Julian Göschl, Bsc

Creation and Evaluation of a Matchmaking Platform for Student Projects

Master's Thesis
to achieve the university degree of
Master of Science
Master's degree programme: Software Engineering and Management

submitted to

Graz University of Technology

Supervisor
Dipl.-Ing. Dr.techn. Michael Krisper

Institute of Technical Informatics
Head: Univ.-Prof. Dipl.-Inform. Dr.sc.ETH Kay Uwe Römer

Graz, April 2023

Danksagung

Ich möchte meinem Supervisor Michael Krisper für die Unterstützung während dieser ganzen Arbeit danken, sowie für die grundlegende Idee des Projekts. Außerdem möchte ich meinen Eltern, sowie meiner Schwester Helena für jegliche Hilfe danken. Weiters danke ich allen Lehrern und Professoren, die mir auf dem Weg geholfen haben. Zuletzt möchte ich meinen Freunden und Kommilitonen danken, sowie allen, die an der User Study dieser Arbeit teilgenommen haben.

Abstract

In this thesis, a matchmaking platform and recommender system for students at Graz University of Technology was created and evaluated. The purpose of this platform is to simplify the process of finding an appropriate and fitting thesis topic for a Master's or Bachelor's Thesis. For that, a mobile application for students, and a web application for lecturers were created. The mobile application was built in Flutter, using the programming language Dart, while the web application's front-end was built in Angular, with the back-end coded in node.js. Moreover, related work and background information on the following topics are covered in this thesis: Software Architecture, Design Patterns, Matchmaking, Recommender Systems, Multi-Platform Development, Databases, and the main frameworks for the two platforms of this project: Flutter and Angular. The software architecture and details of the implementations of this project are discussed in detail, as well as the design patterns in use. Lastly, the created project was evaluated by assessing the project's requirements and a user study was carried out, indicating that the project indeed solves the students' struggle of finding a suitable thesis topic.

Contents

List of Figures

List of Tables

Listings

Chapter 1

Introduction

In this thesis, I tackled a problem every student has during their life at university: finding a fitting topic for their bachelor's or master's thesis. Before finishing a bachelor's or master's degree, students in most majors need to write a thesis on a topic within their field of study. Before hitting that burden, do not realize how hard finding a fitting topic for their thesis often is. For example, within Graz University of Technology, most institutes only offer an outdated, incomplete list of topics, sometimes within PDF files and some institutes do not offer a list on their website at all. This often leads to the best possible way for students to get a good overview of open thesis topics: personally contacting lecturers to request a list of their topics.

This brought up the idea to collect topics and offer the students an overview of those topics within a mobile application called "TU Project Finder", hence greatly simplifying the process of looking for topics interesting to the student. Moreover, this application also may help lecturers accelerate the process of finding students interested in their written out theses, while also avoiding the extra work of answering students' emails asking for a list of their possible theses.

A central focus of this thesis is also looking at design patterns, that describe a problem within software development, alongside a solution, which can be applied to different applications and circumstances. Matchmaking and recommender systems are also central points of this thesis.

1.1 Context

This thesis is a master's thesis at Graz University of Technology, to be more precise the Institute of Technical Informatics.

The implemented applications are specific to the needs of students and lecturers of TU Graz, who are the two target groups of the project.

1.2 Motivation

One main problem this project solves, which has already been talked about, is the struggle students face when looking for a topic for their bachelor's or master's thesis. The other main problem is lecturers struggling to find students to work on their projects.

Currently, students often have to contact lecturers via email to get a list of possible topics. Due to every student having to find and write a thesis to receive their degree, offering an application to them, that gives an overview of possible topics and helps to find a fitting one, would help students greatly.

Moreover, this is also of personal interest to me, the writer of this thesis, due to struggling while looking for a bachelor's thesis, which I ultimately chose, due to other students recommending one. While looking for a topic for the master's thesis, the only way I found to work, was by contacting lecturers, whose lectures were interesting to me.

1.3 Structure of the Thesis

- Chapter 1 is where the structure of this thesis is shown: the introduction.
- Within Chapter 2, background information and related work on subjects important to understanding this thesis are covered. These subjects include:
  - Software Architecture in general
  - Design Patterns and their importance to software development
  - Matchmaking platforms that allow people to connect
  - Recommender Systems that give appropriate suggestions
  - Multi-Platform Development to support multiple operating systems
  - Specific technologies like Flutter, Angular etc.
- In Chapter 3, the architecture and the design of the applications are covered, along with the requirements set for the project.
- Chapter 4 provides an insight into the implementation of mobile and web applications implemented in the process of writing this thesis.
- In Chapter 5, design patterns in this application are explained, as well as the context each was used in.
- Chapter 6 evaluates the project's success in two parts. Firstly, requirements are looked at, whether they were fulfilled or not. Secondly, a user study was carried out, which will be discussed.
- Chapter 7 is the discussion, where limitations and future work will be covered.
- Lastly, Chapter 8 is the conclusion of the thesis.

Chapter 2

Background and Related Work

In this chapter, the background and related work for the thesis are described. This covers the topics of *Software Architecture*, *Design Patterns*, *Matchmaking*, *Recommender Systems* and *Multi-Platform Development* with a major focus on the *Flutter* framework, due to its importance to this work.

2.1 Software Architecture

> "The software architecture of a system is defined as the structure or structures of the system, which comprise software components, the externally visible properties of those components, and the relationships among them." - L. Bass, P. Clements, and R. Kazman [1].

What is Software Architecture? As a method to understand large-scale software structures, software architecture originally emerged in the early 70s by Parnas [2]. It is used as a high-level blueprint for software design throughout all stages of the development, as well as maintenance [3]. Parnas' approach was to make use of modularization software and hide information to improve flexibility and comprehensibility within the software, structure [4].

Nowadays, the main goal of software architecture is, to keep up the software quality throughout the whole lifespan of a software project [4]. To achieve this, extensive planning needs to be done beforehand and every adaption of the original architecture should be well documented. Before creating the architecture of the software requirements must be defined by various shareholders. These requirements can be split in functional and non-functional requirements, with functional requirements describing what the software should do, and what features

it should include, while non-functional requirements describe rather generalized and emerging properties, such as up-time of the software [5]. Software Architecture may also be described as the bridge between the requirements and the code [6].

2.1.1 How does Software Architecture influence Software Development?

Garlan describes the following aspects in which the software architecture influences the software development process [6]:

- Understanding
- Management
- Reuse
- Communication
- Evolution
- Construction
- Analysis

Understanding One goal of software architecture is to simplify a large-scale system by creating an easy-to-understand abstraction of the whole system. To make the system more understandable, software architecture also explains architectural decisions [6].

Management A thought through software architecture often leads to a better understanding of requirements, possibilities and possible risks. Moreover, successful projects often even include “creating a well-structured software architecture” of one of the key milestones of the project [6].

Reuse A well-designed software architecture has multiple reusable parts. It allows for different reusable libraries, as well as the framework itself may be reusable. Something that is often reused or rather reapplied, are Design Pattern, which will be discussed later in this work [6] in section 2.2.

Communication Software architecture allows easier communication between all stakeholders, due to common vocabulary as well as an easy understandable, structured abstraction of the whole system [6].

Evolution By having an accurate description of the system and all of its components, changes, and maintenance is simpler, as well as estimating the costs for the change and maintenance will get easier [6].

Construction When using the abstraction of the software as a blueprint for the development process, creating the system will be more efficient, faster, and more structured [6].

Analysis Analysis of different metrics gets easier with a well-structured software architecture. Clearly defining all system requirements with metrics to verify them helps to check their status constantly and therefore assure a higher level of quality within the software development process [6].

2.1.2 Problems of Software Architecture

The list of problems a team may face within software architecture is long, with some being more widespread than others. The problems this work will discuss are:

- Documentation
- Erosion
- High-level architecture to Code
- Bias
- Complexity and Scalability

Documentation Due to the fact that the requirements and the documentation of systems are often outdated and do not cover every aspect of a system, the development team will get less reliant on it [7]. Moreover, if the requirements are not organized well enough or changes are not communicated, this could lead to struggles, due to not everybody having the same knowledge about those specifications.

Most specifications are file-based, which may simplify the process in the beginning but might end up as an extra burden further down the road. Coordinating the sharing of all files, including requirements and their changes may end up being a lot of extra work [7].

Over time, bad documentation leads to design decisions being unclear to a point where most of them will be untraceable [7].

Erosion Connected to the documentation is something called software architecture erosion. This problem is faced when the implemented architecture ends up not being structured like the intended structure. This often results in complex, hard-to-understand, and hard-to-maintain software with an overall low code quality [8]. This can be solved by checking conformance between implementation and intended architecture regularly.

A study on software erosion published by US Air Force showed that the implementation of the same feature, done in an eroded version of the same software takes about twice as long and the amount of produced errors increased eightfold [9].

Impact of Architecture Decisions to the Source Code High-level architectural requirements are hard to translate to the source code. Moreover, some high-level architectural decisions are made without the technology in mind, such as programming language. Therefore, a developer might be forced to differ from the architectural requirements because the intended architecture simply will not work in some cases. This again often leads to the problems already mentioned in the paragraph about documentation, that changes of the architecture are not documented or hard to trace [8].

Bias Every software architect is biased in multiple ways and therefore will not make completely rational decisions. Cognitive bias is a term, describing a human's inability to decide or reason rationally. There is a broad number of biases that may influence the software architect's decision-making [10].

Anchoring bias, for example, is when decisions are made in a certain context. This context is often assumed implicitly, and every decision is made with that context in mind, meaning the context is the anchor of each decision.

Another frequent bias is confirmation bias. For example, when a software developer tests their source code, they is way more likely to choose test cases that verify their code, rather than faulty ones [11]. If something works once, the architect is biased to keep using it.

Lastly, framing bias should be covered. If, for example, two groups receive the same requirements, one group receives the requirements in a very detailed, descriptive text, while the other group only receives them in a less compulsory way, making them seem more like a list of ideas and possibilities. Only through a different wording of the requirements, the first group will try to stay as close to the requirements as possible, thinking they are mandatory, while the other group tries to come up with creative and new ideas, not focusing too much on the requirements received [10].

Complexity and Scalability In an evolving software architecture, each problem and combined solution may alter or add new requirements or design decisions. The whole system, therefore, co-evolves leading to the system getting progressively more complex.

The scalability of the system suffers from a higher complexity of the architecture. Moreover, some architectural methods are unable to scale well, therefore increasing the complexity of the system in return.

Design Patterns Until now, this work mostly covered high-level software architecture, where mostly the overall structure of the software project is defined. Further down the levels of architecture is the level that defines the architecture of modules and interconnections [12]. This level covers packages, classes, components, as well as design patterns.

Design patterns are a central topic in this work and therefore covered in more detail in section 2.2. As part of the low-level software architecture, design patterns aim at making parts of the code structure to be reusable in many versatile use cases. Each design pattern differs from the others, with some being defined broadly and usable in many cases and others being only viable in a handful of use cases. Besides reusability, another useful aspect of design patterns is that they are an abstraction of more complex design decisions, therefore making them easier to understand as well as allowing a common vocabulary within the software architecture and the development process[13].

A more in-depth description of design patterns is given in the next section, as well as later in chapter 5, when the usage of design patterns within the TU Project Finder will be discussed.

2.2 Design Patterns

> "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice" - Christopher Alexander [14]

What is a Design Pattern? A design pattern is used to describe a software abstraction that can be used to solve similar problems over and over again. Patterns can describe how objects are created and how they communicate among other use cases [15].

The so-called Gang of Four (or GoF) took Alexander's approach that patterns describe problems that appear in different places in our environment, with each also describing the basic solution that can be applied over and over again. Alexander's approach is originally meant for buildings and towns, and the GoF applied it to software development. The GoF created a list of 23 original design patterns [16]. Those patterns are not supposed to be complex or domain-specific, but rather simple and reusable across various systems facing similar problems.

Design patterns are used to capture and define design know-how and reuse it. By doing so, they can improve communication between developers by defining a dictionary of patterns and simplifying structures of highly experienced developers and making them usable for beginners. Moreover, they can help improve implementation speed, speed up maintenance and refactoring, and simplify the code structure [17].

2.2.1 How is a Design Pattern defined?

With way over 100 different design patterns in existence, there needs to be a standardized way to define them. There are many different ways to define them, with the most common ones being [18][19]:

- Alexandrian Form
- Canonical Form
- Coplien Form
- GoF Form
- Compact Form
- POSA1/2/3/4 Forms

- Portland Form

Alexandrian Form Originally not meant for software design patterns but quite commonly used nevertheless. Alexander defined a pattern through seven sections [14]:

- Title
- Prologue
- Problem statement
- Discussion
- Solution
- Diagram
- Epilogue

GoF Form Gamma et al. proposed that graphical notations of design patterns are useful but not sufficient because they do not display the ideas behind the pattern usage, the trade-offs that are needed as well as alternative patterns or other names the patterns are known as. Their suggested form to define patterns is the following [16]:

- Name
- Classification
- Also Known As (not always)
- Motivation
- Applicability
- Structure
- Participants
- Collaborations
- Consequences
- Implementation
- Sample Code
- Known Uses
- Related Patterns

Compact Form The so-called compact form is a commonly used short way to define patterns [18] only including as few sections as needed:

- **pattern name**: a short that can be used to identify the pattern
- **problem**: when should the pattern be applied
- **solution**: describes the elements, their relationships and what they are used for
- **consequences**: results and trade-offs of the pattern

2.2.2 How can Design Patterns be categorized?

The Gang of Four proposed a categorization using two dimensions. The first one, which is about purpose, distinguishing between creational, structural, and behavioral patterns, and the second dimension about the scope makes the distinction between whether the pattern is applied to a class or an object [16].

Using this way to categorize the original 23 patterns, the following table was created:

		Purpose		
		Creational	Structural	Behavioural
Scope	Class	Factory Method	Adapter	Interpreter Template Method
	Object	Abstract Factory Builder Prototype Singleton	Adapter Bridge Composite Decorator Facade Proxy	Chain of Responsibility Command Iterator Mediator Memento Flyweight Observer State Strategy Visitor

Table 2.1: Categorization of the original 23 patterns [16].

Why are Design Patterns useful? Some advantages of using design patterns have already been mentioned, like making complex structures more abstract and

simpler, which also allows lesser experienced developers to implement and make use of those patterns [12].

Software projects can start to rot while growing without taking care of the problems. There are four primary symptoms of that happening, which include rigidity, fragility, immobility, and viscosity [12].

- Rigidity in this context means that the software gets stiff and difficult to change, mostly because changing one small thing will result in the need to adapt different parts of the code [12].
- Fragility is closely related to rigidity, with fragility meaning that fixing one problem of the code will often result in another problem of the code being exposed somewhere else [12].
- In this case, immobility means that one is not able to reuse code snippets of the same or other projects because the risks caused by separating the needed part are too big [12].
- Lastly, viscosity can be split into the viscosity of the design and the environment. An example of viscosity is when a developer needs to make a change and has multiple possibilities, with some preserving the current design and others changing it. In cases where changing it is easier than keeping the current one [12].

All those symptoms can often be avoided by applying fitting design patterns where needed.

Composite Design Patterns Often design patterns only differ in small details and the developer has the choice of which one fits his use case. However, there is also a way to combine the beneficial parts of multiple design patterns for a specific use case to create a new, composite, design pattern [20].

Model-View-Controller can be seen as a widely known composite design pattern because it consists of three patterns combined to be used as one. Composite design patterns combine multiple design patterns and the developer selects those parts that benefit the implementation, allowing for even more flexibility with the downside of often having a more complex structure [20].

Relationships between Design Patterns Some patterns are applied to the same use case, some patterns are combined to create a new composite design pattern, and some patterns are already composite and use another pattern in their implementation. Zimmer analyzed the 23 GoF patterns and tried to find all existing connections between them. He also classified the connection as “one pattern uses the other”, “one pattern is similar to the other” or “one pattern can be combined with the other” [21]. His results can be seen in Figure 2.1

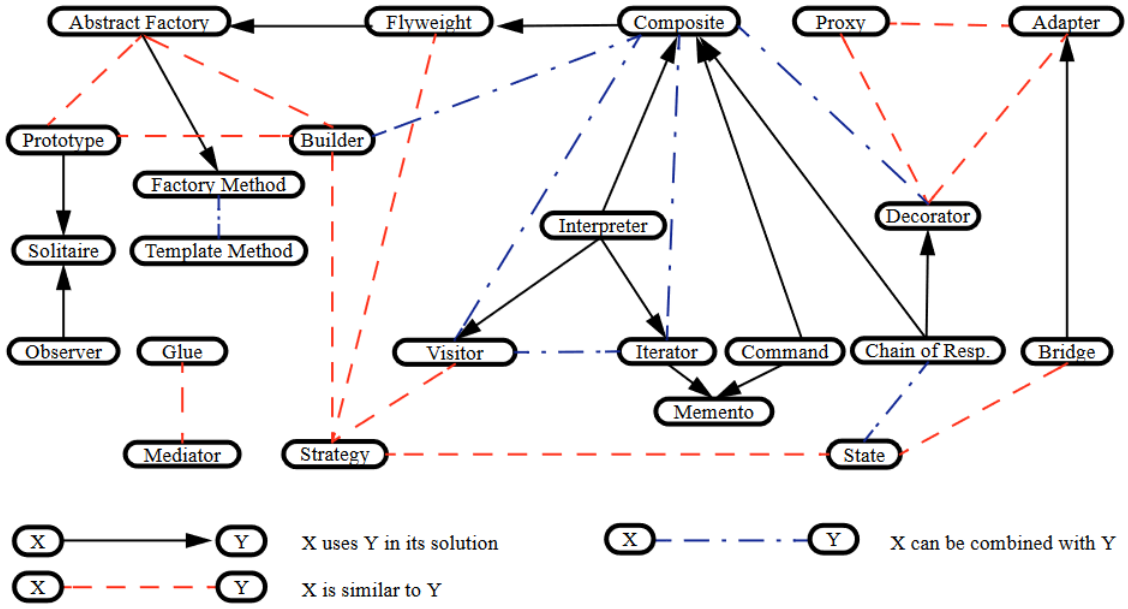

Figure 2.1: Connections between Design Patterns by Zimmer [21].

2.3 Matchmaking

This section covers background information and related work on the topic of matchmaking.

What is matchmaking? Matchmaking is used to match users to other users, services, or groups of other users. By implementing algorithms to optimize that matching process, the developer often faces a multitude of struggles, such as the users thinking that the matchmaking is biased even though it is supposed to be unbiased [22][23]. Moreover, matchmaking algorithms often include prioritization of services or users which gets controlled or could be manipulated by the market operators.

Optimization of matchmaking algorithms also may benefit the global economy, for example, by enabling users to share unused cars or houses. Sharing platforms such as AirBnb, Lieferando, or Uber that use matchmaking algorithms to connect the user to possible customers are predicted to grow their yearly revenue to $335 billion in 2025 from only $14 billion in 2014 [22].

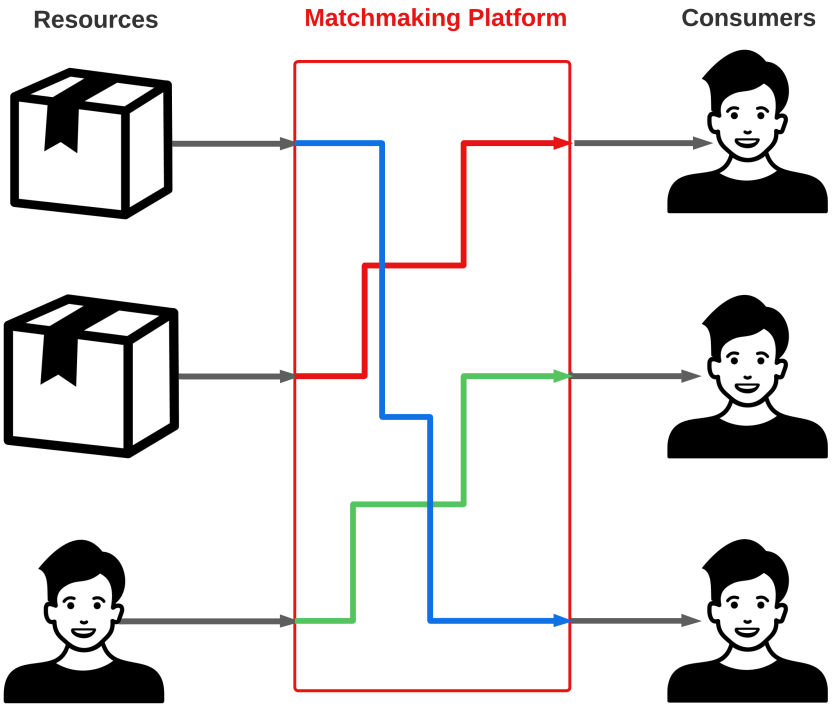

Figure 2.2: Matchmaking process where everything can be a resource and resources are matched with the consumer through the matchmaking platform

Which topics are researched within Matchmaking? Research within matchmaking algorithms is done in different directions. Some of the studied topics, important to this work, are:

- General optimization of matchmaking algorithms
- Decentralization
- Remove algorithmic bias

General optimization Matchmaking does not have a universal implementation for every use case. Different use cases require different algorithms. Some algorithms also got outdated over time and replaced by successor algorithms, which for example is the case for the TrueSkill matchmaking algorithms that are now getting replaced by TrueSkill2. Both algorithms are used for skill rating systems and predicting the match outcome, mostly for online video games. TrueSkill only allows for a match outcome prediction accuracy of 52% while TrueSkill2 predicts the outcome with an accuracy of 68% [24].

Decentralization Matchmaking algorithms such as MATCH focus on making the algorithm safer by decentralizing it. By doing so, MATCH tries to allow the end-user a bigger impact on the choice-making process, while also being sturdy against malicious matchmakers that differ from the match-making policy of the system in use. This also allows the algorithm to always match the user with the best currently available counterpart [22].

Remove bias Since most algorithms are to some degree controlled by the market operators, most match-making processes are influenced [22].

Even when the system itself is as unbiased as possible, it often seems to be unfair or biased to the end user. In dating apps, for example, users are often allowed to select their preferences such as age, race, and religion among others. To name one example, in the dating app CoffeeMeetsBagel a number of users reported only being matched with other users of the same ethnicity, even though they selected ethnic neutrality. This is often perceived to be biased by the user [23]. This is not only the case in dating apps but also, for example, in skill-based matchmaking for multiplayer online games. Despite all players being ranked as being on the same level of skill, some players will consider the matchmaking to be unfair [25].

Which problems are matchmaking algorithms facing? Depending on the specific use case, each matchmaking algorithm faces different problems, and some of them have already been discussed in the previous section.

Bias is among the biggest problems for the end-user. De Vos et al. [22], for example, are trying to remove the influence or bias caused by the market operators. But even without a bias within the algorithm itself, it often seems like there is one.

Sometimes users may perceive matchmaking algorithms as unfair. This not only was the case for dating apps like CoffeeMeetsBagel [23] but also for online video games.

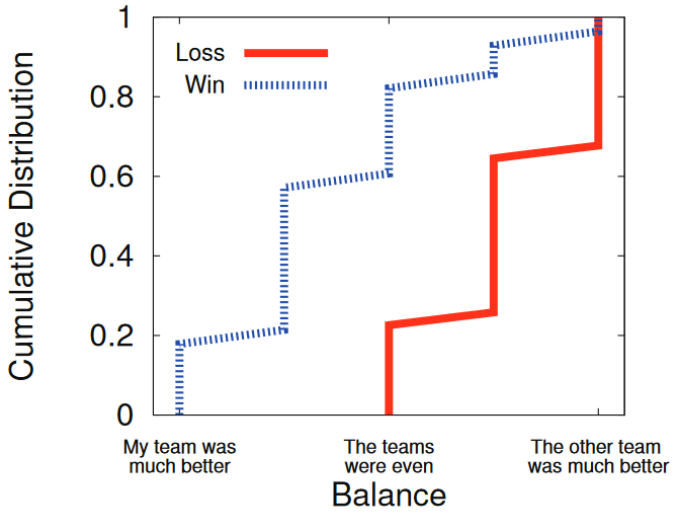

Figure 2.3: Perceived Matchmaking Balance in League of Legends. Showing that users perceive the teams as rather unbalanced when they are on the losing team, while perceiving the teams as balanced when placed in the winning team [25].

As visible in Figure 2.3 there is a huge difference in how the balance of the game is perceived depending on whether the user wins or loses the game. In League of Legends, matchmaking is based on the ELO system, even though the exact algorithm is not publicly known. ELO matchmaking was originally developed for chess with the intention to match chess players to equally skilled players. Some Research suggests that this matchmaking algorithm is not optimal for online video games like League of Legends or Dota 2, which are quite similar [26].

2.3.1 What are use cases for matchmaking algorithms?

The most prevalent use cases for matchmaking algorithms include:

- Web Services
- Online Video Games
- Dating Services

Web Services Matchmaking in Web Services often tries to match the user to either the web services itself or, what is currently more common, to other users within the web service. The immense growth of global revenue when it comes to sharing platforms like Airbnb [22] can be attributed to match-making algorithms to a certain degree. By optimizing those algorithms, this business sector was able to grow by more than 2000% within 10 years. Despite this growth, these algorithms are facing ever-changing problems, some of which were already discussed, ranging from market operators having control over the results to security problems.

Online Video Games Most online video games use matchmaking algorithms to match players with those that are rated as being at the same skill level, this can be one-on-one matching or groups of players as teams. With more than 27 million daily players [25], League of Legends is one of the biggest computer online games, which therefore also receives a lot of research. League of Legends uses a matchmaking algorithm based on the ELO system. Research suggests that the used algorithm can be improved in different ways. ELO rating is mostly based on a score that increases or decreases by a certain value depending on the ranking of the opponent's ELO rating and whether the player won or lost. Suggested improvements for complex video games like League of Legends, for example, include involving expected queue times for the games [25] or due to the diversity in playable characters including the role the player wants to play in matchmaking [24].

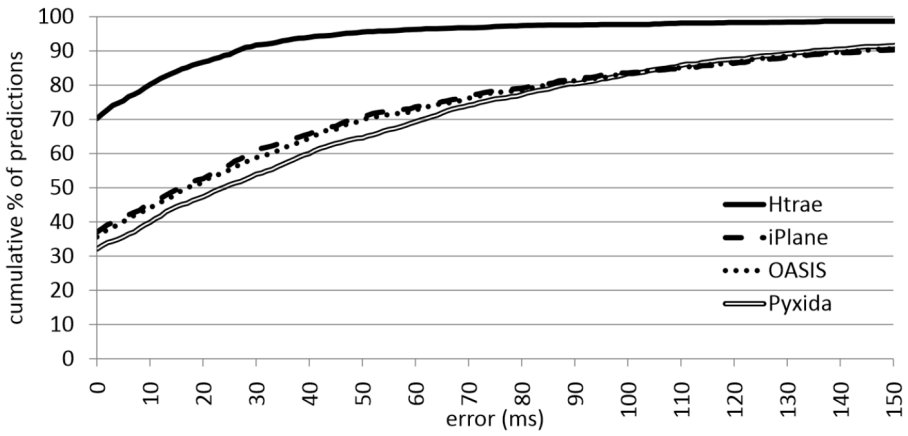

Figure 2.4: Matchmaking for Online Games - Htrae compared to other latency matchmaking algorithms that focus on minimizing the latency between users [27].

Even in video games where no skill-based matchmaking is used, there are matchmaking algorithms in use. Agarwal and Lorch implemented the Htrae algorithm which focuses and optimizing the latency between the users in an online video game, in their case Halo 3. By using Htrae and comparing it to competing algorithms like OASIS, iPlane and, Pyxida on a dataset of 50 million matches, they concluded that they could outperform all the other algorithms. As one can see in Figure 2.4 the average ms of latency between all users in the lobby created by Htrae is significantly lower than in lobbies created by competing algorithms [27].

Matchmaking in video games is not only used to match users to other uses. Sarkar et al. discuss using matchmaking to match the user to levels depending on their calculated skill rating. In their work, they suggest treating tasks or levels like users and trying to give them a skill rating. By doing so, the player is matched with the most fitting level for his skill level. By doing so, in the video game “Paradox” they found that the player will often complete more difficult levels as well as a higher quantity of levels [28].

Dating Services When it comes to dating services matchmaking algorithms, most of these services allow the user to enter his preferences in some way like already covered, this can result in the user being treated unfairly by the algorithm [23]. Moreover, not only the algorithms are influenced by the user’s ratings of proposed matches by the algorithm, but also the user itself. As research suggests, even though the users try to show themselves as authentic as possible, the users often are seen tweaking their profile according to other user profiles they are interested in [29].

Because the matchmaking algorithm of Tinder, the most successful dating service using matchmaking with about 6.5 million monthly downloads[1], is secret, the way it filters and suggests profiles is not clear. One assumption about Tinder is that users can influence their ranking by paying extra money.

2.4 Recommender Systems

> "It is often necessary to make choices without sufficient personal experience of the alternatives. In everyday life, we rely on recommendations from other people either by word of mouth, recommendations letters, movie, and book reviews printed in newspapers, or general surveys." - Paul Resnick and Hal R. Varian [30]

What are Recommender Systems? Recommender Systems are used to predict information a user would like by applying machine learning. Quite a few different techniques are being applied including:

- Content-based filtering
- Collaborative filtering
- Demographic filtering
- Knowledge-based filtering
- Hybrid filtering

All of them are trying to achieve the same goal while using different paths to get the optimal recommendation for each user. Therefore, the suggested information provided by a recommender system varies from user to user, sometimes even for the same user on a different platform. This can be the case due to the fact that the recommender system can give a more detailed suggestion when getting access to the user's smartphone sensors, like GPS, compared to the data from a Computer [31].

[1] https://www.statista.com/statistics/1200234/most-popular-dating-apps-worldwide-by-number-of-downloads/

2.4.1 Which techniques are most commonly used?

When it comes to the real-life usage of techniques applied in recommender systems, the three most prevalent types are collaborative filtering, content-based filtering, and hybrid filtering [32]. As already mentioned, they are all trying to achieve the best possible set of recommendations for each user but differ in the way they are trying to produce those recommendations.

Content-based Filtering This type of recommender system bases the suggestions made on the similarity of items. This means that items, the user liked in the past, are compared to other items, and through different metrics, a similarity is calculated, after which the most similar ones are suggested to the user. The used metrics can include a simple list of keywords, for example, for movies, a similar title, actors, or genres can be included in the decision-making process. Depending on the type of item the recommender system is applied to, different features will be included in each item profile. Hence, the similarity calculation also differs depending on whether the items are, for example, movies, articles, or like in the case of Amazon, a huge list of very different items [32].

Collaborative Filtering Compared to content-based filtering, collaborative filtering focuses more on the history of the user itself by creating a profile for each user and creating relations to users that are interested in the same items [32][33], by doing so, a user-item matrix is created. As displayed simplistically in Figure 2.5, two users previously liked two similar items, hence their user profile similarity is recognized as high by the algorithms. This leads to the item the left user has not yet liked or disliked, being suggested to him, since a similar user bought it [34].

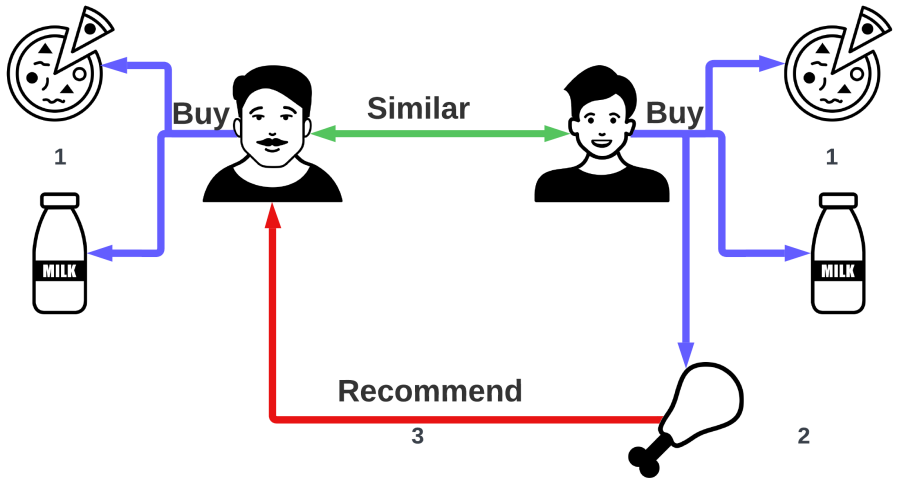

Figure 2.5: Concept of Collaborative filtering inspired by [34]. Step 1: Both users purchase the same products. Step 2: One user purchases a product the other does not purchase. Step 3: The system recommends the unique purchase of one user to the other user.

This process can also be applied to groups of users instead of each individual. Therefore, users are clustered, instead of being looked at individually. After being placed in a group of users, each user included in that group will receive the same recommendations, depending on which items fit their group most. This approach could also be classified as being a demographic recommender system [33][35].

Hybrid Filtering This group can not be generalized in the way they work but rather is a collective of all techniques, which can not be classified as one of the techniques mentioned above. Facebook, for example, uses a hybrid recommender system combining all of the techniques mentioned above [32]. Hybrid filtering focuses on picking aspects of different techniques to optimize the recommendations to best fit the expectations.

2.4.2 Which problems are Recommender Systems Facing?

Most recommender systems have similar struggles, while some problems can be rather specific to a small number of implementations. Some of the most prevalent problems include:

- Cold Start [32]
- Sparsity [32]
- Lack of Data [32]
- Over Specialization [32]
- Scalability [32]
- Changing data or user preferences [32]
- Data poisoning [36]
- Privacy of user [37][38]
- Trust [39]
- Cultural [40]
- Creation of bubbles [41][42]

Cold Start The problem of a cold start can arise in three different ways. The first is when a system is completely new and has little information about the existing users and items to create a reliable set of recommendations. This leads to recommendations for all users will be vague and mostly inaccurate. Secondly, a cold start can happen, when a new user joins the system and there is no information about the preferences of this specific user. This will lead to standardized results, not fitting the needs of the user. Lastly, when a new item is added to the system which is not similar to any item currently in the system. Mostly, this leads to the item not being suggested to anybody because the similarity to both other items in general and the recently searched items by users is very low [32].

Sparsity In collaborative filtering, the problem of sparsity arises when a user rated too few items to successfully create this user's user-item matrix and predict other items they like [32]. It is similar to the cold start because it occurs due to a lack of data

Lack of Data This problem is closely related to the cold start because, especially at the beginning of a system, there is too little information in most cases to produce satisfying recommendations. Moreover, one can face the problem of a lack of data, when the users of the system get inactive, hence producing not enough data to keep the recommendations up to date[32].

Over Specialization When a recommender system focuses on a too specific topic, the quality of the produced recommendations may suffer. When a grocery shop offers a lot of different types of sugar, they will get too hard to distinguish and therefore recommendations will get worse [32].

Scalability Scalability problems may be faced when the system grows, but the software is not optimized to handle bigger amounts of data, therefore slowing down the recommendation process or completely breaking down. But not only software can cause problems in this regard, but also hardware can bring up problems. If the hardware, the recommendations are calculated on has too little RAM or a processor that is too slow, the system will get inefficient [32].

Changing data or user preferences This problem arises, when the attributes of an item change, for example, the product name or ingredients, as well as when the preferences of a user change. These problems should be considered when building a recommender system because in most systems, attributes of an item or user preferences are bound to change over time [32].

Data Poisoning Collaborative filtering recommender systems can be very susceptible to data poisoning, meaning that fake users are generated to achieve a target. These fake users are created in a way to be as similar to a big group of other users while also, for example, liking the item they are trying to promote in a malicious way. These attacks can be detected but by only creating a few fake users that only try to promote a small number of items in a big recommender system, like Amazon, detection gets progressively harder [36].

Privacy of user In today's world, privacy is becoming more prevalent than ever. To enhance collaborative recommender systems, it can be useful to combine the databases of multiple companies to improve the recommendation process. This, on the other hand, can be a huge privacy problem. Canny [37] suggested a privacy-preserving collaborative filtering algorithm. The user's recommendations are generated without the usage of private data. There are many other approaches to solving the problem of privacy in recommender systems, with only a few of them having a high accuracy [38].

Trust Trust is not a simple concept and therefore hard to include in recommender systems. A recommender algorithm should consider the following properties [39]:

- **Strength**: Not every relationship has the same level of trust; therefore differentiation is needed
- **Transitivity**: If user A trusts user B and user B trusts user C, then user A will most likely have a certain degree of trust in user C
- **Composability**: How different recommendations from different users for the same item influence the user the recommendations is created for
- **Dynamicity**: Trust of user A to user B changes of time, therefore strength of trust must be adapted
- **Assymetry**: If user A trusts user B the system can not assume that user B trusts user A in the same way
- **Context dependence**: User A trusting user B on one topic does not always imply that this is the same on another topic

Cultural Most recommender systems as well as the studies done on them are focusing on the English language. Whenever a recommender system is created for other languages, maybe even other writing systems, like for example the Arabic script, research is rarely done. Therefore, the implementation of such a recommender system might face problems due to less research. Although recommender systems for Arabic content are not researched to the same degree as the ones for English content, they are already achieving very satisfying results in, for example, sentiment analysis [40].

Creation of bubbles Research disagrees with the existence of filter bubbles in recommender systems. Due to the fact that collaborative filtering algorithms are trying to optimize the recommendations by grouping users up, some argue that homogeneous bubbles are created, which, especially on social media platforms, may lead to extremists being grouped up and therefore radicalizing more. This often leads to, for example, Facebook displaying less different-minded content and only showing the content of similar-minded users [41].

Röchert et al. [41] showed that right-wing populism networks are created as isolated communities within the social media platform itself. This shared information often even includes misinformation, sometimes called “fake news” [41][42].

2.5 Multi-Platform Development

This section discusses the background and related work of Multi-Platform development, also known as Cross-Platform development.

Multi-Platform Development is currently growing in popularity, mostly within mobile development, due to the popularity of the two mobile operating systems – Android (72% of the market [43]) and iOS (27% [43]). Due to the importance of Flutter and Dart, those frameworks will be the main focus later in this section of this work.

2.5.1 What is Multi-Platform Development?

The need to make use of a type of Multi-Platform Development is growing to cover various platforms and markets with a single code base, instead of two or more codebases with very similar, yet incompatible for other platforms. Even though native applications allow for broader and more optimized use of features of devices, such as optimized cameras, GPS, or calendars, in the end, every platform needs its own code base, exponentially increasing the effort and development cost [44].

There are many ways to implement Multi-Platform applications, some of which will be covered next in this work.

2.5.2 Types of Multi-Platform Implementations

There are various ways to implement Cross-Platform applications. The main focus later on in this section will be on Flutter, but besides that, other approaches to Multi-Platform development are [44]:

- Mobile Web Applications
- Hybrid Applications
- Interpreted Applications
- Cross-Compiled Applications

Mobile Web Applications One way to implement a Multi-Platform Project is to circumvent common problems that occur when adapting to multiple operating systems by running the application in a browser. Such applications are based on common web technologies such as HTML, CSS, and JS, among others. This also allows the application to be run on any device without an installation process.

On the downside, these applications are not optimized for the device and therefore often face restrictions when it comes to the usage of sensors, cameras, and other tools the device would offer to native implementations [44].

Hybrid Applications Similar to Mobile Web Applications, so-called Hybrid Applications, are based on commonly used web technologies like HTML, CSS, and JS. The main difference to Mobile Web Applications is that Hybrid Applications are not run within a browser, but instead through an application that is installed on the device. This enables the application to access more features of the device, like camera or GPS, while allowing the developers to reuse most of the code for different platforms because most of the execution is done through a web container on the device using API calls [44].

While this type of implementation offers a solution to a lot of problems Mobile Web Applications are facing, the execution through web containers still often is significantly slower compared to a native application [45].

One of the most widespread frameworks for such a hybrid application is PhoneGap which is based on Apache Cordova and uses JS, HTML5 as well as CSS3 for its cross-platform hybrid implementations [46][47]. However, PhoneGap faces the same problems most other hybrid applications also struggle with: slow response times and missing a native design, hence not feeling natural to the user [47].

Interpreted Applications A not-as-widespread solution to the implementation of cross-platform projects are interpreted applications. These are applications that are for the most part translated to the native code for each platform, but the remaining part is translated or interpreted during the runtime of the applications [44]. Ruby and Java are the most commonly used languages for such a project, with Appcelerator Titanium being the most popular environment for software development. While being dependent on the platform, this type of Multi-Platform development allows for a native design of interfaces [44].

Cross-Compiled Applications This type of Multi-Platform implementation compiles the application natively to be executable on multiple platforms. Notable examples of such a framework are Microsoft Xamarin, which uses C#, or Applause, which uses its own programming language, namely Xtext [44]. In recent years, Meta, formerly known as Facebook, published their own framework that allows for cross-platform compilation with React Native. Google followed shortly after with their Flutter framework, which makes use of Google's programming language Dart [43].

Cross-Compiled Applications allow for a single code base that compiles to a native application, while also solving problems of the formerly discussed multi-platform approaches, such as execution speed, and not being able to use native interfaces among others.

Due to the importance of Google's Flutter framework, the next section of this work will discuss it in more detail.

2.6 Flutter

Due to Flutter's importance for this work, this section covers background and related work for Flutter, including the peculiarity of Flutter, as well as comparisons to other frameworks.

2.6.1 What is Flutter?

Flutter is mainly used to develop iOS and Android applications, while also allowing the development for Web, Windows, Linux, and macOS with a single codebase.

While allowing for lower development and maintenance costs due to the existence of only a single codebase compared to multiple native ones [43], Cheon and Chavez [48] showed that the needed SLOC (Source Lines of Code) are about 37% in the Flutter codebase, therefore also decreasing costs of development.

Google's Flutter framework' 1.0 version for cross-platform applications was launched in 2018 [2] and, as already mentioned, makes use of Google's programming language, called Dart, which was first released in 2011[3].

2.6.2 Structure of Flutter

Flutter is a widely spread cross-platform framework, which, unlike most of its competitors, offers a close-to-native performance. Before we have a closer look at performance comparisons, this section will take a closer look at how Flutter is structured and how it works.

Flutter allows the user interface to be directly coded, instead of using a markup language, like Android with XML [48]. The main paradigm of the user interface design of Flutter is based on widgets. Pretty much everything within Flutter is a widget, ranging from simple types like text fields, buttons, icons to more complex types, such as gestures, themes as well as padding, with most of them allowing to be nested within each other to allow the construction of the user interface. According to Cheon and Chavez, this paradigm of "everything is a widget" and the extensive use of nesting leads to some developers calling the user interface development process of Flutter a "nested hell". Worth mentioning is the fact that it is possible to flatten this nesting, by implementing and splitting everything into smaller methods, allowing for better maintainability and readability [48].

[2] https://docs.flutter.dev/resources/faq

[3] http://googlecode.blogspot.com/2011/10/dart-language-for-structured-web.html

Widgets can be divided into two big groups: On one hand, there are *StatelessWidgets* and on the other hand, *StatefulWidgets*. Each widget is designed to look like a mixture of iOS design-like Cupertino and Android design-like Material catalog [49].

- **StatelessWidgets**: are a lesser complex type of those two. They basically will not change their state after the creation. Since all the class variables are supposed to be final, the only way to change this type of widget is by calling its constructor with other values, which should be avoided for the most part [49].
- **StatefulWidgets**: allow of different states of the widget without repeatedly recalling its constructor, but instead by calling a designated *setState* method. This type of widget includes two classes, the StatefulWidget itself and another class handling the state of it. Once a change of a class variable leads to a change in the user interface, the setState method is called. By doing so, the state's build method is called, and the widget is rebuilt within the user interface [49].

2.6.3 Architecture of Flutter

Flutter's architecture is based around three main layers[4], each of which is split into multiple layers on their own, as seen in Figure 2.6 [49].

The lowest layer, the Embedder layer, is different for every platform, Flutter can be compiled for. Moreover, this layer allows Flutter to include native plugins and features, such as GPS or camera, allowing Flutter to seem like a native application.

The C and C++-based middle layer, also known as the Engine-Layer, allows Flutter's good performance. Additionally, this layer includes its own rendering engine. Therefore, Flutter applications will look the same on every platform, by not running through each platform's native rendering engine.

[4] https://docs.flutter.dev/resources/architectural-overview

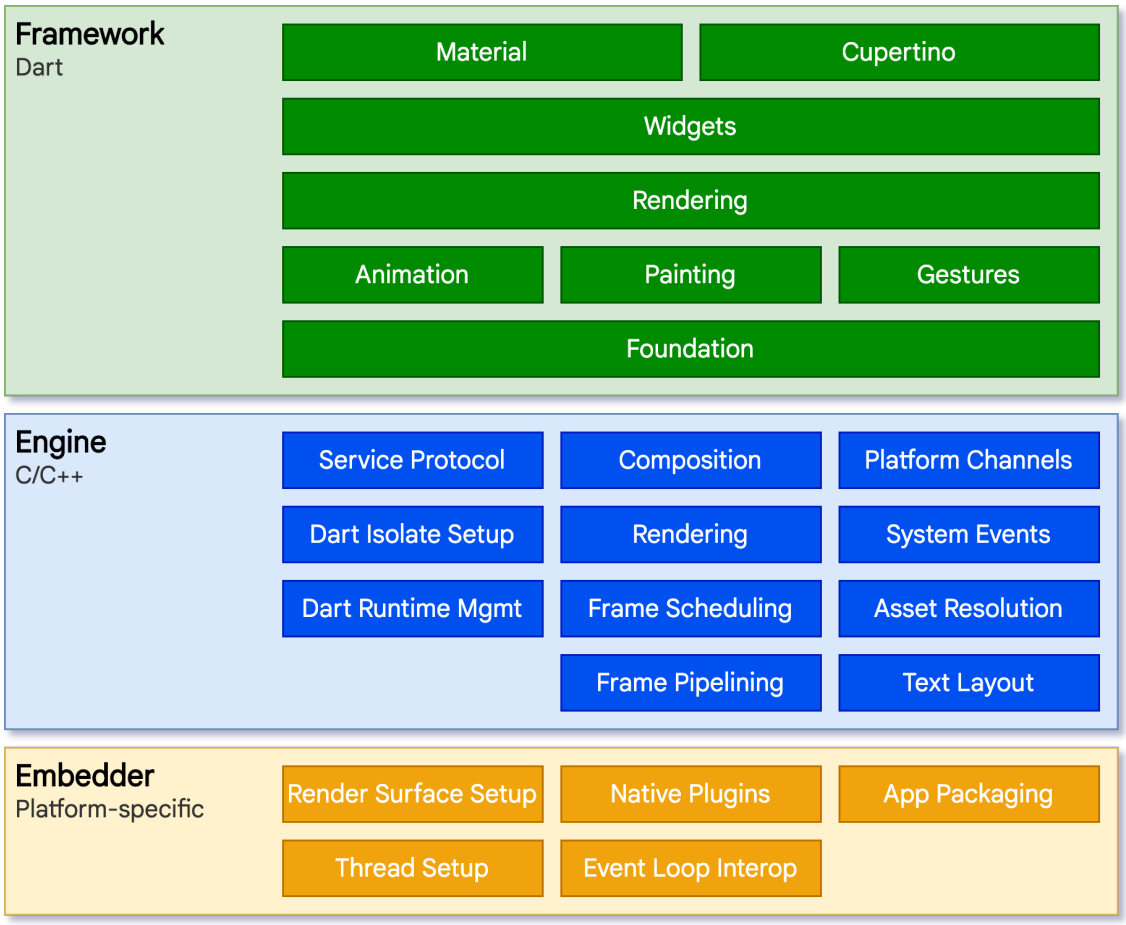

Figure 2.6: Architecture of Flutter by flutter.dev[5].

The layer for the framework of Flutter is based on Dart. This includes the widgets we already looked at in this work with all their different facets, like gestures, animations, looks, and everything else surrounding them.

2.6.4 Libraries for Flutter/Dart

Libraries for Flutter or Dart are called packages. The basic packages included in the Flutter SDK allow for a broad variety of widgets and features but depending on the usage, are often lacking different features [48]. An example of a lacking feature are notifications, which are not possible without an external package. In the development process of Flutter or Dart applications, usage of third-party packages, which can be found on Flutter's own platform[6] is widespread [48].

[5]https://docs.flutter.dev/resources/architectural-overview
[6]https://pub.dev/

Compilation Dart supports two different types of compilation, depending on the compilation mode:

- Firstly, ahead-of-time compilation, which is used in release mode, produces the SDK ready to be run on the end user's device or a VM.
- Secondly, Dart and therefore Flutter allows for just-in-time compilation, allowing for a hot reload within the VM, while in debug mode. By enabling just-in-time compilation, the developer can simply reload the VM instead of restarting it to apply changes to the source code [50].

2.6.5 Testing Flutter applications

Flutter allows its applications to be tested in three different automatic approaches [49]:

- **Unit Tests**: Within Flutter, unit testing allows the developer to test each method and class by defining lambda functions which check if the values returned by each method or class are the expected ones.
- **Integration Tests**: This type of testing is used to control, whether different components work together as intended by the developer.
- **Widget Tests**: Like the name suggests, widget tests are used to test a single widget or a group of widgets and their behavior. Conveniently, this is possible without the use of an emulator of either iOS or Android, but instead directly within Flutter.

2.6.6 Comparison to alternatives

In this part of this work Flutter will be compared to its closest competitor, Facebook's React Native, as well as to native Android implementations.

Comparison Flutter vs. native Android Android applications are mostly written in Kotlin or Java. For their comparison between Flutter and Android, Cheon and Chavez [48] rewrote an Android application written in Java to Flutter. While doing so, they faced a few differences that needed adaption, to name some examples include concurrency, where Android uses multi threading and Flutter isolates, or, as already mentioned, the user interface design, where their Android application used XML as a markup language while Flutter allows for the interface to be coded using Flutter's widgets.

While rewriting the application they found that the final Flutter application, with the same functionality as the Android application, had a 37% smaller codebase, with the share of the codebase for user interface design, network and model interestingly staying approximately the same. Cheon and Chavez did not discuss performance comparisons between Android and Flutter [48].

Comparison Flutter vs. React Native In their comparison between React Native and Flutter, Mota and Martinho mostly focused on the performance side to be more precise on the launch of the application, lists displaying local and remote content, camera for image and video capture, accessing local and remote multimedia content and lastly, various animations [43]. See table 2.2 for their results.

	iOS		Android	
Features	React Native	Flutter	React Native	Flutter
Launch	2 ✓	1 ✗	2 ✗	3 ✓
Local Lists	9 ✓	6 ✗	3 ✗	15 ✓
Remote Lists	3 ✗	12 ✓	8 ✗	10 ✓
Take Photos	0 ✗	4 ✓	0 ✗	5 ✓
Record Video	4 ✓	0 ✗	1 ✗	4 ✓
Access Image	0 ✗	4 ✓	0 ✗	5 ✓
Access Video	0 ✗	4 ✓	1 ✗	4 ✓
Streaming	3 ✓	2 ✗	2 ✗	3 ✓
Animations	2 ✗	3 ✓	0 ✗	6 ✓
Total	23 ✗	36 ✓	17 ✗	55 ✓

Table 2.2: Comparison performance of features React Native vs. Flutter on Android and iOS by Mota and Martinho[43]. "✓" marks the better value in each rating, while "✗" marks the worse value.

Each test case was developed in React Native as well as Flutter and compiled to run on iOS and Android, with each device being connected to the same network.

By appointing points for each test and applying a leverage system, Mota and Martinho show that in a vast majority of use cases, Flutter outperforms Facebook's React Native, with them marking that in some scenarios React Native is the better option in a special case [43].

As seen in table 2.2, on Android every tested feature Flutter outperformed React Native but on iOS for some features React Native has the upper hand.

Interestingly, even though React Native shows a stronger performance on iOS, in general in the test cases for remote lists it did, struggle on iOS while being close to Flutter on Android. Overall, Flutter shows a better performance on both operating systems by scoring about 57% more on iOS and 223% more on Android when compared to the points scored by the React Native applications [43].

Lastly, when comparing the metrics of execution time, CPU usage, RAM usage, and Frames per Second (FPS), and again awarding points as well as applying a leverage system. Mota and Martinho concluded that only in RAM usage, React Native has the upper hand on iOS. Especially being outclassed by Flutter when it comes to FPS, as seen in table 2.3

	iOS		Android	
Metrics	React Native	Flutter	React Native	Flutter
Execution Time	11.45 ✗	13.92 ✓	10.04 ✗	16.84 ✓
CPU	2.09 ✗	15.94 ✓	14.45 ✗	25.64 ✓
RAM	38.98 ✓	8.18 ✗	3.83 ✗	25.51 ✓
FPS	7.54 ✗	28.56 ✓	0.00 ✗	401.24 ✓
Total	60.07 ✗	66.60 ✓	28.32 ✗	469.22 ✓

Table 2.3: Comparison performance during feature tests React Native vs. Flutter on Android and iOS with leverage system applied by Mota and Martinho[43]. "✓" marks the better value in each rating, while "✗" marks the worse value.

A pattern already seen in the feature tests is that React Native is closer to Flutter when compiled to iOS compared to the results for the tests run on Android.

Overall, Mota and Martinho concluded that Flutter is the better alternative to a native implementation due to its good performance in a variety of features on both tested platforms. Worth noting is that on iOS React Native showed to be a good alternative, particularly in RAM usage. Therefore Mota and Martinho recommend React Native for applications with few visual effects and a local SQLite database for iOS while recommending Flutter for cross-platform Android and iOS applications with more complex visual impacts due to its strong performance in the FPS metric [43].

2.7 Databases

This section covers background information on databases, focusing on the difference between SQL and NoSQL databases, with a focus on Google's NoSQL database, Firebase.

2.7.1 What is a database?

The main target of a database is to store data and make it as easy as possible to access, edit and manage. Data within a database can range from personal information about users to GPS locations of places and everything in between.

Fast access to the data within a database is getting increasingly important in the world of today, hence new types of databases and data structures arise. Currently, the most commonly used type of database is a relational database using SQL. Other types of databases include distributed databases, cloud databases, or the currently attention-gaining NoSQL databases [51].

2.7.2 What are relational databases?

Overall, relational databases are the most common type of database in use to date with SQL being the most widely-known type. SQL stands for "Structured Query Language" and is a standardized language to interact with relational databases [52].

Relational databases structure their data within tables split into columns and rows. Each row normally includes all the data for one entry, while each column houses the same type of data for multiple entries. The main functions SQL offers are [52]:

- **Data definition**: user defines the structure of data as well as relations between entries
- **Data retrieval**: user may retrieve data
- **Data manipulation**: user can add new data, edit existing data or delete existing data
- **Access control**: user can restrict access to data for other users
- **Data sharing**: multiple users may access at the same time without concurrency problems
- **Data integrity**: data is protected from corrupting, due to SQL's integrity constraints

SQL databases follow the ACID properties, which include atomicity, consistency, isolation, and durability. The goal of those properties is to ensure the correctness of the database [51].

- **Atomicity**: The atomicity property means that operations within a transaction must either be all executed successfully or none are executed.
- **Consistency**: Consistency means that every transaction is executed in isolation to ensure database consistency.
- **Isolation**: Isolation ensures that concurrent transactions are run isolated – meaning that even if there are interim results, other transactions will not gain access to the results until the whole transaction is done.
- **Durability**: The durability property ensures that after a transaction has been completed, the changes will persist within the database, even in the case of failures in subsequent transactions.

2.7.3 What are NoSQL databases?

NoSQL stands for "**N**ot **O**nly **SQL**" and a variety of NoSQL databases are currently gaining traction, due to their flexibility and support of big data. Most NoSQL databases allow for unstructured data, which is often too complex for typical relational databases within the SQL category. Example use cases are ranging from webcam data, over huge collections of documents to typical SQL database problems [53].

Unlike relational databases, NoSQL databases do not require the same number and types of data columns within a collection, which is the name often used for tables within NoSQL databases. This leads to data within a collection, not having the same characteristics. Moreover, they generally do not follow the ACID rules [54][51].

NoSQL databases offer a diversity of structures, but most may be classified within these three categories [54]:

- Key-Value Stores
- Document Stores
- Graph Databases

Key-Value Stores Key-Value stores make use of a dictionary type of accessing data. Each entry has a unique key that allows the user to access data stored in an entry. The structure of each entry may differ, unlike in relational databases. Example databases would be Voldemort and Redis [54].

Document Stores Document Stores use JSON files or JSON file-like structures within their entries. Each entry, often called documents in NoSQL databases, contains an ID unique to its collection. Due to the JSON-like structure, nesting is possible and allows for documents to contain a theoretically infinite number of sub-documents, with each of those may differ in their stored data and data types.

Databases of this type are for example MongoDB, CouchDB, and Firebase, which will be covered in more detail later in this section [54].

Graph Databases Focusing on linked data, graph databases allow for data to be stored based on relations to other data within the database in graphs. GraphDB, Neo4J, and BigData are databases making use of this structure [54].

Firebase Firebase, or to be more precise, Google's Firestore, is a NoSQL database that can be classified as "document stores" database. Therefore, data is stored in JSON-like nested documents within collections. Each document can have a structure differing from another, while also including sub-documents but not allowing a many-to-many relationship [55].

Firestore allows for real-time updates and unlike SQL does not require any server-side code [56]. Native support for Android, iOS, Windows, macOS, and Linux applications through the support of implementations in Java, Flutter, and C++ among many others, allows Firestore databases to be used for a wide variety of projects [57].

Unlike most other databases, Firebase allows the user to natively cache data and therefore keeps it available while having no network connection and automatically updates all the changed data once a connection is established again [57].

Table 2.4 shows the pricing model of Firebase and Firestore.

Cloud Firestore Pricing		
Pricing Plan	No-cost up to	Pay as you go
Stored data	1 GB total	$0.18/GB/month over No-cost
Network egress	10 GB/month	$0.12/GB over No-cost
Document writes	20.000 writes/day	$0.18/100.000 writes over No-cost
Document reads	50.000 reads/day	$0.06/100.000 reads over No-cost
Document deletes	20.000 deletes/day	$0.02/100.000 deletes over No-cost

Table 2.4: Pricing overview of Google's Firestore service in use as a database within this project

2.7.4 SQL vs. NoSQL

Over recent years, there were a lot of papers comparing SQL and NoSQL databases, some of which will be discussed here.

Li and Manoharan [58] attempted to compare Microsoft SQL with a total of six different NoSQL databases, namely, MongoDB, RavenDB, CouchDB, Cassandra, Hypertable, and Couchbase. Their goal was to compare those seven databases in the four main database operations - instantiate, read, write, and delete. Each database needed to run these operations on different datasets, varying in size from 10 up to 100,000 operations. Their results showed, that only MongoDB and Couchbase could outperform Microsoft SQL in every test case, while all the other NoSQL databases showed performance problems in different aspects. It is worth mentioning that MongoDB outperformed Couchbase and SQL by a factor of five or more when it came to the instantiating test case, which can be seen in Figure 2.7 [58].

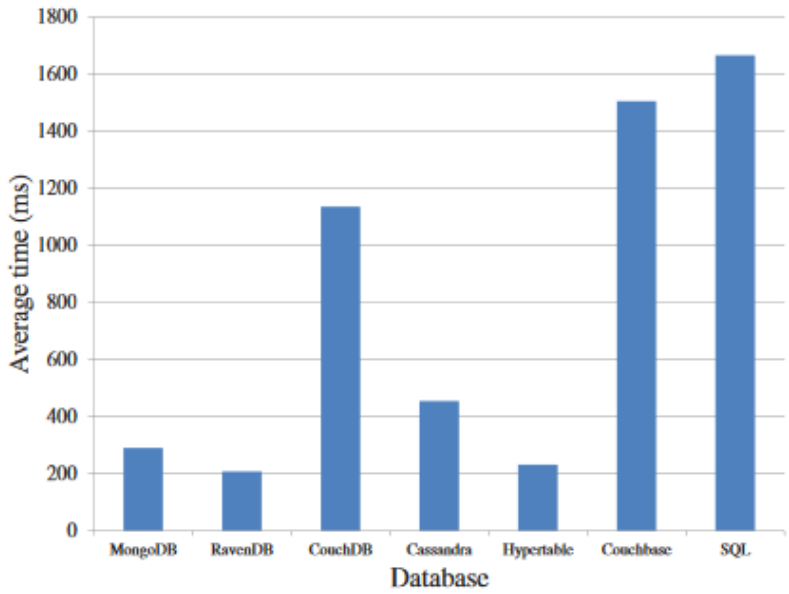

Figure 2.7: Comparison of the seven databases analyzed by Li and Manoharan when it comes to instantiating[58]

Another paper by Parker et al. [53] compared MongoDB and an SQL database not only by running the four operations mentioned earlier, but also by updating non-key attributes and aggregate queries. They found, just like Li and Manoharan, that MongoDB outperforms for inserts, updates and simple queries, but compared to SQL struggles with non-key updates and queries and aggregate queries [53][58].

Last but not least, Ohyver et al. [56] compared the Firebase NoSQL database with a MySQL database as database for their daily nutrition needs mobile application. Their tests compare the two databases by running create, read, update and delete operations on test cases with one to 3000 entries and after applying the Wilcoxon Signed-Rank test to the results, which is of no special interest. Their results show that Firebase Realtime database, also known as Firestore, outperforms MySQL in every tested operation, but for the update and delete operations MySQL slightly outperformed Firestore when it came to test cases with more than about 2500 entries. They concluded that Firebase outperformed MySQL in their tests and was hence deemed more suitable for their mobile application [56].

2.8 Angular

This section will cover some background information on the Angular frontend JavaScript/TypeScript framework.

2.8.1 What is Angular?

Angular was first introduced by Miško Hevery and Adam Abrons as an open-source frontend framework in 2009. Their main goal was to achieve a highly productive web development experience with their JavaScript framework. In 2010, it was released as a front-end web application framework by Google, where 1.8.3 is the currently active version as of 2023 [59].

It is also used by Google, YouTube, Facebook Applications, Nike among a lot of other well-known brands to implement their frontend [60].

There are a few rare or unique solutions within Angular including:

- MV*
- Components & Templates
- Data Binding
- Dependency Injection
- Angular CLI
- First-Party Libraries

MV* Also known as Model View Whatever is a version of the well-known Model View Controller pattern, but instead of a Controller, the programmer may choose whatever he wants, be it MVC, MVVM, MVP [60].

Components & Templates Components are like the building bricks of an Angular project. Each component contains an HTML file, a CSS style file, and a TypeScript class with a Component decorator. Templates are used to tell Angular how to render components. This is done within the HTML file, and each HTML file may refer to other components and therefore to their HTML file. This, for example, allows for the creation of one header component. This header component is then used as a part of each template to create each view, allowing for fewer duplicate code [59].

```
<input type ="text" id="hero-name" [(ngModel)]="hero.name">
```

Listing 2.1: Example of two-way data binding in Angular

Data Binding Data Binding allows for variables to be used within both, the HTML, as well as the TypeScript file. User input directly saves variables within the TypeScript class corresponding to the HTML file.

Moreover, Angular uses two-way data binding, which allows for instantaneous communication between the HTML file and the TypeScript file. Listing 2.1 displays, how the implementation of two-way data binding looks within the HTML file. In this example, the class “hero” has a variable named “name” which may be changed from either the user, within the text field, or from the TypeScript side, for example through a change within a database[7].

[7] https://angular.io/guide/binding-syntax

Dependency Injection Angular's dependency injection pattern allows for the dependencies within TypeScript classes without caring about instantiation. Whenever a TypeScript class depends on another, the file is injected and Angular automatically handles the instantiation[8].

Angular CLI Angular CLI is a command-line interface tool to build, develop, and test Angular applications. It is not the only way to do so, but the recommended way. The most commonly used commands include[9]:

- **ng build**: Builds the Angular application to the output directory
- **ng serve**: Builds the Angular application and runs it within the command line. Also rebuilds on code changes
- **ng test**: Automatically runs unit tests for the project
- **ng e2e**: Like "ng serve" but also runs end-to-end tests

First-Party Libraries Angular offers a wide variety of first-party libraries. This list includes Angular Router, for client-side navigation and routing. Angular Forms is a library to build forms, also supporting the validation of forms. Angular HttpClient supports client-server communications. Alongside a few other easily usable first-party libraries essential to building the client-side of a web application[10].

[8] https://angular.io/guide/dependency-injection
[9] https://angular.io/cli
[10] https://angular.io/guide/what-is-angular

Chapter 3

Architecture and Design

This work consists of two main code bases for applications that are meant to be used by end users. Besides that, a web crawler, a Firebase database, and a planned integration within the TU Graz Single-Sign-On system are part of the project's architecture.

3.1 Elements of the Architecture

There are two targeted code bases within this work: mobile and web.

For the mobile application, the target end users are students looking for projects for their master's or bachelor's thesis. The web platform targets university professors looking for students to work on their suggested theses or other projects.

Moreover, a smaller side project is a web crawler, that is used to collect written-out projects of institutes of TU Graz, to get a broader variety of projects for the students to choose from. The crawler collects projects and theses from the official websites of each institute and stores them in our database, therefore making them visible to the students within the mobile application. Also, lecturers may edit their projects within the web application.

A central part of this work is the database. In our case, a Firebase database is used, which is a NoSQL database system hosted by Google offering a well-documented implementation within the Flutter framework used for our mobile application. This database holds all the information for our code bases. More details and a detailed overview of the database structure will be covered later in this chapter.

Lastly, an integration within the TU Graz Single-Sign-On system is planned, to allow the lecturers to easier access the web platform of this project and therefore allow them to post and edit their projects and theses for the students.

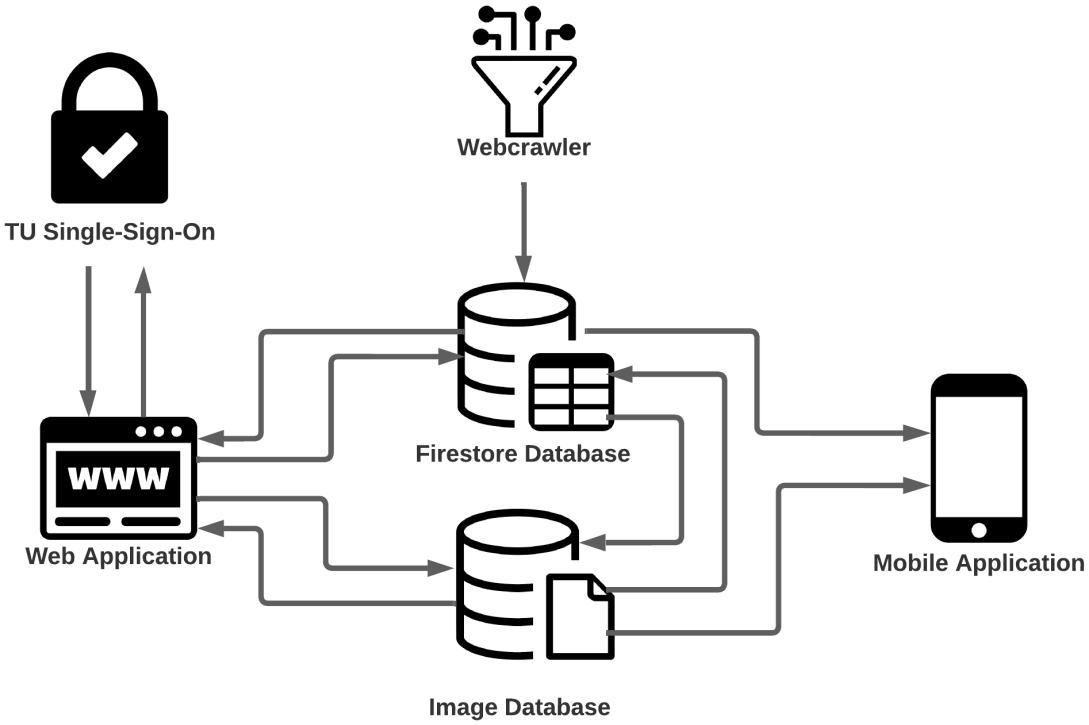

Figure 3.1: Overview of the architecture of this project

How are the main components of this project's architecture working together? As shown in Figure 3.1, the central part of this project is the Firebase database, which consists of the Firestore database storing data for both platforms of this project. It gets its data from the web application by the lecturers, the web crawler as well as the image database.

The Firebase database holds two main tables, one for feedback and one for the projects. Within the feedback table, feedback of students, as well as lecturers, is stored. The project table holds everything needed for projects to be presented to the students. More details on the database can be found in section 3.2.

These two database tables are directly accessed by the mobile and web platforms. The mobile application stores the feedback data anonymously directly into the feedback table while retrieving data from the projects table. Data of students is not stored within the database, but instead, locally on the user's device. This will be described in more detail within chapter 4.

- The mobile application may only save data to the feedback table and only retrieve data from the projects table.
- The web application posts data to the feedback table and retrieves data from the projects table, as well as being able to edit and add new projects to the projects table.
- The web crawler can only access the projects table, while its main focus is to add new projects and to check for duplicates, it may also retrieve existing projects from the projects table.

3.2 Database structure

The database structure is split into two parts, one database with two collections or tables (one for feedback and one for projects) and another database for images.

The database system and its features The service used for the database is Cloud Firestore, which is a sub-product of Firebase and can be classified as a NoSQL database. This means that there is no defined structure and every entry within the database may have different content types and sizes, which is not the case in our project.

Moreover, Firestore allows for entries to have sub-tables with their own entries, which can have sub-tables again, and so on until a depth of 100. On top of that, NoSQL also means that queries are fundamentally different from the more broadly used SQL queries. Firestore does not provide the user with a system of automatically incrementing ID system, but with the possibility to give each entry a randomly generated string as a unique identifier.

Another feature offered by Cloud Firestore is the native support of offline data. Data requested by a device is cached and can be used offline and once a network connection is established again, it will be synchronized with the database again.

Lastly, this system offers real-time updates, which allow data to be changed in run-time and updated instantly once a change is detected in a used entry within the database. Most data in our case is retrieved once, but this feature is still in use in this work and provides a simple and efficient way to update data on run-time without the programmer needing to check due to the native implementation within Firebase's methods.

3.2.1 Terminology of Firestore

Due to Firestore being a NoSQL database, terminology differs from what is known to users familiar with SQL. What would be considered a table within a SQL database may also be referred to as a collection. Each table has entries or documents, which is what they are mostly called in Firestore. These documents, on the other hand, either contain data, which is saved in fields within the document, or sub-tables, also called sub-collections. Each document may hold one or more sub-collections while also having data saved in fields at the same time.

3.2.2 Database tables

Figure 3.2 displays the structure of the Firebase NoSQL database and its two collections: feedback and projects.

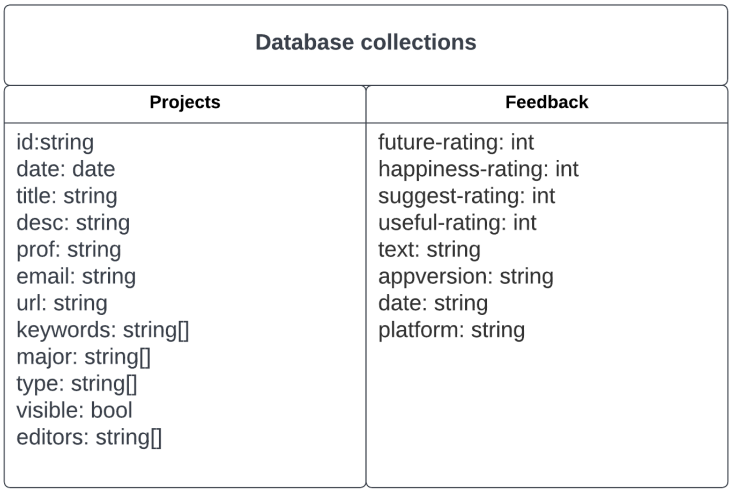

Figure 3.2: Database structure

Feedback collection Within the feedback table, as the name suggests, feedback on our system provided by students and lecturers is collected, and it includes the following content for each entry:

- **future-rating**: whether the user intends to use the software in the future
- **happiness-rating**: how happy the user is with the application
- **suggest-rating**: how likely the user is to suggest the application to others
- **useful-rating**: usefulness of the application to the user
- **text**: text field for the user to add additional feedback
- **appversion**: app version used when posting this feedback
- **date**: date, when feedback was added to the database
- **platform**: web or mobile

The data type of each field within a document of the feedback table can be seen in Figure 3.2. Each int value in this case ranged from 1-4, depending on the provided rating from the user. The range was chosen, so the user has to choose a side, rather negative or positive, so there are no neutral feedback forms.

Projects collection This table holds all information on projects, except the images, which will be covered later in section 3.2.2. Each document of the project's collection has the following fields:

- **id**: unique identifier as a string
- **date**: date of last update
- **title**: title of the project
- **desc**: project description
- **prof**: lecturer in charge of project
- **email**: email contact of lecturer in charge
- **url**: website of lecturer or institute
- **keywords**: list of two keywords to describe each project, such as "AI" or "complex numbers"
- **major**: majors of students the project should be suggested to
- **type**: type of project, may be bachelor, master, project, or other. Length: 1-4
- **visible**: whether the project is visible in the mobile application or only editable in the web
- **editors**: email address of other users allowed to edit the project

Figure 3.2 shows the data types of each data field within an entry of the projects' collection. Some fields, such as the prof, email, or URL field, are mostly self-explanatory, and others, like the keywords or visible, will be explained in more detail, within chapter 4.

Image storage Everything about projects, except the title image of the project, is stored in the projects' collection. These images are stored as PNG or JPG files within the Storage service of Firebase. The service allows files to be stored in a cloud service, which is connected to the project within the Firebase system. Every project has one title image. To connect the file to the project, each image is named with the ID of the project it belongs to.

In some cases there is no image connected to the project. Therefore, the institute's logo or the TUGraz logo will be displayed instead.

3.2.3 Database requirements

The main requirements for this systems' database were:

- Supported by Flutter and Angular/node.js
- Simple to expand
- Stable service - high uptime
- Image storage

3.3 Mobile Application Software Architecture

The main goal of the mobile application's architecture was to be as easy to understand and use for the end user as possible.

3.3.1 Why Flutter?

The design decision to use the Flutter framework was made, due to the possibility to compile to multiple platforms, most notably iOS and Android. Moreover, Flutter offers a wide variety of packages, especially compared to other cross-platform frameworks, which made it the best choice for this project's requirements and plans. Furthermore, the Flutter framework offers simple implementations for a multitude of problems, such as routing and Firebase, which will be covered in more detail in chapter 4.

3.3.2 Requirements of Mobile Application

When discussing the views and features of the mobile application, the following features and requirements were chosen to be essential and will be discussed hereafter:

- Local User - User settings (aka. Start) View
- User feedback - Feedback View
- Swiping Projects - Swipe View
- Overview of liked Topics - Liked Topics View
- More detailed Projects - Project View
- Contact lecturer of project - External Mail-Application
- Same design across all platforms
- Low network traffic
- Matchmaking within Swipe View

All views are displayed in Figure 3.3. The next part of this work will cover more detailed explanations of each requirement or view and their importance for the project.

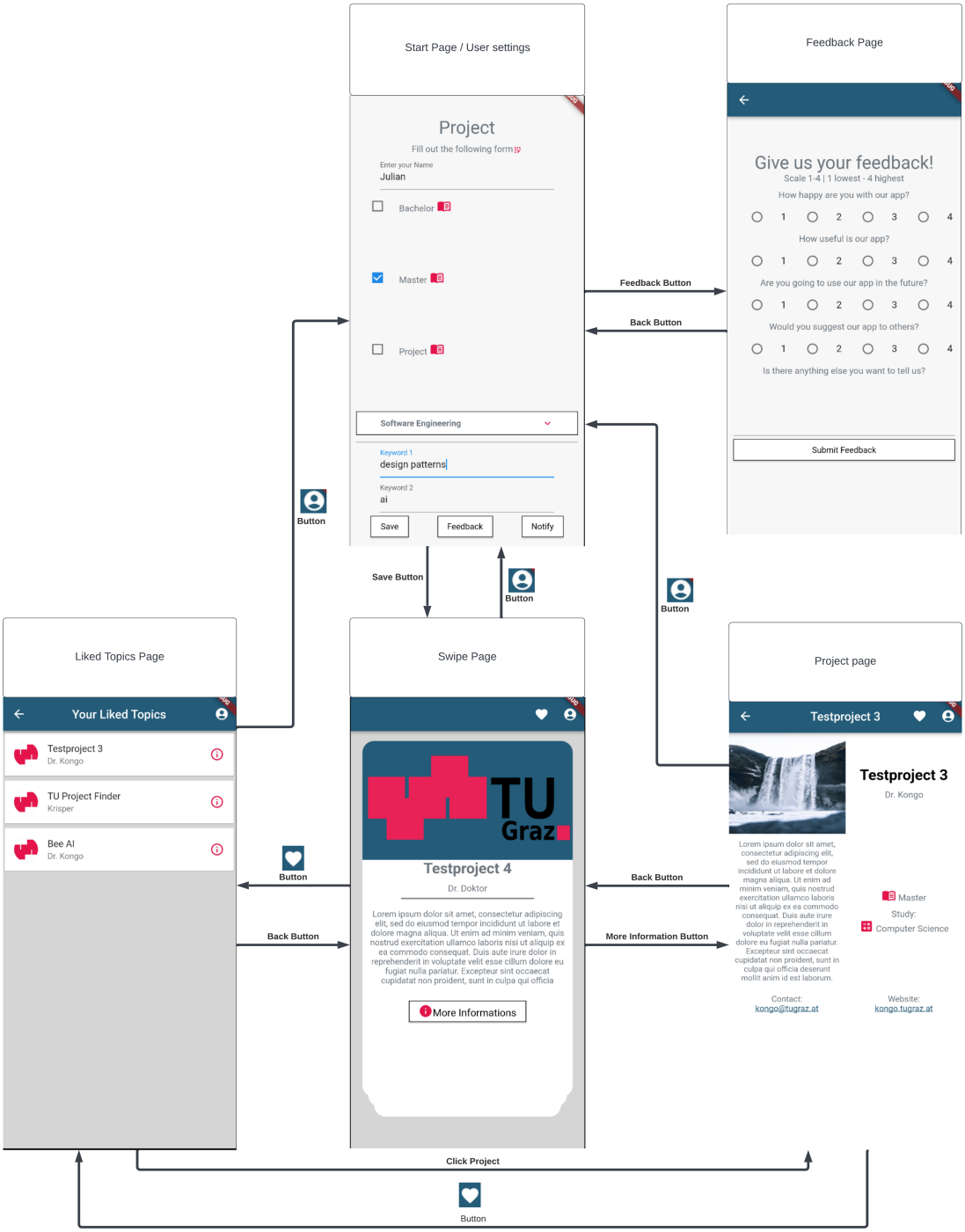

Figure 3.3: Views of mobile application

Local User - User settings View To preserve the user's anonymity, we chose to only locally store user data on the user's mobile device. Besides allowing for better user anonymity and no login system, this minimizes database costs because fewer requests to the Firebase database are needed.

The User settings View is the first view a user sees when opening our mobile application and due to Flutter being cross-platform, it will look the same on iOS and Android. After filling out all the information, which will be covered within chapter 4, the user can save and continue to the swipe page or the feedback page.

User feedback - Feedback View Allowing the users to give direct feedback through a dedicated view. As seen in Figure 3.4, the user is asked five questions. Ratings for the first four questions are from 1-4, where 1 is the lowest and 4 is the highest possible rating, while the last question is open for overall feedback by the user and the questions asked are:

- How happy are you with our app?
- How useful is our app?
- Are you going to use our app in the future?
- Would you suggest our app to others?
- Is there anything else you want to tell us?

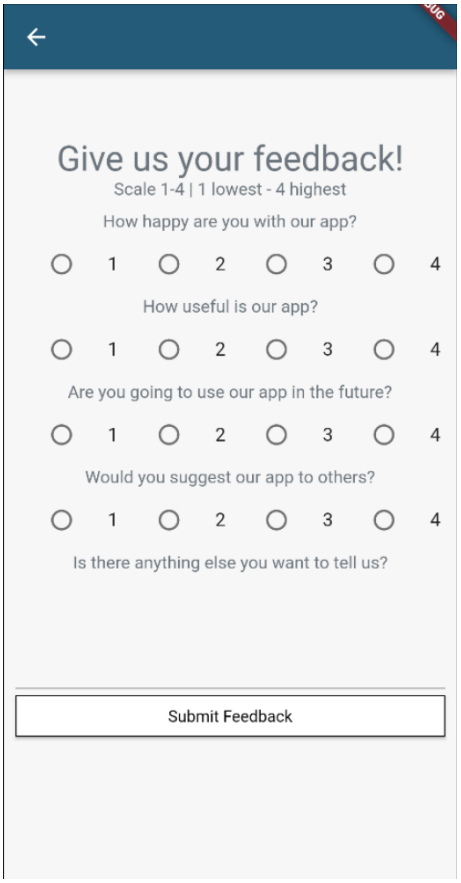

Figure 3.4: Feedback View and its questions

Swiping Projects - Swipe View This view is the key part of our mobile application with its Tinder-like swiping gesture, as well as the cards, that are supposed to look and feel like profiles on Tinder.

An example card can be seen in Figure 3.5, with an image at the top, if uploaded, showing optimally something related to the project itself, or in this case, a filler image of the institute in charge of the project or TU Graz itself.

After there is the title, name of the lecturer in charge, and a short summary of what the project is about. The Tinder-like swiping gesture includes being able to swipe the project's card right or left, where left means "No" or "Not interested" and right means "Yes" or "Interested". After a project gets swiped to the right, it will get added to the user's list of liked projects or topics.

The algorithm behind the order in which the projects will be discussed in more detail in chapter 4.

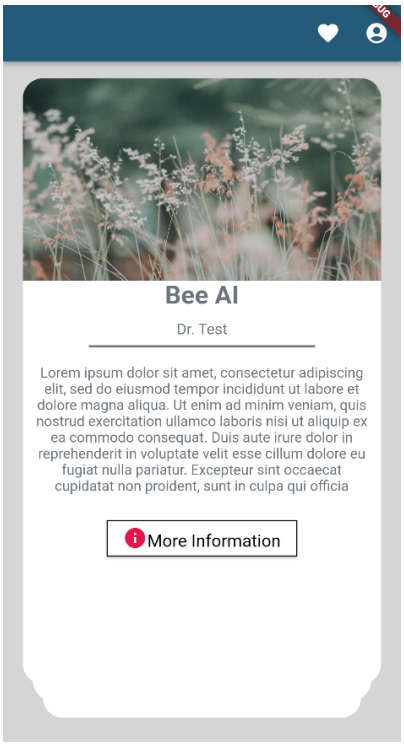

Figure 3.5: Swipe View with an example Card of a Project

Overview of liked Topics - Liked Topics View Another view of the mobile application is a simple overview of all projects or topics the user previously liked. Only the title and the person in charge are shown in a list, which is shown in Figure 3.6. By clicking on a project, the Project View shows a more detailed view and also allows the student to contact the lecturer.

This View can be accessed by clicking the Heart Icon on top of the Swipe View and the Project View.

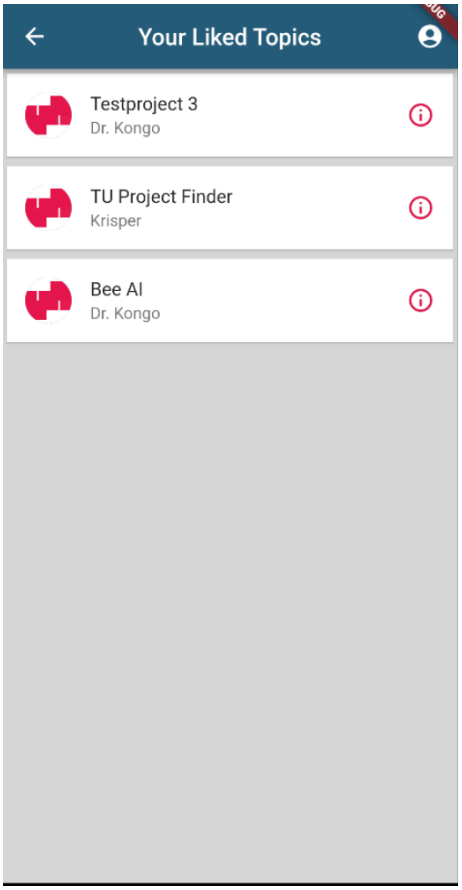

Figure 3.6: Liked Topics View with a few liked example projects

More detailed Projects - Project View This view is accessed through either the "More Information"-Button on a project's card within the swipe view or by clicking a project in the user's liked topics.

The project view contains all the information and the same image as the card within the swipe view. It is the intended way to contact the lecturer, who is in charge of a project. This can be done by clicking on the contact email at the bottom of the view.

This view gives a more detailed view on the project, by including a link to the website of the lecturer or institute, showing the project's types (for example, Master's Thesis or Bachelor's Thesis), and the allowed studies for the project.

Contact lecturer of project Clicking on the email address below the contact allows the student to contact the lecturer in charge of the project. This opens the user's default mail application and prefills it with a predefined short default text, the recipient, and the subject.

The predefined text and subject are meant to be an assistance to the user and can be adapted or completely changed if wanted.

Same design across all platforms Due to using Flutter as the framework, this requirement was fulfilled quite easily. Even though Flutter allows for differing designs on different platforms, it is not its intended way to implement Flutter applications. Therefore, our Flutter application automatically allowed a standard design for Android and iOS.

Low network traffic The mobile application only retrieves data from that database that might become relevant to the user. This was done by implementing the Lazy Loading pattern, which is explained in more detail in chapter 5. A project's title image is only retrieved right before it is needed, reducing network traffic even further.

Matchmaking within Swipe View The last requirement of the mobile application is the matchmaking process. This algorithm's target is to match the user, through the input they made in the User settings View, with the most fitting projects while completely filtering out projects with wrong majors and project types – which is the simple part of the matchmaking process.

This requirement will be thoroughly covered in chapter 4.

3.4 Web Application Software Architecture

This section will cover the requirements of the web application alongside its structure and views.

Framework Front End The Angular framework by Google was used to implement the web application's front end. Worth mentioning is that just like Flutter, it is released by Google.

Moreover, the Angular framework has a native implementation of Firebase, which makes using the already covered Firebase database within the web application quite straightforward, making the decision to pick Angular for the front end clearer. In early design Firebase was meant to be accessed through the front end, due to the simple use through Angular, but it was later swapped to be part of the back end to allow for a cleaner cut between both frontend and backend of the web platform.

Framework Back End As the backend framework express.js, a node.js web application framework was chosen. Mainly this choice was made, due to the flexibility and simplicity of express.js, as well as allowing for a decently simple Firebase database.

3.4.1 Requirements of the Web Application

Requirements for the Angular web application are comparatively simple when looking at those of the mobile application:

- Lecturer adds new Project
- Lecturer edits/deletes existing Project
- Mobile Application Card Preview
- Project Visibility
- Login/SSO
- Feedback

Lecturer adds new Project As already discussed, there are two main ways a new project can be added to the project's database, one of them being that a new project entry is created by a lecturer or someone in a comparable position.

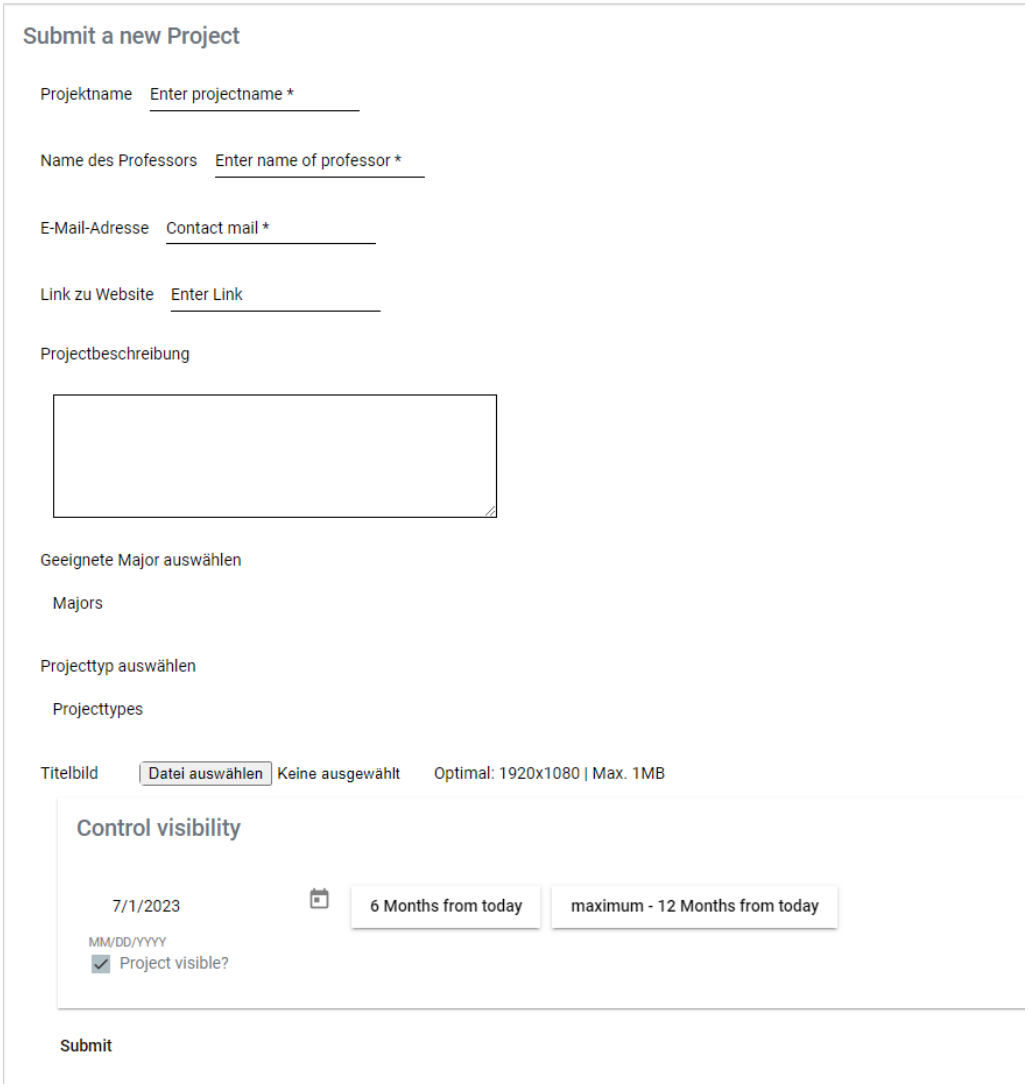

Figure 3.7: Form to add a new project (without the Preview, which can be seen in Figure 3.8)

The information the user can or should enter can be seen in Figure 3.7 and are already covered within section 3.2 and will be covered in more technical detail within chapter 4.

Moreover, on the right side of the view the user can see a preview of how the project may end up looking within the mobile app.

Lecturer edits/deletes existing Project Every lecturer may edit or delete an existing project, they created, or the crawler connected to their account. The view is just like the view to add a new project, but loads existing data for a project from the Firebase database. Unlike in the “Submit a new Project” view, at the bottom the user can choose between saving changes made and deleting instead of submitting a new project.

Mobile Application Card Preview The next requirement of the web application is a live preview of how the project currently being added or edited may look for the users within the mobile application. This card preview includes the uploaded image, as well as entered project name, the name of the person in charge of the project, and the project’s description. Once one of these fields gets updated, the preview is meant to instantly update to show the changes. An example preview of a new project can be seen in 3.8. When there is no input to a field, the name of the field will be displayed in the preview as placeholder and in the case that there is no image uploaded, a placeholder image will be displayed, which can be seen in 3.8.

Figure 3.8: Preview of the Project currently being added or edited

Project Visibility While adding or editing a project, the lecturer may change how long a project should be visible within the mobile application.

As displayed in Figure 3.9, the lecturer may hide the by unchecking the “Project visible?” check mark. Other than that, a lecturer may set a deadline for the project, after which the project will automatically turn invisible to users of the mobile application. This may be done, by directly choosing a date through a date picker, which may not be more than 12 months in the future, or by choosing 6 months or 12 months from the day of editing.

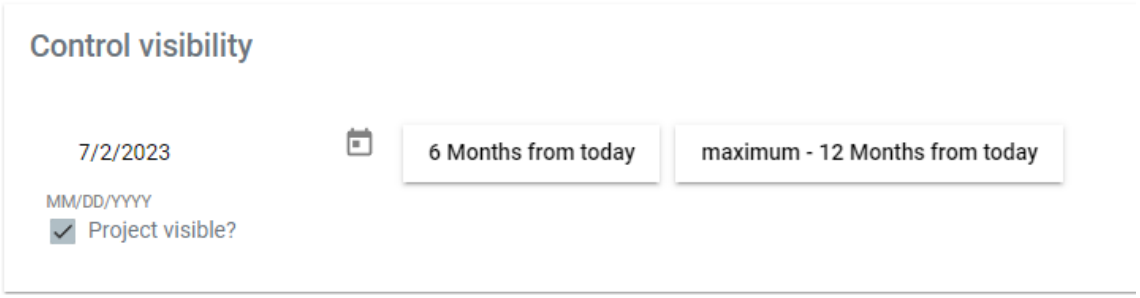

Figure 3.9: Visibility card at the bottom of the add/edit view

The main reason 12 months was chosen as the maximum time span a project can be visible, is that this way the risk of unsupported projects being shown within the web application is reduced. Moreover, this visibility control allows the lecturer to directly control their project’s visibility as well as lifespan.

In the case a person in charge of a project is still looking for a student to work on the project after the set time span ran out or is about to, he may just update it within the edit project view, again up to 12 months from the time of editing.

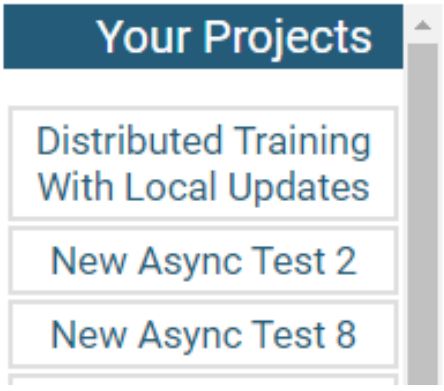

Figure 3.10: Display of the current user’s projects on the left side of the view

Login/SSO The current web application login system is implemented using a simple username and password combination, which is meant to be replaced by a full integration within the TU Graz Single-Sign-On system, to allow lecturers to get access to their project through the login system they are used to. Moreover, as the name suggests, they only need to sign in once per session and the SSO system recognizes them on all platforms within the SSO system.

After logging in, the system allows the user to submit a new project or, as seen in Figure 3.10, the user will get a list of all his projects on the left side of his screen. By clicking on one of those, he may edit or delete it.

Feedback To get more insight into the user's thoughts about this project's web application, the implementation of a feedback form made sense. This may be seen within 3.11 and includes the same questions covered in more detail within the feedback paragraph for the mobile application and feedback collection in the database.

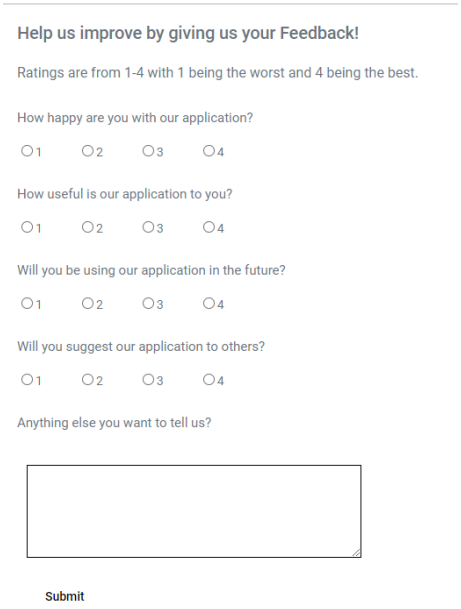

Figure 3.11: Feedback view in web application

3.5 Other Project Requirements

A handful of requirements for the project do not fit within either the web or mobile application's requirements and will therefore be listed here:

- Projects list up-to-date
- Real-time synchronization
- 99% Uptime

Projects list up to date For all currently supported institutes, the web crawler should allow for an always up-to-date projects list. Lecturers or editors will mostly add projects for institutes not supported by the web crawler as well as projects not included in the officially written-out projects by an institute.

Real-time synchronization Whenever a user requests data from the database, the data should be up-to-date. This should be the case for the web and mobile platforms. Realistically, this can not completely be achieved, since most lecturers and editors will not update data to projects, directly after the project is defined. It should be tried to achieve this requirement with the best possible effort.

99% Uptime A high percentage of uptime should be achieved by the database, the web application, and the mobile application.

Due to the database being outsourced, this is out of our hands, but historically this should not be a problem uptime-wise, since firebase pretty much guarantees an uptime of over 99.95%[1]. The web application's uptime will depend on the server it will be run on later. Lastly, the mobile application's uptime mostly depends on the user's network connections and the ability to receive data from the database. However, once data has been retrieved it is possible to use the mobile application even without a working network connection, due to caching.

[1] https://firebase.google.com/terms/service-level-agreement

Chapter 4

Implementation

This chapter will cover the implementation of the mobile application, web application, web crawler, and database. Important packages/libraries and each platform's structure will also be discussed.

4.1 Implementation of the Database Access

Within this section, the ways each platform accesses the database are explained.

Firebase in Flutter Within Flutter, the packages `firebase_core` and `cloud_firestore` are used to access the data from the projects and feedback collections. `firebase_core` allows to initialize the connection to the database with an API key, while `cloud_firestore` allows retrieving and updating data within Firestore. A simplified database access in Flutter is shown in Listing 4.1.

firebase_storage allows for access to the images linked to the projects. This is done by requesting images, which are named like the unique ID of a project.

Firebase in Web Application Due to Firebase allowing for direct access within Angular, it was first planned to implement the database connection through the frontend it was later decided to make a clear cut between the frontend and backend and place the database access in the backend.

```
Query db = FirebaseFirestore.instance.collection("projects").
    where("visible", isEqualTo: true);
await db.get().then(
        (value) {
      for(var project in value.docs){
        List? keyword = [];
        Project tmpproject = Project(
            project.get("title"),
            project.get("prof"),
            desc: project.get("desc"),
            project: project.get("project"),
            projectID: project.reference.id,
            major: project.get("major"),
            keywords: keyword
        );
        projectList.add(tmpproject);
      }
    }
);
```

Listing 4.1: Simplified Firebase access in Flutter, without retrieving every field from the database and catching exceptions.

Within the backend, the package firebase-admin allows for database access. Unlike in the Flutter application, this package allows for both, the access to the collections and the image storage. Again, it gets initialized with a private key before getting access to retrieving, adding, editing, and deleting projects, adding feedback, and adding and changing images.

Firebase in Web Crawler The web crawler uses the python library firebase_admin to access and edit data within the database. Just like in the other platforms of this project, a private key is used to establish a connection to the projects collection.

4.2 Implementation of the Mobile Application

This mobile application's Flutter implementation will be covered in more detail in this section. Important packages, the routing process, matchmaking, as well as the general structure, and other important design decisions will be covered.

Overall thought Overall Flutter's structure is layered, and most Widgets may have child Widgets. Therefore, our implementation is split into layers. Most layers are extracted into their own methods, while layers with multiple child Widgets are often split into multiple methods.

When it comes to views, each view is a file and, depending on whether it is stateless or stateful, contains one or two classes. Besides the views, which were covered in section 3.3, routing, notifications, images, database access, and a main class are used to implement this project.

Important Packages Within this project's Flutter implementation, a variety of packages were used, while only a handful of them have real importance to the project. These include:

- **Tcard**: For creating "Tinder"-like swipe cards
- **shared_preferences**: For storing the user's settings locally
- **firebase_core and cloud_firestore**: To access Firestore data
- **firebase_storage**: To access Firebase's image storage
- **string_similarity**: Calculate ranking of projects depending on keywords
- **flutter_local_notifications**: To create notifications

Why is the user stored locally? The decision to keep the user local and not save any data within the database was made due to data privacy reasons, since the user's data is not stored, besides, when feedback is sent.

Moreover, the application, therefore, is more user-friendly, due to the fact that no login is needed. Once the needed data for filtering is entered, no more data input is needed and everything is stored locally on the user's device through the shared_preferences package.

Routing Routing in Flutter is mostly implemented by pre-defining possible routes shown in Figure 4.1. Routes need to have unique names, except the initial route does not have a name but instead a single slash.

This Rout-class is initialized within the main class to start the application with the initial route. The Rout-class does not need to be included within the other classes using it to navigate between the different screens, due to it being called within the main-functions runApp-call.

```
class Rout{
  Rout();
  Widget setRout(){
    return MaterialApp(
      title: "TUPF",
      initialRoute: '/',
      routes: {
      '/': (context) => const StartScreen(),
      '/swipe': (context) => const SwipeScreen(),
      '/liked': (context) => const LikedScreen(),
      '/info' : (context) => const InfoScreen(),
      '/feedback' : (context) => const FeedbackScreen(),
      },
    ); // MaterialApp
  }
}
```

Figure 4.1: Pre-defining possible routes in Flutter

```
onTap: (){
    Navigator.pushNamed(context, '/info', arguments: {'index': convertIDtoIndex(likedList[index])});
}
```

Listing 4.2: Routing with arguments

Within the LikedScreen-class the `/info` route is called with arguments, due to the InfoScreen-class being dependent on which projects should be displayed, this can be seen in Listing 4.2. Most other calls within the navigator only supply context and the route.

Matchmaking Matchmaking between the user and the projects is done locally. After the user sets their information, mainly the project type, the major and optionally up to two keywords, the matchmaking process starts.

Firstly, the data, for all visible projects of the user's major will be retrieved from the database.

The list of projects will be ranked using the Sørensen–Dice coefficient, which is used by the `string_similarity` package in use. Each of this user's keywords Sørensen–Dice coefficient with each of the project's keywords will be calculated and then averaged out. This average lies between 0 and 1, where the project closest to 1 will be listed as first. Some projects may not contain keywords and therefore will be given a 0 within the string similarity rating.

4.3 Implementation of the Web Application

Important packages for the frontend and backend implementation, as well as the structure of both ends, will be explained in this part of the work.

4.3.1 Angular - Frontend

Here everything noteworthy about the Angular implementation will be covered.

Overall, every part of a view in Angular is split in four files. One html-file for the overall structure, one css-file for the design and structuring, one spec.ts-file for testing and a ts-file for methods behind the view. Moreover, each view can include other modules, for example, in our case a header, which is used in every view.

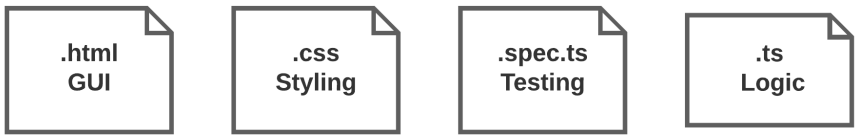

Figure 4.2: Files and their datatypes part of an Angular view

Launching the frontend The Angular frontend of this project is launched by running `ng serve --proxy-config proxy.conf.json` within the command line. Calling `--proxy-config proxy.conf.json` is used to redirect calls to the backend to the right location.

```
{
  "/api/*" : {
    "target" : "http://localhost:3000",
    "pathRewrite" : {
      "^/api" : ""
    }
  }
}
```

Listing 4.3: Proxy config frontend to backend location in `proxy.conf.json`

As seen in Listing 4.3, `target` sets the location of the backend server, which in our case currently is located at `localhost:3000`, as well as rewriting the path. This is done because within the frontend calls include `/api` before the actual command while they are expected to only have the command itself.

Web Application Routing For routing within Angular, possible routes need to be pre-defined, similarly to how it was done in Flutter. Figure 4.3 displays the definition of all routes within the project, as well as the components to which the router redirects the user to.

```
const routes: Routes = [
  { path: '', redirectTo: '/login', pathMatch: 'full' },
  { path: 'login', component: LoginComponent},
  { path: 'test', component: TestComponent},
  { path: '404', component: NotfoundComponent},
  { path: 'projects', component: CreateProjectComponent},
  { path: 'projects/edit/:id', component: EditProjectComponent},
  { path: 'feedback', component: RatingComponent},
  { path: '**', pathMatch: 'full', redirectTo: '/404' },
];
```

Figure 4.3: Setting possible routes within Angular

Worth noting is the last line, here we redirect every unknown path to the `/404` path due to them being unknown. Moreover, the third last row within Figure 4.3 shows variable paths, where `:id` is every possible ID a project may have within the web platform.

Frontend Packages Packages within the frontend are a rarity, due to the diverse native options within Angular. Routing, backend communication, and building forms, for example, are all included within the `@angular` package.

A variety of modules, which are parts used to build the frontend, such as buttons, date picker, or input fields, are also part of the `@angular` package library. All needed modules are imported within a local `app.module.ts` file, allowing all `.html` and `.ts` files to make use of them.

Communication with Backend To communicate with the backend, a service-file is created, which is a class called from other classes or views allowing the usage of within web development widely known GET, PUT and POST calls.

This way, data can be requested from the backend, as well as data, such as feedback and projects, can be sent to the backend. These calls include a path, which is used like a command, which is where the proxy-file from earlier comes into play, as well as variables that are part of the path, which mostly is data entered by the user.

4.3.2 node.js - Backend

This part covers important implementation thoughts about the backend, implemented with node.js.

Commands Possible calls the frontend may send to the backend to handle mainly include:

- **GET project** - Retrieving projects from database and pass to frontend
- **POST project** - User adds new project and adds to database
- **POST feedback** - User enters feedback to be stored to database
- **POST edit** - User edits existing project to be edited within database
- **PUT delete** - User deletes existing project in database
- **POST image** - User adds image to a new or existing project

Moreover, it is worth mentioning that all used methods, `GET`, `POST`, `PUT` are http methods.

4.4 Implementation of the Web Crawler

This section will cover the web crawler's implementation.

Structure of the Web Crawler The web crawler is focused on crawling new projects within the ITI institute of TU Graz.

It is a web crawler, starting at the institute's open projects page and crawling for all projects that are displayed there. Each project is scanned for the type, like Master, Bachelor, or other, as well as the major, which in this institute's case mostly is Computer Science, Software Engineering and Management, or Electrical Engineering. Figure 4.4 shows a list of the currently available theses topics, which are shown by opening the "Teaching" tab.

Before adding projects to the database, the projects are checked for existence within the database, and if they are not found, the crawler adds the newly found projects to the Firebase database.

Due to the fact that ITI does not add images to their written-out projects, the project's image will be the ITI logo by default.

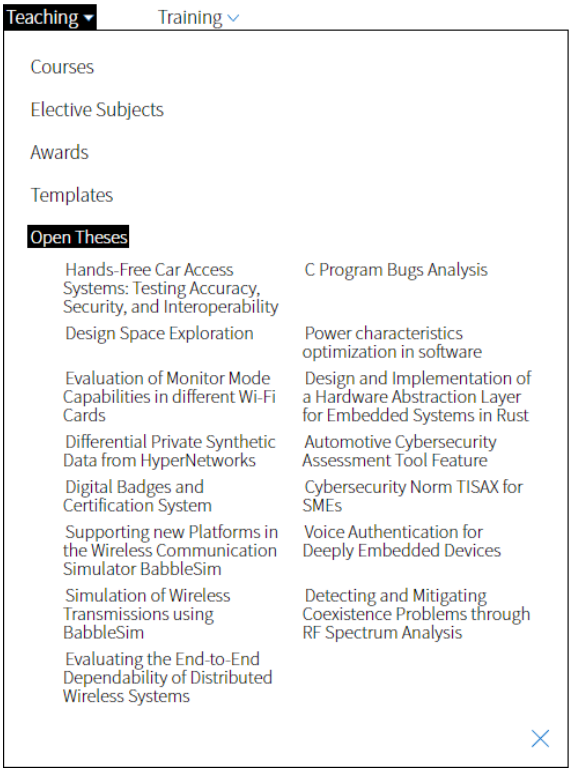

Figure 4.4: List of crawlable Projects on ITI website[2].

[2]https://www.tugraz.at/en/institutes/iti/home

Chapter 5

Design Patterns

This chapter covers a variety of Design Patterns used in this project. Firstly, patterns used in both, the mobile and the web applications will be covered, afterwards patterns exclusively used within the mobile application and lastly, patterns that were used within the web application will be discussed.

5.1 Design Patterns in Mobile and Web Application

This section will cover most Design Patterns that were implemented in both the mobile and the web application:

- Iterator
- Async Await
- Observer
- Singleton
- Reactor
- Eager Loading

Iterator The iterator pattern is among the most widely known and used patterns, and possibly is the most widely spread pattern. It is used to iterate over a list, array, or a similar data structure to access or manipulate the items within these data structures.

Async Await Async Await is used in both applications to wait for a response from the database. Once await is called, the program executes other code while waiting for the response from, in our case, the database. This pattern has a lot of different applications and is not only applicable to databases or other external responses.

Observer An Observer pattern was used within both, the mobile and the web applications, but the use cases are different. This pattern is used to observe one or more variables for changes.

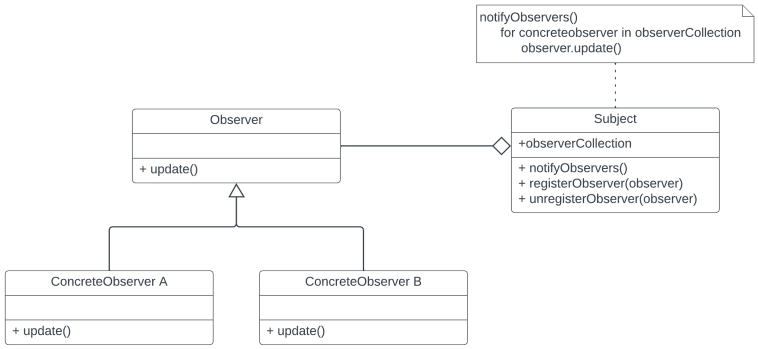

Figure 5.1: Observer Pattern UML

In the mobile application, the observer pattern was used to track the number of cards left for the user to swipe, once this number reached five or fewer remaining cards, the observer will call a method to get more projects from the database to get more cards.

Within the web application, this pattern was used to observe changes within the project that is currently being created or edited. Changes will lead to the observer calling a method to change the preview card if needed.

Singleton Just like the Observer pattern, the Singleton pattern was implemented in both platforms with different use cases. This pattern only allows a single object of a class, which is defined to be a singleton, each constructor call after the first will lead to the already existing object being returned to the caller.

```
static final NotificationService _notificationService = NotificationService._internal();
factory NotificationService() {
  return _notificationService;
}
```

Figure 5.2: Flutter implementation of the Singleton Design Pattern

Within the mobile platform, there are two main use cases. The database and the notification system, of which the Flutter implementation, can be seen in Figure 5.2. The web application uses the singleton pattern, for example within the app-service, which is the communication point of the frontend to the backend.

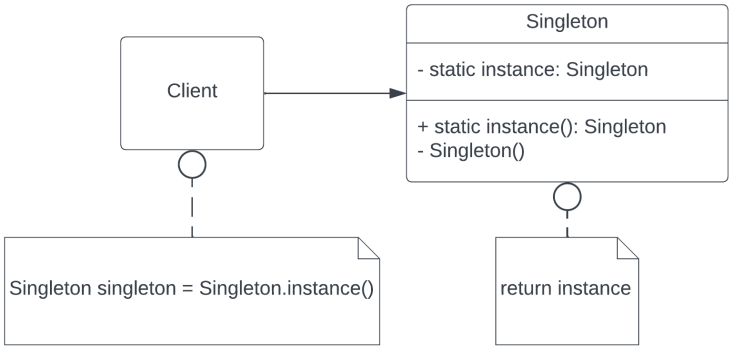

Figure 5.3: Singleton Pattern UML

Reactor The Reactor pattern allows for service requests to be delivered concurrently by multiple clients. This pattern is not directly implemented within either of the platforms but used to to the Reactor pattern being part of the Firebase database service.

Eager Acquisition Whenever everything that might be needed in the future is directly downloaded from the database, one would call that Eager Loading.

In the web applications case, this is everything related to the logged-in user. Whenever a user is logged in, the data for all his projects is retrieved from the database, and within the mobile application Eager Loading is used to download the data of all projects within the user's selected major and project type, all data besides the images that is.

Figure 5.4: Eager Acquisition UML

5.2 Design Patterns exclusive to Mobile Application

Within the Mobile Application, a variety of Design Patterns were used. In contrast, others were implemented completely intentionally – a full list of all patterns that may be found within the mobile application includes:

- Strategy
- Future
- Lazy Acquisition/Loading
- Caching
- Null-Safety
- Extract Method
- Half-Sync/Half-Async

Strategy The Strategy pattern is used to choose one algorithm to run out of a group of interchangeable algorithms.

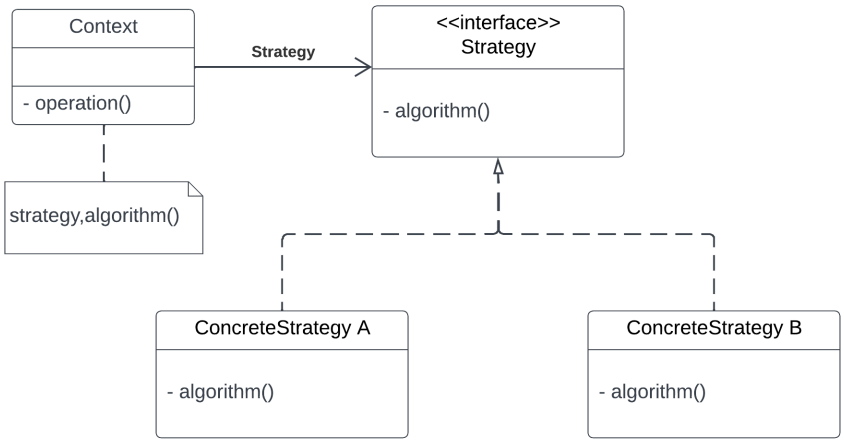

Figure 5.5: Strategy Pattern UML

In this project, it was used to differentiate between iOS and Android – firstly, because during compilation there are differences, and secondly, due to the notifications working in different ways.

Future A Future is a data object that is returned to the client from a service before the computation is finished. After completion, the client receives the final object. Within Flutter, the client receives a Future object after requesting data from the Firebase database. In our case, we know exactly how the final object will be structured, and therefore may already call the functions with the Future as input.

Lazy Acquisition Lazy Loading is used to only download data from the database when it is really needed.

The implementation of this pattern within this project is used to only download the images of a project when its card is about to be shown. This is done to minimize the application's network usage.

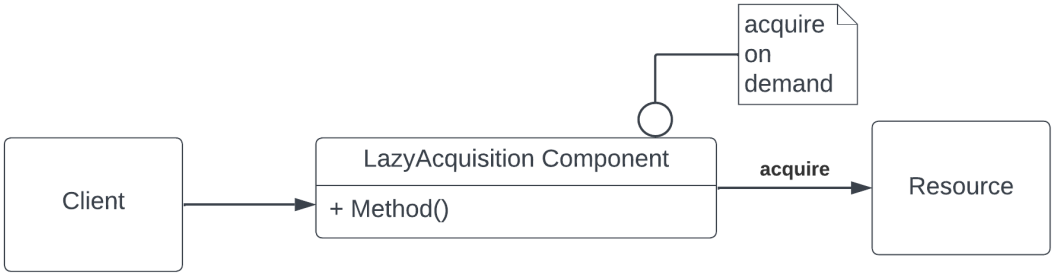

Figure 5.6: Lazy Acquisition Pattern UML

Caching After acquiring data, the data is stored locally and therefore does not need to be acquired from a database but instead is loaded from the local storage.

Caching is used to minimize network usage in this project. Firebase supports caching natively. Everything requested from the Firebase databases, including the projects themselves, as well as their images, will automatically be cached in Firebase's cache.

Null-Safety Null-Safety is a pattern applied by Flutter and Dart. It ensures that every variable does not contain `null` as a value unless it is initialized to be `null`. Types like int can not have `null` as a value overall.

Null-Safety can also be classified within the Null-Object pattern as it is a way to handle objects with no value.

Extract Method This is a refactoring design pattern that was applied. By extracting and splitting code into new methods with self-explaining method names, we allow for smaller methods and simpler-to-understand code.

This pattern was applied to most views because in the beginning, it was easier to build the basic GUI within in a handful of methods, but after they were split into more, once the design got more detailed.

Half-Sync/Half-Async This pattern allows for two layers, one being synchronous, while the other layer is asynchronous. These layers may communicate directly, like in the case of this project, or with another layer in between for queuing.

Most of the project is reliant on the asynchronous layer, because of its dependency on the data from the Firebase database. A small part, of the local user data, is synchronous and both layers communicate directly, due to the direct access of the asynchronous layer to data of the synchronous layer.

5.3 Design Patterns exclusive to Web Application

Design patterns which are exclusively found within the web application are covered in this section.

Encapsulated Field Whenever this pattern is implemented, the variable can only be accessed and changed through getter and setter methods. Within the backend of the web application, all changes to the data may only be done through getter and setter methods.

Command Command pattern implementation encapsulates requests as an object and then forwards the object to a receiver to execute them accordingly.

```
postFeedback(feedback : any){
    this.http.post("/api/feedback", feedback, httpOptions).subscribe(response => {
```

Listing 5.1: Implementation of the Command Design Pattern in the Web Application

As seen in Listing 5.1, the Command pattern was used for the communication between the frontend and the backend. The part of the url after '`/api/` is the command, which in return is handled by the backend. In the case of this example, the feedback entered by the user will be stored in the database.

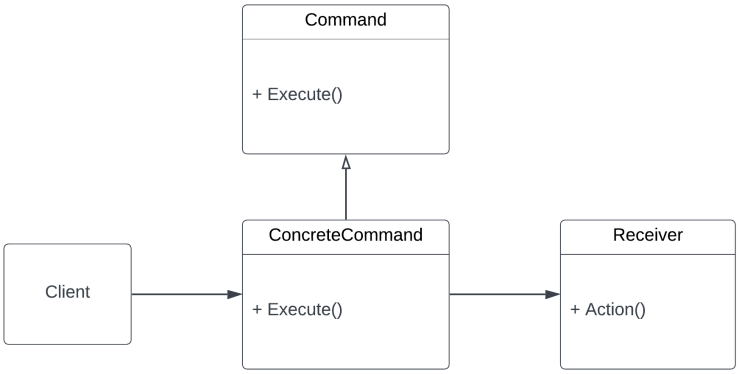

Figure 5.7: Command Pattern UML

Client-Server Like the name suggests, this pattern includes a client and a server, where the client requests a service and the server fulfills it. The Client in our project is the frontend and the backend, including the database, is the server in our case.

Chapter 6

Evaluation

This chapter will cover an evaluation of the project's goals and achievements. Firstly, the requirements defined within the Software Architecture (chapter 3) will be revisited and analyzed. Afterwards, other capabilities of the system will be discussed. Lastly, a user study was done with students of TU Graz to get feedback on the mobile application.

6.1 Assessment of Requirements

Within this section, most requirements listed within chapter 3 will be discussed, and whether they are completely fulfilled. Moreover, capabilities will be discussed.

6.1.1 Assessment of Requirements

This section will cover the requirements of the project and assess them. Most requirements are qualitative requirements, therefore, each will be rated on a fulfillment scale of 1-5, depending on how the implementation worked out.

6.1.2 Mobile Application requirements

Within this part of this work, the requirements of the mobile application will be covered. The following requirements will be looked at:

- Local User
- Swiping Projects
- Contact lecturer of project
- Same design across all platforms
- Low network traffic
- Matchmaking within Swipe View

Local User The architectural decision to keep the user only in the local storage was implemented successfully and can therefore be rated a 5 on a scale of 1-5. This allowed for more anonymity for the user, as well as simplified the matchmaking process, due to it being moved to the local device. Moreover, there is not much data being stored, therefore, the usage of the user's storage is limited to a few strings.

Swiping Projects The Tinder-like swiping process was implemented according to the requirements. By making use of the Flutter package `tcard`, which not only allowed for the animation of swiping cards either left or right but also for handling the user's input in the background. Also, the card design allowed by `tcard` is quite simple, due to each card being handled like a normal view in Flutter. This allows for a lot of freedom in the design process, as well as a known format for the developer. This completely fulfilled the requirement but some small additions, such as adding buttons for liking and disliking, might improve the implementation of this requirement, therefore this requirement is a 4/5.

Contact lecturer of project Contacting the lecturer with a predefined email text and therefore fulfilling this requirement was done using the `url_launcher_string` package for Flutter. This requirement can be rated a 5/5, since it met all expectations.

Same design across all platforms Successfully implementing this requirement was quite simple, while achieving a 5 on the fulfillment scale of 1-5. By implementing the project in Flutter, this requirement was fulfilled passively, due to, as already explained in section 2.6, Flutter allowing the code to be compiled to be executed on iOS and Android, along other platforms.

Low network traffic Lowering the network traffic was done by only retrieving text data of visible projects before ranking them and filtering the project of interest to the current user. Images, which are the main network traffic of this project, are only retrieved once they are needed. Moreover, caching allows for even lower traffic, due to images being cached, as well as text data for liked topics. There are most likely always possibilities to lower the network traffic of our platform, therefore this requirement only achieved a 4 on our fulfillment scale.

Matchmaking within Swipe View Overall the matchmaking is done locally, besides the filtering of non-visible projects. The ranking is done using the Sørensen–Dice coefficient, which is used by the Flutter package `string_similarity`, depending on the keywords entered by the user and stored within each project.

The matchmaking process was chosen to be implemented in a simplistic way, due to only a few dozen projects being in question at any time. With the project growing and the number of projects shown to each student might grow and a different matchmaking process might be better, therefore, due to the current algorithm working well for the current scope of the application, a rating of 4 on our 1-5 fulfillment scale was selected.

6.1.3 Web Application requirements

Next, most requirements for the web application will be covered, which include:

- Add new project
- Edit/Delete existing project
- Preview card
- Project visibility
- Login/SSO
- Feedback

Add new project This requirement was fulfilled with Angular's first-party libraries for forms and form validation. The form for the creation of a new project includes data displayed to the user, such as the project's name or the lecturer in charge, data needed for the matchmaking process, such as the keywords and lastly, visibility control. Moreover, a preview card is included in this view, which will also be covered in this section of the paper. On the fulfillment scale, this requirement achieved a 5.

Edit/Delete existing project Quite similar to the view for the creation of a new project, this requirement was implemented only using Angular's first-party libraries and in a very similar design to the one seen in the view for adding a new project, therefore, like the "Add new project" requirement, a score of 5 on the fulfillment scale is achieved. The main difference to the "Add new project" requirement is that instead of a button to submit the new project at the bottom of the view, two buttons can be found in this view. One to save the changes made to the project loaded and another one to delete the whole project from the database.

Preview card Using Angular's first-party method `mat-card`, a preview card was created, with the goal of giving lecturers a viewing of what the project will look like to students seeing the project within their Flutter-built Android or iOS application. The information displayed only includes data entered by the lecturer or loaded from the database. This preview card can be seen in both the add new project view, and the edit/delete project view. Overall, this preview card is quite similar to the one displayed in the mobile application, but due to small, visible differences, especially the "More Information" button, a score of 4/5 was achieved.

Project visibility Completing this requirement was done within the functionality of Angular's form-building and form validation features. With a check, the editor may choose to turn the project visible or invisible to the students, but unlike deleting, this allows the editor to still edit the project and turn it visible later on. Moreover, the editor needs to set a date, after which the project automatically turns invisible. The maximum duration for visibility is 12 months, which is done to avoid "dead" projects within the mobile application. Just like projects manually being set to invisible, each project may be set to visible again after being set to invisible automatically. Due to the implementation of this requirement fulfilling everything expected, a score of 5 on the fulfillment scale was chosen.

Login/SSO The requirement for the login system is fulfilled, but they project is not yet integrated into the TU Graz SSO, therefore only achieving a 3/5 on the fulfillment scale. The current login system is very simple. Moreover, currently, the project editing is bound to the email addresses of the lecturer or designated editors. Each user can only open the edit/delete view for each project they are assigned.

Feedback Like every other form within the web application, this application's feedback form was built with only Angular's first-party form and form validation libraries and scored a 5 on the fulfillment scale.

6.1.4 Other requirements

Lastly, other requirements for this project included:

- 99% Uptime
- Real-Time Synchronization
- Project list up to date

99% Uptime For the mobile application, a 99% uptime is guaranteed by Firestore guaranteeing an uptime of at least 99.95%[1], and everything else depends on the user's network connection. Moreover, even if the user does not have a working network connection, he may use the mobile application due to caching, which may lead to some projects not having title images, but other than that the application works as normal.

Within the web application, uptime can not be guaranteed at the moment. This is the case, due to the web application not yet permanently being hosted. In the case the application will get hosted by TU Graz, a 99% uptime should be realistic.

Real-Time Synchronization Projects being up-to-date is achieved due to Firestore's real-time updates. Therefore, every change by an editor or lecturer within the web application will automatically update projects within the mobile application in real-time. This is also the case for projects added through the web crawler. This, of course, can only happen if the mobile user has a working network connection. This requirement achieved a 5 on the fulfillment scale.

[1] https://firebase.google.com/terms/service-level-agreement

Project list up to date Projects added by a lecturer or editor are within their area of responsibility to keep up to date and therefore cannot be guaranteed to be up-to-date. Projects posted on institutes' official projects list will be automatically added to the projects' database. This, however, is only the case if the institute is supported by the web crawler, which at the moment is only the case for the ITI institute of TU Graz. The fulfillment of this requirement is hard to rate, due to future work with institutes, therefore this requirement can only be rated with a 3/5 score.

6.1.5 Summary of requirements

Table 6.1 summarizes every requirement of the project and its rating on the fulfillment scale, from 1 to 5, except one, which can not be rated at the moment.

Requirements		
Requirement	Platform	Requirement met?
Local User	Mobile	5/5
Swiping Projects	Mobile	4/5
Contact lecturer of project	Mobile	5/5
Same design across all platforms	Mobile	5/5
Low network traffic	Mobile	4/5
Matchmaking within Swipe View	Mobile	4/5
Add new project	Web	5/5
Edit/Delete existing project	Web	5/5
Preview card	Web	4/5
Project visibility	Web	5/5
Login/SSO	Web	3/5
Feedback	Web	5/5
99% Uptime	Other	Possible
Real-Time Synchronization	Other	5/5
Project list up to date	Other	3/5

Table 6.1: Summary of all requirements and their fulfillment score on a 1-5 scale

6.2 User Study

In this section, the user study, which was carried out with TU Graz students, will be covered. Firstly, the questions asked why they were asked, and the expectations will be covered. Subsequently, the results of the user study will be shown and analyzed.

6.2.1 Experimental Design

The user study was carried out with 13 students, each student individually, and only focused on the mobile applications side of the project, due to the web application being irrelevant to the students.

Firstly, a presentation about the features and design of the mobile application was held, followed by each student filling out a form on their opinion on the project through questions, which are covered next. These questions were answered without any guidance and anonymously by each student.

Questions asked After the presentation, each student filled out a survey, including questions about their thoughts on the mobile application.

The questionnaire contained five statements, which each student could rate from 1-6, where 1 meant they do not agree with the statement at all, and 6 meant they fully agree. This scale also was selected due to it not allowing a neutral value or opinion on a statement. Lastly, there is an open feedback field, which allows each attendant to give their opinion on the project and ideas they may want to see implemented.

Questions asked in the questionnaire are meant to collect data on the opinions about the mobile application, especially from the view of a student, and contain the following statements rateable from 1-6:

- This application overall seems useful to me.
- I would use this application when looking for a thesis.
- This application would simplify the process of finding a topic for a thesis.
- This application would work among students at TU Graz.
- I would recommend this application to fellow students.

As well as the open question “Are there any comments or suggestions you would like to add about the TU Project Finder that would help improve it?”.

Expectations Positive feedback was expected, due to the problems students usually face when looking for a thesis. Most institutes of TU Graz only offer outdated, unstructured, or no lists at all. Therefore, it is to be believed, that most students would like a platform offering a central point to look for possible thesis topics, which at the moment, does not exist at TU Graz. Therefore, overall, very positive feedback on the statements within the form can be expected.

Moreover, the Tinder-like swiping process makes it simple for students to explore possible projects, even to some degree "gamifying" the process, allowing for a process, which at the moment is a real struggle for most students, to possibly even be an enjoyable experience.

Lastly, the possibility for students to enter any feedback they would like, there might be some ideas to be implemented in the project in the future.

6.2.2 Results

After each of the 13 students rated the statements listed above on a scale of 1-6, the results were really good. Each statement's average rating was above at least 5.3, with the lowest-rated statements being "This application would work among students at TU Graz." and "I would recommend this application to fellow students.", meaning on average, each student on average would agree with this statement. 5.3 being the worst rated statements are really promising results for this user study and the project overall.

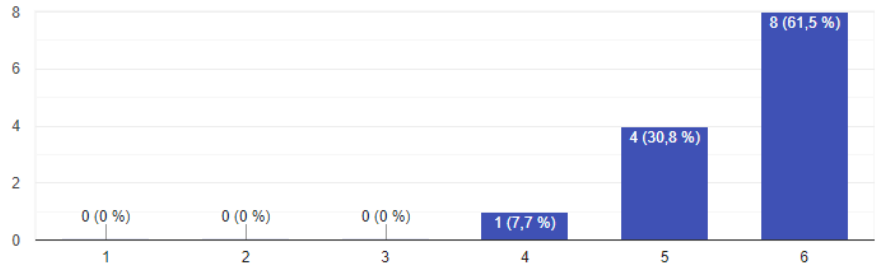

Figure 6.1: User study results for the statement "This application would simplify the process of finding a topic for a thesis."

The two statements with the highest agreement are "This application would simplify the process of finding a topic for a thesis." and "I would use this application when looking for a thesis.", which are really well-rated, with an average above 5.5 each. In Figure 6.1, for the first of those two top-rated statements, "This

application would simplify the process of finding a topic for a thesis." the exact ratings can be seen. Ratings for "I would use this application when looking for a thesis." look the same.

When it comes to the open feedback field at the end of the user study, some interesting comments were made, namely:

- Student website, TUG online or Studo
- Like/Dislike statistics for lecturers
- Suggest Keywords
- Current finding process

Student website, TUG online or Studo This was the most suggested improvement in the feedback field. Three students suggested implementing TU Project Finder as part of the Studo application, which is an application most students at TU Graz use to organize their lives at university, including emails, calendars, and registering for lectures, among other things.

Other related suggestions were also offering the student's application as a website, so they would not need to install another application on their phone, as well as suggesting integration into TUG online, which is the online platform for students and lecturers to organize their lives at university, so to speak the browser version of Studo.

Due to the broad usage of Studo, an integration into this application could definitely make sense and should be discussed in the future.

Like/Dislike statistics for lecturers Offering lecturers a monthly update on the Like/Dislike statistics of their projects could be an interesting feature, to also offer the lecturers' feedback on how their topics perform among students. Although this feature might be unpopular among some lecturers, overall this could help increase the quality of topics written out at TU Graz and further improve the process of finding topics for students.

Suggest Keywords Moreover, feedback included that the application should suggest keywords for students, which could definitely work and should be looked into.

Current finding process Overall, each feedback discussing the current finding process showed that there is a place for this application, due to the current process not being structured well, and students being clueless about the best way of finding topics. Problems in the current process found within the user study include:

- Own ideas
- Emails
- Other ways

Own ideas Some students said that it is common in their major that students approach lecturers with their own ideas and possible topics, which can be hard, due to students often not knowing the scope of a bachelor's or master's thesis. This was listed by students majoring in architecture and teaching.

Emails Another feedback student gave, majoring in teaching and one that did not name their major, was that they found possible topics mainly through emails by lecturers, whose lectures they attended in the past. Even if this works in some cases, it is possibly not the way most students want and should rely on, due to most lecturers not regularly sending out their topics.

Other ways Other students found their topics in the past through fellow students recommending them to a lecturer or even through an outdated bulletin board at an institute.

Chapter 7

Discussion

This chapter will cover possible limitations in the future, as well as planned future work to further improve this project.

7.1 Limitations

Possible limitations of this project may include:

- Generalization of students
- Adaptation of institutes
- Growing user base

Generalization of students Overall, a generalization of the results to all students of TU Graz is limited by a small number of students, which were part of the user study. Moreover, even with those students covering a few different majors, a lot of other majors may work differently in the way their process of looking for a thesis is.

Adaptation of institutes Another possible limitation of the project is the willingness of institutes of TU Graz to publish their projects in a well-structured way on their website, where they could be collected with a web crawler, or directly add projects to our database through the web application.

Growing user base Starting to grow a user base at TU Graz might be a struggle at first, but this could be countered by possibly integrating the mobile application as part of the at TU Graz widely used Studo application, which was a suggestion part of the feedback from some students during the user study.

7.2 Future Work

There are a few possible ways this project may be expanded or improved on in the future, including:

- TU Graz Single-Sign-On
- Web Crawler
- Studo

TU Graz Single-Sign-On Integrating the web application into the TU Graz SSO system would largely benefit the project, due to easier access for lecturers and editors to their projects within the system. Currently, all sites part of the TU Graz network allow login with the same login credentials and once logged in, you stay logged in on the device for a while and across the whole network. Moreover, this would not only make it easier for lecturers, but most likely also increase their willingness to make use of the web platform.

Web Crawler Currently, the web crawler is only fully implemented for the ITI institute of TU Graz. Even though, expanding the crawler to all institutes of TU Graz is impossible, due to a lot of institutes not offering an up-to-date list, or no list at all. However, institutes maintaining their lists may be integrated into the web crawler pretty quickly and therefore minimizing effort for lecturers and editors of an institute, which in turn could possibly motivate other institutes to do the same.

Studo Some students suggested this during the user study, which makes sense since most students already use the Studo application to handle their calendar, mail, exams along other things at university. Students possibly want to avoid having more applications than are needed on their phones. Moreover, integration into the Studo application could, even though it would probably be a lot of effort, simplify the process of growing a user base and making our application known to students at TU Graz. This possible cooperation is yet to be discussed with them.

Chapter 8

Conclusion

This project worked on solving the common problem of most students at TU Graz, which is trying to find a suitable topic for their bachelor's or master's thesis. On the other hand, it should be as easy as possible for lecturers to publish topics within this platform.

A Flutter application for iOS and Android was chosen as the optimal solution to help the students find possible topics in an easy and gamified way. To solve the lecturers' side of the problem, an Angular web platform was built to allow the lecturers themselves or their designated editors to add, edit, and remove from our database. Moreover, to collect more topics for the students to swipe through, a web crawler was implemented. At the moment only works for the ITI institute of TU Graz as a proof of concept, but should be expanded to other institutes offering an online list of topics in the future.

Overall, after carrying out a user study with students from various majors, overwhelmingly positive feedback was gathered. Every student agreed with the statements that "This application would work among students at TU Graz." as well as "This application would simplify the process of finding a topic for a thesis.", which shows the possibly bright future of this project.

In the future, working on an efficient solution to keeping projects within the database up to date, mostly through web crawlers for more institutes as well as helping institutes add their new projects to our database on their own, should be looked at. Moreover, implementing TU Graz Single-Sign-On for the web application is needed, to allow lecturers and editors to add new projects with as little work as possible. Lastly, possibly implementing TU Project Finder within the studo app would allow for a wider availability for students, due to the broad usage of the application at TU Graz.

Bibliography

[1] L. Bass, P. Clements, and R. Kazman, *Software Architecture in Practice*, 3rd ed. Addison-Wesley Professional, 2012. → Cited on page 4.

[2] D. L. Parnas, "On the criteria to be used in decomposing systems into modules," *Commun. ACM*, vol. 15, no. 12, p. 1053–1058, 12 1972. [Online]. Available: https://doi.org/10.1145/361598.361623 → Cited on page 4.

[3] P. Zhang, H. Muccini, and B. Li, "A classification and comparison of model checking software architecture techniques," *Journal of Systems and Software*, vol. 83, pp. 723–744, 05 2010. → Cited on page 4.

[4] L. Dobrica and E. Niemela, "A survey on software architecture analysis methods," *IEEE Transactions on Software Engineering*, vol. 28, no. 7, pp. 638–653, 2002. → Cited on page 4.

[5] F. Solms, "What is software architecture," in *ACM International Conference Proceeding Series*, 10 2012. → Cited on page 5.

[6] D. Garlan, *Software Architecture*. John Wiley & Sons, Ltd, 2007. [Online]. Available: https://onlinelibrary.wiley.com/doi/abs/10.1002/9780470050118.ecse375 → Cited on pages 5 and 6.

[7] A. Tang, P. Liang, and H. Vliet, "Software architecture documentation: The road ahead," in *Proceedings - 9th Working IEEE/IFIP Conference on Software Architecture, WICSA 2011*, 06 2011. → Cited on page 6.

[8] M. Silva and I. Perera, "Preventing software architecture erosion through static architecture conformance checking," in *2015 IEEE 10th International Conference on Industrial and Information Systems (ICIIS)*, 12 2015, pp. 43–48. → Cited on page 7.

[9] M. Dalgarno, ""when good architecture goes bad," methods and tools," *Method and Tools*, vol. 17, pp. 27–34, 2009. [Online]. Available:

https://www.methodsandtools.com/archive/archive.php?id=85 → Cited on page 7.

[10] H. Vliet and A. Tang, "Decision making in software architecture," *Journal of Systems and Software*, vol. 117, 01 2016. → Cited on pages 7 and 8.

[11] W. Stacy and J. MacMillan, "Cognitive bias in software engineering," *Commun. ACM*, vol. 38, no. 6, p. 57–63, 6 1995. [Online]. Available: https://doi.org/10.1145/203241.203256 → Cited on page 7.

[12] C. R. Martin, "Design principles and design patterns," *objectmentor*, 2000. → Cited on pages 8 and 12.

[13] P. Petrov and U. Buy, "A systemic methodology for software architecture analysis and design," in *2011 Eighth International Conference on Information Technology: New Generations*, 2011, pp. 196–200. → Cited on page 8.

[14] C. Alexander, S. Ishikawa, and M. Silverstein, *A Pattern Language: Towns, Buildings, Construction*. New York: Oxford University Press, 8 1977. [Online]. Available: http://www.amazon.fr/exec/obidos/ASIN/0195019199/citeulike04-21 → Cited on pages 9 and 10.

[15] J. O. Copien, *Software Design Patterns: Common Questions and Answers*. The Press Syndicate of the University of Cambridge, 1998. → Cited on page 9.

[16] E. Gamma, R. Helm, R. Johnson, and J. M. Vlissides, *Design Patterns: Elements of Reusable Object-Oriented Software*, 1st ed. Addison-Wesley Professional, 1994. [Online]. Available: http://www.amazon.com/Design-Patterns-Elements-Reusable-Object-Oriented/dp/0201633612/ref=ntt_at_ep_dpi_1 → Cited on pages 9, 10, and 11.

[17] W. Tichy, "A catalogue of general-purpose software design patterns," in *Proceedings of TOOLS USA 97. International Conference on Technology of Object Oriented Systems and Languages*, 01 1997, pp. 330–339. → Cited on page 9.

[18] W. Cunningham, "Pattern forms," 2005. [Online]. Available: https://wiki.c2.com/?PatternForms → Cited on pages 9 and 11.

[19] M. Krisper, “Finding the right design pattern using binding time properties,” 2016. → Cited on page 9.

[20] D. Riehle, “Composite design patterns,” in *Proceedings of the 1997 Conference on Object-Oriented Programming Systems, Languages, and Applications (OOPSLA’97). ACM Press*, vol. 32, 10 1997, pp. 218–228. → Cited on page 12.

[21] W. Zimmer, *Relationships between Design Patterns*. USA: ACM Press/Addison-Wesley Publishing Co., 1995, p. 345–364. → Cited on page 13.

[22] M. de Vos, G. Ishmaev, and J. Pouwelse, “Match: A decentralized middleware for fair matchmaking in peer-to-peer markets,” in *Proceedings of the 21st International Middleware Conference*, ser. Middleware ’20. New York, NY, USA: Association for Computing Machinery, 2020, p. 74–88. [Online]. Available: https://doi.org/10.1145/3423211.3425678 → Cited on pages 14, 15, 16, and 17.

[23] D. Paraschakis and B. Nilsson, “Matchmaking under fairness constraints: a speed dating case study,” in *Communications in Computer and Information Science (CCIS,volume 1245)*, 04 2020. → Cited on pages 14, 15, 16, and 18.

[24] T. P. Minka, R. Cleven, and Y. Zaykov, “Trueskill 2: An improved bayesian skill rating system,” in *Technical Report MSR-TR-2018-8. Microsoft*, 2018. → Cited on pages 15 and 17.

[25] M. Claypool, J. Decelle, G. Hall, and L. O’Donnell, “Surrender at 20? matchmaking in league of legends,” in *2015 IEEE Games Entertainment Media Conference (GEM)*, 10 2015, pp. 1–4. → Cited on pages 15, 16, and 17.

[26] M. Myślak and D. Deja, “Developing game-structure sensitive matchmaking system for massive-multiplayer online games,” in *Lecture Notes in Computer Science (LNISA,volume 8852)*, 11 2014. → Cited on page 16.

[27] S. Agarwal and J. Lorch, “Matchmaking for online games and other latency-sensitive p2p systems,” in *ACM SIGCOMM Computer Communication Review*, vol. 39, 08 2009, pp. 315–326. → Cited on page 18.

[28] A. Sarkar, M. Williams, S. Deterding, and S. Cooper, “Engagement effects of player rating system-based matchmaking for level ordering in human compu-

tation games," in *FDG '17: Proceedings of the 12th International Conference on the Foundations of Digital Games*, 08 2017. → Cited on page 18.

[29] J. Ward, "What are you doing on tinder? impression management on a matchmaking mobile app," *Information, Communication & Society*, vol. 20, pp. 1–16, 11 2016. → Cited on page 18.

[30] P. Resnick and H. R. Varian, "Recommender systems," *Commun. ACM*, vol. 40, no. 3, p. 56–58, 3 1997. [Online]. Available: https://doi.org/10.1145/245108.245121 → Cited on page 19.

[31] O. Artemenko, V. Pasichnyk, and N. Kunanec, "E-tourism mobile location-based hybrid recommender system with context evaluation," in *2019 IEEE 14th International Conference on Computer Sciences and Information Technologies (CSIT)*, vol. 2, 2019, pp. 114–118. → Cited on page 19.

[32] N. Mishra, S. Chaturvedi, A. Vij, and S. Tripathi, "Research problems in recommender systems," *Journal of Physics: Conference Series*, vol. 1717, p. 012002, 01 2021. → Cited on pages 20, 21, 22, and 23.

[33] F. Ricci, L. Rokach, and B. Shapira, *Recommender Systems: Introduction and Challenges*. Springer Science+Business Media New York 2015, 01 2015, pp. 1–34. → Cited on pages 20 and 21.

[34] A. Pujahari and V. Padmanabhan, "Group recommender systems: Combining user-user and item-item collaborative filtering techniques," in *2015 International Conference on Information Technology (ICIT)*, 2015, pp. 148–152. → Cited on pages 20 and 21.

[35] M. A. Ghazanfar and A. Prugel-Bennett, "A scalable, accurate hybrid recommender system," in *2010 Third International Conference on Knowledge Discovery and Data Mining*, 2010, pp. 94–98. → Cited on page 21.

[36] L. Chen, Y. Xu, F. Xie, M. Huang, and Z. Zheng, "Data poisoning attacks on neighborhood-based recommender systems," *Transactions on Emerging Telecommunications Technologies*, vol. 32, 06 2021. → Cited on pages 22 and 23.

[37] J. Canny, "Collaborative filtering with privacy via factor analysis," in *Proceedings of the 25th Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '02. New

York, NY, USA: Association for Computing Machinery, 2002, p. 238–245. [Online]. Available: https://doi.org/10.1145/564376.564419 → Cited on pages 22 and 24.

[38] J. Zhan, C.-L. Hsieh, I.-C. Wang, T.-S. Hsu, C.-j. Liau, and D.-W. Wang, "Privacy-preserving collaborative recommender systems," *Systems, Man, and Cybernetics, Part C: Applications and Reviews, IEEE Transactions on*, vol. 40, pp. 472 – 476, 08 2010. → Cited on pages 22 and 24.

[39] M. Raizada, "Survey on recommender systems incorporating trust," in *2022 International Conference on Applied Artificial Intelligence and Computing (ICAAIC)*, 2022, pp. 1011–1015. → Cited on pages 22 and 24.

[40] A. Al-Ajlan and N. AlShareef, "A survey on recommender system for arabic content," in *2022 5th International Conference on Computing and Informatics (ICCI)*, 2022, pp. 316–320. → Cited on pages 22 and 24.

[41] D. Röchert, M. Weitzel, and B. Ross, "The homogeneity of right-wing populist and radical content in youtube recommendations," in *SMSociety'20: International Conference on Social Media and Society*, 07 2020, pp. 245–254. → Cited on pages 22 and 25.

[42] C. Stöcker, *How Facebook and Google Accidentally Created a Perfect Ecosystem for Targeted Disinformation*. Springer Nature Switzerland AG, 01 2020, pp. 129–149. → Cited on pages 22 and 25.

[43] D. Mota and R. Martinho, "An approach to assess the performance of mobile applications: A case study of multiplatform development frameworks," *Proceedings of the 16th International Conference on Evaluation of Novel Approaches to Software Engineering (ENASE 2021)*, pp. 150–157, 2021. → Cited on pages 25, 27, 28, 32, and 33.

[44] L. Delia, N. Galdámez, P. Thomas, L. Corbalán, and P. Pesado, "Multiplatform mobile application development analysis," in *2015 IEEE 9th International Conference on Research Challenges in Information Science (RCIS)*, 05 2015, pp. 181–186. → Cited on pages 25, 26, and 27.

[45] P. Gokhale and S. Singh, "Multi-platform strategies, approaches and challenges for developing mobile applications," in *2014 International Conference on Circuits, Systems, Communication and Information Technology Applications (CSCITA)*, 04 2014, pp. 289–293. → Cited on page 26.

[46] A. Keus, S. Noichl, and U. Schroeder, “Vergleich von technologien zur entwicklung barrierefreier mobiler lernapplikationen,” in *DELFI 2019*. Gesellschaft für Informatik e.V., 09 2019. → Cited on page 26.

[47] C. Rieger and T. A. Majchrzak, “Towards the definitive evaluation framework for cross-platform app development approaches,” *J. Syst. Softw.*, vol. 153, no. C, p. 175–199, 7 2019. [Online]. Available: https://doi.org/10.1016/j.jss.2019.04.001 → Cited on page 26.

[48] Y. Cheon and C. Chavez, “Converting android native apps to flutter cross-platform apps,” in *2021 International Conference on Computational Science and Computational Intelligence (CSCI)*, 2021, pp. 1898–1904. → Cited on pages 28, 30, 31, and 32.

[49] J. Franz, “Evaluation von zustandsverwaltungssystemen für das mobile cross-plattform-framework flutter,” p. 83, 2022. → Cited on pages 29 and 31.

[50] L. Dagne, “Flutter for cross-platform app and sdk development,” p. 37, 5 2019. → Cited on page 31.

[51] B. Lutkevich, “database (db).” [Online]. Available: https://www.techtarget.com/searchdatamanagement/definition/database → Cited on pages 34 and 35.

[52] J. Groff, P. Weinberg, and A. Oppel, *SQL The Complete Reference, 3rd Edition*, ser. The Complete Reference. McGraw Hill LLC, 2008. [Online]. Available: https://books.google.at/books?id=mvAcp4dM_FQC → Cited on page 34.

[53] Z. Parker, S. Poe, and S. Vrbsky, “Comparing nosql mongodb to an sql db,” in *ACMSE ’13: Proceedings of the 51st ACM Southeast Conference*, 04 2013. → Cited on pages 35 and 38.

[54] R. Hecht and S. Jablonski, “Nosql evaluation: A use case oriented survey,” in *2011 International Conference on Cloud and Service Computing*, 2011, pp. 336–341. → Cited on pages 35 and 36.

[55] I. k. g. Sudiartha, I. N. Indrayana, I. W. Suasnawa, S. Asri, and P. Sunu, “Data structure comparison between mysql relational database and firebase database nosql on mobile based tourist tracking application,” *Journal of Physics: Conference Series*, vol. 1569, p. 032092, 07 2020. → Cited on page 36.

[56] M. Ohyver, J. Moniaga, I. Sungkawa, B. Subagyo, and I. Chandra, “The comparison firebase realtime database and mysql database performance using wilcoxon signed-rank test,” *Procedia Computer Science*, vol. 157, pp. 396–405, 01 2019. → Cited on pages 36 and 38.

[57] A. T. Kabakus, “A performance comparison of sqlite and firebase databases from a practical perspective,” *Düzce Üniversitesi Bilim ve Teknoloji Dergisi*, vol. 7, pp. 314–325, 01 2019. → Cited on page 36.

[58] Y. Li and S. Manoharan, “A performance comparison of sql and nosql databases,” in *2013 IEEE Pacific Rim Conference on Communications, Computers and Signal Processing (PACRIM)*, 08 2013, pp. 15–19. → Cited on pages 37 and 38.

[59] R. Branas, *AngularJS Essentials*. Packt Publishing, 8 2014. → Cited on pages 38 and 39.

[60] S. Delcev and D. Draskovic, “Modern javascript frameworks: A survey study,” in *2018 Zooming Innovation in Consumer Technologies Conference (ZINC)*, 05 2018, pp. 106–109. → Cited on pages 38 and 39.