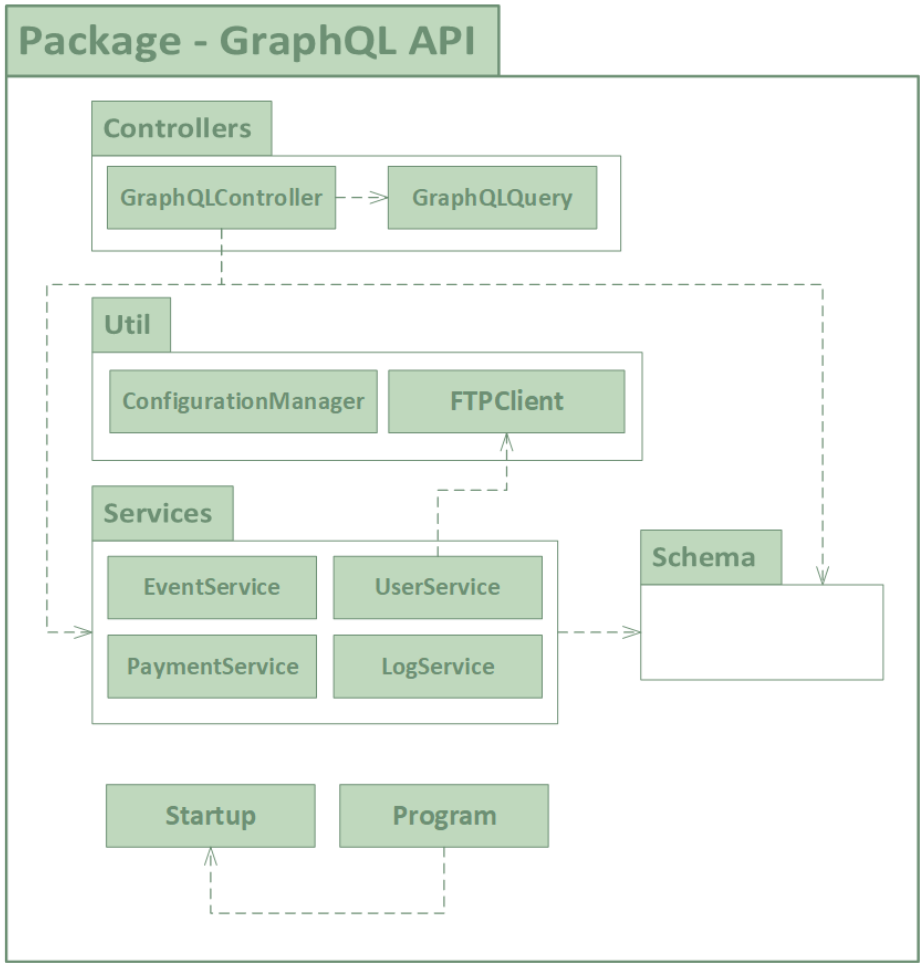


Figure 5.10: Package diagram - GraphQL API

5.4.2 Design class diagram

The design class diagram for the GraphQL API can be seen on Figure: 5.11. The two most important classes *EventLinkQuery* and *EventLinkMutation* are shown in the Schema package. The missing packages are mostly GraphQL data models corresponding to the same models as in the database component, hence they are omitted.

The *IUserService* contains significant more methods in comparison to the other service interfaces. This is due to the simple reason, that if a method requires user information, then it would be implemented in the *IUserService*. Notice that most of the functionality of the *IUserService* are direct requirements from the analysis phase. In the end, it's the *EventLinkQuery* and *EventLinkMutation* classes that contain the functions that can be invoked through the API.
The *GraphQLController* uses logging extensively to log information whenever a query is made to the system, mostly exceptions and other errors.
No custom exceptions were made for the API services due to the fact that the specifics of which kind of exception is thrown is not too important here. The error message will just be sent back with the response. Only a specific exception was made for the *FTPClient* class. The reason some classes are shown in this diagram and not the package diagram is that, for example, the *FTPClientException* is an inner class inside *FTPClient*.
It is important to mention that the *PaymentService* is never used in practice, as implementation to purchase tickets from the system has not yet been implemented. In addition to that, the compositions to the *Startup* class is due to registrations of the other classes. These are necessary for the API to function.

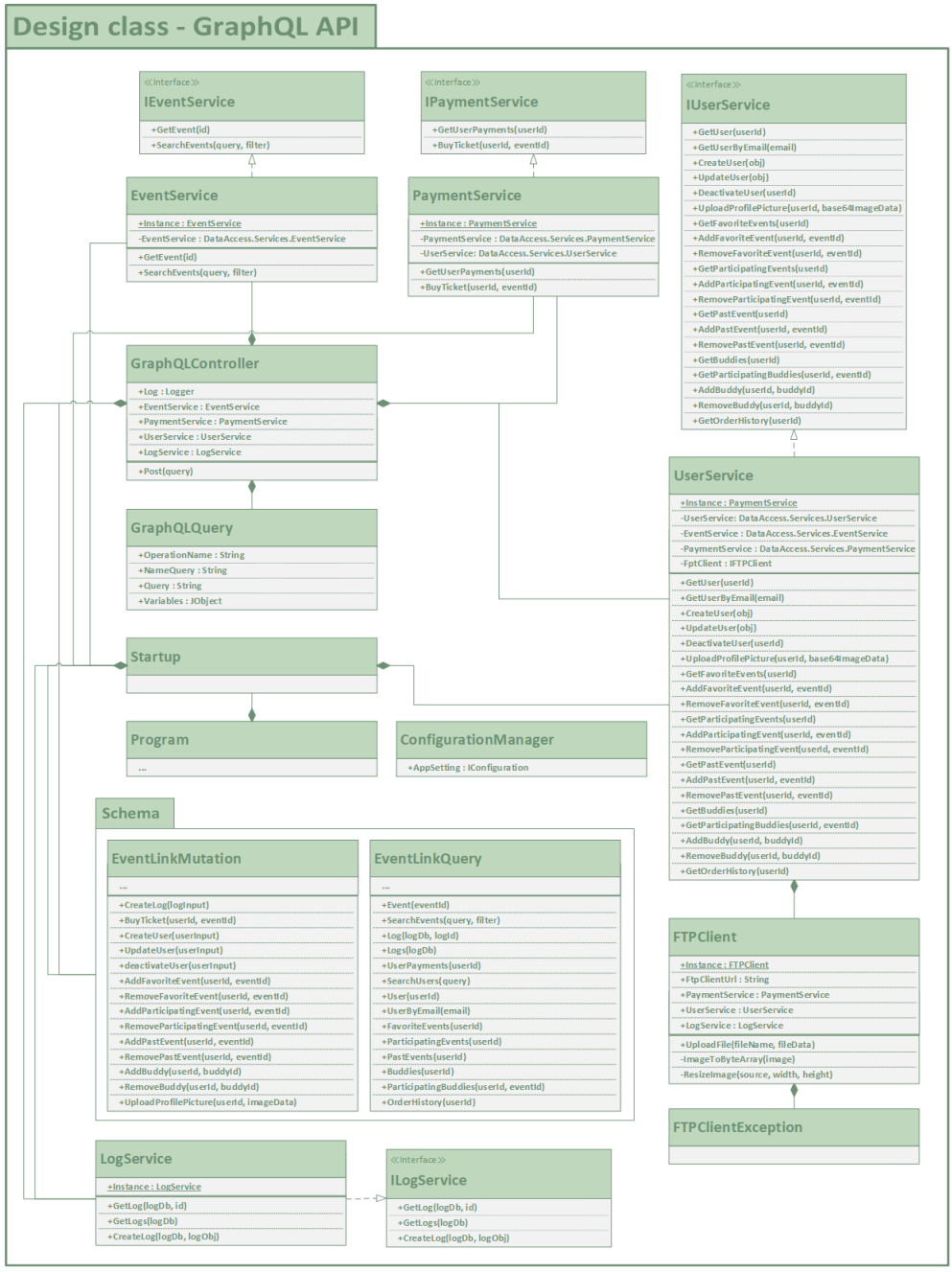


Figure 5.11: Design class diagram - GraphQL API

5.4.3 Implementing GraphQL queries and mutations

Due to the chosen design patterns and architecture, the actual implementation of the GraphQL queries and mutations is very simple. The general structure of a query or a mutation is a constructor that contains the required services, and then numerous field expressions that implements the actual API functionality. The *EventLinkQuery* and *EventLinkMutation* both make extensive use of the four API services as can be seen in Figure: 5.12.

```
1 reference | 0 exceptions
public EventLinkQuery(IEventService eventService, ILogService logService,
    IPaymentService paymentService, IUserService userService)
{
    Name = "EventLinkQuery";

    /**********************************
     * EVENT QUERIES
     **********************************/
    Field<EventType>("event",
        arguments: new QueryArguments(new QueryArgument<NonNullGraphType<StringGraphType>> { Name = "eventId" }),
        resolve: context =>
        {
            try
            {
                var eventId = context.GetArgument<string>("eventId");
                return eventService.GetEvent(eventId);
            }
            catch (Exception e)
            {
                context.Errors.Add(new ExecutionError(e.Message));
                return null;
            }
        }
    );
```

Figure 5.12: Code snippet - EventLinkQuery

Looking at the query in the diagram, it can be seen that it is called *event* and takes a single *string* argument called *eventId*. The program then extracts the argument and gets the value with the use of the *GetArgument* method. Next, the *EventService* is used to get the desired event. If any errors occur, the exception will be caught and sent back as a response. The reasoning behind this choice of design is not going to be evaluated any further, as it has been plenty in the thesis so far.

An example of a mutation can be seen on Figure: 5.13. Notice that the implementation of a query and a mutation is the exact same. In fact, it would be possible to put the code seen inside this mutation in a query, since there is no implementation difference between a query and a mutation. It is solely a conceptual organization of the functionality.

This mutation corresponds to the *Participate event* use case from the analysis. It requires two arguments, a *userId* and an *eventId*, both of type *string*. Again, the values are extracted and the appropriate API service method is invoked.

```
Field<StringGraphType>(
    "addParticipatingEvent",
    arguments: new QueryArguments(new QueryArgument<NonNullGraphType<StringGraphType>> { Name = "userId" },
        new QueryArgument<NonNullGraphType<StringGraphType>> { Name = "eventId" }),
    resolve: context =>
    {
        try
        {
            var userId = context.GetArgument<string>("userId");
            var eventId = context.GetArgument<string>("eventId");
            userService.AddParticipatingEvent(userId, eventId);
            return eventId;
        }
        catch (Exception e)
        {
            context.Errors.Add(new ExecutionError(e.Message, e));
            return null;
        }
    }
);
```

Figure 5.13: Code snippet - EventLinkMutation

5.4.4 Challenges

Two significant annoyances were encountered during the development of the GraphQL API. The first being exceptions due to wrong type parsing in the GraphQL execution engine and the second data model inconsistencies between the GraphQL API and the database software component.

Graph Query Language (GraphQL) execution engine This problem requires a brief explanation. When making a GraphQL query, you can either specify arguments as direct values or you can can pass them as parameters to the query. Usually, when calling the API functions from a client, the easiest method is to pass them as arguments. During implementation, this was tested extensively with GraphiQL (More in Section: 7.7). The problem was, that sometimes the API calls would work when giving values as direct input, but would fail when given parameters, even though it was the same values. It took many hours to debug and find the cause, but in the end it was fixed with a small change in GraphQLs type parsing. [mrp20]

Data model inconsistencies As explained previously, GraphQL has its own set of data models that it uses, which are equivalent to those of the MongoDB software component. As seen in this chapter, the event data model

is very complex with a lot of nested data. For the two paired models to be able to be parsed correctly, they have to correspond exactly with each other, so all changes to one has to made to the other as well. This has been a problem at times, where a quick change has been made in one of them and then forgotten in the other. This has contributed to extra hours of debugging.

5.5 Access control API

Like the previous components, the access control API will have a package diagram and a design class diagram as well. Additionally, the implementation details of JWT will be discussed. At last, challenges regarding JWT refresh tokens will be explained.

5.5.1 Package diagram

The package diagram for the access control API can be seen on Figure: 5.14. It has not been changed from the design phase.

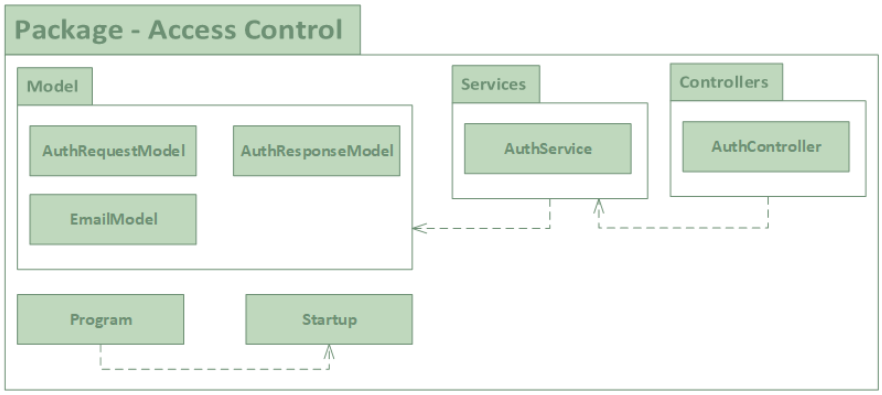


Figure 5.14: Package diagram - Access control

The diagram includes the data models for request-response authorization as well as its service and corresponding controller. Some new classes not seen before in the design are the *Startup* and *Program* class. *Program* is the entry of execution

and is required since it is an API. *Startup* is used for configuration of the API controller, it has been seen in previous diagrams as well.

5.5.2 Design class diagram

The design class diagram for the access control API can be seen on Figure: 5.15.

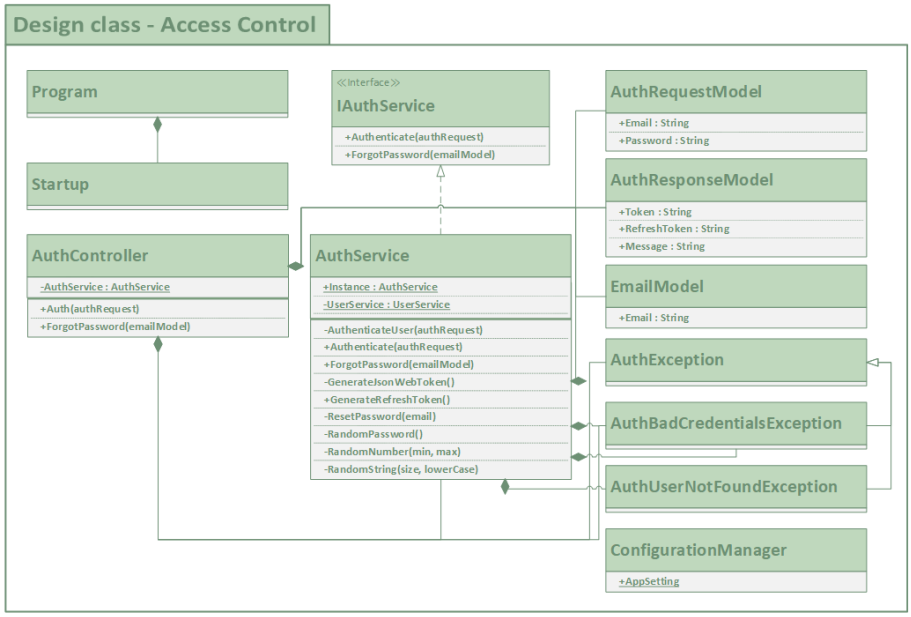


Figure 5.15: Design class diagram - Access control

There has been created an interface for the *AuthService* service class, and the class implementation contains various auxiliary methods for user authorization and JWT generation. Custom exceptions has been implemented here as well for the same reasons as explained for the previous design class diagrams.

5.5.3 Implementing JSON web tokens

In Section: 4.7.2 the validation parameters used for verifying a JWT was discussed. On Figure: 5.16 the implemented validation parameters can be seen.

```
services.AddAuthentication(JwtBearerDefaults.AuthenticationScheme)
    .AddJwtBearer(options =>
    {
        options.TokenValidationParameters = new TokenValidationParameters
        {
            ValidateIssuer = true,
            ValidateAudience = true,
            ValidateLifetime = true,
            ValidateIssuerSigningKey = true,
            ValidIssuer = Configuration["Jwt:Issuer"],
            ValidAudience = Configuration["Jwt:Issuer"],
            IssuerSigningKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Configuration["Jwt:Key"]))
        };
    });
```

Figure 5.16: Code snippet - JWT validation parameters in GraphQL API

As discussed in the design chapter, the issuer, audience, lifetime and signing key are validated for a token. The issuer and audience is considered classified information, so these are loaded from files that are not public. How this is done will be explained later. It is important to reiterate that this code snippet is taken from the GraphQL API as that is the component that verifies JWT tokens. Additionally, to secure the GraphQL API with the tokens, an authorization annotation is required in the *GraphQLController* class. This can be seen on Figure: 5.17.

```
namespace EventLink.API.Controllers
{
    [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)]
    [Route("/graphql")]
    [ApiController]
    1 reference
    public class GraphQLController : ControllerBase
    {
        private readonly Logger _log = Logger.Instance;
```

Figure 5.17: Code snippet - JWT annotation in GraphQL API

The generation procedure for the tokens can be seen on Figure: 5.18.

```
1 reference | 0 exceptions
private string GenerateJsonWebToken()
{
    var securityKey = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(Config["Jwt:Key"]));
    var credentials = new SigningCredentials(securityKey, SecurityAlgorithms.HmacSha256);
    var token = new JwtSecurityToken(Config["Jwt:Issuer"], Config["Jwt:Issuer"],
        null, expires: DateTime.Now.AddMinutes(120), signingCredentials: credentials);
    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Figure 5.18: Code snippet - JWT token generation

A JWT token is generated by encoding a secret key, that is contained within the system. If this key is leaked, none of the issued tokens would be considered secure, so keeping this key secret is very important. Then, signing credentials are made by hashing the security key. Next, the actual token is created with the issuer, expiry time and credentials. Notice that in the figure the expiry time is set to 120 minutes. The reasoning behind this will be explained in the following challenges section. As explained before, in practice, it would be advised to keep it to 20 minutes. An improvement to the token generation could be making it user specific, such that only tokens issued for a specific user can be used by that user.

5.5.4 Challenges

Due to the simple implementation of the JWT tokens, there was not encountered any significant challenges. However, one thing that was shown to be problematic was the implementation of refresh tokens.

It was attempted to implement refresh tokens, such that users could be signed in to the application forever, in theory. However, during the testing of the attempted implementation, the refresh tokens were never used by the system. The reason for this is still unknown, and it was shortly after decided to scrap the implementation of refresh tokens due to time constraints. Instead, the user gets automatically signed out of the application when their token expires. This is the reason behind setting the expiry time to two hours instead of 20 minutes for now.

5.6 Flutter mobile application

In this section, a package diagram and design class diagram for the mobile application will be shown. Additionally, the implementation behind OAuth2 and the main screen be will be presented. At the end, some of the encountered challenges during the implementation will be discussed.
A necessary disclaimer for this section is that the Flutter application has not been implemented very well, meaning that the cohesion and coupling of the classes is dreadful. Therefore, some compromises have been made with the diagrams.

5.6.1 Package diagram

The package diagram for the mobile application can be seen on Figure: 5.19. Due to the implementation of the application, only the most important interactions between the packages and classes are shown, otherwise the diagram would not provide any useful information.

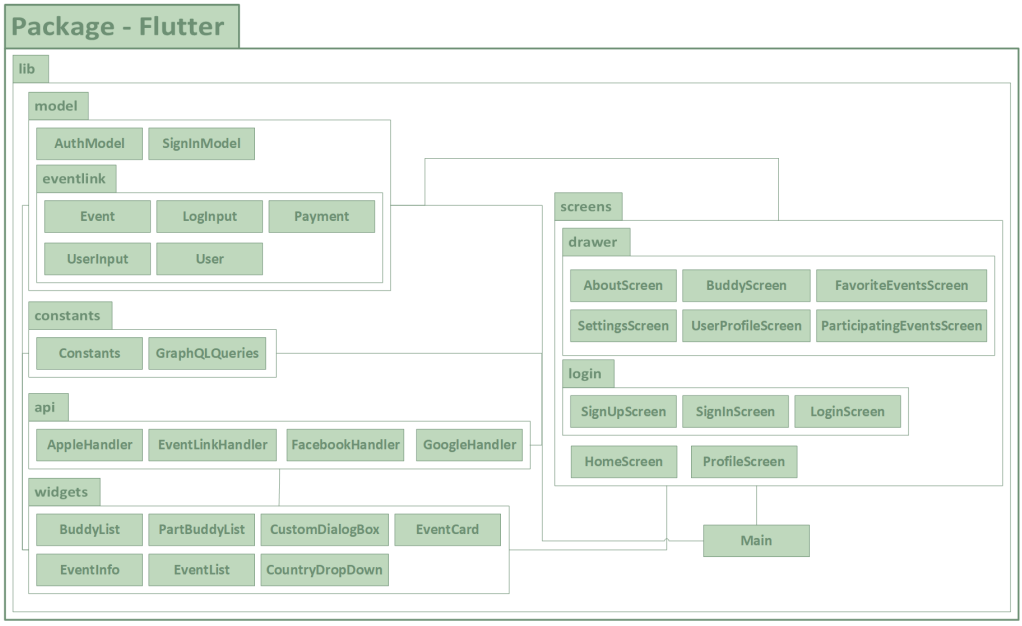


Figure 5.19: Package diagram - Flutter

Since the GraphQL API is using a specific data model to send out event data, it would only make sense to implement such a model in Flutter as well, which has been done in the *model* package. The package also contain models for the access control API as well.
Additionally, a *screens* package exists, containing all the different screens the user can navigate to within the system. It can be seen, for example, that the *screens* package has an association to the *widgets* package, because the screens are used to display the widgets, and some widgets use the screens in order to navigate the user to them.
The *api* package contains all the API handlers, a single one for EventLink and three others for the OAuth2 providers. To avoid confusion, only the classes seen in the *widgets* and *screens* package are actually Flutter widgets. The rest are ordinary Dart classes.

5.6.2 Design class diagram

The design class diagram can be seen on Figure: 5.20. All of the associations and compositions are present in this diagram, however the *model.eventlink*, *screens.drawer* and *widgets* packages have been omitted and will be shown in their own respective diagrams. This has been done to simplify the main diagram. Packages and classes that communicate with a class inside any of these three will be shown with an association. The *screens.drawer* and *widgets* packages can be seen on Appendix: B.8 and Appendix: B.9 respectively. *model.eventlink* will not have a diagram, as the data models are equivalent to the ones of the GraphQL API and datastore.

Taking a first glance at the diagram, it can clearly be seen that the associations and compositions between the classes are very complicated. The precise reasons for this will be explained in the challenges section.
Considering the API handlers at the top of the diagram, they are Singleton classes with methods for signing in and out using OAuth2. The *EventLinkHandler* class is implemented differently, as it is a GraphQL API consumer and not OAuth2.
All of the screen classes such as *SignInScreen*, *HomeScreen* and *SignUpScreen* mostly contain auxiliary methods to help building smaller widgets contained within the screen widget itself. The *GraphQLQueries* class contain premade *GraphQL* queries that are used throughout the whole application. The only thing changing in the queries are the parameters given.

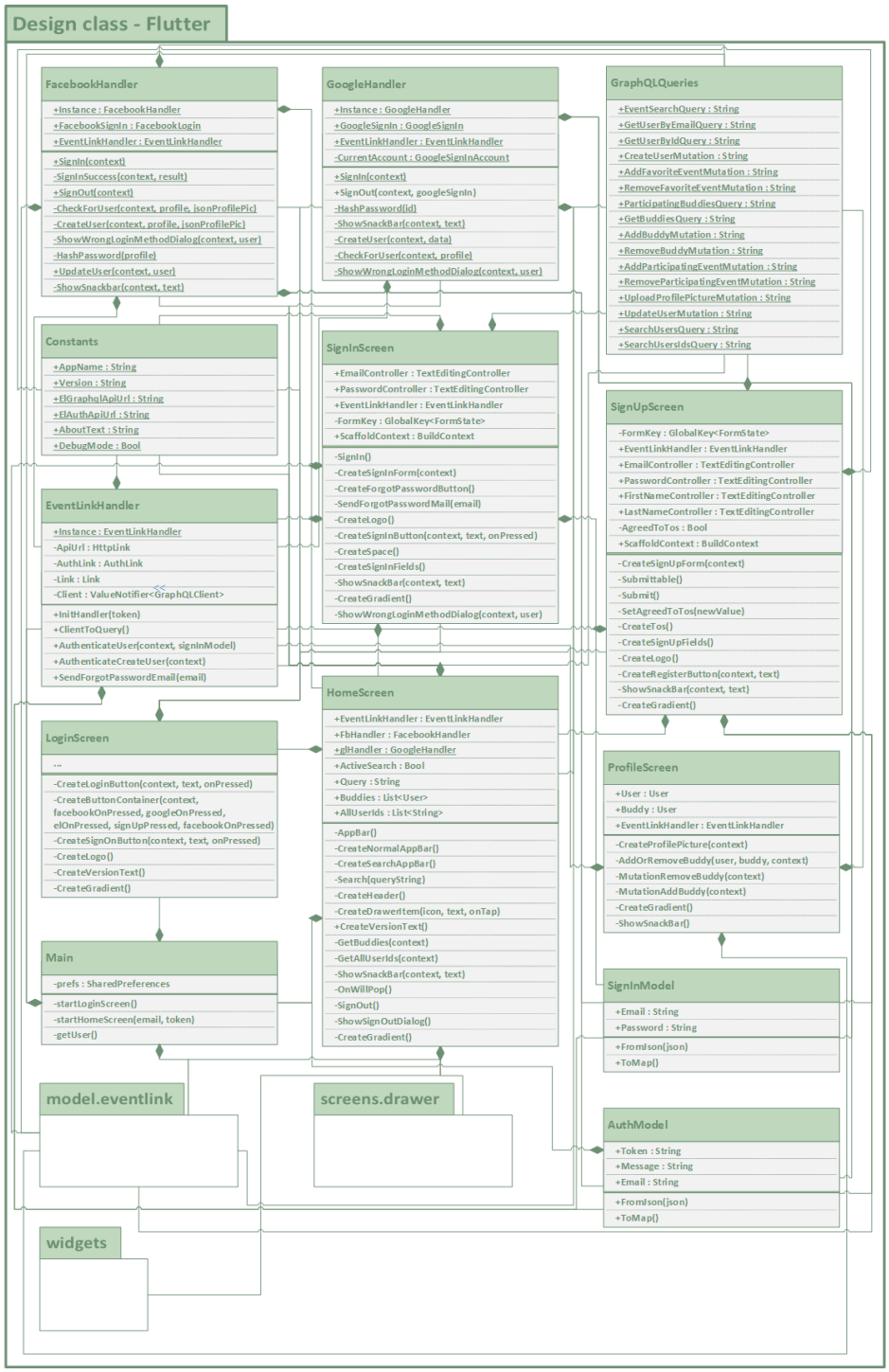


Figure 5.20: Design class diagram - Flutter

5.6.3 Implementing OAuth

The OAuth2 implementation was done with the libraries given in Section: 4.8.1. This made it simple to get access to API functionality, however a lot of logic still had to be implemented. As an example, the sign in method for Facebook OAuth2 can be seen on Figure: 5.21.

```
static Future<Null> signIn(BuildContext context) async {
  eventLinkHandler.authenticateCreateUser(context);

  final FacebookLoginResult result =
      await facebookSignIn.logInWithReadPermissions(['email']);

  switch (result.status) {
    case FacebookLoginStatus.loggedIn:
      _signInSuccess(context, result);
      break;

    case FacebookLoginStatus.cancelledByUser:
      Exception('Login cancelled by the user.');
      break;

    case FacebookLoginStatus.error:
      Exception(
        'Something went wrong with the login process.\n'
        'Here\'s the error Facebook gave us: ${result.errorMessage}',
      );
      break;
  }
}
```

Figure 5.21: Code snippet - OAuth2 Facebook SignIn method

Firstly, an asynchronous call is made to the Facebook API, logging in with read permissions for the users email. Depending on the result from this request, the method will either continue the sign in process to EventLink or throw an exception. If the login request is successful, the logic shown on Figure: 5.22 will be executed.

```
static void _signInSuccess(
    BuildContext context, FacebookLoginResult result) async {
  final accessToken = result.accessToken.token;

  _showSnackBar(context, "Signing in with Facebook...");

  /* Find information from the facebook GraphQL API. */
  var urlQuery =
      'https://graph.facebook.com/v4.0/me?fields=first_name,
      middle_name,last_name,address,email,birthday,name,id&access_token=$accessToken';
  final graphResponse = await http.get(urlQuery);

  /* Decodes the GraphQL response, so it can be used in the auth model. */
  final profile = json.decode(graphResponse.body);

  final profilePicUrl = 'https://graph.facebook.com/' +
      profile['id'] + '/picture?type=large&redirect=false';

  final jsonProfilePic = await http.get(profilePicUrl);
  final profilePicture = json.decode(jsonProfilePic.body);

  var newEmail =
      profile['email'].toString().replaceAll(new RegExp(r'\\u0040'), "@");
  profile['email'] = newEmail;

  /* Check if user is already is in DB (email) */
  _checkForUser(context, profile, profilePicture);
}
```

Figure 5.22: Code snippet - OAuth2 Facebook SignInSuccess method

The access token given by Facebook is acquired, and a query to Facebook is made, retrieving data such as first name, middle name, last name, address, email, etc, essentially data that a user of EventLink consists of, such that a user can be created in the system. Additionally, profile picture from Facebook is fetched and all the data is together put into a new method that will check whether the user already exists in the database and other practicalities. Afterwards, if everything succeeds, the user would then be signed in to EventLink.

5.6.4 Implementing EventLink main screen

A small code snippet from the main screen of the mobile application can be seen on Figure: 5.23. This piece of code is shown as it builds the *EventList* widget on the main screen that will display all the events.

```
body: Builder(
  builder: (context) => Container(
    decoration: _createGradiant(),
    child: Container(
      child: GraphQLProvider(
        child: EventList(
            query: query,
            filter: '',
            type: EventListType.Regular,
            user: widget.user,
            buddies: buddies,
            scaffoldContext: context),
        client: eventLinkHandler.client,
      ),
    ),
  ),
),
```

Figure 5.23: Code snippet - Main Screen EventList

The *EventList* widget is wrapped inside a *GraphQLProvider* widget, which in turn is wrapped in a *Container* widget. This is a very common nested widget structure in Flutter. The *GraphQLProvider* makes sure that all the GraphQL requests inside *EventList* will work. The client associated with the GraphQL API is given as a parameter as well. For the *EventList*, a search query is given, an empty filter and an enumeration that tells the list that this is a regular event list (other ones exist only showing favorite events, etc.). Additionally, the currently signed in users object is given as well. Inside the implementation of *EventList*, the following code on Figure: 5.24 will then be executed.

```
Widget _eventListView(BuildContext context, QueryResult result) {
  try {
    final events = parseEvents(result);
    return ListView.builder(
      itemBuilder: (ctx, index) {
        final event = events[index];

        List<User> partBuddies = new List();

        for (var buddy in buddies) {
          if (buddy.participatingEvents.contains(event.id))
            partBuddies.add(buddy);
        }

        return EventCard(
          event: event,
          user: user,
          type: type,
          partBuddies: partBuddies,
          scaffoldContext: scaffoldContext,
        );
      },
      itemCount: events.length,
    );
  } catch (e) {
    return Container(
      height: double.infinity,
      width: double.infinity,
      decoration: _createGradiant(),
      child: Center(
        child: Loading(indicator: BallPulseIndicator(), size: 100.0),
      ),
    );
  }
}
```

Figure 5.24: Code snippet - EventList EventListView

This code snippet shows how a list of *EventCard*s are build, using the result from the query seen in the previous diagram. The result is formatted in JSON, so it has to be parsed to the Dart event model as the first step. A *ListView* building widget is formed which creates a single *EventCard* for each event, as well as gathering required information such as participating buddies, etc. If any of this logic fails, a loading screen is shown. Failure might occur, for example, if the data has not yet reached the application.

5.6.5 Challenges

During the development of the Flutter application, a variety of different issues have been encountered. This includes problems due to asynchronous programming, visual studio code debugging not working, missing documentation, mobile data usage, display scalability, bugs in Flutter source code, OAuth2 release and test keys and much more. A select few of those will be elaborated further.

Asynchronous programming Due to the way that the GraphQL client library works, it was required to make a lot of asynchronous calls to the GraphQL API. This caused many race condition-like issues. For example, if one call was required to finish before another, but it did not, then all sorts of issues would occur due to missing data. This has accounted for several extra hours of debugging and implementation, but in the end it was possible to fix the most significant ones of those issues.

Visual Studio Code debugging An issue related to asynchronous programming is debugging in Visual Studio Code. When the asynchronous calls were not working, they were attempted to be debugged. However, for unknown reasons, Visual Studio Code would freeze most of the time when trying to debug Flutter. Sometimes the debugging would work, but most of the time it would make both the editor and application unresponsive. This issue occurred on two different computers and there was never found a solution for it.

OAuth2 release and test keys When using OAuth2s single sign on functionality, you are required to construct test keys for the associated accounts (Facebook, Google) that should be allowed to sign in through your application. The generation of those have been slightly complex and have not always worked correctly. For example, at one point in time it was only possible for one of the developers to log in through Facebook. Additionally, for a pseudo release of the application, it was wanted to generate a release key. This was however not possible, due to all sorts of requirements, including a terms of service document etc. This means, that only the two developers are able to sign in through Facebook and Google.

Display scalability Making the Flutter application scale well on arbitrary phone display sizes have been a difficult task to achieve. The application has been developed on emulators of varying size as well as physical phones, but it has not yet been successful with establishing a good display scalability. Therefore, the application only scales well on bigger phones, as the two developers of EventLink have phones with large displays (around $105cm^2$).

5.7 Logging

This section will explain the implementation of the logging system by explaining a few code examples.

5.7.1 Implementing the logging system

The implementation of the logging system was quite simple. It consists of a single Singleton class which has access to the datastore, and it also contains a single method called *Log*, shown on Figure: 5.25.

```
39 references | 0 exceptions
public override void Log(LogDb logDb, string parentName,
    string functionName, string message, LogLevel logLevel, bool isPrinting)
{
    var logMessage = "[" + DateTime.Now + " | Database: " + logDb + " | Parent: " + parentName +
                     " | Function: " + functionName + " | " + logLevel + "]: " + message;
    var logObj = new Log(parentName, functionName, message, logLevel);

    /* Always log to the system log */
    LogService.CreateLog(LogDb.System, logObj);

    /* If the log is pointing to anywhere other than the system, log there as well. */
    if (!logDb.Equals(LogDb.System))
    {
        LogService.CreateLog(logDb, logObj);
    }

    if (isPrinting)
    {
        Console.WriteLine(logMessage);
    }
}
```

Figure 5.25: Code snippet - Log method

The method takes arguments corresponding to the same as in the log data model, plus an extra boolean value called *isPrinting*, which decides whether the log is printed to the console. The method constructs a log message and then a log object, it then logs it to the *System* log collection in the database and if the log is desired for some other log collection it also gets logged to that. The various log collections that exist in the database can be seen on Figure: 5.26.

```
99+ references
public enum LogDb
{
    System = 1,     /* Logs everything that is related to the system */
    Event = 2,      /* Logs everything related to events (Crawler, fetching of events, etc.) */
    Payment = 3,    /* Logs everything related to payments (Orders, etc.) */
    User = 4,       /* Logs everything related to users (User updates, authentication, etc) */
    Statistics = 5 /* Logs everything related to statistics (Crawler statistics, user auth tries, user activity, etc.) */
}
```

Figure 5.26: Code snippet - LogDb enumeration

The *System* collection contains all the logs that are logged in the system. The *Event* log collection contain everything regarding events and so on. The *Statistics* collection contain statistics regarding crawler collection of events. As mentioned before, these collections are made such that managing the logs would be a more simple task.

5.8 General challenges

This section will describe the challenges that have occurred throughout the development of EventLink, that do not fit into the previous challenge categories. This includes issues regarding cyclic dependencies and storage of secretive information.

Cyclic dependencies In the early stages of the design phase, it was wanted for the datastore to be dependent on the logging component, such that the datastore could log every single operation it performed. However, the problem was that the datastore was already dependent on the logging component, which makes a cyclic dependency. This is obviously due to bad design, so the idea was revised and compromises had to be made. Luckily, as it was early in the design phase, nothing was implemented yet, so the impact of the issue was not significant.

Storing secretive information As previously mentioned, it has been necessary to store secretive information such as API keys, JWT keys, datastore credentials and more. It was desired to read these properties from a file, such that they could be easily accessible. Therefore, it was decided to use a *appsettings.json* file to store the data inside. A class was then created for each of the components called *ConfigurationManager* that could read values from this file. The small annoyance with this, was that if one developer updated it, it had to be sent to the other one through e-mail or similar, since it was not going to be committed to version control for

obvious reasons. Additionally, at one point in time the file got extremely large, and it would take the system a long time for it to be loaded. This has since then been fixed.

5.9 Design sequence diagrams

In order to show how system interaction has been implemented, two diagrams has been created. These take basis from the design sequence diagrams previously made for two use cases.
They will show how the different components of EventLink communicates, and not how the classes communicate directly. Therefore they are a different from ordinary UML design sequence diagrams, but due to the simplicity of the components, it was decided that this would produce a better overview. The first diagram on Figure: 5.27 will show how the user signs in to EventLink. This will be a prerequisite for the next diagram, which will show how a user searches for and participates in an event. As with the previous sequence diagrams, these will only show success criteria.

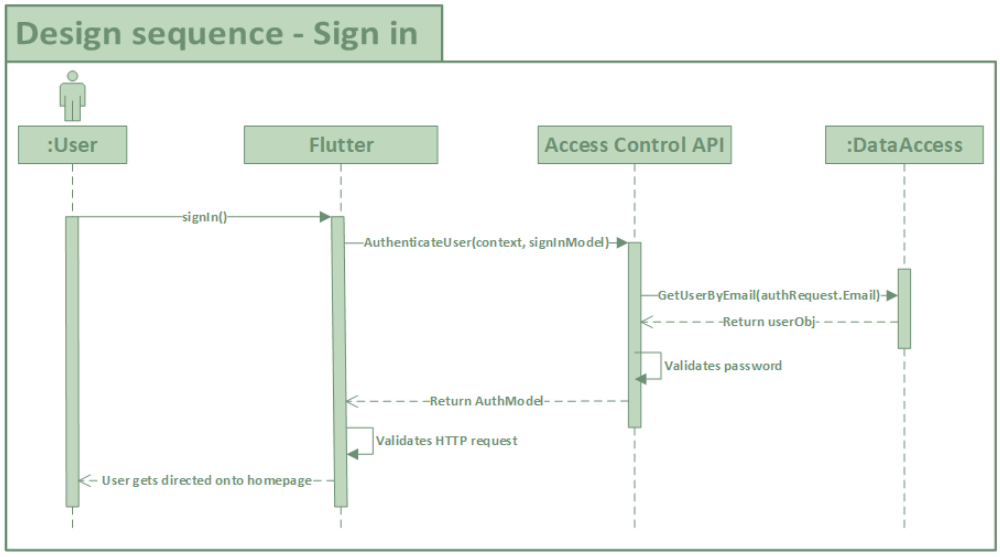


Figure 5.27: Design sequence diagram - Sign in

The diagram starts with the user choosing to sign in within the Flutter application. Then the application sends an *AuthenticateUser* request to the access con-

trol API. Once this is received, the access control API asks the datastore (here called DataAccess due to implementation) for a user object with a matching e-mail that the user gave when signing in. When the user object is retrieved, then the access control API validates the users password. Once the user is validated it returns an *AuthModel* (in practice a *AuthResponseModel*) to the application. If the request is valid, then the user gets redirected onto the main screen of EventLink, because the user was then successfully authorized. The flow was implemented this way to both make sure that the given user actually exists, plus having the right credentials.

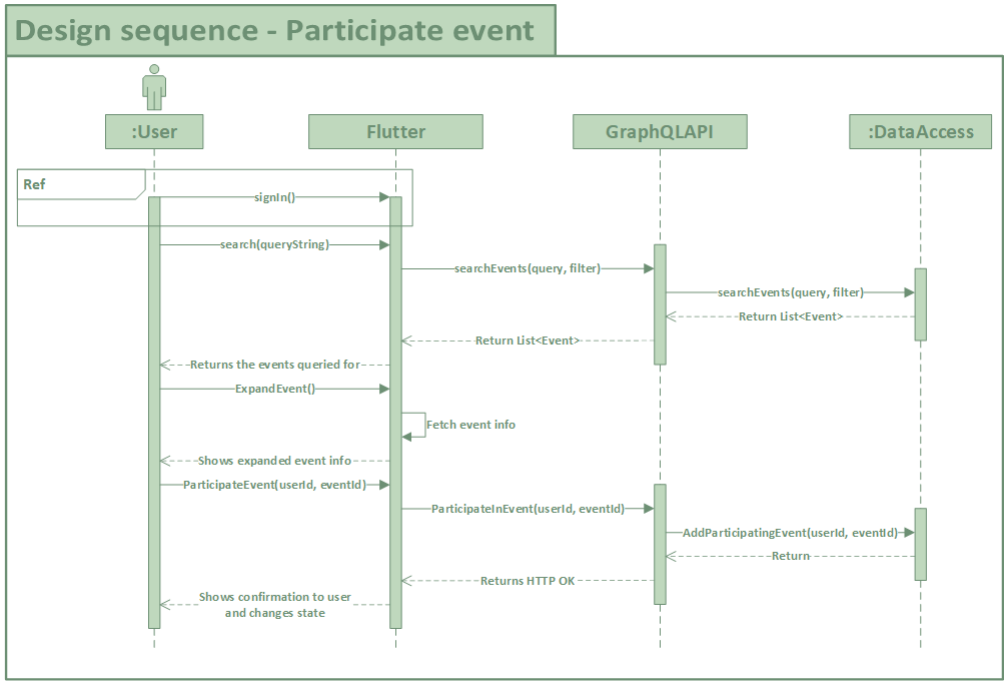


Figure 5.28: Design sequence diagram - Participate event

The diagram for event participation can be seen on Figure: 5.28. As stated previously, the sign in sequence is a prerequisite for this sequence as shown with the *ref* block. Firstly, the user uses the *search(queryString)* method to search for an event in the Flutter application. This method then executes a query with the *queryString* as parameter to the GraphQL API, specifically *searchEvents(query, filter)*. The GraphQL API then gives the query further to the datastore which then returns a list of events, reaching the Flutter application in the end.
The user can then interact with the events, and in order to participate in a

event, the event has to be expanded. This is done locally within the Flutter application with the user pressing on a button. Once expanded, the user can click to participate in the event, which then is done by making a query with a mutation to the GraphQL API. Once the mutation is successful, it will return a *HTTP OK* status, which Flutter will react to, and change the state of the screen in order to give visual feedback to the user.

5.10 Summary

This chapter has shown how the system is implemented by the help of a deployment diagram, which shows the domain of the virtual machines, how they communicate as well as where each component resides.
Each of the components have been elaborated with a package diagram, a design class diagram, an important feature and encountered challenges.
For the MongoDB component, the implementation of the event database service has been explained and the implementation of event parsing has been explained for the event crawler. Additionally, for the GraphQL API, the implementation for the queries and mutations have been described and for the access control API it was JWT. Lastly, for the Flutter application, the implementation of OAuth2 and its main screen was shown. The implementation of the logging system was explained as well. General challenges encountered in the implementation was also explained, which was cyclic dependencies and storing secretive information. Then two design sequence diagrams were build for the *Sign in* and *Participate event* use cases.

The implementation phase has been a success. There was not experienced any major issues, only lesser significant challenges regarding the mobile application, as it quickly became unmanageable. Despite its poor implementation, the application works both seamlessly and intuitively. The backend supports all the features that were required and it has been designed such that it is easily maintainable for future developers.

Chapter 6

Test & Performance

This chapter will document the testing of EventLink. This will be done through a variety of different testing methods, starting out with unit and integration testing. These methods will primarily be used to test the services in the system, such as the datastore *EventService*. Additionally, monkey tests will be performed on the mobile application and at the end user and accept tests will be held as well.

6.1 Unit tests

To make sure that EventLink is working as intended, every component has got a test suite. Each of these test suites holds a considerable amount of different unit tests, used to test the core functionality. Due to the service oriented design of the components, it was decided to primarily test the functionality of the services, as these include the primary functions.
The approach to unit testing was done such that once some new functionality was developed, for example a method in a service, it would briefly be manually tested, then unit tests would be made for it. If the service was included in a milestone, the whole test suite would be tested at the end of that milestone.
The test cases are not descriptive test cases, as it was decided that those were not needed for the services. The datastore software component, event crawler and access control API have been tested. Unfortunately there was no time available to test the rest of the system. This is the reason why the datastore was the first component to be tested, as the other components are dependent on it. This way, it is assured that the datastore works in all of the components. The test cases for the datastore *EventService* can be seen on Table: 6.1.

Id	Test case	Result
TC001	GetDocument_FoundDoc	OK
TC002	GetDocument_IdNull	OK
TC003	GetDocument_IdNotFound	OK
TC004	GetDocuments_FoundDocList	OK
TC005	GetDocuments_DocListNull	OK
TC006	CreateDocument_DocCreated	OK
TC007	CreateDocument_DocProviderEventIdNull	OK
TC008	CreateDocument_DocAlreadyExists	OK
TC009	CreateDocument_CheckInvalidIdFormat	OK
TC010	UpdateFormat_DocWasUpdated	OK
TC011	UpdateFormat_DocWasNotFound	OK
TC012	DeleteDocument_DocWasDeleted	OK
TC013	DeleteDocument_DocIsNull	OK
TC014	DeleteDocument_DocIdIsNull	OK
TC015	DeleteDocument_DocNotFoundWrongId	OK
TC016	DeleteDocument_DocNotFoundWrongProviderEventId	OK
TC017	DestroyDocument_DestroySingleDoc	OK
TC018	DestroyDocument_DestroyNullDoc	OK
TC019	DestroyDocument_DestroyNullStringId	OK
TC020	CheckEvent_NullDAObject	OK
TC021	CheckEvent_EssentialData_Name	OK

Table 6.1: Test case - Datastore EventService

All of the test cases have been implemented such that they should result with *OK*, only and only if their result is correct. This goes for both the negative and positive tests. Considering the first three test cases of the table, they all test *read* functionality. **TC001** tests whether an existing document (event object) can be found, **TC002** tests what happens if one tries to get a document with an id of null, and **TC003** tests what happens if the given id is not found in the database. The other test cases consists of similar tests but for the remaining CRUD operations. The rest of the test cases can be seen in Appendix: B.4.

As an example, the implementation of **TC001** can be seen on Figure: 6.1.

```
[TestMethod]
0 references | 0 exceptions
public void GetDocument_FoundDoc()
{
    Event e = null;
    Event e1 = null;

    try
    {
        e = eventService.GetEvent("507f191e810c19729de860ea");
    }
    catch (Exception ex)
    {
        Assert.Fail(ex.Message);
    }

    try
    {
        e1 = eventService.GetEvent("507f191e810c19729de860ae");
    }
    catch (Exception ex)
    {
        Assert.Fail(ex.Message);
    }

    Assert.IsNotNull(e);
    Assert.IsNotNull(e1);
}
```

Figure 6.1: Test case code - EventService FoundDoc

The method attempts to retrieve two existing events (these are created when the test suite is ran) and it then assures that they are non-null. It is a simple test case, but it is worth to remember that the service is implemented such that if anything goes wrong, it will throw an exception, which is also the reason why the test case would fail if that was the case.

6.2 Integration tests

Due to the service oriented design of the components, integration tests have been done through the component unit tests. For example, since the access control *AuthService* makes use of the datastore software component, testing *AuthService* tests its integration with the datastore. This has been decided to be feasible, as it tests the integration between the components, and is therefore deemed as integration tests. Each time a new feature has been implemented within a component, it must be redeployed to its server. Once deployed, the integration can be tested through the unit tests and thereby verify that no harm has been done.

On the deployment diagram found on Figure: 5.1 the integration between the components can be seen. This should give an overview on which part of the system is dependent on each other. Additionally, the integration tests for the *AuthService* can be seen on Appnedix: B.24.

6.3 Monkey tests

To check the sturdiness of a mobile application, a monkey test can be used. A monkey test is a test which gives the application a series of random touch inputs. This is done to make sure that the application does not crash, even under considerably large amounts of touch events. The test would ensure that the application is capable of handling large amounts of random input.
Unfortunately, as the application is created with the use of Flutter, no monkey test frameworks or libraries are available at this point in time. This is most likely due to Flutter being a relatively new and young framework.

6.4 User tests

Four different people were asked to try out the application and give their thoughts on the design and experience with EventLink.
The tests found on Table: 6.2, Table: 6.3, Table: 6.4 and Table: 6.5 are made by giving the user a brief description of what they should find or do within the application. The test will describe what the user does and at the end describe what they thought of the application. The user tests was created after the implementation of the mobile application, and should therefore help with describing possibilities for further development.

Test id	UT01
Test case	Find a new buddy.
Brief description	Sign in to EventLink, find a user that you are not friends with and add them as a friend.
Users process	The user does not have a profile, and therefore cannot sign in. The user tries to create a profile with Facebook, but is denied. Then they create a profile with EventLink. The user navigates to the drawer and clicks on buddies. A list of users is shown, and the user clicks on one which they are not friends with and clicks "add buddy".
Users thoughts	1. It would be nice to be able to sign in with Facebook. 2. The drawer is easily found. 3. The buttons in the drawer makes sense, nothing unnecessary.

Table 6.2: User test - Add a buddy

Test id	UT02
Test case	Find and participate in an event.
Brief description	Sign in to EventLink, find the event "Thriller" which is performed on the 18th of January and participate in it.
Users process	The user tries to sign in with Facebook. They are not allowed to do this, so they try Google. This is not allowed either and they are instructed to use the EventLink sign up. The user tries to scroll down to the event, but there is too many and it takes too long. The magnifier glass is spotted and the event is searched for. Once the event is found, the user clicks on the learn more button and clicks participate.
Users thoughts	1. It would be nice to sign in with Facebook or Google. 2. It would be nice to display a description on the event.

Table 6.3: User test - find and participate in event

Test id	UT03
Test case	Change your name within EventLink.
Brief description	Sign in to EventLink and change your name.
Users process	The user signs up with EventLink, and proceeds to the main screen. The user spots the drawer rather quickly and opens it. The profile button is clicked and the name is changed. Upon clicking save changes nothing happens.
Users thoughts	1. Saving the changes does not work.

Table 6.4: User test - Change name

Test id	UT04
Test case	Favorite events.
Brief description	Sign in to EventLink and add a couple of events to your favorites.
Users process	The user signs up with EventLink and is presented with the main screen. The user clicks the learn more button of the event. Upon further inspecting the events, the user notices the small heart icon. Upon clicking on the heart, the user favorites the event.
Users thoughts	1. The favorite icon could be more eye catching. 2. The events lack description.

Table 6.5: User test - Favorite events

As seen on the tests above the most frequent comments are regarding the OAuth2 sign in, the lack of information on an event and that saving a profile changes does not work. These will be elaborated further upon below.

OAuth2 As previously mentioned, the OAuth2 sign in does not work for other people than the developers of EventLink. This is because there is not generated a release key for Facebook or Google. There is used a test key, and therefore no other profiles are allowed to use this.

Lack of event description This is a problem with Ticketmasters data. It does not provide event descriptions even though the data model says so. The events created with Eventful has descriptions.

Saving profile changes The problem was located. The functionality was created but the method was never called upon clicking on the save changes button.

6.5 Accept test

The acceptance test will be based on the amount of requirements and MVPs that are currently satisfied. Each of the components will be examined below.

MongoDB database The MongoDB software component supports basic CRUD operations on the four data models, and the database contains a variety of collections including the ones in the MVP, hence the MVP (Table: 3.6) is satisfied. In addition to that, requirement **F01** and **F06** (Table: 3.3) are satisfied as well.

Event crawler The event crawler satisfies its MVP (Table: 3.8) by being able to fetch data, parse it, store it in the datastore and do this within a given time interval. Additionally, it also satisfies requirement **F07** from the requirements specification.

GraphQL API The GraphQL API does not directly satisfy its MVP (Table: 3.9) as it does not contain CRUD for events and users. This is however not an issue, as not all operations from CRUD are needed, so the MVP is slightly outdated. Regardless, the GraphQL API satisfies all of its MVP. Additionally, it satisfies requirement **F05**, **F08**, **F09**, **F10**, **F11**, **F12**, **F13**, **F14**, **F15**, **F16**, **F17**, **F18**, **F22**, **F23**, **F24** and **F25** from the requirements specification.

Access Control API The access control API indirectly satisfies all of its MVP (Table: 3.10), since the OAuth2 functionality was moved to the Flutter application. Additionally, it satisfies requirements **F02**, **F03** and **F04** from the requirements specification. **F04** can be considered to be satisfied indirectly, as it is not possible for the user to request to be signed out, but will be when their JWT token expires.

Mobile application The Flutter mobile application almost satisfies all of its MVP (Table: 3.11). The only element that is not satisfied is the OAuth2 sign on with Apple, which is okay as it was scrapped. In addition to that,

requirement **F26** is satisfied here as well, as the application lets a user change their password.

As a small summary, the requirements **F01-F18**, **F22-F26** are satisfied. Requirement **F19** and **F20** will not be satisfied as the functionality for purchasing tickets was omitted. **F21** has not been implemented. Whether a component satisfies an arbitrary requirement can be argued against, for example, does the GraphQL API really satisfy requirement **F09**? Yes, it does, as the API is able to present the required data for user profiles.

RID	Datastore	Crawler	GraphQL	Access Control	Flutter
F01	×				
F02				×	
F03				×	
F04				×	
F05			×		
F06	×				
F07		×			
F08			×		
F09			×		
F10			×		
F11			×		
F12			×		
F13			×		
F14			×		
F15			×		
F16			×		
F17			×		
F18			×		
F19					
F20					
F21					
F22			×		
F23			×		
F24			×		
F25			×		
F26					×

Table 6.6: Requirements satisfaction matrix

An overview over requirement satisfaction can be seen on Table: 6.6. Note that only functional requirements have been considered for the accept test, as these are deemed to be the most important ones. The rest of the requirements will be assessed below.

U01 This is not satisfied, due to the display scalability issues explained in Section: 5.6.5.

U02 This is satisfied, but there is room for improvement.

U03 This is satisfied, which can be seen in Section: 6.4.

U04 This is satisfied, as the user can contact support through the mobile application.

R01 This is not possible to measure as the system is not under any load. However, at the time of writing, the system has been deployed for 2 months with zero downtime.

P01 This is not possible to measure either, but the system has been designed to use as little resources as possible.

P02 This has not been measured directly, but significant stalling has not been experienced.

S01 This is satisfied.

I01 This is satisfied as well.

L01 The system does not comply to GDPR, as there is no terms of service that explains the acquired data. However, this is not required at the moment, as the system is not in production.

As the majority of the requirements and MVPs are satisfied, the implementation is deemed to be successful.

Chapter 7

Tools

This chapter will introduce and describe the plethora of software applications and other relevant tools that has been used during the development of EventLink and the writing of this thesis. The tools were used for project management, version control, software development, testing, remote access, communication and much more.

7.1 GitKraken Glo

Glo is a software developed by GitKraken and has been the primary tool used for managing the project. Glo is a collection of shared boards that can be used to manage tasks, organize calendar deadlines and in general help keeping a project organized. Glo has been used as an issue board to keep track of development tasks and milestone achievements.

There has been created a board for each component of EventLink along with a general board for tracking other relevant elements such as meeting notes, storing of diagrams, useful links etc. Each of these boards contain two essential lists, a MVP and a NTH list. These lists contain notes regarding functionality considered to in the MVP and NTH respectively. This has given a clear and easy to understand overview of the requirements and expectations of each of the projects and therefore helped with achieving their respective milestones. [Git19b]

A snippet of the access control API board can be seen on Appendix: B.10.

7.2 GitHub

GitHub has been used as the primary version control repository for the source code for EventLink. GitHub provides hosting for software development version control using the Git version control system. This allows for easy to use version control, backup, code branching, issue tracking, change history and more. [Git19a]

GitHub allows their users to create organizations. An organization on GitHub is a collection of related, version controlled repositories, that can be used if a single repository is not sufficient. In the development of EventLink, a GitHub organization was created, since two separate but related repositories were needed. One for the backend system and another one for the Flutter mobile application.

GitHub has primarily been used to give an overview of the continuous changes in the codebases, which has made it easier to detect failing code and errors. Feature branches have been used when new features have been developed to avoid merge conflicts as much as possible. Additionally, GitHub has been the primary backup provider for the source code as well. [2]

7.3 Visual Studio & Visual Studio Code

Visual Studio and Visual Studio Code have been the primary development environments used. Visual Studio is a full-fledged IDE whereas Visual Studio Code is a lightweight code editor. Both of the editors are created by Microsoft.

Visual Studio is heavily based around the .NET ecosystem and was therefore a clear choice when it came to finding a suitable development environment for the backend system. Visual Studio Code has been used for creating the Flutter mobile application, as it is lightweight and has great support for both Dart and Flutter. [Mic19b] [Mic19c]

Both of these IDEs contain a lot of useful functionality, which has been a great help in the general development of both the Flutter application and the backend.

7.4 MobaXTerm

Since EventLink consists of five different servers, it was necessary to have a way to organize, manage and connect to these in a simple way. This has been achieved by the usage of MobaXTerm. MobaXTerm is an advanced terminal emulator for Windows which provides a GUI to manage terminal sessions either over the internet or over a local network. MobaXTerm provides various network tools to the user such as SSH, FTP etc. Additionally, MobaXTerm gives the user a Linux-like terminal experience on a Windows machine.

MobaXTerm has given a single place to hold all the server connections, as well as giving a quick way to upload data to the servers. This has been very useful to upload the DLL files that the servers have to host for EventLink. In MobaXTerm, it is possible to save connections with their corresponding SSH keys etc., which in this case has been a great help. With a single click, you're logged in to your desired server. [Mob19]

7.5 Robo 3T

Robo 3T is a native and cross-platform database management tool for MongoDB. Robo 3T has been used to create and manipulate data on the database server.

Throughout the development, Robo 3T has been used as the primary tool to manage the database. This includes tasks such as the creation of database collection, wiping of said collections, data checking and much more. This tool has been a necessity, due to the need to store and manage the data within the system together with being an easy and simple way to connect to the database. [Lab19]

7.6 Freenom

Freenom is a registry operator that administers the .ml and .tk top level domains. The domain name *eventlink.ml* has been acquired through Freenom. This has been done for the ease of using a domain instead of IP-addresses. Since there was no need for .com addresses, it was chosen to go with the free .ml domain. If EventLink was to be used as an actual production system, it would be more professional to go with a .com domain instead of .ml, however for developmental purposes a .ml domain works just fine. [Fre19]

7.7 GraphQL

GraphQL provides a variety of tools for the development of GraphQL APIs. In the development of EventLink there has been used two such tools. The first one being GraphiQL which helps creating and debugging queries and mutations, and the latter one being Voyager which helps to get an overview of the underlying GraphQL data model of the system.

7.7.1 GraphiQL

To be able to query and manipulate data with GraphQL, GraphiQL has been issued as a simple interface. This is GraphQLs own in-browser IDE which allows the user to see which parameters the queries and mutations accept. GraphiQL uses intelli-sense to help the user with creating queries and mutations. Additionally, it shows the query results in a organized and understandable way.

This tool was chosen due to the fact that it is a native tool from the GraphQL ecosystem. GraphiQL has been used a lot to state if the queries returned what they were supposed to, to check if the right data was queried and correctly returned etc. This has been one of the most important tools in the development

of EventLink, since the GraphQL API is the heart of the project. GraphiQL has been an essential player in the manual testing of the API as well. [Gra19].

7.7.2 Voyager

Voyager is another tool from the GraphQL ecosystem. Voyager is a simple tool that shows a dynamic diagram of the underlying datamodel for the system at hand. Since the diagram is interactive, it allows for visual debugging and error checking of data relationships. Voyager has been used to check whether the programmed data model was interpreted as wanted by the system.

A snippet of EventLinks datamodel can be seen on Figure: B.11. Additionally, the interactive Voyager diagram can be found at Appendix: [4] but requires a valid JWT token to be seen.

7.8 Insomnia

The most essential service of EventLink is the GraphQL API. To ensure the performance and reliability of the API, it was necessary for it to be tested thoroughly. Since GraphiQL is hosted on the same server as the API resides on, it was necessary to find another tool to query the API from outside the same network.

Insomnia is a Windows application used for managing all kinds of HTTP requests. It has built in tools for REST, GraphQL and much more. The user is able to store and save the requests that have been run. This makes it easy to go back and test the same requests after some potential code changes, etc. Insomnia offers different tools to both GraphQL and HTTP. This was chosen over the common PostMan, due to the support for GraphQL. Insomnia was primarily used to test query functionality after redeployments, code changes, etc. [Inc19]

7.9 Visio

Visio is produced by Microsoft and is a part of the Microsoft Office package. Visio is a modeling tool used to create diagrams for all uses and purposes. This means that it is not used for modeling software only, but can be used to model all sorts of interactions.

Visio has been used to create all the diagrams within this thesis. Visio was chosen as the diagramming tool due to the modern look of the diagrams and seemingly ease of use. Visio Professional is the specific version that has been used. [Mic19a].

During the usage of Visio there has been encountered a few problems. Visio has problems running if some specific programs are running on the same machine. This has been narrowed down to the *Nahimic* audio program. If this program was running, then Visio would exit.

7.10 Overleaf

Overleaf is a online LaTeX editor that allows for easy editing and collaboration on projects and has therefore been used to write this thesis. Overleaf includes a variety of useful features such as multiple users editing the same document, document history, document reversion and more. Overleaf has been used due to the freedom of the document configuration as well as due to its ease of use. [Ove19]

7.11 Spectrum

Spectrum is a community and communication platform, that was requested by Martin, the external advisor, as a way of communicating. Spectrum helps index all communication, supports real-time messaging and gives the opportunity to join different communities.

Spectrum has not been used a lot, but has been the primary link of communication with the external advisor, whenever questions and other related elements have arisen. The Spectrum chat can be found at Appendix: [3]. [Spe19]

Chapter 8

Result

This chapter will summarize the current state of both the backend system components and the mobile application. This will discuss elements such as working functionality, non-working functionality, requirements coverage and MVPs. Additionally, further development will be discussed with the focus on what would be the next steps if EventLink were to be developed further.

8.1 Backend system status

The general status of the backend components is seen to be great. As mentioned in the accept test in Section: 6.5, the system satisfies most of the requirements as well as MVPs. The system contains a lot of useful functionality that can easily be developed further upon due to the design and implementation choices. It is believed that if EventLink were to be transferred to other developers, that it would simple to understand and manage the source code. Therefore, the backend system has been a success.

8.2 Mobile application status

The application works without any major issues, except for the ones mentioned in Section: 6.4. One of the biggest problems with the mobile application is its implementation, which is not visible to the user. However, the architecture of the Flutter code could be improved tremendously.
The application has been deemed a success, since the application is reliable and is intuitive to use, also shown on the user tests. The MVP has been fulfilled except for two elements. These would be *The system must contain a button for buying a ticket to an event* and *the system must contain a button for signing in using Apple OAuth2*. Recall, the MVP for the GUI can be found on Table: 3.11. The reason why these have not been implemented have been explained previously.

8.3 Further development

A couple of critical features could be implemented further, as well as some NTH features. First, the critical features will be shown, then NTH. It is important to note that these are not critical in order for the system to function, but more to ensure a better user experience as well as better ways to handle some functionality.

Data validation Currently, most of the data that is exchanged between the components are not validated in any way, only a few important data fields. The system could be improved by increasing the amount of data validation that happens both in the mobile application and in the backend.

Increased security As of now, JWT, OAuth2 and user login are the only security measures in the system. This could be improved with elements such as e-mail or phone number validation, encrypted communication between the components and more.

JWT refresh tokens As stated previously, JWT refresh tokens, should be implemented in order to ensure that a user is not signed out in the middle of using EventLink.

Better display scalability As previously mentioned, the scalability on different devices (especially smaller devices) is not the best. EventLink works best on bigger devices. So before taking the application into production, better device support and display scalability should be achieved.

Next up is the NTH features. These would give the user an overall better experience using the mobile application.

Calendar integration In order to get a better overview of events, a calendar integration could become useful. This could integrate with the users device calendar, in order to see if they are available to participate in a specific event on a specific date.

Geo-location If a user is on the go with the application, geo-location services could be useful. This could check where the person is located and show events close by. This should also locate the country the person is in. For example, if the user is traveling, it would show events in the given country.

Open/closed profiles A open/closed profile could be comfortable for a user. If a user searches for a buddy with an open profile, the user should be able to see all the events the buddy participates in etc. If the profile is closed, the user cannot see anything other than the picture and the buddies name. This could perhaps also be used to be anonymous in the application, such that the user would not show up in search results.

Recommendations The system could provide recommended events to the user, based upon the events the user participates in. This could be done through the usage of machine learning or just simply by creating some statistical functions which check on the users past event genres.

Filtering A filter could be useful to implement. This could be done in the drawer menu on the main screen. This should give the user a possibility to search for specific locations, event genres and so on.

Better buddy system To upgrade the buddy system, a chat system could be implemented. This should make it easier for a user to ask their buddies

along to an event. There could also be created a way to check which friends the user has on social medias, in order to recommend them to the user. Additionally, a person should have to accept a buddy request if another person wants to be their friend.

Buy tickets through the application A feature previously described in the analysis chapter, is to be able to buy tickets through the application. This was however later omitted. It ss still a relevant feature, so it should be implemented in the future.

Chapter 9

Conclusion

At the beginning of the project, an analysis phase was entered in order to create a foundation for solving the problem statement at hand. During this phase, a sea of possibilities were encountered, but in the end a single solution was found consisting of a mobile application. The application would provide a single point of entry for browsing all kinds of events. To support this application, a backend system was needed in order to gather event information. Then, necessary technologies were chosen in order to accomplish this. A requirements specification was created together with potential users and an initial prototype was created as well. The analysis phase made a great foundation, without any complications, for entering the design phase.

In the course of the design phase, the gained knowledge from the analysis phase was built further upon. The design phase did not feel like an ordinary design phase, as more time was spent on researching how the chosen technologies worked in practice, rather than designing the system. This is likely due to the simplicity of the individual components, as looking at them one by one, they are not very complex. Therefore, analysis class diagrams and a deployment diagram were enough to get the gist of their design. Additionally, design patterns were chosen to base around the architecture, such that each component would be manageable and sound. The initial prototype was redesigned from concept to a more real world application. Even though the design phase did not feel ordinary, does not mean it was a failure. In the end, most of the needed information was

gained, such that the implementation phase could start.

In the implementation phase EventLink was implemented and deployed to a single server containing five virtual machines. The virtual machines would host the components, which consists of a MongoDB database, an event crawler, a GraphQL API, an access control API and lastly a VSFTP server. This was the first major milestone that was reached, as now it was possible to see how the designed system worked in practice. The first major issue was encountered when implementing the Flutter mobile application. Large amounts of unmanageable dependencies was increasing in the source code and it kept getting worse as development went on. This could possibly have been prevented, by spending more time in the design phase, designing and understanding how Flutter worked in practice. Despite the problem with Flutter the implementation of the other components went well and turned out to be simple and manageable systems as desired.

The testing of the system components went well as well, which was done with unit and integration testing. Unfortunately, unit and integration tests were not created for all components due to time constraints, but this is okay as the most important components such as the MongoDB software component was tested. In the end, $88,46\%$ of the requirements were satisfied, whereas the rest were either omitted or not implemented. In addition to that $92,86\%$ of the MVPs were met. The reason that not 100% of the MVPs and requirements were met, is due to some of the requirements was scrapped during the development. As all of the tests have been succeeding verifying system logic and no major issues have been encountered, the development of EventLink has been deemed a success. This is enforced by the user tests as well, which was done without any major issues.

In the introduction two problems were introduced, attention span and usage of applications, and it was said that EventLink would solve these problems. It can now be said, that EventLink solves these problems by being a single application that is intuitive enough to use, such that it advocates for minimal usage of it. This is achieved both with the user experience of the application and the collection of events from various event providers.
The result of the project is satisfiable, however could have been improved with more emphasis on understanding Flutter. Additionally, more communication with IT Minds could possibility have improved the development as well, but as the system was very clear, they were not used that much, which is unfortunate. There is a tremendous room for further development of both the mobile application and the backend system, for improving the concept of EventLink as a whole. In the end, it is believed that EventLink is a product with great potential, but needs the last refining bits before it is ready to be released into the world.

Appendix A

Links

[1] EventLink Adobe XD UI Prototype
`https://xd.adobe.com/view/05ae69ed-486c-44ea-5688-0a77ae2504f5-739f/`

[2] EventLink GitHub Organization
`https://github.com/Event-Link`

[3] EventLink Spectrum chat
`https://spectrum.chat/eventlink`

[4] EventLink Voyager
`http://api.eventlink.ml/ui/voyager`

Appendix B

Diagrams, Figures & Tables

B.1 Development Analysis

B.1.1 Milestone plan

Day	Activity	Milestone start	Milestone en	Notes
02/09-2019	Analysis of project and datastore.	Datastore		
06/09-2019				
09/09-2019				
13/09-2019	Exception handling in datastore. Start unit tests.			First meeting with advisor Daniel Kolditz.
16/09-2019	Finalize implementation of unit tests for datastore.		Datastore	
20/09-2019	Analysis of data collection.	Data collection		
23/09-2019	Start unit tests of data collection.			First meeting with IT Minds advisor.
27/09-2019	Finalize implementation of data collection unit tests.		Data collection	Thomas in Germany for concert.
30/09-2019				
04/10-2019				
07/10-2019				
11/10-2019				
14/10-2019				
18/10-2019				
21/10-2019				
25/10-2019				
28/10-2019				
01/11-2019				

Figure B.1: Milestone plan snippet

B.1.2 Use case descriptions

Id	UC02
Use case name	Sign in
Scope	EventLink
Level	User goal
Primary actor	User
Description	A user signs into the system.
Stakeholders and interests	**EventLink user:** Wants an intuitive sign in screen, so they can sign in easy and quickly. **Company:** Wants the users to sign in and use the system as much as possible.
Preconditions	The user has access to the system and has already performed UC01.
Postconditions	The user has been signed in to the system and can now use it.

Table B.1: Brief use case - Sign in

Id	UC03
Use case name	Deactivate user
Scope	EventLink
Level	User goal
Primary actor	User
Description	A user deactivates their account in the system.
Stakeholders and interests	**EventLink user:** Wants a way to deactivate their account if they are no longer going to use the system. **Company:** Wants to give the user the freedom to deactivate their account at any given moment.
Preconditions	The user has access to the system and has already performed UC02.
Postconditions	The users account is now deactivated and they can no longer sign in.

Table B.2: Brief use case - Deactivate user

Id	UC04
Use case name	Participate event
Scope	EventLink
Level	User goal
Primary actor	User
Description	A user participates in an event.
Stakeholders and interests	**EventLink user:** Wants to be able to participate in a given event in the system. **Company:** Wants users to purchase tickets and participate in events from EventLink to grow the platform and get recognized.
Preconditions	The user has access to the system and has already performed UC02.
Postconditions	The user is now participating in a event.

Table B.3: Brief use case - Participate event

Id	UC05
Use case name	Find buddy
Scope	EventLink
Level	User goal
Primary actor	User
Description	A user performs a search for users, selects a user and adds them as a buddy.
Stakeholders and interests	**EventLink user:** Wants to connect with their friends in the system, so that they can plan to go to events together. **Company:** Wants to give the users the opportunity to connect with other users and hopefully create social relationships.
Preconditions	The user has access to the system and has already performed UC02.
Postconditions	The user now has a buddy.

Table B.4: Brief use case - Find buddy

B.1.3 System sequence diagram

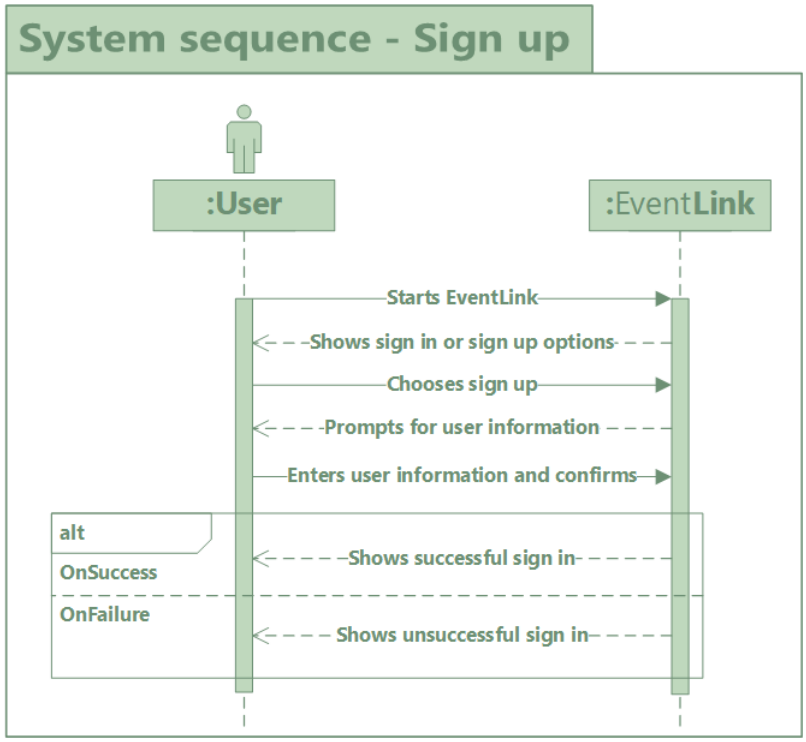


Figure B.2: System sequence diagram - Sign up

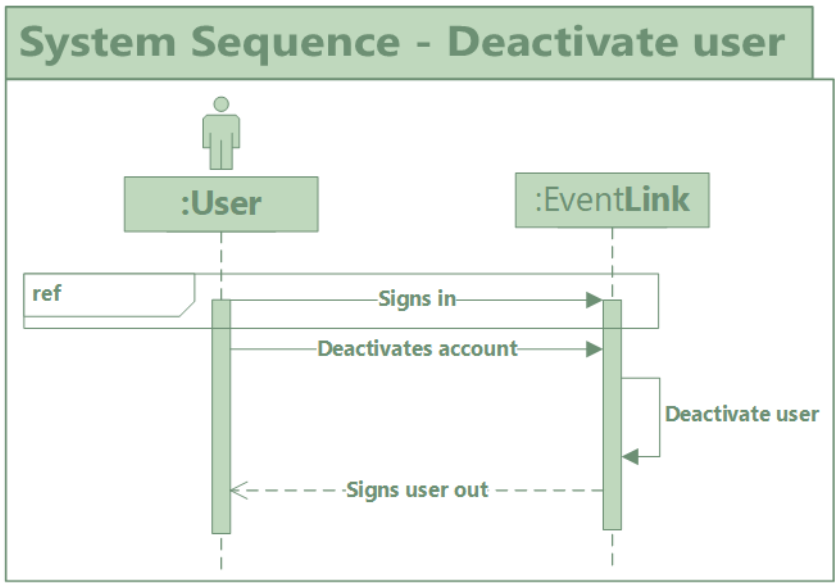


Figure B.3: System sequence diagram - Deactivate user

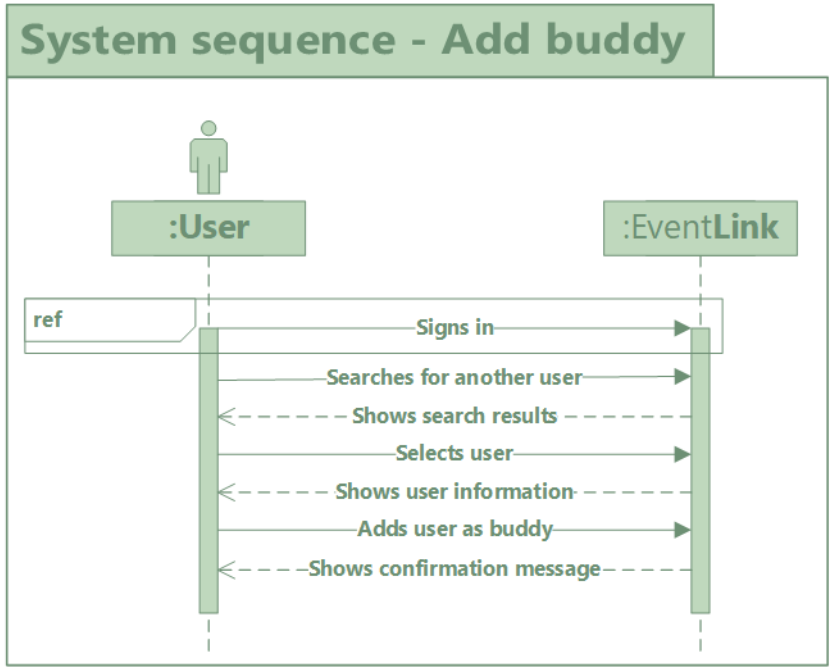


Figure B.4: System sequence diagram - Add buddy

B.1.4 Risk matrix

Risk Matrix	Minor	Moderate	Major	Critical
76-100%	4	5	6	7
51-75%	3	4	5	6
26-50%	2	3	4	5
0-25%	1	2	3	4

Table B.5: Risk matrix

B.2 Design

B.2.1 Ticketmaster data model

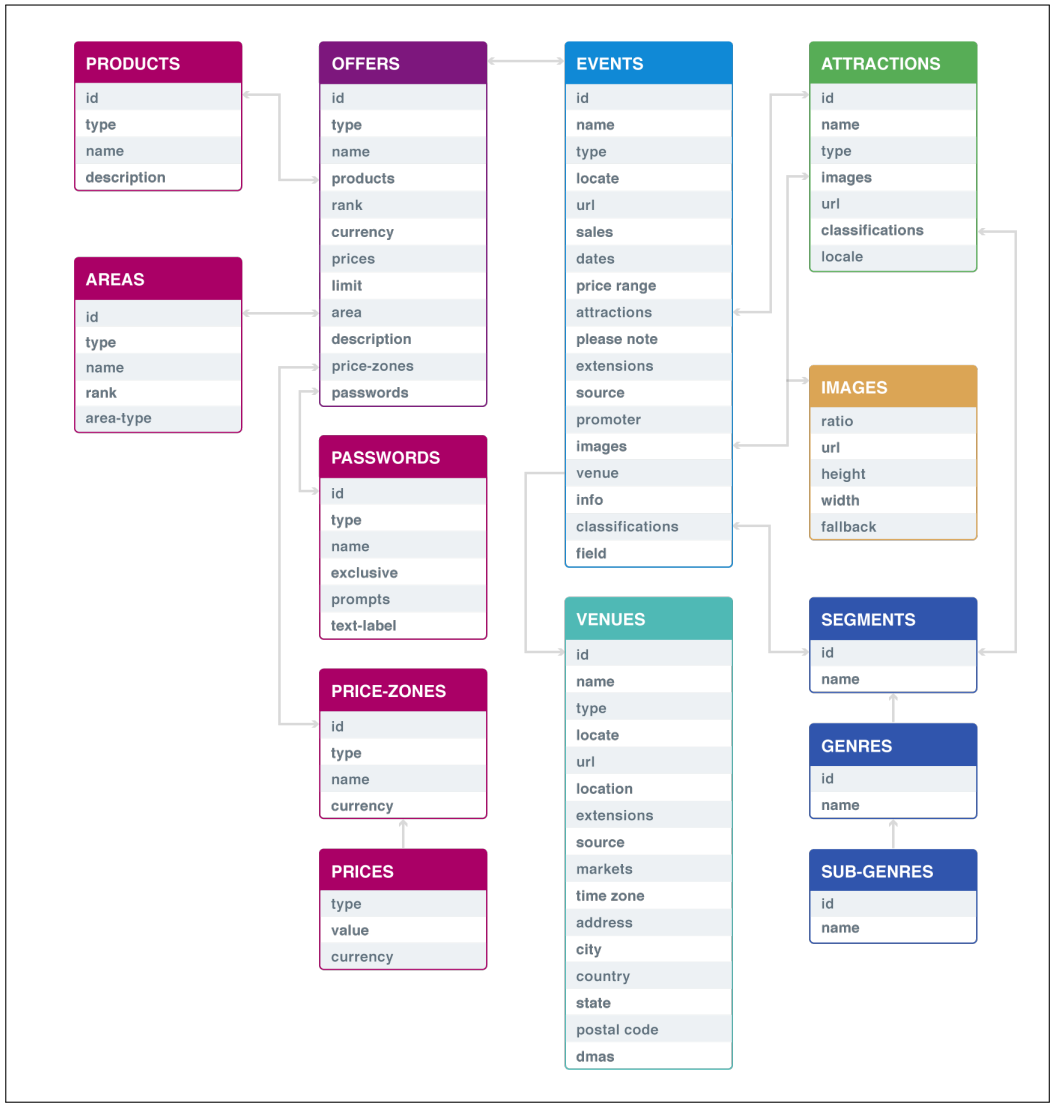


Figure B.5: Ticketmasters data model

B.2.2 Event data models

Datatype	Name	Description
DateTime	StartDateTime	Date for when the ticket sale starts.
bool	StartTBD	Start date to be determined.
DateTime	EndDateTime	Date for when the ticket sale ends.

Table B.6: Data model for Sales

Datatype	Name	Description
string	LocalStartDate	Local date for when the event starts.
string	Timezone	Timezone where the event takes place.
string	StatusCode	Whether tickets are on sale or not.
bool	SpanMultipleDays	Whether the event spans multiple days or not.

Table B.7: Data model for Dates

Datatype	Name	Description
bool	Primary	Local date for when the event starts.
bool	Family	Timezone where the event takes place.
Segment	Segment	Contains event classification type info.
SubGenre	SubGenre	Contains sub-genre related information.

Table B.8: Data model for Classification

Datatype	Name	Description
string	Id	Segment type id.
string	Name	Name of segment type.

Table B.9: Data model for Segment

Datatype	Name	Description
string	Id	Genre type id.
string	Name	Name of genre type.

Table B.10: Data model for Genre

Datatype	Name	Description
string	Id	Genre type id.
string	Name	Name of genre type.

Table B.11: Data model for SubGenre

Datatype	Name	Description
string	Id	Promoter id.
string	Name	Name of the promoter.

Table B.12: Data model for Promoter

Datatype	Name	Description
string	Type	Type of this pricerange.
string	Currency	The currency used for this pricerange.
double	Min	Minimum price.
double	Max	Maximum price.

Table B.13: Data model for PriceRange

Datatype	Name	Description
string	Id	Unique identifier for this venue.
string	Name	Name of the venue.
string	Type	Type of the venue.
string	Url	Url for the venues website.
string	Locale	Locale of the venue.
string	Timezone	Timezone where the venue is located.
City	City	City where the venue is located.
Country	Country	Country where the venue is located.
Address	Address	Address where the venue is located.

Table B.14: Data model for Venue

Datatype	Name	Description
string	Name	Name of the city.

Datatype	Name	Description

Table B.15: Data model for City

Datatype	Name	Description
string	Name	Name of the country.
string	Code	Country code.

Table B.16: Data model for Country

Datatype	Name	Description
string	Line	Address line.

Table B.17: Data model for Address

Datatype	Name	Description
string	Id	Unique identifier of the attraction.
string	Name	Name of the attraction.
string	Type	Attraction type.
string	Locale	Attraction locale.
Externallinks	Externallinks	Social media links.

Table B.18: Data model for Attraction

Datatype	Name	Description
List<string>	Youtube	Youtube URLs.
List<string>	Twitter	Twitter URLs.
List<string>	Itunes	Itunes URLs.
List<string>	Lastfm	Lastfm URLs.
List<string>	Facebook	Facebook URLs.
List<string>	Wiki	Wiki URLs.
List<string>	Instagram	Instagram URLs.
List<string>	Homepage	Homepage URLs.

Table B.19: Data model for Externallinks

B.3 Implementation

B.3.1 Package diagram

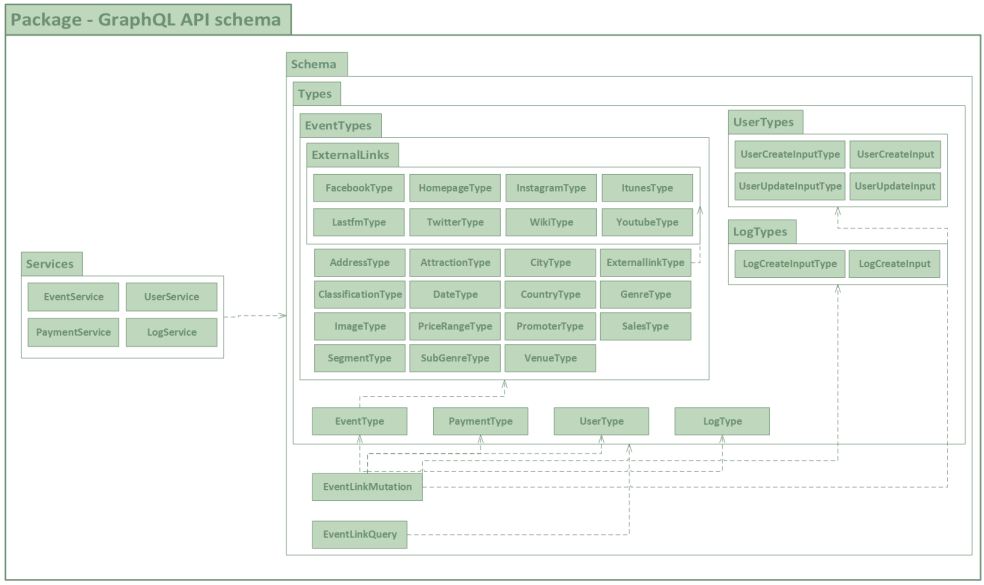


Figure B.6: Package diagram - GraphQL API Schema

B.3.2 Design class diagram

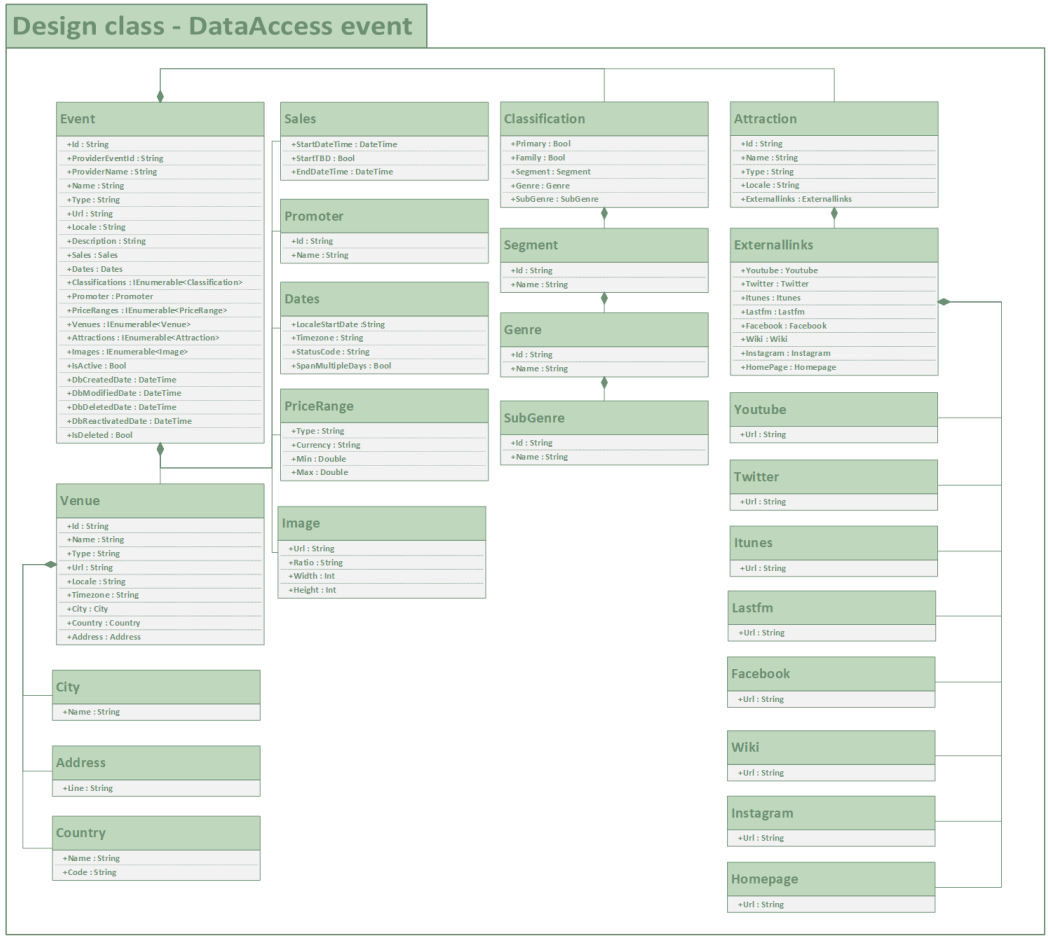


Figure B.7: Design class diagram - DataAccess event model

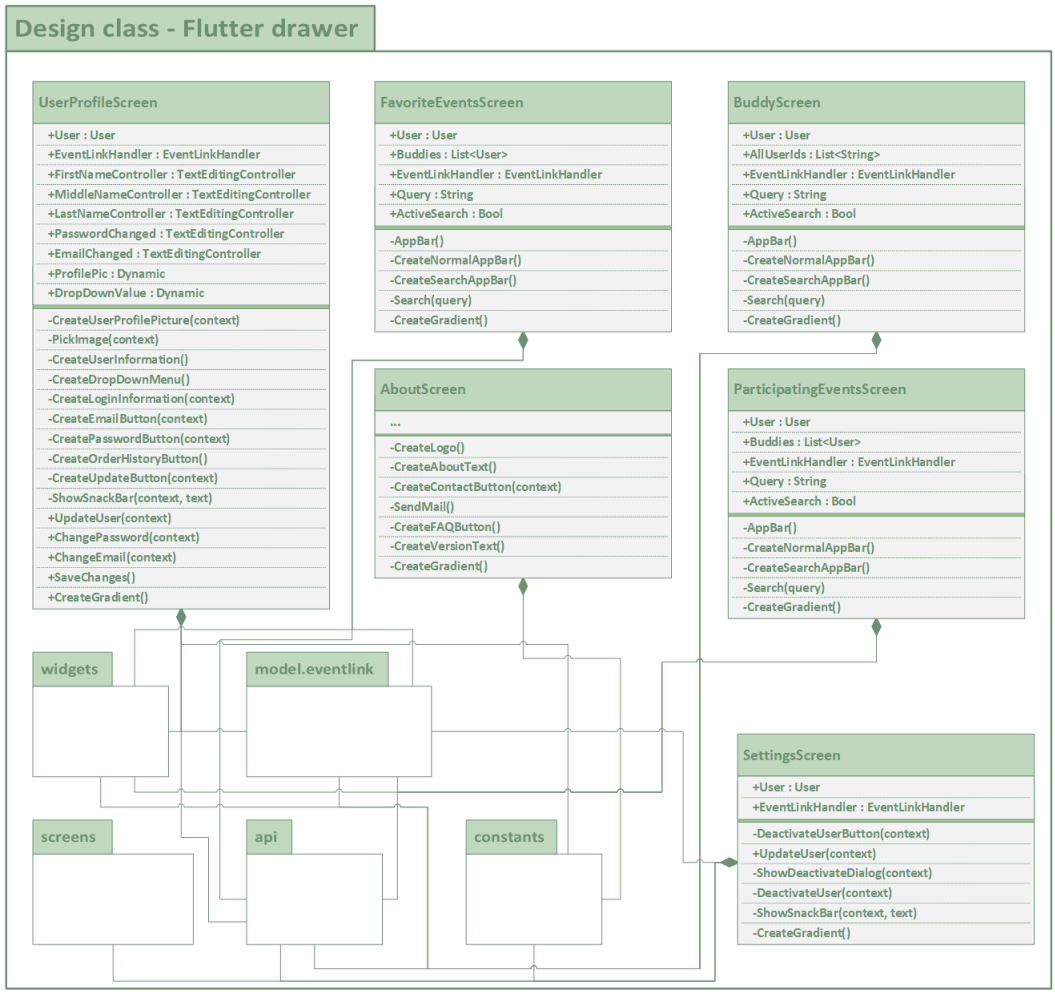


Figure B.8: Design class diagram - Flutter drawer

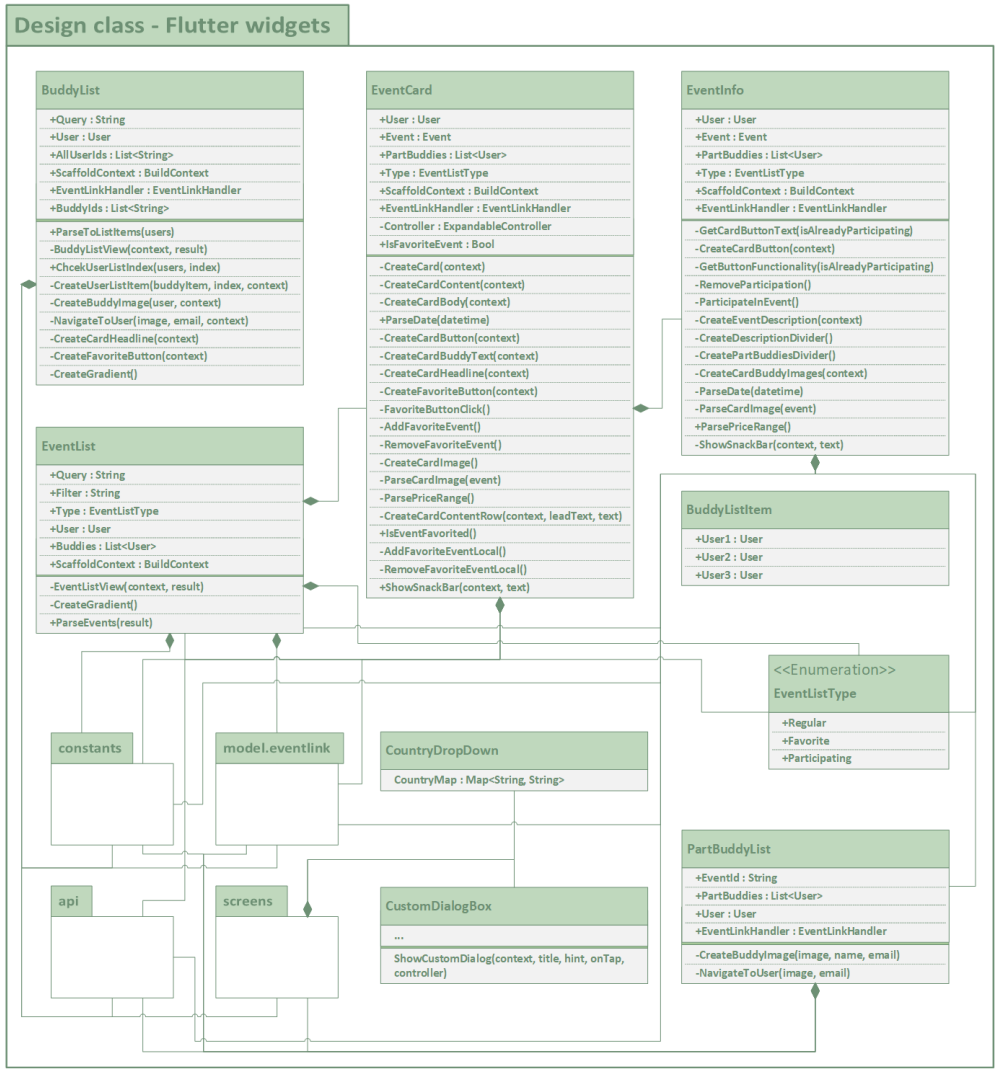


Figure B.9: Design class diagram - Flutter widgets

B.4 Test

B.4.1 Testcases

Id	Test case	Result
TC022	GetDocument_FoundDoc	OK
TC023	GetDocument_IdNull	OK
TC024	GetDocument_IdNotFound	OK
TC025	GetDocuments_FoundDocList	OK
TC026	GetDocuments_DocListNull	OK
TC027	CreateDocument_DocCreated	OK
TC028	CreateDocument_DocAlreadyExists	OK
TC029	CreateDocument_CheckInvalidIdFormat	OK
TC030	UpdateFormat_DocWasUpdated	OK
TC031	UpdateFormat_DocWasNotFound	OK
TC032	DeleteDocument_DocWasDeleted	OK
TC033	DeleteDocument_DocIsNull	OK
TC034	DeleteDocument_DocIdIsNull	OK
TC035	DeleteDocument_DocNotFoundWrongId	OK
TC036	DestroyDocument_DestroySingleDoc	OK
TC037	DestroyDocument_DestroyNullDoc	OK
TC038	DestroyDocument_DestroyNullStringId	OK
TC039	CheckLog_NullDAObject	OK
TC040	CheckLog_EssentialData_ParentName	OK
TC041	CheckLog_EssentialData_FunctionName	OK

Table B.20: Test case - Datastore LogService

Id	Test case	Result
TC042	GetDocument_FoundDoc	OK
TC043	GetDocument_IdNull	OK
TC044	GetDocument_IdNotFound	OK
TC045	GetDocuments_FoundDocList	OK
TC046	GetDocuments_DocListNull	OK
TC047	CreateDocument_DocCreated	OK
TC048	CreateDocument_DocAlreadyExists	OK
TC049	UpdateFormat_DocWasUpdated	OK
TC050	UpdateFormat_DocWasNotFound	OK
TC051	DeleteDocument_DocWasDeleted	OK

Id	Test case	Result
TC052	DeleteDocument_DocIsNull	OK
TC053	DeleteDocument_DocIdIsNull	OK
TC054	DeleteDocument_DocNotFoundWrongId	OK
TC055	DestroyDocument_DestroySingleDoc	OK
TC056	DestroyDocument_DestroyNullDoc	OK
TC057	DestroyDocument_DestroyNullStringId	OK
TC058	CheckPayment_NullDAObject	OK
TC059	CheckPayment_EssentialData_EventId	OK
TC060	CheckPayment_EssentialData_UserId	OK

Table B.21: Test case - Datastore PaymentService

Id	Test case	Result
TC061	GetDocument_FoundDoc	OK
TC062	GetDocument_ProviderEventIdNull	OK
TC063	GetDocument_DAIdNotFoundException	OK
TC064	GetDocuments_FoundDocList	OK
TC065	GetDocuments_DocListNull	OK
TC066	CreateDocument_DocCreated	OK
TC067	CreateDocument_DocAlreadyExists	OK
TC068	UpdateFormat_DocWasUpdated	OK
TC069	UpdateFormat_DocWasNotFound	OK
TC070	DeleteDocument_DocWasDeleted	OK
TC071	DeleteDocument_DocIsNull	OK
TC072	DeleteDocument_DocIdIsNull	OK
TC073	DeleteDocument_DocNotFoundWrongId	OK
TC074	DestroyDocument_DestroySingleDoc	OK
TC075	DestroyDocument_DestroyNullDoc	OK
TC076	DestroyDocument_DestroyNullStringId	OK
TC077	CheckLog_NullDAObject	OK
TC078	CheckLog_EssentialData_Fullname	OK
TC079	CheckLog_EssentialData_Name	OK

Table B.22: Test case - Datastore UserService

Id	Test case	Result
TC080	EventParser_SameDataLength	OK
TC081	EventParser_SameProviderEventIds	OK

Id	Test case	Result
TC082	EventParser_SameNames	OK
TC083	EventParser_SameUrls	OK
TC084	EventParser_SameTypes	OK
TC085	EventParser_SameDateLocalStartDate	OK
TC086	EventParser_SamePriceRanges	OK
TC087	EventParser_SameSales	OK

Table B.23: Test case - Event Crawler Parser

Id	Test case	Result
TC088	TestAuthenticate_ExistingUser	OK
TC089	TestAuthenticate_NonExistingUser	OK
TC090	TestAuthenticate_ExistingUserWrongPasswordRightEmail	OK
TC091	TestAuthenticate_ExistingUserWrongEmailRightPassword	OK
TC092	TestForgotPassword_ExistingUser	OK
TC093	TestForgotPassword_NonExistingUser	OK

Table B.24: Test case - Access Control AuthService

B.5 Tools

B.5.1 GitKraken Glo

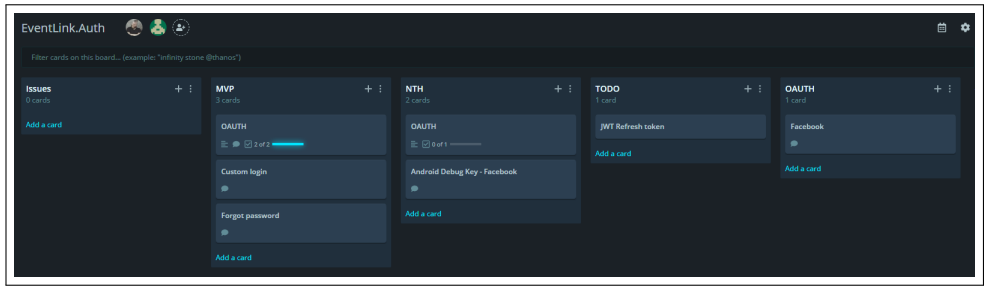


Figure B.10: Glo authentication board

B.5.2 Voyager

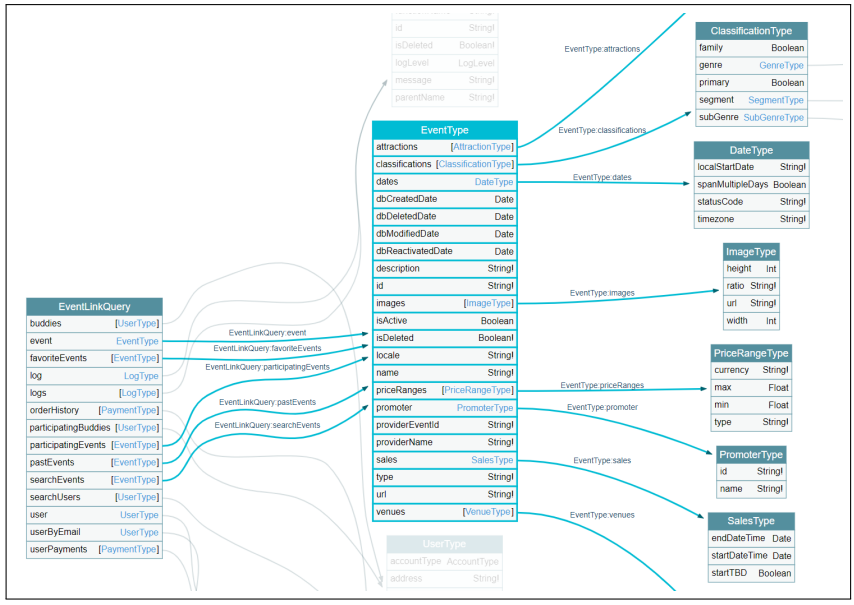


Figure B.11: Voyager datamodel snippet

Acronyms

API Application Programming Interface. 18, 20–23, 25, 26, 34, 36–39, 45–50, 55, 58, 67, 70, 72, 74–78, 80, 81, 83, 85, 87, 89, 91, 92, 94, 99, 100, 104, 106, 107, 114

AWS Amazon Web Services. 40

CRUD Create, Read, Update, Delete. 26, 34, 42, 48, 61, 64, 95, 99

DLL Dynamic Link Library. 105

DTU Technical University of Denmark. v, vii, 40, 41, 55

EU European Union. 44

FTP File Transfer Protocol. 105

FURPS+ Functionality, Usability, Reliability, Performance, Supportability, Design constraints, Implementation constraints, Interface constraints, Physical constraints. 12

GDPR General Data Protection Regulation. 12, 13, 42, 44, 55, 101

GraphQL Graph Query Language. 20, 21, 23, 25, 29, 36–40, 48–51, 55, 70, 72, 74, 75, 78, 80, 81, 85, 87, 91, 92, 99, 100, 106, 107, 114

GRASP General Responsibility Assignment Software Patterns. 40

GUI Graphical User Interface. 14, 105, 110

HTTP Hyper Text Transfer Protocol. 20, 21, 39, 58, 107

IDE Integrated Development Environment. 105, 106

IP Internet Protocol. 106

JSON JavaScript Object Notation. 21, 22, 39, 42, 43, 55, 66–69, 86

JWT JSON Web Token. 21–23, 25, 26, 29, 36, 49, 50, 76–79, 89, 92, 99, 107, 111

LINQ Language Integrated Query. 42, 64

MVP Minimum Viable Product. 5, 25, 27–29, 51, 99, 101, 104, 109, 110, 114

NoSQL Non Structured Query Language. 19, 39, 42

NTH Nice To Have. 104, 110, 111

OOP Object Oriented Programming. 18

RAM Random Access Memory. 40

REST Representational State Transfer. 20, 21, 39, 107

SOAP Simple Object Access Protocol. 20, 21

SQL Structured Query Language. 19, 21

SSH Secure Shell. 105

T-SQL Transact Structured Query Language. 19

TCP Transmission Control Protocol. 39

UDP User Datagram Protocol. 39

UML Unified Modeling Language. 5, 14, 31, 33, 37, 61, 90

VSFTP Very Secure File Transport Protocol. 58, 114

XML Extensible Markup Language. 20, 21

Glossary

.NET .NET is a free, cross-platform, open source developer platform for building many different types of applications. 18, 19, 21, 48, 49, 105

.NET Core NET Core is a cross-platform version of .NET for building websites, services, and console apps. 18, 29, 39

C# C# is an type-safe object-oriented language that enables developers to build a variety of secure and robust applications that run on the .NET Framework. 18, 19, 23, 29, 42, 61

code review Code review is a software quality assurance activity in which one or several persons check a program mainly by viewing and reading parts of its source code. 6

crawler A crawler is a program that gathers data from Web sites, APIs and other sources. 20, 65, 66, 69, 89, 92, 94, 99, 114

dart Dart is a client-optimized programming language for fast apps on any platform. 23, 80, 86, 105

datastore A datastore is a repository for persistently storing and managing collections of data. 14, 17, 19, 21, 23, 25, 26, 31, 34–39, 42, 53, 54, 66, 81, 88, 89, 91, 93, 94, 96, 99

DigitalOcean DigitalOcean provides developers cloud services that help to deploy and scale applications that run simultaneously on multiple computers. 40

eventful Eventful is an online calendar and events discovery service owned by Entercom. 2, 46, 47, 55, 67–69, 99

EventLink EventLink is the name of the solution that has been developed during this thesis. 1–3, 5–7, 10, 11, 16, 17, 21–23, 28, 29, 32, 40–43, 45–47, 49, 50, 57, 70, 80, 83, 84, 87, 89–91, 93, 94, 96–98, 103–107, 109–111, 114, 118–120

extreme programming Extreme Programming is an agile software development framework that aims for higher quality of life for the development team. 6

F# F# is a functional programming language. F# programming primarily involves defining types and functions that are type-inferred and generalized automatically. 18

flutter Flutter is Google's UI toolkit for building natively compiled applications for mobile, web, and desktop from a single codebase. 22, 23, 25, 29, 50, 51, 55, 79, 80, 85, 87, 90–92, 96, 99, 104, 105, 110, 114

functional programming functional programming is a programming paradigm, that treats computation as the evaluation of mathematical functions and avoids changing-state and mutable data. 18

handler A handler is a software component that can handle requests and responses between two or more systems. 14, 32

identity server IdentityServer4 is an OpenID Connect and OAuth 2.0 framework for ASP.NET Core. It has support for a variety of access control mechanisms. 21, 22

iteration planning An iteration plan consists of the given tasks that are to be completed in a iteration. 6

iterative development Iterative development is a way of breaking down the software development of a large application into smaller chunks. 6

JavaScript JavaScript, often abbreviated as JS, is a high-level, just-in-time compiled, object-oriented programming language. 22, 23

LaTeX LaTeX is a document preparation system. Which helps style and format a document. 108

linux Linux is a open source Unix-like operating system. Created in 1991 by Linus Torvalds. 41, 58, 105

MongoDB MongoDB is a cross-platform document-oriented database program. Classified as a NoSQL database program. 19, 21, 23, 25, 29, 34, 39, 42, 55, 60, 61, 63, 65, 67, 75, 92, 99, 105, 114

MSSQL Microsoft SQL Server is a relational database management system developed by Microsoft. 19

MySQL MySQL is an open-source relational database management system developed by Oracle. 19

OAuth2 OAuth 2.0 is the industry-standard protocol for authorization. It focuses on client developer simplicity while providing specific authorization flows for applications. 21–23, 25, 27, 29, 36, 49–52, 55, 79–81, 83, 87, 92, 98, 99, 110, 111

pair programming Pair programming is an agile software development technique in which two programmers work together at one workstation. 6, 29

react native React Native is an open-source mobile application framework created by Facebook. It is used to develop applications for Android, iOS and more. 22, 23

reactjs React is a JavaScript library for building user interfaces. It is maintained by Facebook and a community of individual developers and companies. 22, 23

scrapy Scrapy is a web-crawling framework written in Python. Originally designed for web scraping, it can also be used to extract data using APIs or as a general-purpose web crawler. 20

SeatGeek SeatGeek is a mobile-focused ticket platform that enables users to buy and sell tickets for live sports, concerts and theater events. 46

sequential development The stages of development for any process of growth happen in a prescribed sequence. 6, 29

testing Software testing is defined as an activity to check whether the actual results match the expected results and to ensure that the software system is defect free. 6

ticketmaster Ticketmaster Entertainment, Inc. is an American ticket sales and distribution company based in Beverly Hills, California. 2, 46, 47, 55, 67–70, 99

ubuntu Ubuntu is a free and open-source Linux distribution based on Debian. 41, 58

unified process Unified Process is an iterative and incremental software development process framework. 6

waterfall model The waterfall model is a breakdown of project activities into linear sequential phases, where each phase depends on the deliverables of the previous one. 6

Bibliography

[ApS19] GDPR.DK ApS. General data protection regulation. https://www.gdpr.dk, 2019.

[Aut19] Auth0. Json web token introduction. https://jwt.io/introduction/, 2019.

[Cle19] J. Clement. Number of apps available in leading app stores as of 3rd quarter 2019. https://www.statista.com/statistics/276623/number-of-apps-available-in-leading-app-stores/, October 2019.

[Con19] Intersoft Consulting. General data protection regulation. https://gdpr-info.eu/, 2019.

[Eve19a] Eventful. Api documentation eventful api. https://api.eventful.com/docs/events/get, 2019.

[Eve19b] Eventful. Eventful api. https://api.eventful.com/, 2019.

[Fou19] GraphQL Foundation. Graphql, a query language for your api. https://graphql.org/, 2019.

[Fre19] Freenom. Freenom registry. https://www.freenom.com/, 2019.

[Git19a] GitHub. Github. https://github.com/, 2019.

[Git19b] GitKraken. Glo dashboards. https://www.gitkraken.com/glo, 2019.

[Gra19] GraphQL. Graphiql. https://github.com/graphql/graphiql, 2019.

[Ide19] IdentityServer. Identityserver. https://identityserver.io/, 2019.

[Inc19] Kong Inc. Insomnia. https://insomnia.rest/, 2019.

[Lab19] 3T Software Labs. Robo 3t. https://robomongo.org/, 2019.

[Lar04] Craig Larman. *APPLYING UML AND PATTERNS An Introduction to Object-Oriented Analysis and Design and Iterative Development.* John Wait, third edition, 2004.

[Mic19a] Microsoft. Visio. https://products.office.com/en/visio/flowchart-software, 2019.

[Mic19b] Microsoft. Visual studio. https://visualstudio.microsoft.com/, 2019.

[Mic19c] Microsoft. Visual studio code. https://code.visualstudio.com//, 2019.

[Min19] IT Minds. It minds. https://it-minds.dk/, 2019.

[Mob19] Mobatek. Mobaxterm. https://mobaxterm.mobatek.net/, 2019.

[Mon18] Paul Monk. To sql or not to sql. https://capgemini.github.io/design/sql-vs-nosql/, May 2018.

[Mon19] MongoDB. Mongodb c#/.net driver. https://docs.mongodb.com/ecosystem/drivers/csharp/#introduction, 2019.

[mrp20] mrpink76. Problem passing parameters. https://github.com/graphql-dotnet/graphql-dotnet/issues/608, January 2020.

[.NE19] GraphQL .NET. graphql-dotnet/graphql-dotnet: Graphql for .net. https://github.com/graphql-dotnet/graphql-dotnet, December 2019.

[Ove19] Overleaf. Overleaf, online latex editor. http://overleaf.com/, 2019.

[Par19] Aaron Parecki. Oauth 2.0. https://oauth.net/, 2019.

[Per17] Sarah Perez. Smartphone users are using 9 apps pr day, 30 per month. https://techcrunch.com/2017/05/04/report-smartphone-owners-are-using-9-apps-per-day-30-per-month/, May 2017.

[Scr19] Ltd Scrapinghub. Scrapy api. https://scrapy.org/, 2019.

[Spe19] Spectrum. Spectrum. http://spectrum.chat/, 2019.

[Tic19] TicketMaster. Getting started - the ticketmaster developer portal. https://developer.ticketmaster.com/products-and-docs/apis/getting-started/, 2019.

[Wor18] Digital Information World. The human attention span. https://www.digitalinformationworld.com/2018/09/the-human-attention-span-infographic.html, September 2018.