



ulm university universität
uulm

Ulm University | 89069 Ulm | Germany

**Faculty of Engineering,
Computer Science
and Psychology**
Institute of Databases and
Information Systems

Development of a Distributed Workflow-based Analysis Service for Metadata of Mobile Applications

Master's thesis at Ulm University

Submitted by:

Jörn Hofschlaeger
joern.hofschlaeger@uni-ulm.de
791591

Reviewer:

Prof. Dr. Manfred Reichert
Prof. Dr. Rüdiger Pryss

Supervisor:

Michael Stach

2020

Version of June 29, 2020

© 2020 Jörn Hofschlaeger

This work is licensed under the Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International (CC BY-NC-SA 4.0) License. To view a copy of this license, visit <https://creativecommons.org/licenses/by-nc-sa/4.0> or send a letter to Creative Commons, PO Box 1866, Mountain View, California, 94042, USA.

Set: PDF-L^AT_EX 2_ε

Abstract

Smartphones have multiple functions and can, therefore, be used for many different applications. An interesting use case is in the area of mHealth apps. Therapists can use mHealth apps for special treatments in the field of meditation, depression, or tinnitus, to name just a few. This poses a problem because the ratings of apps are not objective and can be misleading. This leads to the difficulty that therapists are not able to make a good decision based on the information provided.

To improve this situation, a workflow-based service for the use case of the analysis of metadata for mHealth apps is developed. This service retrieves the metadata of the apps from the Google Play Store and allows to save the metadata of different points in time. In addition, experts can rate the apps using the MARS questionnaire in order to extend the existing data. The advantage of this service is that more information about mHealth apps is available, and the evaluations are more objective. By using the metadata of an app at different points in time, changes can be analyzed. The analysis of app metadata and the questionnaire data is used for the creation of a user interface that provides an overview of the changes and shows the results of the questionnaires. Consequently, this will help therapists to determine whether the app is suitable for a specific case.

In this thesis, a workflow engine for the orchestration of microservices is used. This is a modern approach to achieve a maintainable and scalable solution. The core of the presented solution is based on Zeebe, a product for orchestration of service tasks defined in a BPMN 2.0 workflow. Each service task is implemented as a microservice. The service tasks are implemented using the Zeebe client and are developed in the modern programming language Go. To store the data, CouchDB is used. An open-source web scraper written in JavaScript is used to retrieve the app metadata from the Google Play Store. The prototype presented in this thesis shows that a service for analyzing metadata of mobile applications can be based on the technologies used.

Contents

1	Introduction	1
1.1	Motivation	1
1.2	Problem Analysis	2
1.3	Contribution	5
1.4	Outline	6
2	Fundamentals	7
2.1	Mobile Health App Database	7
2.2	Databases	8
2.2.1	Relational Databases	8
2.2.2	Non-Relational Databases	9
2.3	Microservices and their Orchestration	10
2.3.1	Distributed Architecture	11
2.3.2	Choreography and Orchestration	12
2.3.3	Building Workflows	13
2.4	Retrieve Data From App Stores	14
2.4.1	Using an API	14
2.4.2	Using a Scraper	15
3	Requirement Analysis	16
3.1	Functional Requirements	16
3.1.1	Account	17
3.1.2	Search	18
3.1.3	App	19
3.1.4	Review	20
3.1.5	Analysis	21
3.2	Non-Functional Requirements	22

4	Concept	23
4.1	Microservice Orchestration	24
4.1.1	Zeebe	24
4.1.2	Zeebe Client	25
4.2	Database	27
4.2.1	CouchDB	27
4.2.2	Fauxton and API	29
4.3	User Interface	30
5	Implementation	32
5.1	Google Play Scraper	32
5.1.1	Fixing the Google Play Scraper	33
5.1.2	RESTful API	36
5.2	Workflows and Microservices	37
5.2.1	Implementing the Play Scraper	38
5.2.2	Implementing the Persistence Layer	41
5.2.3	Implementing Database Search	45
5.2.4	Implementing Database Updates	47
5.2.5	Implementing Ratings	52
5.2.6	Implementing Metadata Analysis	54
5.3	User Interface	56
5.3.1	User Login	57
5.3.2	Searching and Adding Apps to Database	58
5.3.3	Show Saved Apps	64
5.3.4	Update Apps	69
5.3.5	Rate Apps	70
5.3.6	Analysis of Metadata	71
6	Compliance with Requirements	75
6.1	Functional Requirements	75
6.2	Non-Functional Requirements	76
7	Conclusion and Future Work	77
7.1	Conclusion	77
7.2	Future Work	79

Contents

A Software Used and Its Versions	81
Bibliography	82

1 Introduction

This chapter first motivates the chances that are being created when adding additional information to smartphone applications. Afterwards, problems are getting discussed that are cumming up when creating a distributed service with distribution and complexity. Additionally, a general problem with ratings in app stores is getting explained. Then, it gets explained what the contributions of this thesis are. Last, the outline of this thesis is presented.

1.1 Motivation

Nowadays, more than three of four German citizens over 16 years own a smartphone [2]. Additionally, the market with apps, services, and infrastructure for smartphones grows by two percent per year. Therefore, this market is getting more and more significant.

For adding functionalities to a smartphone, applications are needed. Those applications are mostly distributed by the Apple App Store¹ for Apple devices and the Google Play Store² for devices running Android. Those platforms are called *app stores*.

However, for some cases, like for mobile health (mHealth) apps, the ratings in those app stores do not help decide which application is the best to treat a specific disease, e.g., an app that has a lot of surveys that must get completed between the regular app content can have a poor rating but still can be very helpful [39]. This is the case since users describe in those ratings how they liked the app while they do

¹<https://www.apple.com/de/ios/app-store/>

²<https://play.google.com/store/apps>

not differentiate between categories that are important for deciding which application is best for treatment [27].

To generate data that is more meaningful for therapists, there is the possibility to create an additional service, where the information from the original app stores is getting enriched with ratings and further additional information by experts. Those expert ratings are being created using a standardized questionnaire to rate apps under certain aspects which helps to make ratings more comparable.

To prevent having to copy or rewrite all information which is already given in the app stores, some platforms offer an application programming interface (API) which can get used for getting this information.

1.2 Problem Analysis

When writing a service that can get used by many users at the same time, scalability is a first big issue [4]. This can be solved by distributing the load. To make it possible to distribute the workload, different techniques can get applied.

One way to distribute the workload is by offering the same service multiple times. This idea can get implemented easily using a multi-site load balancer, which lets always balance the workload and capacity of each host running the service and lets the users use a host having enough capacity. How this can get implemented is shown in Figure 1.1.

However, in most cases, not the whole service is affected when more users are using it at the same time [11]. Still, it can be difficult to find out which part of a monolith service has problems with scaling. Thus, a solution can be splitting one monolith service into smaller and distributed microservices. This allows detecting the parts that scale badly and replicate only those. This is especially possible because the microservices are implemented independently. Furthermore, when splitting a service into microservices, this allows distributing the single services onto different servers. How this could get implemented is shown in Figure 1.2. Doing this increases not only scalability but the availability since when one microservice is not reachable this does not mean that the whole service cannot work. This is the case since every microservice is a single process, and users who do not need to utilize this microservice do not use the capacity of those.

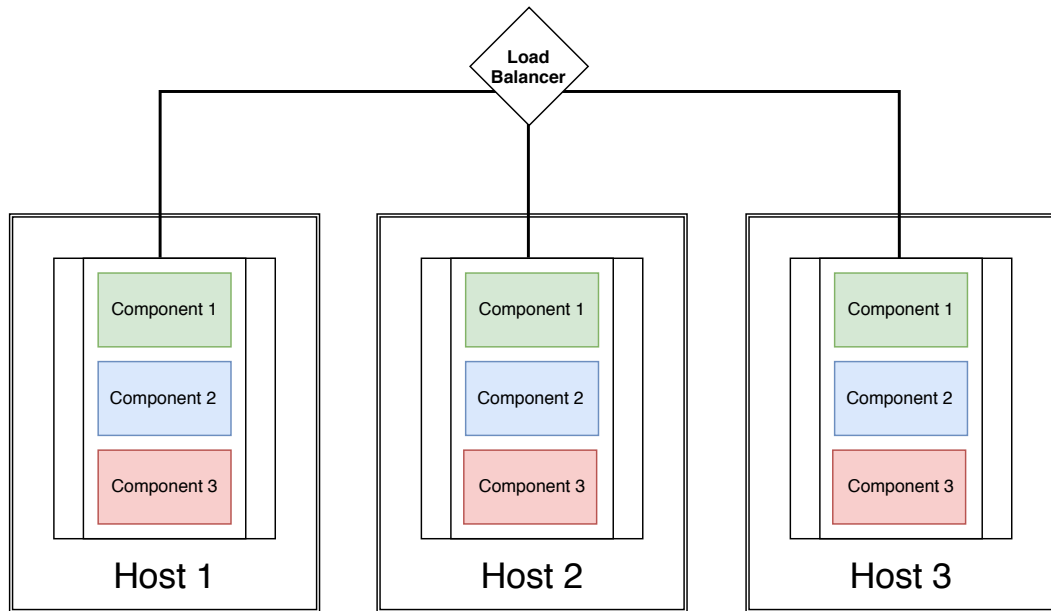


Figure 1.1: A monolithic service, having three components running on three hosts. The load balancer decides which of those gets used by each user to balance the workloads for the hosts. Based on [11].

Another important issue is complexity. Nowadays, activities that are required to perform are becoming more sophisticated and need the interaction of many persons or systems [7]. A possible approach to visualize and allow simplification of the business can get reached using *business processes*. Those business processes are describing all necessary activities and interactions. To allow automation and administration of business processes, those can get implemented in workflows [40]. Galler define workflows as the implementation of business processes [14].

When defining workflows it is essential to design and document every workflow so that it is unambiguous [7]. This means that every reader understands the same when reading a business process. Unambiguity is even more critical when implementing a business process into a workflow since here a machine needs to understand the process.

However, having a well-designed workflow, it allows getting more insight into all process activities [7]. A workflow allows identifying problems and areas where optimization is possible. Second, redundancies can get determined by using the gained insights. Third, by having responsibilities defined in business processes, it is clear who has to fulfill each task by when. Finally, it is always clear what to do with the

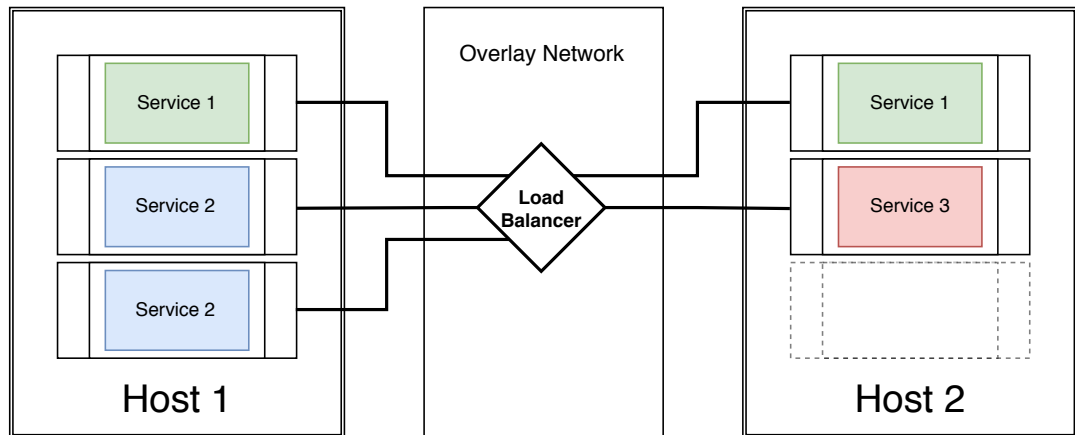


Figure 1.2: Microservices are running on two hosts. A load balancer decides which service is used for each user. Services can run multiple times or just once independent from the host. Based on [11].

results, i.e., what is the next task to get performed with the results.

In summary, using business processes makes it easier to track and improve all processes. When automating and turning business processes into workflows, the likelihood of human errors gets reduced, and the efficiency of the tasks can get improved even more.

In some cases, the data that is given by the app stores comes not with enough information since ratings and comments are not objective and often contain personal views of the users. To solve this issue, additional data can get gathered by letting experts rate apps using standardized questionnaire, which allows reaching an objective level of ratings. However, apps are getting updated, and when having many apps already rated by those experts, it is essential to review those ratings when the app gets changed.

Therefore, it is essential to check for updates and what is even more important to check if the quality of the app changes. To make this possible without letting experts check every app on each update, this can be solved analyzing the metadata of the apps. Thus, when the ratings in the app store change a lot after an update, the chance that this app was altered in a way that it needs to get reviewed is higher than in other cases where nothing changes. Here it is essential to track the ratings since, in the app store, it is not possible to get the ratings by time, i.e., it is not possible to get whether ratings are made before or after an update [27]. The downloads and comments can get further insights into what was changed, but it is hard to get

if changes are essential from those automatically. Furthermore, the investigation of the update rates can be attractive, since this allows recognizing if an app is still getting improved.

All this gathered data can get visualized and therefore help to decide if an app review potentially needs to get updated and help to choose an app to use when more than one app could be used in the same context.

1.3 Contribution

The aim of this thesis is the development of a prototype for a workflow-based analysis service. This service analyses the metadata of mobile apps to create additional information. When the service is implemented, a use case is to create a service for mHealth apps. Therefore, the service can help in cases a mobile app should get used for treatment. For creating a service that helps to decide which app is best for treatment, the metadata of apps is getting analyzed to help seeing changes, e.g., in ratings from one update to the next. Additionally, experts can add ratings that are more objective and have different categories in which the app gets rated.

A therapist can use the service via an internet user interface to search for apps. In many cases, there is not only one app that can help. To help to decide which app is best other experts rate the apps, and the ratings and changes in the ratings get analyzed. Having decided for an app, the therapist can add a rating afterwards, which helps other users of the service.

Using workflows for the service implementation allows letting services run multiple times and on different hosts. Doing this helps, e.g., when scraping the Google Play Store since here, the number of requests per host is limited. Additionally, using independent microservices allows developing and scaling each of the services individually when it is necessary. For testing the use case of the service, a user interface is implemented. In addition, the user interface allows the presentation of the results of the metadata analysis and the advantages of this.

1.4 Outline

The remainder of this thesis is getting divided into six chapters. Chapter 2 introduces all the fundamentals and ideas this thesis is based on. Subsequently, the requirements for the service to be developed are defined in Chapter 3. In Chapter 4, the concept the later work is going to build on is explained. Chapter 5 is dedicated to the implementation of the concept and the presentation of the UI. Chapter 6 presents the degree of implementation compared to the requirements defined in Chapter 3. Eventually, in Chapter 7, the thesis is summarized, and conclusions are getting drawn. Furthermore, this chapter presents possible future work.

2 Fundamentals

In this chapter, the fundamentals this thesis is based on are getting discussed. In the following the mobile health app database is getting motivated. Then, different types of databases and two techniques to implement the combination of microservices are getting discussed. Also, two alternatives for retrieving data from app stores are reviewed.

2.1 Mobile Health App Database

One field of application of this thesis is for mHealth apps. Those apps can get used for medical treatments or in medical studies where the information which apps are useful for special treatment is essential. That information is not given in app stores where everyone can rate an app by personal opinion and without always looking at the aspects which are essential for treatments or studies.

Additionally, mHealth apps became more and more popular over the last years [20]. Those apps aim to help the users to improve their health through many different functionalities, e.g., monitoring health data or training of body functions. Very often, however, the quality of content and data security are difficult to assess, so that risks, misinformation, and adverse developments cannot get ruled out when using these mHealth apps [38]. The Mobile Health App Database (MHAD) is a database system that tries to generate more transparency regarding the quality of apps in the health sector [38]. By generating this transparency, patients and practitioners are getting supported to make informed and quality-assured decisions which app to use.

Each published mHealth app has been reviewed by two specially trained experts using a validated diagnostic tool called MARS [38]. These reports are getting reviewed by a third independent scientist and then released.

For adding new apps to MHAD, those can get searched by a privileged user in the

Google Play Store and Apple App Store. Having an app found it can get added to the MHAD review process. After going through that process, the app is getting added to the MHAD.

The MHAD can still get improved. In this thesis, a workflow-based service is getting designed and implemented that has the potential to do so. However, integrating the workflow-based service into the MHAD is not part of this thesis. Therefore, for the prototype implementation of this thesis, a simple user interface will get implemented. This user interface will demonstrate the usage of this service.

2.2 Databases

In this thesis, an analysis service is going to get planned and implemented. However, there are a lot of different databases for each type of database. Since the choice of database changes how data is stored, how to talk to the database, and a lot more, it is essential to compare the possibilities.

2.2.1 Relational Databases

Since 1970 there was the idea of ordering data in sets of tuples [22]. Those databases were so-called relational databases where data gets organized in tables.

Additionally, data values must get typed to be, e.g., strings, dates, or numeric [32]. Once a table gets created, whereby the types are set, these types are enforced by the system. Having tables and types inside set by creation, it is essential to plan the usage, and all that will be filled in before [32]. This means it is always clear which datatype is presented in each field and how to join different tables.

Furthermore, in relational databases, a lot of the logic is already implemented in the database, which makes it easier to implement it in programs. On the other hand, this can be a problem when the data structure changes or things should be added later on.

Additionally, splitting up relational databases after creation, is not possible with modern relational databases. Therefore the potential of scalability is limited.

When the type of data or the fields can change later, a relational database may not

be the right choice. This is the case since a change in a field would make it necessary to overwork the already saved data. Therefore, it would be better to choose a database without schema which allows reacting to changes in the database using the software implementation.

2.2.2 Non-Relational Databases

Over the last decade, the requirements for databases changed [1, 33]. This change in requirements happened because the quantity and frequency in which data is generated and has to get saved within services has grown exponentially. As a result, new database technologies have been developed to address these new needs. Those so-called *not only SQL* (NoSQL) databases are therefore highly scalable and offer different other optimizations for getting used with many data [34].

Those newly developed databases allow, e.g., to split up databases on many different servers, to change the fields in the database from time to time, or save data in its form even when the data is deeply hierarchical [32].

The most straightforward idea of NoSQL databases were key-value databases [33]. Here key-value pairs are saved like they are in maps or hashtables or modern programming languages [32]. Key-value databases are very simple structured and allow, therefore, very performant operations. However, this type of database is not useful when complex queries or aggregation is needed.

Columnar databases are designed in a way that always the data from a given column is getting saved. This significantly improves the access time through optimized input/output operations. Therefore, columnar databases are the opposite of a relational database where the information of a row is stored together. Using a columnar database is very useful for analysis scenarios. Therefore, better when otherwise storage full of null values would need to get stored. However, when looking only on the structure, columnar databases are the midway between relational and key-value databases.

Document databases store documents including the nested structures [32]. This structure allows for reaching high flexibility. Additionally, a document database imposes only a few restrictions on data that should be saved. Each database differently solves indexing, ad hoc querying, replication, consistency, and other design decisions.

Lastly, there are graph databases that are rarely getting used [32]. This database-type consists of nodes and their relations to other nodes. Graph databases are very good at traversing the nodes through the following relationships.

In summary, all those types of NoSQL databases have different advantages and are structuring the data differently [33]. However, all NoSQL databases have in common that the specification of a data type for fields is not mandatory. Not knowing the types of data by the database moves some of the work into the programs that are using the database. Therefore, it is easier to create the database but it can get harder to implement it.

2.3 Microservices and their Orchestration

When programming projects, usually, the codebase grows when adding features [28]. Thus, after some time projects are becoming increasingly complex, and dependencies between components lead to undesirable side effects.

One way to solve this is cohesion and the use of vertical services. Different levels of abstraction are getting added to allow to change things just in one place of at least knowing where the code has to get adapted. Therefore, microservice architectures are a current trend in implementing distributed systems. Dragoni et al. define microservices as independent and small processes that are communicating directly or using an event bus using messages [11]. Therefore, implementing a project using microservices means that the project gets divided into smaller sub-projects, which can do their work independently but communicating with the others to fulfill the same target as the one big project would. Therefore, microservices are a modern way of implementing software architectures.

Implementing software using a microservice architecture allows having small teams to implement small parts that can get combined very flexible to one system [18]. Furthermore, since those services are independent and are communicating over messages, it is possible to use for each microservice the technology which fits best - which would typically not be possible in a monolithic project. Therefore, once having a microservice for a task implemented, it can get used for many independent systems. What is more, each service can get scaled individually to fit the needs. Having a lot of small microservices those can get used for implementing a distributed system.

2.3.1 Distributed Architecture

As already mentioned, using microservices, each of these services has its task, which can get fulfilled without the others. When implementing every task in an independent microservice, this results in the possibility of distributing the services using APIs. For this, each service can run independently on one or more servers which adds even more scalability.

However, having those services independently running each microservice need to fulfill several characteristics resulting in guidelines which are getting stated and explained in the following [16].

First, each microservice needs to have an independent life cycle. For each of these services, it must be possible to develop a service in an independent team. There is not supposed to be any necessity to coordinate with other teams. Additionally, it must be possible to start and stop or even change a microservice independent from others.

Second, each microservice must provide stable interfaces that do not change during updates. In cases interfaces are getting incompatible for adding more features or reflecting changes at other parts, this should be done by versioning the interface.

Third, there are cases a microservice must communicate with other microservices. Here, the communication must get implemented in a way that expects the other service not to be able to answer immediately. One way to solve this would be by using asynchronous messages and a message broker for guaranteed message delivery.

Fourth, each microservice needs to be robust and fault-tolerant. Therefore, when one microservice causes problems, other microservices are not affected. If one instance of a microservice has a problem, also its other instances must not be affected by this either. In most cases, it is better to let one instance of service fail or restart than letting the entire system fail and stop working.

Fifth, for being able to fulfill their work, microservices often keep local copies of the data they need. Storing local copies mainly helps in meeting the other policies such as independence.

Last, for allowing independent scalability, it must be possible to provide each service with the needed resources while not affecting others. However, not every microservice must fulfill all of those mentioned guidelines but reach the same targets to make it possible for the microservices to work together.

2.3.2 Choreography and Orchestration

Microservices can get combined to workflows that define the composition of microservices, which are needed, to reach a defined condition. For implementing the workflows, microservices need to get orchestrated or choreographed [26].

When using choreography, the focus is on organizing the collaboration of the services. Here, each service calls the next one and hands over the results which are needed for the next service. However, in cases, one service cannot finish its work, a fail message needs to get sent. This is necessary to let the other processes know what problem happened at which service. Therefore, choreographing microservices means that each service needs to have implemented which service to call after finishing and what to do in error cases. This implies that each service needs to get adapted for every workflow. This concept is shown in Figure 2.1 a).

Orchestration prevents that adaption is getting necessary by adding a layer which takes care of handing messages from one service to the next. This layer is mostly implemented in a workflow engine. Additionally, when orchestrating workflows, it is always clear which service does not work as intended in some error cases, and it is possible to debug such cases since the information that was handed to the service is available at the orchestration layer. Furthermore, retries can get implemented, e.g., when the service does not answer at a specified time or when an error is getting returned.

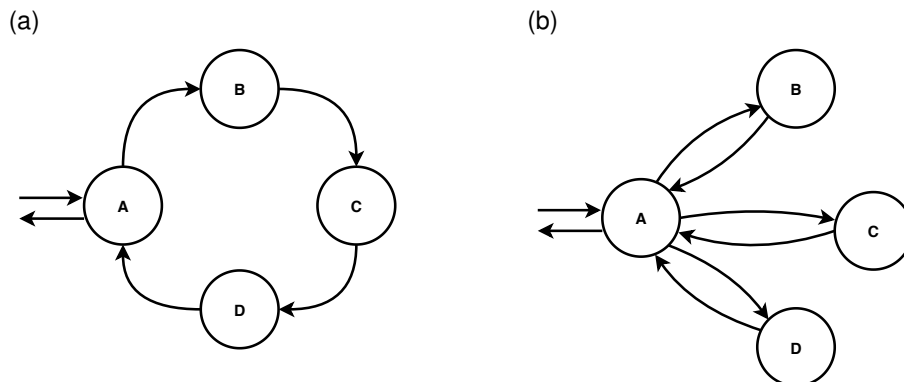


Figure 2.1: The difference of using choreography (a) and orchestration (b) for workflow composition. When using choreography all microservices communicate directly, whereas orchestration adds one more layer for controlling.

Therefore, having a service implemented it does not need adaptations for different workflows since the combination to a workflow, and the handling of messages get moved to another layer. Additionally, when a microservice is always getting used for a task, errors must get fixed in one place only. The concept of this idea is shown in Figure 2.1 b).

To sum it up, for being able to use microservices multiple times without having to change some parts, orchestration has to get used. This is due to the necessity of combining all services directly. On the other hand, this makes an additional layer necessary, which orchestrates the microservices. Therefore, both concepts have reasons to get used.

2.3.3 Building Workflows

When using a microservice architecture, the implementation of workflows is an important aspect [16]. As already mentioned, workflow engines can get used for this. For capturing business processes, Business Process Modell and Notation (BPMN) is a standard [19]. BPMN was published in 2004 by the Business Process Management Initiative (BPMI) as Process Modelling Notation. After the BPMI and the Object Management Group (OMG) merged in 2005, the BPMN was published as an OMG specification for the first time in 2006. When introducing BPMN 2.0, the name was changed to Business Process Modell and Notation to highlight the introduction of execution semantics. The current version of BPMN is 2.0.2.

BPMN combines the advantages of using Extensible Markup Language (XML) Process Definition Language (XPDL) and Unified Modeling Notation (UML) and is therefore used in many companies for creating and automating workflows [9].

To allow using the idea of implementing workflows, an approach is to extract the information of BPMN workflows by software [29]. Extracting the information using software microservices can get orchestrated following the workflow without having to implement the orchestration. However, the implementation of the service tasks is still needed.

To make this process even more comfortable, Camunda created a microservice orchestration tool called Zeebe that allows importing the BPMN workflow directly [16]. To be able to import BPMN processes, those are getting written in a modeler, which saves the workflows in XML. For defining which microservice can get called how

this BPMN workflow can get imported to a client. Using the client workflows can get enriched with information on how to call each microservice.

2.4 Retrieve Data From App Stores

Applications for smartphones are shared in app stores, mainly the Google Play Store – for Android devices – and Apple App Store – for iOS devices. When the data of these stores should be analyzed, it must be retrieved from there in some way. Getting the data allows saving and enriching the data alongside the analysis. For getting the necessary data from the app stores, two different techniques are getting discussed here.

2.4.1 Using an API

One technique to get data from an app store is to use an API that is given by the provider. Such an API allows getting the requested content via specific calls over a web interface [23]. For the user, it is just essential how to use it and in which form the necessary data is presented. For making it easy to work with an API, therefore, proper documentation is needed. Additionally, to make it possible working in programs with such an API, the data is presented mostly in JavaScript Object Notation (JSON).

Some years ago, for interchanging data on the web, XML was the primary format [24]. However, JSON is simple to read and allows translating objects easier to concepts that are known by software developers such as arrays, objects, and key-value pairs. Therefore JSON objects are fitting better for the nowadays mostly used object-oriented design and development.

What is more, JSON documents are faster transmittable and, therefore, more efficient to process than the same data written in XML. Thus, when there is an API for an app store, this should be used. This is the case since fields in an app store are changed sometimes – which can cause problems when not changing the software. This is not the case for an API, for this, the provider mostly does not change the interface and ensures that the API will still produce the wanted outcome.

However, using an API is not always free. Some APIs like the one of the Apple

App Store need to get paid, some others allow a low usage for free, but when more requests are wanted in a certain period this needs to get paid.

Contrarily, sometimes there is not even a paid API, which is the case for Google's Play Store. In this case, where no API is present another way to get the necessary data has to get found. One solution to still get the data is using a scraper which collects the information from websites.

2.4.2 Using a Scraper

Having no API given by Google, a way to get the data is to scrape the app store by using the Play Store's website. A scraper offers the functionality which is wanted by an API when no API is given or not accessible [37]. A scraper collects the information that is presented on websites and orders them in a way that programs can use it. Reordering the information to new clusters allows performing different operations on the data. Thus, the use of scrapers differs for each purpose.

Therefore, it is essential to know that a scraper written for one page does not work for another website [37]. This is the case since it has to get modified to scrape the necessary information from each site. This adaption is required because the scraper gets a full website and tries to detach all necessary information from it. To make this possible, it needs to get implemented where on the site which information can get found.

If not only information from one page is to be scrapped, but all information that a website offers, a scraper must also scrape different sub-pages to get to the necessary information. In some cases, it is furthermore necessary to combine the knowledge of different sites. For controlling a scraper, one could implement it into the service that needs the information. This allows calling the scrapers functions directly. Another possibility, which is not always given, is to use an API for this task. This possibility is not always given since the scraper needs to offer an API, which means additional work when implementing it. An advantage of this is that it opens the potential of using more than one scraper for the same instance of a workflow to distribute the web calls one scraper needs to do. However, for getting able to use a scraper as a microservice, it needs to implement an API. Therefore, a scraper can not get implemented as a microservice directly. However, a solution can be to add a API layer which allows using the scraper as microservice.

3 Requirement Analysis

In this chapter, the requirements for this thesis are getting analyzed and defined. With this, a distinction is made between functional and non-functional requirements. In order to improve the overview of the requirements, these are divided into modules. Each requirement has a code and a description that can be used to refer to the requirements. Additionally, each requirement is assigned a priority according to the MoSCoW method [5].

3.1 Functional Requirements

The functional requirements define all functionalities that the application has to fulfill. These requirements affect all later decisions that relate to the implementation of the application. Each module of requirements is described with a use case diagram. All actors that are used in the use case diagrams are defined in the following.

Guest: Every actor that is not logged in to the application is a guest. Guests are only allowed to log in and register.

User: After registering, every actor is a user. Users of the application can view all app metadata and analysis that are saved to the service. Users can rate apps using a standardized questionnaire. Also, users can view all saved questionnaires.

Operator: Operators have all rights of a user. Also, operators can search the app store for applications and add them to the application.

Administrator: Administrators have all rights of the operator. Additionally, administrators can administrate users.

Code	Description	Priority
Account:		
F01	Authorization	COULD
F02	Registration	COULD
F03	Authentication	COULD
F04	User Administration	COULD
Search:		
F05	Search for Apps	MUST
F06	Save App Metadata to Database	MUST
App:		
F07	Show All Saved Apps	MUST
F08	Update App Metadata	MUST
Review:		
F09	Review App	SHOULD
F10	Show All Surveys per App	SHOULD
F11	Show Survey Data	SHOULD
Analysis:		
F12	Analyse App Metadata	MUST
F13	Analyse App Survey Data	SHOULD

3.1.1 Account

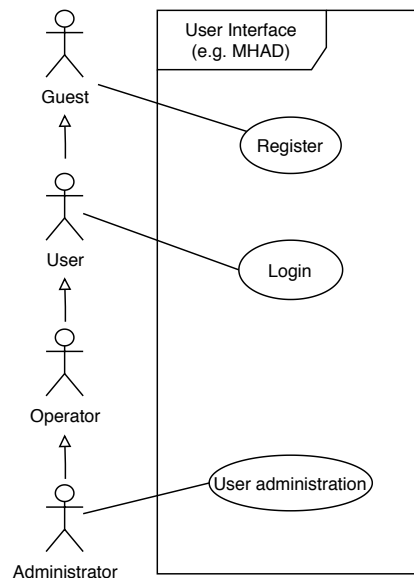


Figure 3.1: Use case diagram for the account module.

F01 (Authorization): When an actor is not logged into the service, access is blocked on all pages except the login and register page.

F02 (Registration): Guests can log in to the application.

F03 (Authentication): Registered users can log into the application and can use all functionalities granted to their user group.

F04 (User Administration): Administrators can administrate users. Which means that they can change for example the user group for every user.

3.1.2 Search

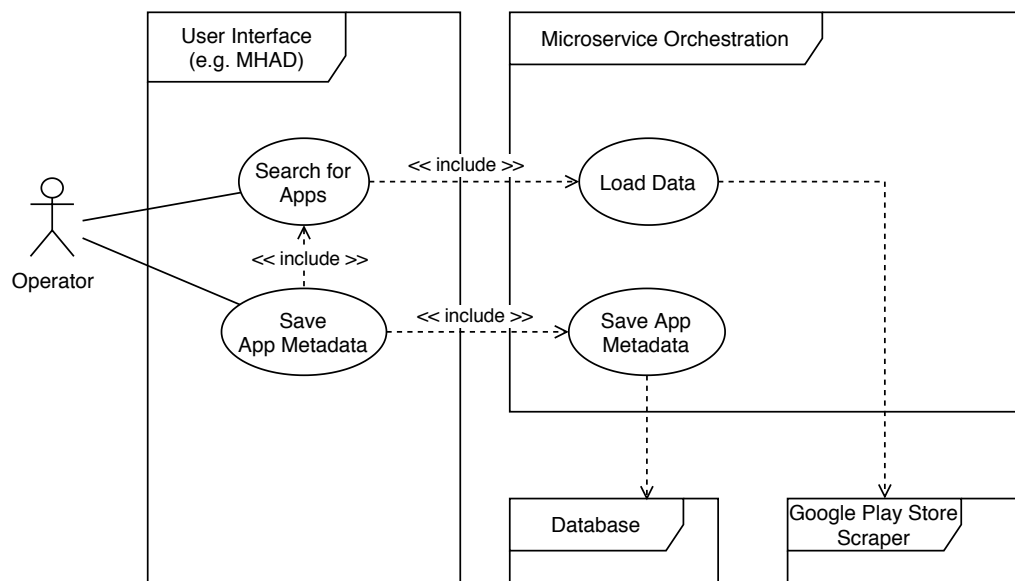


Figure 3.2: Use case diagram for the search module.

F05 (Search for Apps): An operator can search for apps. For this task a microservice queries the Google Play Store scraper and returns the results.

F06 (Save App Metadata to Database): For saving the metadata of an app to the database, an operator searches for app data as described in F05. Using a microservice, the app data of a chosen app can then get saved to the database.

3.1.3 App

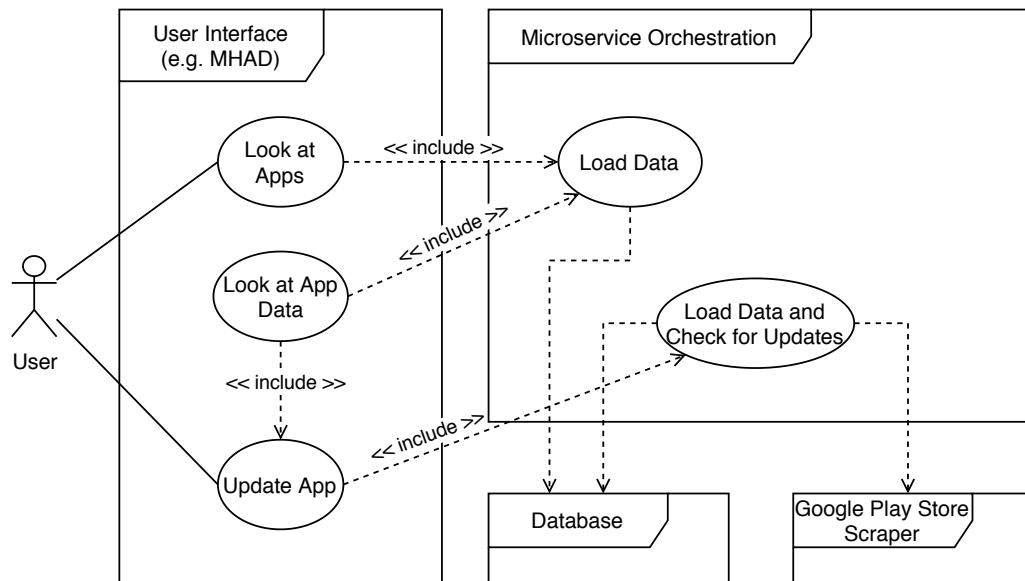


Figure 3.3: Use case diagram for the app module.

F07 (Show All Saved Apps): A user can view all saved apps that are saved to the database. For this, a microservice loads the data of all saved apps from the database.

F08 (Update App Metadata): For updating the metadata of an app that is already saved to the database, a user can trigger that process on the user interface where all data of an app is shown. When triggering that process, a microservice loads the data that is saved to the database and current data using the Google Play Store scraper. When both data is queried, it gets checked if the database data should get updated. In case the data should get updated the microservice is saving the new data to the database.

3.1.4 Review

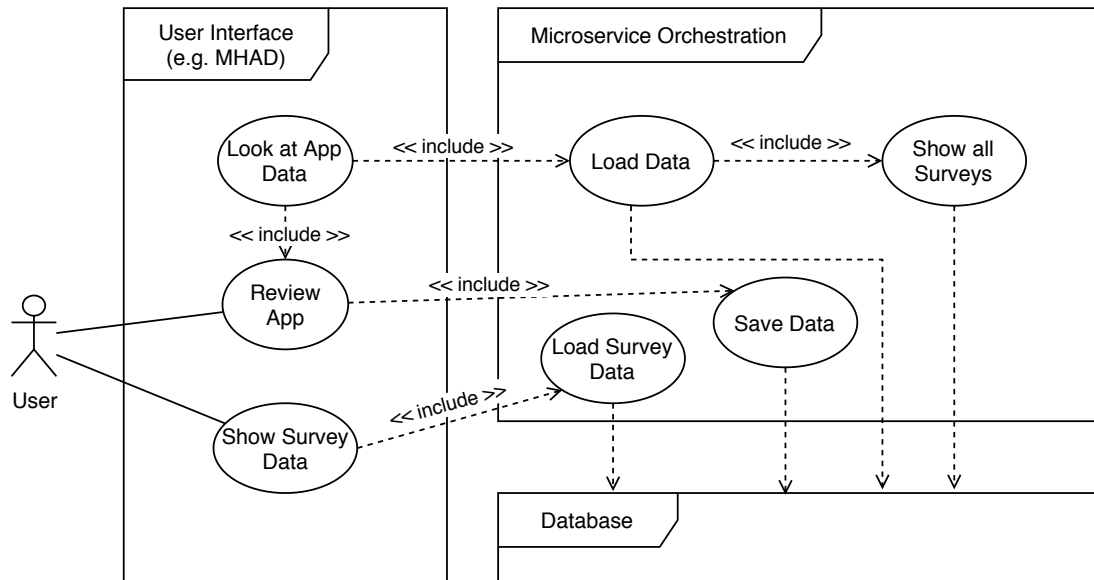


Figure 3.4: Use case diagram for the review module.

F09 (Review App): A user can review apps, by triggering this process from the app page in the user interface. When the questionnaire is completed it is send to a microservice that saves the answers to the database.

F10 (Show All Surveys per App): For seeing all reviews that are saved for an app, opens the app page in the user interface. Doing this, the data for the app gets retrieved from the database. Also, the corresponding surveys are getting retrieved.

F11 (Show Survey Data): To get all data of a specific review, the user can choose one of the in F10 retrieved entries. A microservice is triggered by this, that retrieves the wanted data from the database.

3.1.5 Analysis

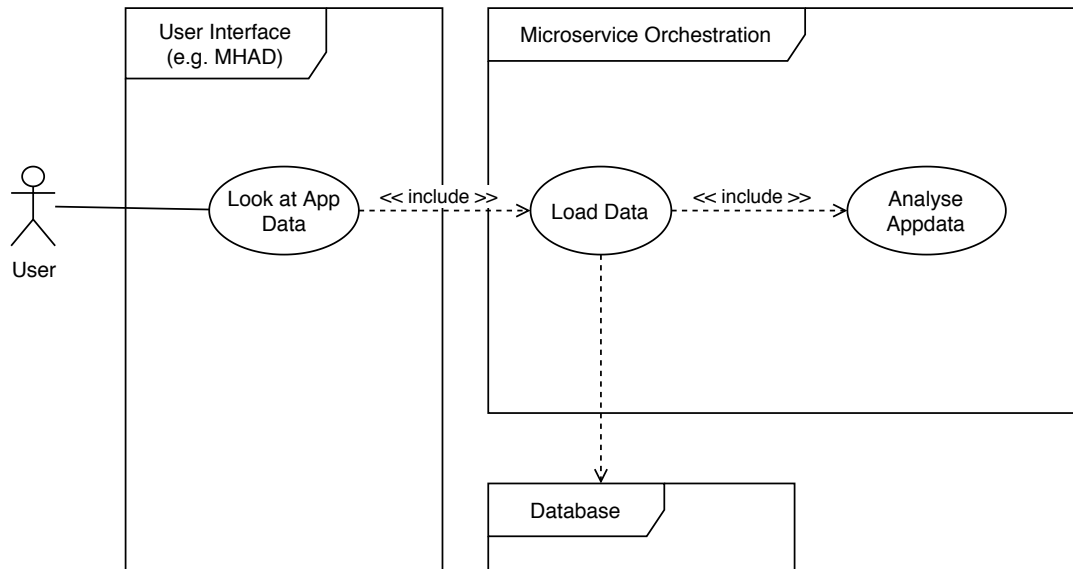


Figure 3.5: Use case diagram for the analysis module.

F12 (Analyse App Metadata): When a user looks at the app data in the user interface, the metadata of this app gets retrieved from the database. In addition, the data is getting analyzed or getting returned in a form that it offers additional insights for the user.

F13 (Analyse App Survey Data): A user can get the analyzed survey data when looking at the app data in the user interface. Here, when retrieving the survey data, it gets analyzed, or it gets returned in a form that offers additional insights for the user.

3.2 Non-Functional Requirements

The non-functional requirements are listed below. They define the quality in which the required functionalities are to be provided.

Code	Description	Priority
System:		
NF1	Reliability	MUST
NF2	Scalability	MUST
NF3	Maintainability	SHOULD
NF4	Extensibility	SHOULD
NF5	Robustness	SHOULD

NF1 (Reliability): The application has to be reliable in any situation. Therefore, the application must be accessible in all possible situations.

NF2 (Scalability): The application must be scalable. For this purpose, all parts of the application should be replicable, and the software used should have been implemented with scalability in mind.

NF3 (Maintainability): The implementation of the application must be well documented. This allows the application to be maintained at a later point in time.

NF4 (Extensibility): Adding additional features to the application should be possible. For this, documentation of is essential.

NF5 (Robustness): The application should be able to process or intercept incorrect user input. Therefore, the application should react in a predefined way.

4 Concept

In this chapter, the concept of this thesis implementation is getting explained. Therefore, in this chapter, the idea, technologies, and the decisions about why each technology was chosen is getting discussed.

The focus of this thesis is on the development of a workflow-based service and its functionalities. For implementing the workflows service tasks, microservices are used. Therefore, an essential part of this thesis is the implementation of a microservice orchestration. This orchestration makes it possible to use microservices that are using the Google Play Scraper to query app metadata. Also, the service will not only save app metadata but add additional analysis, e.g., saving the version of the app when a rating was given, having data about the updates frequency and the app store rating in connection to this.

The service consists of four parts. First, there is the user interface that every user uses to interact with the service. An example of the user interface can be the MHAD explained in Section 2.1. Second, the microservice orchestration. The microservice orchestration is the most critical part of this thesis. Here the tasks which are requested by the users are getting carried out. Third, the database, which stores all apps and the analysis of those. Fourth, the Google Play Store scraper, which is getting the app metadata for android devices.

When looking at apps that are already in the database, first, the data is directly loaded from there. Additionally, it is getting checked if there are any updates for this app in the app stores. In the case of a newer version, the saved app metadata is getting updated. Then changes are getting analyzed. When a user wants to rate an app, the data is loaded using the same process. Additionally, the user can now rate the app. The resulting rating is getting saved to the database. When an operator is adding a new app to the database, the app stores are getting searched for the desired apps. The results are getting shown the operator who chooses which of the results should get added to the database.

In the following, the concept and decisions for the implementation of the microservice orchestration and the database are getting explained. These are particularly important for the concept because the decisions made here also have a substantial impact on the implementation. Last, the concept for implementing a prototype UI is getting presented.

4.1 Microservice Orchestration

Dividing the service into microservices allows better scalability since this makes distributing the workload more manageable. Additionally, to prevent having to work on implementations that are needed more than once, the microservices are getting orchestrated.

4.1.1 Zeebe

When looking for an orchestration software that is lightweight and focuses on scalability, Zeebe is something that comes up. Zeebe is open-source under the Zeebe Community License Version 1.0. The official clients are open-source under the commonly known Apache License, Version 2.0. Being open-source, everyone can help to find and fix issues that come up when using Zeebe. As mentioned earlier in Section 2.3.3, Zeebe allows working with BPMN directly. Therefore, the BPMN workflows are getting written using a modeler that allows adding labels which can later be referenced when implementing each service task.

Zeebe allows to orchestrate microservices in workflows [15]. Since workflows are written in BPM, it gets ensured that each task is well defined. By doing this, Zeebe is scalable and fault-tolerant. Zeebe can be used directly or, what is the recommended way, with Docker¹. Using Docker no installation of software is needed since all this is coming with Zeebe in the Docker container [25].

When using Zeebe with Docker, several compose files are already given. Using Docker allows starting Zeebe for a lot of different purposes, i.e., Zeebe can get started for development or production. The difference is, e.g., that for developing an

¹<https://www.docker.com/>

additional web-monitor is started with Zeebe. This so-called "simple monitor" allows seeing each implemented workflow and its current state. In Figure 4.1, a workflow is shown in the simple monitor. On the left-hand side, metadata of the workflow is getting presented. In the center, the workflow is shown with the running and finished instances. In the bottom, the data of the selected instance gets presented. In the tabs, different data of this instance can get selected.

However, when implementing a service task, it can be helpful to see what data was sent in each step, but for production, this monitor needs additional performance and should, therefore, not run at all times.

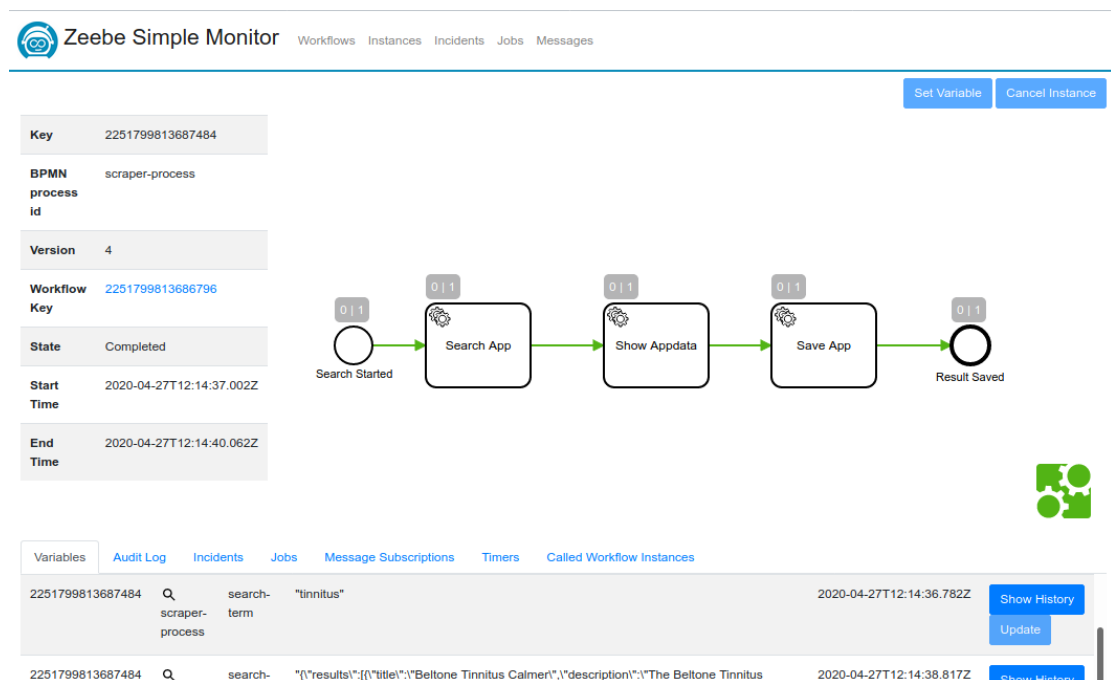


Figure 4.1: The simple monitor is a web-based interface for Zeebe which allows seeing all current workflows, their progress, and which data was handled.

4.1.2 Zeebe Client

Clients for Zeebe can be used in many different programming languages. However, the clients who are officially supported are written in Java and Go. For implementa-

tion, the Go client is going to get used, since using an official client ensures that this is going to get supported for new versions directly. Also, community support could end at any point when the main developer changes to another product or just has no time to adapt the code.

Using Java was the first idea since Java is a programming language that is often used to explain most of the concepts of programming. Thus, Java is very well known and allows implementing code in a known way. Java gets used for an explanation because it was written already in the 1990s and is very popular since [12]. Java is a general-purpose programming language that is very powerful [13]. Additionally, Java is very stable, which made it very popular with enterprises and large shops. However, code written in Java is not very compact and thus not always easy to read. Switching from Java to Go has some advantages too. The design of Go was started in 2009 by three persons working at Google [8]. They designed Go as a language that can solve common issues that appear when scaling distributed systems. When implementing functions in Go, many errors get caught by the compiler, e.g., implementing a package and not using it creates an error [41]. For preventing different styles of using curly braces when programming in Go, there is just one right way of using those. In Go, the opening curly brace has to be in the same line as the function it is opening since, in other cases, it is not going to compile. Another thing that makes code written in Go very readable is that defining a new variable is different than when using an already used one. For creating a new variable `:=` is getting used, for defining a value to an already known variable `=` is used.

For implementing a concept based on microservices, it is essential to reuse the methods. Therefore, the implementation of service tasks needs to get implemented not directly in the same file or package where the workflow is getting defined and sent to the workflow engine. Thus, the service tasks are getting implemented in an extra package which gets imported.

Importing functions from other packages in Go is different than in other programming languages [41]. In Go, every function that should get exported and therefore is possible to import has to start with a capital letter. Any functions that do not start with a capital letter are not accessible from outside the package. Second, importing own packages can get done using several options. The easiest way is by using relative paths. Another possibility is compiling the package that should get imported, which saves the compiled package to the local cache. Using relative paths allows preventing that step of having to compile the package for each change, but since

the path could be not the same on another machine. Therefore, using relative paths for import should only get used for testing. Thus, the service tasks are getting still adapted, for imports relative paths are used for now.

To sum it up, code written in Go is very readable and better scalable than code in Java since scalability is one of the main focuses of Go. Some things are different than in other programming languages; this is going to create some issues when starting to work with Go. However, accepting the differences, these concepts enhance writing code that is readable and understandable by others. Therefore, Go is going to get used for the implementation of this project of the Zeebe client.

4.2 Database

When implementing a service that needs to save data, one of the most critical decisions is which database going to get is used. This decision is essential since each database has advantages and disadvantages compared to others that are more or less important for the desired usage [32]. Thus, even if all databases can save data, it is essential to choose a database that focuses on the aspects that are needed for the service.

First, it is essential to have the possibility to save data not always in the same format - the data which is obtained from app stores could change over time. The database should allow saving data in different formats and let the implementation solve possible issues with changes. Second, creating a distributed service, it is hard to predict the amount of data or users before. Therefore, the database should be scalable. Third, it should be possible to search the database fast for the wanted results, e.g., all data of one app and the differences in metadata. A database that fits the explained needs is CouchDB.

4.2.1 CouchDB

For the implementation, CouchDB, a document-based database, is used. Describing CouchDB, one word is very important; it is "relax" [3]. Relaxation in the context of CouchDB means many different things. First, the core concepts of CouchDB are focused on ease of use. Therefore those concepts can get understood quite easily

by everyone who has been working on the web. Second, CouchDB has an internal structure that is fault-tolerant. Thus, failures that occur get controlled and stay isolated in single requests. Third, CouchDB is built for handling varying traffic. In cases of traffic spikes, CouchDB will take more time to answer, but all requests are getting answered. Additionally, after those spikes, CouchDB will return to regular speed. Fourth, the hardware running CouchDB can get scaled easily. Therefore, when more requests are expected from a certain point in time, hardware can get scaled up without any problems. In other cases, or when less traffic gets expected, the hardware can get scaled down the same way. Last, some features which would result in preventing to allow scaling are left out by design. Thus, some ways of implementing are not possible using CouchDB which results that some things have to get done not as it would be normal with other databases.

For scaling the database, one of the core features of CouchDB is replication [21]. Databases can get replicated very quickly, which allows bringing the data closer to the user or adding more capacity to the database. When replicating a database, it is possible to replicate not all data in a database, which can get useful in cases only a part of documents is needed more often or for clients closer to the replicated database. For preventing documents from getting replicated, they can get created as local documents that get never replicated. Another possibility to choose which documents should get replicated is including a selector object to the documents that should get replicated. Last, a filter function can get used for filtering the documents from a database that should get replicated. Replication can be continuous or just made once. When replicating, the changes will get transferred only in one direction. However, a continuous replication can get archived when setting up a replication task on both nodes in the opposite direction.

Another essential feature of CouchDB is design documents [36]. Design documents are a particular document type which contains application code. Since this code is running directly inside a database, the application API runs very efficiently. In design documents, views and other application functions can get created. These views getting created using JavaScript and are the primary tool for querying. Views consist of a map and a reduce function, where the reduce function is optional.

For aggregating information that gets stored in CouchDB views are needed [6]. Views allow converting single documents into a list of information. This list can then get used for querying and selecting information, or group the information by selected values. Thus, views allow to index and query information which gets stored

in documents. Furthermore, views allow producing lists of specific elements of a document. Another use case is using views to create tables or lists of information, which is summarizing the data of documents. Views can further get used for extracting or filtering information from documents. Last, views can get used for calculating, summarizing, or reducing documents. For each database, multiple design document, each including multiple different views can get created to get the wanted information.

When using views, it is essential to know that views get only updated when accessing the view [6]. Meaning, views do not get updated when adding data to the database or when updating it. Thus, when accessing a view the first time, or after many changes in the database, it will take some time until the index gets created or updated, and the view gets generated. However, in most cases, the update process is comparatively fast. Another important aspect of views is that they are working with B-Trees, which are bound to the design document. Therefore, accessing a view, all views indexes that are bound to the same design document are getting updated, even when only one view was accessed. Using a B-Tree for storing the information makes it very efficient to get an item based on a key, or even a range of keys. Furthermore, a view can get queried while grouping by the key. Grouping the data allows querying a view only for a given key, or for all keys at the same time but get the results subdivided for each key.

4.2.2 Fauxton and API

When working with CouchDB, there are two ways of communicating with it. First, there is Fauxton², a web-based interface which is built into CouchDB. Second, there is an API.

Using Fauxton to communicate with CouchDB allows using the majority of the functionality, including the ability to create and update views and documents. Additionally, CouchDB can get configured using Fauxton. Therefore, Fauxton can get used for configuring CouchDB and allows looking at the data in the databases. How Fauxton looks like is shown in Figure 4.2. Additionally, Fauxton allows creating and testing views without the necessity of writing the requests to query the view with all options.

²<https://couchdb.apache.org/fauxton-visual-guide/>

Name	Size	# of Docs	Partitioned	Actions
_replicator	2.3 KB	1	No	
_users	2.3 KB	1	No	
app	54.9 KB	17	No	
survey	6.7 KB	3	No	

Figure 4.2: Fauxton, a native web-based interface for CouchDB.

However, for implementation, the CouchDB API can get used. Having an API given, microservices using CouchDB can get implemented easily, as explained earlier. All aspects that are needed for implementing the API get explained in detail when using them the first time.

4.3 User Interface

For using the analysis service not only for testing, a web user interface (UI) is also implemented. However, the focus of this thesis is not the UI. Therefore, the UI is going to get implemented functionally as a prototype.

To allow creating a UI that is created fast, it is getting written in HTML templates, implementing a CSS stylesheet that makes the HTML elements looking good without further work. The stylesheet which is going to get used is MVP.css³ a minimal-

³<https://andybrewer.github.io/mvp/>

ist stylesheet. MVP.css is getting used here since it does not change the HTML based on classes as other stylesheets do – it does change the HTML elements directly. Thus, the HTML template does not need to get changed to make use of this stylesheet.

The functionality of the UI is written in Go. Using Go for the functionality is going to allow using the same Zeebe client as for the implementation to start workflows and accessing the workflow variables. Thus, the code that is implemented before testing the workflows can get reused in some parts. Additionally, using Go here means that the expertise of using Go is getting more over the whole work which will result in the best possible implementation at the end.

For being able to implement some of the analysis parts in the UI, JavaScript is also used. Using JavaScript will allow representing app metadata dynamically for each app. For creating charts for the analysis ChartJS⁴ is used. ChartJS is an open-source JavaScript library that allows creating different sorts of charts in HTML5. Thus, the analysis changes on each update of the app metadata, which is getting saved to the database. Furthermore, JavaScript is used for implementing the expert rating generated with the library SurveyJS⁵. SurveyJS is a JavaScript library that allows showing a survey that gets generated from JSON data. The data requested in the survey is getting returned in a text form that can get transferred to JSON directly. Having the data in JSON allows saving it to CouchDB in an afterward easily accessible format.

⁴<https://www.chartjs.org/>

⁵<https://surveyjs.io/>

5 Implementation

In this chapter, the implementation of the service will be explained. First, the parts needed for the service will get explained. Afterward, the implementation of these parts in the analysis service and the implementation of the service tasks are shown and explained.

5.1 Google Play Scraper

For this thesis, an scraper for the Google Play Store is getting used. Using a scraper is necessary since Google does not have an API that can get used. Using an already working scraper allows focusing less on the scraping and more on additional services.

Therefore, an open-source Node.js scraper, called Google Play Scraper [31], which is used by more than 300 other projects, is used. Using this scraper, which has a supportive community, ensures that even after the end of this thesis, the scraper will be adapted to work with the Google Play Store. Updating the scraper is necessary since the Google Play Store gets adapted every once in a while which would result otherwise in a not working play scraper.

Furthermore, for this project, there is the possibility of using an API that allows consuming the data produced by the scraper. Using the scraper and the corresponding API is going to allow using the data from the Google Play Store in the service, which is going to get implemented in this thesis.

However, this play scraper has some open issues. One of those, which can affect the quality of the service first had to be fixed.

5.1.1 Fixing the Google Play Scraper

A problem with the scraper was that it could not get app data with full detail when searching for an app by name. An executed search only returns the data, which is visible when searching for an app in the browser. However, this is a known issue¹ that came up during a restructuring of the code at the beginning of 2019.

Since this can be an issue for the service that is going to result from this work, solving this issue is an excellent first step. Furthermore, this is an excellent possibility to get to know the scraper and its possibilities better.

Before solving this issue, it made no difference if *fullDetail* was set to true when using the search function, which is shown with its results in Listing 5.1.

```

1 g.search({term: 'panda', num:1, fullDetail: true })
2   .then(console.log);
3 [ {
4   title: 'Panda Pop! Bubble Shooter Saga & Puzzle Adventure',
5   appId: 'com.sgn.pandapop.gp',
6   url: 'https://play.google.com/store/apps/details?id=com.sgn.
      pandapop.gp',
7   icon: 'https://lh3.googleusercontent.com/n-N5SeSks0VootPWoyDGek1iY
      -_EeoL46hhW0gnxbbP0eypVMqMlJt0wAZSzN-yDo0t0',
8   developer: 'Jam City, Inc.',
9   developerId: '5509190841173705883',
10  priceText: 'FREE',
11  free: true,
12  summary: 'Aim, match, shoot and pop in this free bubble shooter
      classic saga adventure!',
13  scoreText: '4.4',
14  score: 4.4345155
15 } ]

```

Listing 5.1: An example search using the Google Play Scraper with not working fullDetail option.

As it gets shown here, there is already some information visible. Even more data, is missing, e.g., which Android versions are supported, how often the app is installed, and the full app description.

For solving this issue, the first step was to understand the issue. The problem when using full detail was that the additional wanted information is not in the HTML response when using the search on the page, in which the search function is scraping.

¹<https://github.com/facundooolano/google-play-scraper/issues/311>

Therefore an additional call for each wanted app using the apps id is needed at the right place.

In Listing 5.2, the code where the full detail option should be looked on can be seen. Here a different treatment for the results should be applied when the option was set.

```
1 function search (getParseList, opts) {
2   return new Promise(function (resolve, reject) {
3     if (!opts || !opts.term) {
4       throw Error('Search term missing');
5     }
6
7     if (opts.num && opts.num > 250) {
8       throw Error("The number of results can't exceed 250");
9     }
10
11    opts = {
12      ...
21    };
22
23    initialRequest(opts)
24      .then(resolve)
25      .catch(reject);
26  });
27 }
28
29 module.exports = search;
```

Listing 5.2: An excerpt of lib/search.js showing the code which results in not working fullDetail option.

When starting to implement the functionality, it was hard to understand the Node.js semantics and especially how to work in this case with asynchronous calls. The problem with those asynchronous calls was that they often returned a pending *promise*. In Javascript a promise object is used for asynchronous computations. A promise represents a value which was not available at the point in time it was requested. Before understanding and then solving the issue of getting those objects returned, it happened even that the written functions returned an array of pending promises or a pending promise object including other pending promises.

Though, it was needed that the functions wait for the promise to get fulfilled and not return it before. When returning a value pending promises, it should be waited for pending promises. However, this is just working for one promise. Therefore, for

each return value, just one promise should be pending. In cases this is not possible, there is the possibility to wait for those promises, but this is more complicated than just returning this. Another thing which has to be in mind is that getting apps with all information can take a lot longer. Therefore, this additional information should only be called for apps in which the user is interested. Also, this has to be in mind, when testing this functionality since for tests there is often a timer which prevents waiting forever for a test. At the end, a solution was found for the given problem. This solution has been simplified after revising it multiple times.

For this solution, the apps are getting enriched with full detail directly before they would be returned otherwise. Not trying to add the details earlier makes sure that the app Id is saved for each requested app. Additionally, at this point, only the requested apps are getting treated. For this, a map function is used to call the app module to load the additional data. Finally, before returning the promises, which would be returned otherwise, are getting resolved.

However, the additional needed time comes from here. For each app where full detail is wanted, a new call for this app is getting created. This new call for each app is necessary since the wanted data is not shown on the overview page but the apps page. This page is getting scraped using the app module. To make this possible the app module is called for each app using the apps identifier.

The implementation is shown in Listing 5.3.

```
1 function search (getParseList, appData, opts) {  
  ...  
23   initialRequest(opts)  
24   .then(resolve)  
25   .catch(reject);  
26 }).then((results) => {  
27   if (opts.fullDetail) {  
28     // if full detail is wanted get it from the app module  
29     return Promise.all(results.map((app) => appData({ ...opts,  
30       appId: app.appId })));  
31   }  
31   return results;  
32   });  
33 }  
  
35 module.exports = search;
```

Listing 5.3: An excerpt of lib/search.js showing the proposed solution for searching apps with full detail.

After having the issue with the not working full detail in search solved, a pull request was made in the original repository² to make this solution available for everyone using the Google Play Scraper.

When having the pull request submitted, the code was reviewed by the maintainer of the scraper. The maintainer found that even when having now code that is very short and easy to read, using the app metadata and importing this functionality could be a problem when using the scraper together with a memoization functionality that allows caching the results of a query to prevent scraping for the same apps more often than necessary. To prevent this, it was needed to hand the import over from `index.js` where the functionality of each function is set. A problem with this was that here Ramda [35] is used to simplify the code. Ramda allows writing functional JavaScript, which is very elegant to write and read once it is understood.

When having this final problem solved, the pull-request was accepted. Therefore, apps can get with having full details from the play store. As a result, the previously used query now provides much more information, which can be seen in Listing 5.4. The fix was released shortly after in Google Play Scraper version 7.1.2. Therefore, the fix, which was implemented here can now get easily implemented using this version release.

5.1.2 RESTful API

For implementing the Google Play Scraper, it offers a RESTful API [30]. The API can be used directly or in a Docker container, which makes it easily set up and usable.

Having the API running, this allows calling the API with HTTP GET calls and to get the wanted app metadata back in a JSON format. Before the API can be used, it is essential to check the versions of the packages it uses. Checking the version is necessary especially for the play scraper, where it is using version 6.2.7 in the current version. Therefore, the earlier explained pull request is not implemented here. However, this can be changed easily in the APIs package file where all dependencies are defined. When the API is running, the requests that are sent to it are passed to the play scraper. The API returns the responses. Thus, the API can get used with the play scraper as a microservice with the orchestration.

²<https://github.com/facundooolano/google-play-scraper/pull/384>

```
1 g.search({term: 'panda', num:1, fullDetail: true })
2   .then(console.log);
3 [ {
4   title: 'Panda Pop! Bubble Shooter Saga | Blast Bubbles',
5   description: 'Enjoy playing free bubble shooters? \r\n\r\nGet
6     ready to pop bubbles and beat ... ',
7   descriptionHTML: 'Enjoy playing free bubble shooters? <br><br>
8     Get ready to pop bubbles and beat ... ',
9   summary: 'Aim, match, shoot and pop in this free bubble shooter
10     classic saga adventure!',
11   installs: '50,000,000+',
12   minInstalls: 50000000,
13   score: 4.6056385,
14   scoreText: '4.6',
15   ratings: 1069607,
16   reviews: 319152,
17   ...
69   released: 'Jan 9, 2014',
70   updated: 1581534902000,
71   version: '8.7.100',
72   recentChanges: '- Pop on with more new levels<br>- NEW
73     Collection Babies to collect!<br>- NEW Event Badge
74     Organization<br>- Valentines Day Theming',
75   ...
80   editorsChoice: false,
81   appId: 'com.sgn.pandapop.gp',
82   url: 'https://play.google.com/store/apps/details?id=com.sgn.
83     pandapop.gp&hl=en&gl=us'
84 } ]
```

Listing 5.4: An excerpt of an example search using the Google Play Scraper with working 'fullDetail' option. Everything which was in Listing 5.1 is included here but may be left out in this excerpt.

5.2 Workflows and Microservices

For using Zeebe, a docker-compose file is offered in a git repository³ [17]. Using this, all docker containers that should get started can be configured in a YAML file. After having the Docker container started, workflows can be modeled using a BPMN 2.0 modeler. This BPMN 2.0 modeler is exporting the workflows in XML which can be executed by Zeebe.

For sending the workflows to Zeebe, a program needs to get implemented. This program contacts the Zeebe engine and hands over the workflow and all needed

³<https://github.com/zeebe-io/zeebe-docker-compose>

methods. For implementing this program and all functions, the programming language Go will get used. When using Zeebe first, a BPMN 2.0 workflow needs to get created. Once the workflow is written, this is getting implemented using the Zeebe client, where it gets added. For implementing functionality and adding microservices, for each BPMN service task, a worker can get created. This worker can get programmed using the client. Thus, for each service task, microservices are getting created which fulfill their job. Each microservice individually can get scaled when it is needed.

Workflows for Zeebe are, as already mentioned, defined using BPMN 2.0. For defining those workflows, Zeebe offers a BPMN modeler called Zeebe Modeler which is based on BPMN.io. Using this modeler workflows can be defined in a graphical editor that exports the workflow in XML. Having the workflow implemented it can then be imported to the Zeebe client where all service tasks can get defined. The first defined workflow allows getting data from the Google Play Scraper by using its API. Therefore, the scraper does not have to get used as a microservice but is getting queried by the implemented microservices.

5.2.1 Implementing the Play Scraper

The first workflow will get app metadata from the play scraper API and hands this data in a wanted format to the database. As a first step, a workflow needs to get defined. The workflow should only search for app metadata with given commands and then show the results. This workflow is shown in Figure 5.1.

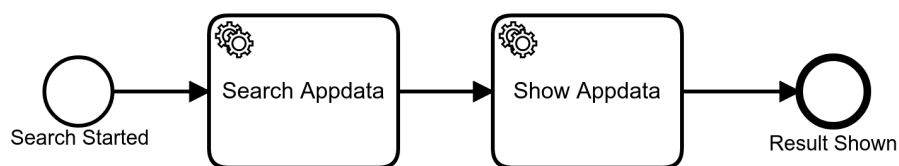


Figure 5.1: Here the BPMN workflow which is getting used in Zeebe for searching for app metadata using the play scraper is shown.

Once having a workflow defined, its service tasks need to get implemented. To do this, first, the play scrapers API needs to get understood. The API can get used only with GET requests. Therefore different options are getting separated using an

"&." An example request is shown in Listing 5.5. Other examples without linkage were presented in the API's git repository readme⁴.

```
GET /api/apps/?q=tinnitus&num=1&fullDetail=true&lang=de
```

Listing 5.5: An example call to get the first app which is found using the keyword 'tinnitus.' The result should be presented in the german language.

For implementing this workflow, first, the microservice *Search Appdata* is getting defined using the Zeebe client implementation. The service task of this microservice first tries to get the data that is necessary to call the API. In case this is successful, the API is called using a web request. The returning data is saved to a map that is getting used to hand data from one microservice to the next. Saving the data into a map makes it possible to call the values by a given name and add additional information later. In error cases, a method gets called, which sets the job to failed and reduces the retries if the service has them by one. The implementation can be seen in Listing 5.6. For later usage, this implementation must be revised for making it possible not always to set each of the variables and allow setting additional ones. In the revised implementation, a user should be able to set these variables in an interface. At first, in this service implementation, the job key and the handed variables are saved into a local variable for later usage. When using Go, the second return parameter here is for the error, which is set in error cases.

Therefore, if the error is not nil, the job fails, meaning that the retry counter will get lowered by one, and the method returns. Next, the search terms are getting used for creating a string that gets used for the following API call. Here, it is essential that using an app id for using the play scraper. The search term needs to get created differently than when using a search term. Additionally, using an app id, the returning app metadata cannot get adapted using further search terms. In other cases, all search terms are getting added to the mentioned string, which is then used to call the API. When there is no error during this method, the response of the API is saved to hand it back to the handler. At the end, the job gets marked as finished.

The second microservice in the workflow is just printing in the first microservice collected data to the console. This service task was implemented to allow testing the handling of data from one microservice to the next. Additionally, all examples in the official tutorial just were using one microservice. Therefore, it was necessary to find

⁴<https://github.com/facundooolano/google-play-api/blob/master/README.md>

out how to implement more than one microservice. This issue could get solved by adding jobs to the job worker.

```
1 func SearchScraperService(client worker.JobClient, job entities.  
   Job) {  
2   jobKey := job.GetKey()  
3   //get variables from job  
4   variables, err := job.GetVariablesAsMap()  
5   if err != nil { ... }  
8   //build search string  
9   builder := ""  
10  ...  
20  if variables["app_id"] != nil{  
21    builder = variables["app_id"].(string)  
22  }else {  
23    for term, content := range variables {  
24      builder += "&" + term + "=" + content.(string)  
25    }  
26  }  
  
28  //use play scraper API  
29  response, err := http.Get("http://localhost:3000/api/apps/?q=" +  
    variables["search-term"].(string) +  
30  "&num=" + variables["number"].(string) + "&fullDetail=" +  
    variables["full-detail"].(string))  
31  if err != nil { ... }  
  
39  //get results of response  
40  data, _ := ioutil.ReadAll(response.Body)  
41  variables["search-result"] = string(data)  
42  //return results to job  
43  request, err := client.NewCompleteJobCommand().JobKey(jobKey).  
    VariablesFromMap(variables)  
44  if err != nil { ... }  
  
51  log.Println("Complete job", jobKey, "of type", job.Type)  
52  ctx := context.Background()  
53  request.Send(ctx)  
54 }
```

Listing 5.6: An excerpt of the implementation of the first microservices service task which uses the play scraper API (written in Go)

5.2.2 Implementing the Persistence Layer

Next, the data needs to get saved to a database, which allows further usage of the data. For saving the data, the first workflow gets revised. The adapted workflow is shown in Figure 5.2. To address the in Section 5.2.1 mentioned issue and allow treating results which are not limited to a single app, an intermediate event got added. This event waits for a message containing the selection of one of the apps which got returned by the first service task. Second, the service task which printed the results got removed. Last, a service task which saves the results to a database was added.

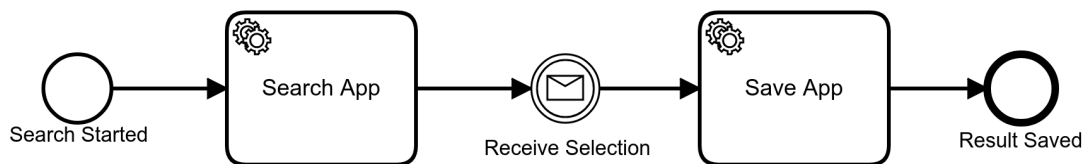


Figure 5.2: The workflow shown in Figure 5.1 got adapted. Adapting the workflow allows choosing which apps data should get saved. Therefore, an intermediate event and a new service task got added. Additionally, the service task for printing the app metadata got removed.

For being able to implement the persistence layer, several issues need to get solved. First, it had to get understood how to use the database using the Go Zeebe client. Second, the data which is getting handed over from the scraping microservice needs to get treated in a way that it can get saved to the database. As mentioned earlier in Section 4.2 it was already known that for communication with the database, an API will get used. However, the API of CouchDB is more comprehensive than the API of the already used play scraper. Thus, more things can be done using the API of CouchDB. Besides, the API makes use of different HTTP methods, e.g., for sending data to the database for saving it, the HTTP PUT method is getting used. For understanding the API, the CouchDB documentation⁵ was very helpful. It is essential to demand authentication for the database to prevent allowing everyone to save and get the data from the database. For authenticating to the database, a POST request is getting used to sending the credentials. In return, a session cookie is passed back. This session cookie can then get used to authenticate the following requests. For being able to use the cookie, it was essential always to use

⁵<https://docs.couchdb.org/en/stable/>

the same client for all requests and additionally add a cookie jar to the client. A cookie jar is saving all cookies so that they can get reused. Thus, using this cookie jar, the authentication cookie can get saved and then send with every request to the database API from there. When requesting an authentication cookie, the rights for the current session are getting set. Thus, it is essential for later, that not every user is allowed to add data or even delete the whole database.

For saving the play scraper APIs data to the database, it is very convenient that both APIs are using JSON formatted data to communicate. Nevertheless, the data which is getting sent by the play scraper API is still not in a format that should get saved to the database directly. The data from the play scraper can only get used directly when an app is getting fetched using the unique id. In all other cases, a JSON results object is getting returned in which all apps that are returned are included. This structure is also used in cases where only one app was fetched. Therefore, for saving each app the same way, nevertheless, it gets fetched by app id or using a search, the format was getting changed in the Zeebe client.

For implementing the service which saves the data to the database, an additional service task gets used. The implementation is shown in Listing 5.7.

At the beginning of the implementation, the results of all earlier tasks are getting saved into a local variable. Then this data is getting unmarshaled into JSON. After having the data converted to the wanted format, it is getting checked if data of more than one app was handed over. In this case, the selected app gets used, where the selection comes from the earlier mentioned intermediate message event. When there is just one app, this is getting taken. Next, a database client is getting created and authenticated at the database using the *createClient* function, which is getting shown in Listing 5.8.

Then the data from before is getting saved to the database using the *myRequest* function, which can get found in Listing 5.9. To make this possible, the earlier created client, the id, which is used to saving the app. Then, the app metadata gets handed over to this function. Last, the service task is getting ended.

When creating a client for the database, first, a cookie jar needs to get created. As mentioned earlier, a cookie jar allows to save the cookies that are getting returned by requests. Thus, a cookie jar allows using the authentication of an earlier request for all following requests. Next, a HTTP client gets created, to allow sending requests to the database. Then, the client is getting authenticated to the database using the *login* function which is shown in Listing 5.10.


```

1 func SaveService(client worker.JobClient, job entities.Job) {
2     ...
10    //get results from scraper
11    data := variables["search-result"].(string)
12    var dat map[string]interface{}
13    if err := json.Unmarshal([]byte(data), &dat); err != nil { ... }
23    //prepare variables
24    timestamp := time.Now().UnixNano() / 1000000
25    var appId string
26    var appData map[string]interface{}

28    //if there is more than one app
29    if dat["results"] != nil {
30        results := dat["results"].([]interface{})
31        appData = results[int(variables["appInList"].(float64))].(map[
            string]interface {}
32        ...
34    } else {
35        appData = dat
36        ...
38    }
39    //create authenticated Client with cookie jar for database
        interactions
40    DbClient, err := createClient(variables["database-addr"].(string
        ))
41    if err != nil { ... }
47    //send database request
48    result, err := dbRequest(DbClient, http.MethodPut, variables["
        database-addr"].(string) + variables["database-name"].(
            string) + appId, appData)
49    if err != nil { ... }

54    //return variables and end job
55    ...
71 }

```

Listing 5.7: An excerpt of the service tasks implementation that saves data of selected apps to the database.

Last, in case the login function is not returning an error, the client is getting returned with nil errors. The login function gets the client handed in, which is getting used to calling the database. Then the login data is saved in a map format that can get transformed later into JSON data, which is necessary to send it to the database. After having the login data saved into the wanted format, an HTTP POST request is called using the function shown in Listing 5.9. The data and error which is returned from the request function are returned by this login function.

5 Implementation

```
1 func createClient(databaseAddr string) (*http.Client, error) {
2     //create cookieJar
3     cookieJar, err := cookiejar.New(nil)
4     if err != nil { ... }
5
6     //create a client for CouchDB using the cookieJar
7     CouchClient := &http.Client{
8         Jar: cookieJar,
9     }
10    if err := login(CouchClient, databaseAddr); err != nil { ... }
11
12    return CouchClient, nil
13 }
```

Listing 5.8: This function is used to create a database client which is necessary for calling the database.

```
1 func dbRequest(client *http.Client, method string, url string,
2     values map[string]interface{}) (map[string]interface{}, error)
3 {
4     data, _ := json.Marshal(values)
5     //create http request, using the handed values
6     req, err := http.NewRequest(method, url, bytes.NewBuffer(data))
7     if err != nil { ... }
8     //set HTTP content type
9     req.Header.Set("Content-Type", "application/json")
10
11    //send request
12    resp, err := client.Do(req)
13    if err != nil { ... }
14    //read response
15    body, _ := ioutil.ReadAll(resp.Body)
16    //pack response status header and body to a map
17    re := map[string]interface{}{"Status": resp.Status, "Headers":
18        resp.Header, "Body": string(body)}
19
20    return re, nil
21 }
```

Listing 5.9: The database request function which creates a request in the format which is needed to use the database API.

The request function translates the handed data from a map structure to JSON. Then this JSON data is used for creating an HTTP request. After adding the content type in the header to let the database know that the data JSON formatted, the database is getting called. In error cases, the error gets returned to the calling

```
1 func login(CouchClient *http.Client) (map[string]interface{},  
    error){  
2     username := "*****"  
3     password := "*****"  
4     values := map[string]interface{}{"username": username, "password"  
        ": password}  
  
6     return myRequest(CouchClient, http.MethodPost, "http://localhost  
        :5984/_session", values)  
7 }
```

Listing 5.10: The login function which authenticates to the Database. Being authenticated allows reading and writing data of the database.

method. Finally, the data which is getting returned by the HTTP request gets saved to a map, which is then getting returned with nil errors. Doing this all returned values are accessible by name while having them stored together in one value.

After having all this implemented, the workflow can get used to get data from the play scraper and then save the results to CouchDB. Saving the data to the database allows working on the data and add further information, e.g., reviews or metadata analysis to it.

5.2.3 Implementing Database Search

Now, being able to search and save app metadata using the play scraper API, an important next step is to find data that was already saved to the database. Being able to find this data is essential for several aspects. First, local searches are essential for users that are not allowed to add apps to the database to see which apps are already added for each keyword. Additionally, being able to search for apps allows further steps which are planned, e.g., updating apps in the database and analyzing the metadata.

A problem that comes up when implementing a search is that CouchDB does not come with a full-text search. For implementing a full-text search, an additional indexer is needed to get this. Another option to solve the issue would be to work with views which allow searching for the items of the view.

However, at this point, it is not important how this issue is getting solved since the focus will be first on implementing the search for app ids and exact matches to other

fields. Additionally, it is possible to use lower equals or higher equals method. This search method allows searching, e.g., for all apps which got downloaded more than 100 000 times or only got apps that have a rating higher than three of possible five.

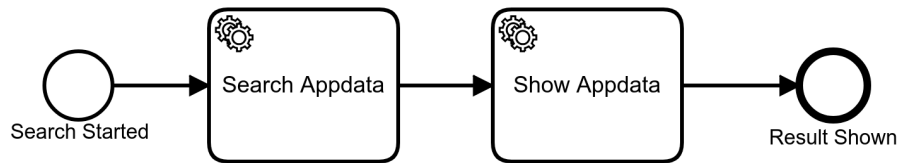


Figure 5.3: Here, the BPMN workflow, which is getting used in Zeebe for searching the database for app metadata, is shown. The first service task is getting the data from the database. The second one is showing the results.

For implementing the search task, a new workflow was created, which is shown in Figure 5.3. The workflow looks the same as the workflow shown earlier in Figure 5.1 since both are searching data and showing the results. However, there is a difference between the implementation of those two workflows. The differences are due to the now implemented workflows service tasks are searching for the data in the local database and is not using the play scraper. Thus, the implementation of this service task is completely different. The implementation of the service task for database search is shown in Listing 5.11.

In the implementation, first, the database connection is created. How the connection is implemented is not shown, since it was already shown earlier. Then the selector, which is getting used to decide which data will get selected from the database, is getting saved from the by Zeebe-client. Then the data handed over a variable to a local one. With this, the variable is saved in JSON format, which allows, as before, to hand it to the earlier written request method. This time the request method is called using the HTTP POST method using the API find method. When no error comes up, the returning values are getting saved and send back to the Zeebe-client.

```
1 func SearchDbService(client worker.JobClient, job entities.Job) {
2     ...
11     //fail if no search selector is given
12     if variables["selector"] == nil{ ... }
18     //create authenticated Client with cookie jar for database
        interactions
19     DbClient, err := createClient(variables["database-addr"].(string)
        ))
20     if err != nil { ... }
25     //prepare database search
26     values := map[string]interface{}{"selector": variables["selector"]
        } // "fields": fields
27     //search database for search-term
28     resp, err := dbRequest(DbClient, http.MethodPost, variables["
        database-addr"].(string) +
29     variables["database-name"].(string) + "_find", values)
30     if err != nil { ... }
36     //save database response
37     log.Println("Database status:", resp["Status"])
39     variables["databaseApps"] = resp["Body"]
40     //return variables and end job
41     ...
57 }
```

Listing 5.11: An excerpt of the service tasks implementation which allows searching the database for a given selector.

5.2.4 Implementing Database Updates

When searching apps in the local database, it is important to check if the apps which are saved in the database are still up-to-date. In cases the app got updates, the database needs to get updated too. Additionally, there could be cases in which an app gets removed from the app store. In this case, it is important to remove the app from the database or mark the app as removed.

For checking if there are updates that should get saved to the database in the app store, it is essential to find out which data changes when an app gets updated. For this, the app metadata which is going to get saved needs to get viewed. This data is getting returned in JSON and shown in Listing 5.12.

The first finding in this data is if there is a new version in the app store, the field

```

1 {
2   "title": "Beltone Tinnitus Calmer",
3   ...
6   "installs": "100.000+",
7   "minInstalls": 100000,
8   "score": 4.5797873,
9   "scoreText": "4,6",
10  "ratings": 1889,
11  ...
12  "histogram": {
13    "1": 40, "2": 50, "3": 60, "4": 361, "5": 1376
14  },
15  ...
22  "androidVersion": "5.0",
23  "androidVersionText": "5.0 and up",
24  ...
46  "released": "Jan 6, 2015",
47  "updated": 1581582234000,
48  "version": "5.2.4",
49  "recentChanges": "Japanese localization",
50  ...
58  "appId": "com.beltone.tinnitus",
59  ...
63 }

```

Listing 5.12: An excerpt of the response using the search workflow described in Section 5.2.1

updated and the field *version*, which are shown when getting an app with full detail. Since an app might get an update without changing the version number, the field *updated* seems to be more promising. The updated field shows a number which is representing the time in milliseconds counting from 1 January 1970, which is often used for timestamps in Unix systems. In the shown example *1581582234000* represents 13 February 2020 09:23:54.

However, even when there is no update to the app, there could be interesting changes, e.g., the ratings or download counts. This fields may not be as important as a new version but changes in ratings could be important when choosing an app in cases more than one app could get used for the same.

When starting the implementation of the updates, two workflows were getting created. One of those workflows is updating a selected apps data; the other is updating all apps data. The workflow for updating a selected apps data is shown in Figure 5.4. The first task in this workflow is searching the database for an entry.

Next, an intermediate event is waiting for a message, including the selection of an app. After having the app selected, the data of this app is getting searched in the app store. As a fourth step, the data is getting compared. For detecting an update was made the earlier mentioned fields in the metadata get used. Thus it can get detected if an update has been made since adding the app or since the last update. In case an update gets detected, the app metadata gets updated. Additionally, the app gets updated in cases when the last update to the database was more than a week ago.

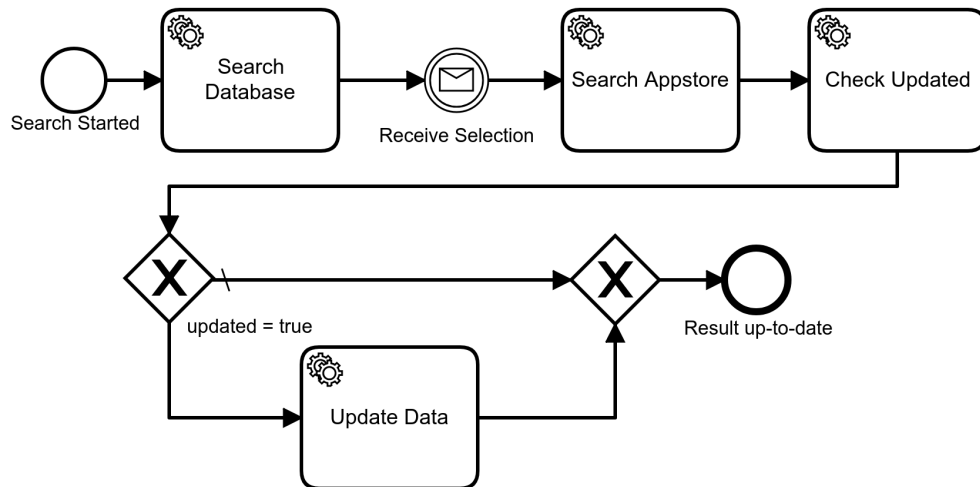


Figure 5.4: The workflow searches in the database. Then one app gets selected, which gets searched app store to detect updates. In case of updates, those are getting saved to the database.

This workflow is a lot more complex than the earlier ones, which results in more effort to implement it. However, some of the service tasks were already implemented for earlier tasks, which allows reusing them and their implementation. However, since those service tasks were not implemented in a way to work together, they need to get overworked. First, they have to get adapted to have their outputs named unambiguously. Second, it is important to hand the variables always into a service task the same way. Therefore, the client needs to implement this, i.e., before calling a service task, it needs to get ensured that all inputs are handed to the service tasks the right way. However, the changes to the service tasks have to get implemented only once. At least when using the same code for each implementation, which is one of the enhancements when using the concept of orchestration.

However, the first service task that was not already created checks if the app which

was got from the database differs from the returns of the app store search. For this, the earlier mentioned updated field gets checked. Additionally, a field which includes the timestamp when adding the app metadata to the database is getting used to test if the last update to the app was done more than a week ago. This process returns whether the app was updated, or the last update was more than a week ago. The implementation of this service task can be seen in Listing 5.13.

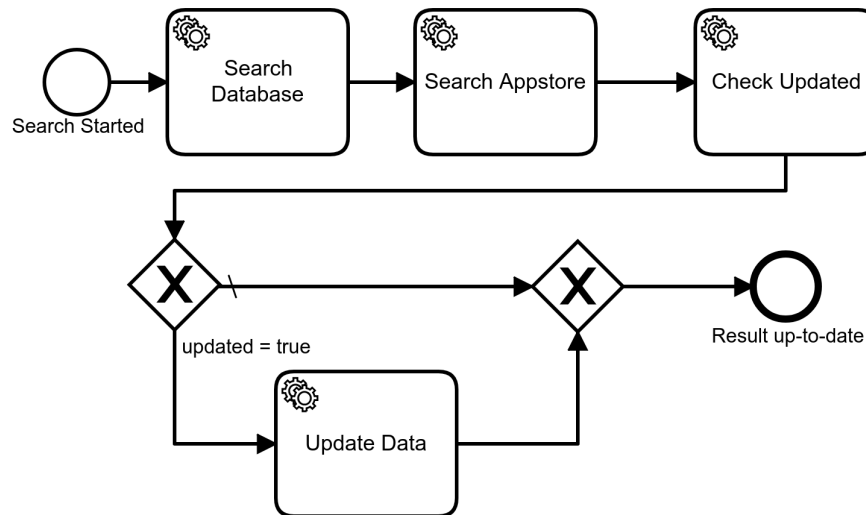


Figure 5.5: The workflow searches in the database and app store to detect updates. In case of updates, those are getting saved to the database.

The second workflow for updating all apps, under the earlier mentioned conditions, is shown in Figure 5.5. In contrast to the workflow, which is used to update a single app, here, the selection of an app is not necessary. Therefore, the intermediate event is left out. Even when this does not make much difference in the workflow, it is important when implementing to have both workflows in mind. To have both workflows is important because treating single apps data is a lot different than a list of apps in many cases.

In case of changed the xor path to the service task to update, the data gets used. The first idea was to update the data in the database. However, when doing this, the original data is not always accessible. Even when allowing to access old versions to solve conflicts, the original data is not accessible on replications. Additionally, when using compaction, all old versions are lost. Therefore, all data that is not getting analyzed before the differences cannot get analyzed at any later point in time. Thus, the solution is to save the app metadata every time in total. When saving the


```
1 func CheckUpdatedService(client worker.JobClient, job entities.Job
   ){
   ...
10  //if one of both variables is not set, fail and retry
11  if variables["updated_Db"] == nil || variables["updated_Scraper"
   ] == nil{
12      failJob(client, job)
13      return
14  }

16  //if variables differ, the app was updated
17  if variables["updated_Db"] != variables["updated_Scraper"] {
18      variables["updated"] = true
19  }

21  request, err := client.NewCompleteJobCommand().JobKey(jobKey).
      VariablesFromMap(variables)
   ...
34 }
```

Listing 5.13: The service task implementation which checks if the app got updated.

metadata of each version in total, it is important to know that all apps need a different database id, which is not the case in the earlier implementation. For solving this issue, when adding the app to the database first, nothing gets changed, and the app id gets used as the database id. When updating, the current timestamp gets added to the database id. The resulting service task implementation is shown in Listing 5.14.

In the shown function, first, the data which was got by from the database and scraper is getting saved to local variables. After that, the app id and the revision id are taken from the database data. Both ids are used for enriching the data, which was obtained from the scraper since this is the more current data. For being able to address the data, which was got from the database and scraper, it needs to get converted into JSON. Adding the ids to the scrapers data is necessary to allow updates, as mentioned earlier. Additionally, a timestamp gets added to allow analyzing when the data got updated last. Then an HTTP client is getting created and authenticated with the database. Finally, the enriched data is getting saved to the database.

After having the database updated, the saved results get shown using the already known service task to do so.

```
1 func UpdateDbService(client worker.JobClient, job entities.Job){
2     //get variables from job
3     ...
4
5     //get results from scraper
6     data := variables["search-result"].(string)
7
8     var appData map[string]interface{}
9     if err := json.Unmarshal([]byte(data), &appData); err != nil {
10         ... }
11
12     //timestamp in milliseconds
13     timestamp := time.Now().UnixNano() / 1000000
14     appId := appData["appId"].(string)
15
16     appData["addedToDb"] = timestamp
17
18     //create authenticated Client with cookie jar for database
19     //interactions
20     DbClient, err := createDbClient(variables["database-addr"].(
21         string))
22     if err != nil{ ... }
23
24     result, err := dbRequest(DbClient, http.MethodPut, variables["
25         database-addr"].(string) +
26         variables["database-name"].(string) + appId + "-" + strconv.
27         FormatInt(timestamp, 10), appData)
28     if err != nil{ ... }
29
30     //save variables and end job
31     ...
32 }
33
34 }
```

Listing 5.14: An excerpt of the service tasks implementation for the database update.

5.2.5 Implementing Ratings

One aspect which got mentioned in the concept is the possibility to add expert ratings to the app metadata in the database. Since the expert is always reviewing one version of the app at a certain point in time, the review should include the version and timestamp of the reviewed app.

A hard decision was how to save the ratings. The first idea was adding the review data to the app metadata. When adding the data to the app directly, it would always be clear and easy to see which app version got rated. However, doing this has some drawbacks. When only the data of the review is needed, the app metadata needs

to get queried too. A second idea was attaching the rating to the app metadata. Attaching the data as a document has the advantage that a document does not need to get updated to add a rating. However, the problem of querying is the same as when adding the data to the document itself. The third idea was saving the ratings to their database. Saving the ratings to another database allows querying the data directly or using views without looking at the app metadata. Therefore, the problem using an own database is that the linking between rating and a specific app and timestamp of this is not given. For creating such a link, the app id and timestamp needs to get added to the rating data. Since the problems were best solved with the third idea, this was implemented.

The workflow which gets used here is very simple, it is getting the rating data and saved this to the database. The workflow is shown in Figure 5.6. Using simple workflows having just one service task, is a result of using a workflow for querying the database and returning the results. This is done, to allow to scale the service by only adding more instances of the same microservice without the necessity having to change the implementation.



Figure 5.6: The workflow which is saving the Rating to database.

The implementation of this workflows service task can get seen in Listing 5.15. In the shown function first, the handed variables were obtained from the job, and a database client is getting created. Then, the rating data is obtained from variables. This data needs to get unmarshalled to JSON to allow adding further data. As mentioned before, the app id is saved to allow linking the rating to an app. Additionally, a current timestamp is getting added to allow ordering the ratings. When having all data set up, the rating is saved to the database using the already known *dbRequest* function.

The data of the rating will get created using the earlier mentioned MARS questionnaire. This survey will get implemented in JavaScript that will get served to the user in the UI. The data of this survey is getting handed over in JSON directly. Therefore, no further implementation is necessary.

```

1 func SaveSurveyService(client worker.JobClient, job entities.Job){
2     //get variables
3     ...
4     //create authenticated Client with cookie jar for database
        interactions
5     DbClient, err := createDbClient(variables["database-addr"].(
        string))
6     if err != nil{ ... }
10    //get rating/survey data and unmarshall to string
11    var surveyData map[string]interface{}
12    if err := json.Unmarshal([]byte(variables["surveyData"].(string)
        ), &surveyData); err != nil { ... }
16    //get variables needed to save rating to db
17    databaseId := variables["databaseId"].(string)
18    appId := strings.Split(databaseId, "-")[0]
19    //get timestamp in milliseconds
20    timestamp := time.Now().UnixNano() / 1000000
21    //add timestamp and app id to rating data
22    surveyData["addedToDb"] = timestamp
23    surveyData["appId"] = appId
24    surveyData["rated"] = databaseId
25    //save data to database
26    result, err := dbRequest(DbClient, http.MethodPut, variables["
        database-addr"].(string) +
27    variables["database-name"].(string) + "-" + timestamp,
        surveyData)
28    if err != nil{ ... }
32    //save results to variables and end job}
36    ...
    }

```

Listing 5.15: An excerpt of the service tasks implementation which saves the ratings to database.

5.2.6 Implementing Metadata Analysis

The most critical and final step is to implement the analysis of metadata of the saved apps. However, it is essential to implement the analysis last, since now there are apps and updates of those saved in the database now. When having the metadata retrieved from the database, it is possible to test the analysis on this metadata and add more data using the earlier implemented workflows.

Since the updates are not saved to one document, it is necessary to get the differences from different documents but group it for the same app to prevent getting data mixed from different apps. For querying all updates, CouchDB views are get-

ting used. As explained in Section 4.2.1, views allow searching and aggregating data from different documents. Additionally, to allow grouping the results, reduce functions that need to get added. Therefore, the central part of analyzing differences is getting written in JavaScript which is used for defining those views. However, using a view the data is not getting analyzed but it is possible to get the wanted information that is necessary to perform the analysis from all apps at the same time. An implemented view is shown in Listing 5.16.

```
1 function (doc) {  
2   emit(doc.appId, {'minInstalls': doc.minInstalls, 'updated': doc.  
   updated, 'version': doc.version, 'addedToDb': doc.addedToDb,  
   'score': doc.score});  
3 }  
  
5 function(keys, values, rereduce) {  
6   if (rereduce) {  
7     return values.reduce(function(a, b) {  
8       return [].concat(a, b);  
9     }, [])  
10  } else {  
11    return values;  
12  }  
13 }
```

Listing 5.16: 'The implemented view which returns the emitted data of each app grouped by app id. This is getting used for analyzing changes between different versions.'

In the top, the map function gets presented. In the map function of a view, it gets defined which data is getting emitted when the view gets called. In views always a key, value pair gets emitted. Therefore, the app id is the key, and a map of the other elements is the value. Below the map function, a reduce function is shown. The reduce function gets used when a view gets called with grouping. When the data for analysis of all saved apps should get grouped by the app id, this is wanted. Therefore, a so-called reduce function got implemented. For understanding a reduce function, it is important to know that views are getting represented as a b-tree. This tree gets reduced using the reduce function. In cases that reducing the tree, the first time is not enough to get the result re-reduce is set to true. In the shown reduce function, the values are getting returned in the not re-reducing case. In the case of re-reducing, the values are getting concatenated in an array of all emitted values. The data is concatenated until there is just one element or, in case of group-

ing, one element per group left. The resulting data in an array of all selected fields are wanted for analyzing the changes of the app version for all apps grouped by app id if grouping is used. Additionally, all changes of a single app can get queried using this view when using grouping and setting an app id as a map key.

The workflow for retrieving the analysis data is shown in Figure 5.7. The workflow has only one service task. This is due to the idea of using the view to get the data directly. Therefore, this data has only get retrieved in one task. The implementation



Figure 5.7: The workflow which is used to get the data that gets used for analysis.

of the service task is shown in Listing 5.17. In this code, first, the variables are getting retrieved from the job. Next, a database client is getting prepared for usage. In case there was no error, the view shown in Listing 5.16 gets queried by enabling grouping and using the app id as a key. Here it was imperative that the app id, which is used as the key, needed to get send in quotation marks. For adding those to a string, they need to get escaped. When using the grouping and key, only the elements having the app id as key are getting returned. The data that is getting returned from the view is saved to the variables which are getting returned to the job at the end. The implementation of the analysis is mainly will get implemented in the view directly using JavaScript. The other parts will get implemented in the functionality of the view.

5.3 User Interface

This user functionality of the UI is written in Go. As described earlier, this allows using the Go Zeebe client. Using Go allows writing the visible parts of the UI in as templates in HTML. Thus, the UI can get accessed using a web browser.

When implementing the UI functionality, it is essential for the concept, that – as before – every data that is getting saved or retrieved using Zeebe workflows. Doing

```

1 func GetAnalysisDataService(client worker.JobClient, job entities.
   Job){
2   jobKey, variables := getVariables(client, job)
3   //get appId from variables
4   appId := variables["appId"].(string)
5   //create database client
6   DbClient, err := createDbClient(variables["database-addr"].(
   string))
7   if err != nil { ... }
12  //get id of the newest version of each app in db from view
13  resp, err := dbRequest(DbClient, http.MethodGet, variables["
   database-addr"].(string) + variables["database-name"].(
   string) + "_design/" + variables["doc"].(string) + "/_view/"
   + variables["view"].(string) + "?group=true&key=\""+ appId +
   "\"", nil)
14  if err != nil { ... }
19  //unmarshall results
20  var dat map[string][]map[string]interface{}
21  if err := json.Unmarshal([]byte(resp["Body"].(string)), &dat);
   err != nil { ... }
26  //add only data to return if there is something returned from
   the database
27  if len(dat["rows"]) > 0 {
28    variables["analysisData"] = dat["rows"][0]["value"].([]
   interface{})
29  }
30  //end job and return variables
31  ...
33 }

```

Listing 5.17: The implementation of the service task used for retrieving the data for analysis from the database.

this allows using multiple workers for the same service task which creates, e.g., the possibility of using different servers for scraping the app store.

When starting to create a web UI, first, a web server is needed. However, using Go, this can get done by implementing only a few lines of code. The code for implementing the web server and add the main page are shown in Listing 5.18.

5.3.1 User Login

In order to restrict the use of the service to registered users, a login page has been implemented. To be able to use the rest of the service a login on this page is nec-

```
1 func main() {
2     //call Index function when accessing /
3     http.HandleFunc("/", Index)
4     //open web server on port 8080
5     if err := http.ListenAndServe(":8080", nil); err != nil {
6         fmt.Println(err)
7     }
8 }
9
10 func Index(w http.ResponseWriter, r *http.Request) {
11     //parse index.html when accessing /
12     tmpl := template.Must(template.ParseFiles("html/index.html"))
13     if err := tmpl.Execute(w, nil); err != nil {
14         fmt.Println(err)
15     }
16 }
```

Listing 5.18: The implementation of a simple webserver which offers an index HTML page using templates.

essary. The login page is shown in Figure 5.8. When a user uses the service for the first time, he can register for the service using the link at the top right of the navigation bar.

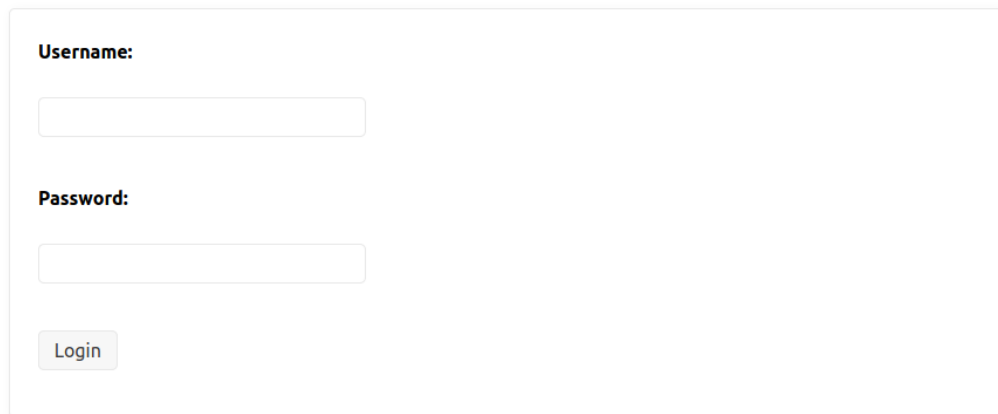
For the service, three different roles of users are implemented: user, operator, and administrator. The user can look at saved apps, update and rate those. The operator can also search for new apps and save them to the database. In addition, an administrator can manage the registered users. When being logged in, for each user only the permitted applications are getting shown in the navigation. However, only hard-coded user credentials are implemented. Where this is enough to show the different roles a user can have, for using the system this should get changed to credentials that are getting encrypted and loaded from the database.

5.3.2 Searching and Adding Apps to Database

The functionality to search for app metadata using the play scraper API was the first functionality that was implemented in the UI. For this, the workflow is shown in Figure 5.2 was used. Using the Go Zeebe client, a problem was that accessing the values of a workflow was not possible. However, it was possible accessing the vari-

[Home](#)[Register](#)

Enter your Credentials



The login form is a light gray rectangular box. Inside, the label 'Username:' is followed by a text input field. Below that, the label 'Password:' is followed by another text input field. At the bottom left of the form is a 'Login' button.

Figure 5.8: The login page.

ables at the end of a workflow. Thus, the workflow was divided into two processes. Additionally, to the workflows, two HTML pages were created first to allow entering search terms and then saving results.

Dividing the workflow has several advantages. The necessity to wait for a message from the user which app should get saved. Thus, it is not a problem when users do abort the process without sending the message. Another important aspect is that creating smaller workflows allows balancing the workloads better to the services.

When a new app should get saved to the database, first, an HTML page gets loaded, which is shown in Figure 5.9. Here the user can enter a search term and the number of results that should get retrieved. When clicking on 'Submit Query,' a workflow gets started to get the wanted information.

The UI function implementation for searching the play scraper is shown in Listing 5.19. In this code, the search UI is getting executed, if the HTTP call handed to the function was not a POST request. When submitting a search, the data which got entered into the fields is getting handed in a POST request to this method. Therefore, those handed values are getting saved into a map, which then gets handed to a function creating a new workflow instance for calling the play scraper with this data.

Search for new Apps

Search Term:

Tinnitus

Num:

4

Submit Query

Figure 5.9: Using the UI to search the Play Store to find Apps.

```

1 func Search(w http.ResponseWriter, r *http.Request) {
2     //prepare html file, if it cannot get found stop with error
3     tmpl := template.Must(template.ParseFiles("html/search.html"))
4     //if the request to page was not POST send search page
5     if r.Method != http.MethodPost {
6         if err := tmpl.Execute(w, nil); err != nil { ... }
7         return
8     }
9     //preventing error cases
10    ...
11    //create map of search terms
12    details := SearchTerms { searchTerm: r.FormValue("searchTerm"),
13                             num: r.FormValue("num") }
14    //use play scraper to get the app metadata
15    res := startScraper(details.searchTerm, details.num)
16    //get the data from the before returned JSON
17    app := getValues((*res).Variables)
18    //parse results page
19    t, _ := template.ParseFiles("html/results.html")
20    //execute new template while handing over the app metadata
21    if err := t.Execute(w, struct { Success bool; Application []App
22                                   }{true, app}); err != nil { ... }
23    ...
24 }

```

Listing 5.19: An excerpt of the implementation used for searching apps to show the results in the UI.

The returned apps are in a JSON string, which needs to get unmarshalled to get the variables of the apps that is necessary to show the search results. When this is done, the results are handed to a template which is serving the results page. The results in the UI are shown in Figure 5.10.

Search Results

AppId	Name	Platform	Rating	
com.beltone.tinnitus	Beltone Tinnitus Calmer	Android	4.72	submit
com.gnresound.tinnitus	ReSound Tinnitus Relief	Android	4.6	submit
com.music.tinnitusoundtherapy	Tinnitus relief app. Sound therapy.	Android	4.41	submit
nl.appyhapps.tinnitusmassage	Tonal Tinnitus Therapy	Android	3.93	submit

Figure 5.10: The results of a search are getting shown in the UI.

The workflow that was used for searching the Play Store can be seen in Figure 5.11. It is the shortest workflow possible since it only has one service task. When starting the workflow, the 'Search App' service task is searching the Play Store using the search terms that a user enters in the UI. The implementation of this workflow was already shown and explained earlier in Section 5.2.1. This workflow could get used without changes.

When the workflow is finished, the results are getting shown. A page showing the results is shown in Figure 5.10. Here only the app id, the name, and the current rating are shown. However, this can get adapted easily when more information is needed for deciding which apps data should get added to the database. In the last column on the right-hand side, a submit button is shown. When clicking on the sub-



Figure 5.11: The simplified workflow for retrieving app metadata using the play scraper.

mit button, the app in this line gets saved to the database. For saving, the second workflow gets triggered. This workflow is shown in Figure 5.12.

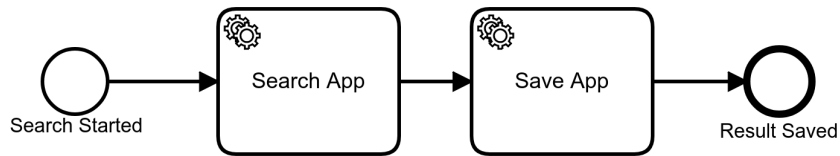


Figure 5.12: The workflow used for saving the app metadata to the database.

The part where the data gets saved to the database is shown in Figure 5.12. In this workflow, the app metadata gets searched with full detail in the first service task, using the Play Scraper. Searching the apps in full detail is necessary due to searching before the apps without full detail. The first search is not done with full detail because this can take much additional time. When having retrieved the app metadata, it is getting saved to the database.

The implementation of both service tasks was already shown before when dealing with the implementation of the play scraper (Section 5.2.1) and the implementation of the database (Section 5.2.2).

When having saved the app metadata, the button in the UI gets disabled, and the shown text in the button changes to *saved*. For this, Asynchronous JavaScript and XML (AJAX) gets used. Using AJAX for this task allows changing the button without having to reload the whole site. Furthermore, using AJAX allows saving all of the shown apps by clicking on the corresponding submit button. The UI, when having saved the data of one of the apps to the database, is shown in Figure 5.13.

Here it is visible that the *submit* button of the app metadata, that was saved, is getting disabled, and the label is changed to *saved*. Thus, it is always clear which app was yet saved to the database. Additionally, it is prevented that a submit button gets clicked multiple times. However, when trying to save an app that was already saved to the database, the app gets not saved again to the database. Therefore, in a later version, the submit button for apps that are already saved to the database is already disabled when the results get shown. Additionally, the label will show that the app is saved.

As mentioned, for saving the app to the database, AJAX is getting used. When clicking on the submit button, a POST request is getting performed, sending the app id to another function, which gets shown in Listing 5.20.

Search Results

AppId	Name	Platform	Rating	
com.beltone.tinnitus	Beltone Tinnitus Calmer	Android	4.72	<input type="button" value="saved"/>
com.gnresound.tinnitus	ReSound Tinnitus Relief	Android	4.6	<input type="button" value="submit"/>
com.music.tinnitusoundtherapy	Tinnitus relief app. Sound therapy.	Android	4.41	<input type="button" value="submit"/>
nl.appyhapps.tinnitusmassage	Tonal Tinnitus Therapy	Android	3.93	<input type="button" value="submit"/>

Figure 5.13: When an App gets saved to the database the button to save it gets disabled and its value changes to show that the saving was successful.

```

1 func receiveAjax(w http.ResponseWriter, r *http.Request) {
2     if r.Method == http.MethodPost {
3         //saveId = appId
4         if r.FormValue("saveId") != "" {
5             res := saveAppData(r.FormValue("saveId"))
6             //unmarshall results
7             var docs map[string]interface{}
8             if err := json.Unmarshal([]byte((*res).Variables), &docs);
               err != nil { ... }
9             log.Println("Save App to Db resp:", docs["dbStatus"])
10        }
11        ...
21    }
22    ...
24 }

```

Listing 5.20: An excerpt of the implementation used for saving the app metadata to the database.

Here the handed app id is used when calling a function that starts the workflow to save the app metadata of the selected app to the database. The called function to save the app metadata is shown in Listing 5.21. The function returns the result from Zeebe, which gets used for logging the database status. The database status could get used later to implement treating errors in a way that the UI shows an error in error cases.

```
1 func saveAppData(appId string) **pb.  
    CreateWorkflowInstanceWithResultResponse{  
2 //create Zeebe client to send the workflow  
3 zbClient, err := zbc.NewClient(&zbc.ClientConfig{  
4     GatewayAddress:      BrokerAddr,  
5     UsePlaintextConnection: true})  
6 if err != nil { ... }  
  
9 ctx := context.Background()  
  
11 //prepare necessary variables and prevent errors  
12 variables := make(map[string]interface{})  
13 variables["appId"] = appId  
    ...  
16 //create workflow call  
17 request, err := zbClient.NewCreateInstanceCommand().  
    BPMNProcessId("save-selected-process").LatestVersion().  
    VariablesFromMap(variables)  
18 if err != nil { ... }  
20 //create new workflow instance and return back its results  
21 res, err := request.WithResult().Send(ctx)  
22 if err != nil { ... }  
  
25 return &res  
26 }
```

Listing 5.21: An excerpt of the 'saveAppData' function called in Listing 5.20.

In this method, first, a Zeebe client gets created. Then the necessary variables which should get handed over are getting saved into a map. When creating a new instance of the workflow, the map is handed over to this. After having this done, the workflow is getting sent. When the workflow is finished, its results are getting returned. Last, these returned results are getting returned to the calling function.

5.3.3 Show Saved Apps

A next functionality that is essential for the UI is showing all apps that are currently saved in the database. Implementing this task is more complicated than it sounds in the first place. That saving the data to the database is complicated is due to the way of saving updates to the database. In the database, each update is saved as an own document, which was explained earlier when the updates were implemented

in Section 5.2.4. Thus, each app should get shown only once. For achieving to retrieve each app only once from the database, a view can get utilized. In the implementation of the view, all apps are getting reduced to only the newest version of the app where the app id from the Google Play Store is getting used as an identifier. The view which was used for that is shown in Listing 5.22.

```
1 function (doc) {
2   emit(doc.appId, doc._id);
3 }
4 function (keys, values, rereduce) {
5   if (rereduce) {
6     //return the maximum of both handed in timestamps
7     return values.reduce(function(a, b) { return Math.max(a, b) },
8       -Infinity)
9   } else {
10    return (function() {
11      var timestamp = 0;
12      values.forEach(function (value) {
13        //split database id to get the timestamp
14        var res = value.split("-");
15        //if there is a timestamp
16        if (res.length > 1){
17          //save the maximum timestamp
18          timestamp = Math.max(timestamp, res[1])
19        }
20      });
21      return timestamp;
22    })()
23  }
24 }
```

Listing 5.22: The implementation of the view which is returning the newest database id for each app id.

The map function of this view is emitting only the app id and the documents database id. The database id consists of the app id followed by a timestamp. Therefore, the database id can be divided into two parts, of which it consists using a split function. Splitting the database id allows using the reduce function to return the maximum timestamp. Thus, the view is returning the app id as the key and the newest timestamp as value. Knowing both allows querying the newest app metadata for each app from the database.

After having the view implemented, a workflow got created to get this data. The workflow is shown in Figure 5.14.

This workflow has only one service task. However, when creating a prototype, not everything possible is implemented, which leads to simple workflows. Since showing all apps newest version is an issue that was not solved before, the service task needed to get implemented. The implementation can get seen in Listing 5.23.

The service task first gets the variables from its job. When it uses the DB request method to get the data from the earlier shown view. After having this done, for each of the returned apps, the app metadata is got from the database, using the returned app id. For returning the app metadata, it is saved in a map that allows returning all wanted data in one object.

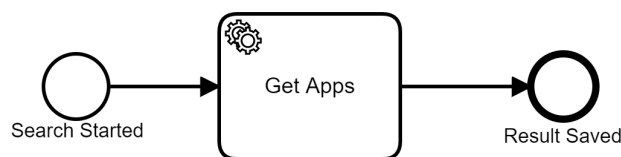


Figure 5.14: The workflow to get the newest versions metadata of each app.

Additionally, using a map for saving the app metadata, the app id is the key to accessing the values. Thus, the app id is handed back without having to save it into another value. The app metadata gets used to show the newest version of all saved apps and some of its data on an overview page, which is shown in Figure 5.15. Here the title, platform, current rating, and when its last updated in the database gets presented for each app that is saved in the database. The shown timestamp that shows when the app was last updated is showing the date and time which was especially important for testing the UI. However, if showing the time is necessary depends on the time after which another update can get saved. Additionally, a *more information* button leads to a UI page that is presenting more details about this app. When clicking on the *more information* button, the information of the chosen app gets retrieved using another workflow, which gets displayed in Figure 5.16. This workflow gets the database id handed in and queries the corresponding app metadata from the database.

The UI for showing more information on the app metadata is shown in Figure 5.17. Here, at the top, the header image of the app is displayed. Directly thereunder, the title of the app gets presented. Below that, a summary describing what the app is made for is shown. When clicking on this summary, the full app description gets


```

1 func GetAllAppsNewestVersions(client worker.JobClient, job
    entities.Job){
2     //get variables and create database client
    ...
5     //get id of the newest version of each app in db from view
6     resp, err := dbRequest(DbClient, http.MethodGet, variables["
        database-addr"].(string) + variables["database-name"].(
            string) + "_design/appdata/_view/latest?group=true", nil)
7     if err != nil { ... }
    ...
26    apps := make(map[string]interface{})
27    //get all newest apps data
28    for elem := range data["rows"] {
29        //get app metadata from database
        ...
52        apps[appId] = appData
53    }
54    //save resulting apps to variables
55    variables["apps"] = apps
57    //end job and return variables
    ...
60 }

```

Listing 5.23: An excerpt of the service tasks implementation used to retrieve all apps newest app metadata from database.

Saved Apps

Title	Platform	Rating	Last Updated	
Beltone Tinnitus Calmer	Android	4.72	10:18 June 12, 2020	More Information
ReSound Tinnitus Relief	Android	4.59	10:16 June 11, 2020	More Information
All Mental Disorders and Treatment	Android	4.29	11:23 June 12, 2020	More Information
1010 Active: CRM for Fitness and Physiotherapy	Android	4.17	11:23 June 12, 2020	More Information

Figure 5.15: The UI page showing basic information of the newest version of each saved app.

shown. Since this description is very long for some apps, this is hidden when accessing the page. Below the summary and description, three boxes with additional information on the app are getting displayed. In the left-hand box, general informa-

tion of the app is shown, e.g., who is the developer of the app, when was the app released, what is the current rating of the app. In the central box, it gets displayed when the app was last saved to the database and information to the costs if apps are supported, and the minimum Android version to use the app. Additionally, in this box, an *update* button is shown. However, at this point, it is just a dummy which later will get the functionality to save a current version of the app to the database. The right-hand side box, a rating histogram, is shown. In this histogram, it can get seen at a glance the distribution of the ratings from one star up to five stars.



Figure 5.16: The workflow used for retrieving the chosen apps data from the database.

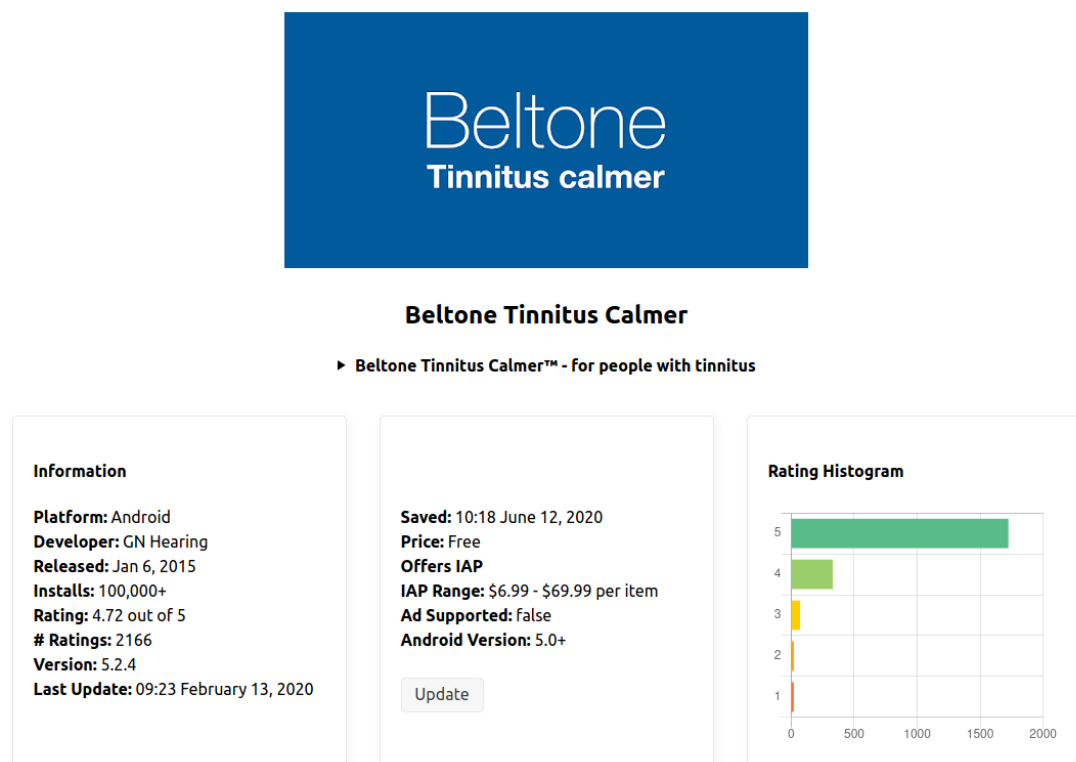


Figure 5.17: The UI app page, showing information which is retrieved from the database.

On this page, further information could get displayed. However, since this is just a prototype, it should get shown how information can get displayed.

5.3.4 Update Apps

For updating the apps, the update button is shown in Figure 5.17 needs to get functionality implemented. For doing this, first, the workflow that was shown earlier in Section 5.2.4 for updating app metadata got overworked. The resulting workflow is shown in Figure 5.18. The main difference to the earlier workflows to update is the addition of the first service task *check version*. In this service task, it gets checked whether the app page the update process gets triggered from is the newest app version, which is saved in the database. For this, the view that was shown in Section 5.3.3 gets queried to get the newest app database id of this app. When having retrieved the newest versions database id, the corresponding data is getting queried in the next service task. Then, the current data is getting retrieved from the app store. After having retrieved the most current data from the database and app store, it gets checked whether the app was updated since the last database update, or if the last update was more than a given time slot ago. For testing, the time slot was set to one hour. In case one of the conditions is true, the data from the app store is saved as a new version in the database. Otherwise, the new data is getting discarded.

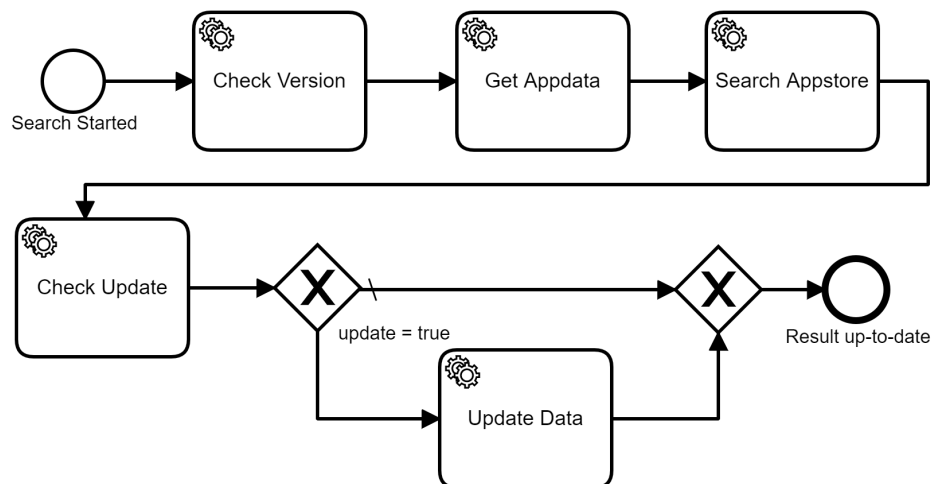


Figure 5.18: The workflow used to update the app metadata.

For implementing the update functionality, as before for saving the data, AJAX is getting used. When for this, the app id used to instantiate the workflow shown earlier to save the current version of the app to the database. When having saved the data, the page is getting reloaded. Reloading the page allows recalculating the analysis data, which will get implemented later.

5.3.5 Rate Apps

For implementing the rating in the UI, the workflow and service task implementation shown and described earlier in Section 5.2.5 were used. Therefore, only the UI and its functionality have to get implemented.

For showing the questionnaire to rate the app SurveyJS was used. The data from this questionnaire gets handed back in JSON, which allows saving the data directly to the database. SurveyJS was already used for creating the MARS questionnaire [10]. Thus, the questionnaire could get directly implemented.

In addition to saving the ratings of the questionnaire, it could be interesting to look at the saved questionnaires. For presenting all saved questionnaires, a box got implemented. In this box, a button gets shown with the date when the rating was saved. Clicking on the button leads to the questionnaire where all filled values are getting shown. The box that gets shown in the UI gets presented in Figure 5.19. Here only two ratings are saved for the app that is currently selected. When saving more ratings the box gets expanded downwards.

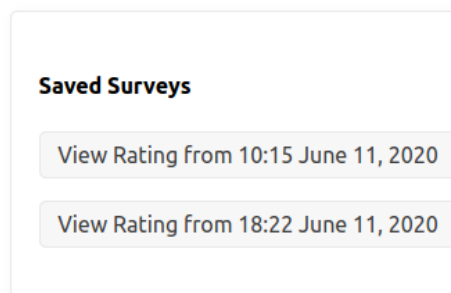


Figure 5.19: The UI element that shows the saved ratings and allows to access those.

For the implementation of this, the rating data got queried from the database. When querying the database, using a view was required since all ratings for a specific app should get queried at once. The view used for this task is shown in Listing 5.24.

```
1 function (doc) {  
2   emit(doc.appId, doc._id);  
3 }  
  
5 function(keys, values, rereduce) {  
6   if (rereduce) {  
7     return values.reduce(function(a, b) {  
8       return [].concat(a, b);  
9     }, [])  
10  } else {  
11    return values;  
12  }  
13 }
```

Listing 5.24: The view which is used to get all rating database ids by app.

This view is similar to the view used for retrieving the analysis data before. In the map function in the top, the app id and the document id get emitted. In the reduce function, the values are getting concatenated. Querying this view results in getting an array of all rating database ids for each app when grouping by app id. Regarding Zeebe, the workflow is shown in Figure 5.7 could get used again. Therefore, it was not necessary to implement a new workflow. Then the data could get implemented using SurveyJS.

5.3.6 Analysis of Metadata

For being able to see changes in the app metadata directly, the metadata which got gathered in the database needs to get analyzed. The data is getting queried from the database using the workflow and implementation described in Section 5.2.6. After having the data fetched from the database, different possibilities can get used for analyzing. First, the average values of specific data can get calculated. Those average values then can get handed to the UI. Second, the data can get handed to the UI directly. The analysis then can get implemented in JavaScript which allows showing line-graphs or dots in a graph to represent changes or specific points in a timeline.

```

1 func getAppAnalysis(appId string) ([]App, []Update) {
2     //create Zeebe client and prepare variables
3     ...
19    request, err := zbClient.NewCreateInstanceCommand().
        BPMNProcessId("analysis-process").LatestVersion().
        VariablesFromMap(variables)
20    if err != nil { ... }
24    //start workflow instance and return results
25    res, err := request.WithResult().Send(ctx)
26    if err != nil { ... }
30    //unmarshall response to JSON
31    var resVariables map[string]interface{}
32    if err := json.Unmarshal([]byte(res.Variables), &resVariables);
        err != nil { ... }
34    //save the app metadata for analysis to prepared App struct
35    var retData []App
36    data := resVariables["analysisData"].([]interface{})
37    for elem := range data {
38        analysisData := data[elem].(map[string]interface{})
39        //add current apps data to retData
40        retData = append(retData, App{ ... })
46    }
47    //sort slice of apps
48    sort.Slice(retData, func(i, j int) bool {
49        return retData[i].Timestamp < retData[j].Timestamp
50    })
51    //special treatment for update data
52    var updateAnalysis []Update
53    updated := ""
54    for elem := range retData {
55        update := retData[elem].Updated
56        //if field updated changed save it
57        if update != updated {
58            updated = update
59            updateAnalysis = append(updateAnalysis, Update{ ... })
62        }
63    }
64    return retData, updateAnalysis
65 }

```

Listing 5.25: An excerpt of the implementation that retrieves the data for analysing the app metadata.

In the implemented prototype, the data was handed over without prior calculations. However, such calculations will get implemented in a later step for analyzing different aspects. The implementation of the function is creating the workflow instance, and treating the data to allow handing them to the UI is shown in Listing 5.25.

In this implementation first, the Zeebe client gets created, and the needed variables get prepared. Then, the workflow instance is created and then started. The results contain the app metadata that is getting emitted by the view shown earlier in Listing 5.16. This data is getting unmarshalled to JSON. After having the data transformed in JSON format, the data is getting saved into a slice of an app object. This slice then gets sorted by the timestamp, which is the date when it was added to the database. Sorting elements is necessary since the data from CouchDB is not sorted by this timestamp, and for visualizing the data, it is easier to treat sorted data. At the end, it is checked for each element if it contains a new update. In case this is true, this gets saved to another prepared slice to allow visualizing the app update timestamps later.

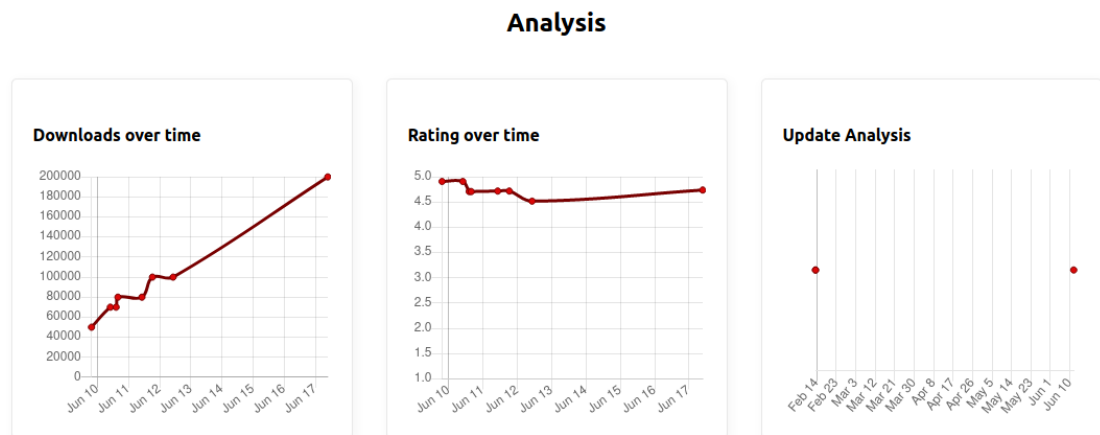


Figure 5.20: An example app metadata analysis.

The resulting UI is shown in Figure 5.20. As mentioned in the concept, ChartJS was used for creating the graphs. On the left-hand side, a line chart was created, showing how the downloads of the app changed by time. Each update in the database creates a new point of data. The date in the x-axis is changing dynamically to present the given points in time in a readable form. Thus, the x-axis is showing minutes, hours, days, or months depending on the data that is handed in. In the center box, another line chart was created to represent the changes in the rating over time. The right-hand box shows how often updates were made to the app. Showing how often updates were made allows seeing if the app is getting developed or not. Further app metadata could get analyzed using the same techniques used here. The shown UI is only representing some fields that are already getting analyzed.

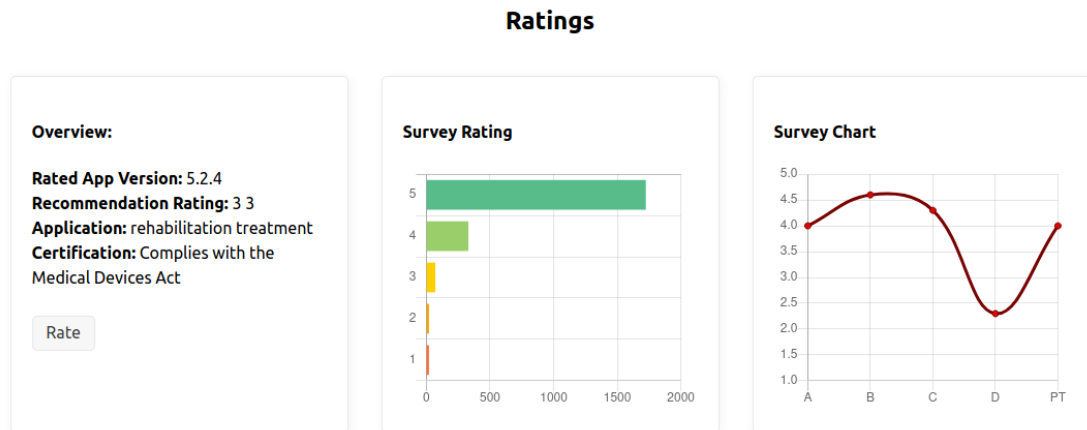


Figure 5.21: An example app rating analysis.

In addition to the analysis of the app metadata, the rating by the surveys can get analyzed. Analyzing the ratings allows seeing differences in the ratings from different times. Furthermore, even if only one rating is available, adding some visual analysis can help to get an overview of the rating without having to look at the details – the implementation is similar to the earlier shown to get the app metadata. Also, for analyzing the ratings, average data gets calculated. The resulting UI is shown in Figure 5.21. As for the analysis of the app metadata, three boxes got implemented. In the left-hand side box, general data of the most current rating get shown. The centered box shows ratings from all surveys are getting shown in a bar chart. The right-hand side box shows the average ratings of different categories rated in the survey. For getting further insights analyzed, more boxes with analyses can get added. However, this analysis can give a good impression on what is possible using charts and text to analyze rating data.

6 Compliance with Requirements

In this chapter the compliance of the implemented analysis service with the functional and non-functional requirements is checked. The conformity is measured in the following six levels:

- (5) The requirement was met in full.
- (4) The requirement was satisfactorily fulfilled.
- (3) This requirement was not satisfactorily met.
- (2) The requirement has not yet been completed.
- (1) The requirement was prepared.
- (0) The requirement was not met.

6.1 Functional Requirements

When measuring the compliance with the functional requirements, the focus is on the implementation. That is because the implementation realizes the functions. For comparing the in Section 3.1 defined functional requirements with the implementation these are getting listed below in a table.

Code	Description	Priority	Fulfilled
Account:			
F01	Authorization	COULD	5
F02	Registration	COULD	1
F03	Authentication	COULD	3
F04	User Administration	COULD	1

Code	Description	Priority	Fulfilled
Search:			
F05	Search for Apps	MUST	4
F06	Save App Metadata to Database	MUST	5
App:			
F07	Show All Saved Apps	MUST	5
F08	Update App Metadata	MUST	5
Review:			
F09	Review App	SHOULD	4
F10	Show All Surveys per App	SHOULD	5
F11	Show Survey Data	SHOULD	5
Analysis:			
F12	Analyse App Metadata	MUST	5
F13	Analyse App Survey Data	SHOULD	4

6.2 Non-Functional Requirements

To measure the compliance with the non-functional requirements, architecture and implementation are given equal weight. This is done since wrong decisions in the design of the architecture can have a significant impact on non-functional requirements. For comparing the in Section 3.2 defined non-functional requirements with the architecture and implementation these are getting listed below in a table.

Code	Description	Priority	Fulfilled
System:			
NF1	Reliability	MUST	4
NF2	Scalability	MUST	5
NF3	Maintainability	SHOULD	4
NF4	Extensibility	SHOULD	4
NF5	Robustness	SHOULD	3

7 Conclusion and Future Work

In this chapter, a conclusion of this thesis is drawn. Additionally, future work is pointed out.

7.1 Conclusion

In this thesis, a workflow-based service that is communicating over HTTP was implemented. This service can get used for saving and analyzing the metadata of mobile applications from the Google Play Store. For retrieving the data from the play store, an open-source play store scraper was used. The structure of the Google Play Store changes every once in a while which would result in having to overwork the implementation when implementing a scraper oneself. Using the open-source play scraper only an update is needed.

The workflows used in this thesis were implemented using Zeebe. Zeebe enables implementing workflows in BPMN 2.0. Furthermore, Zeebe orchestrates service tasks. The service tasks of the workflows were implemented using the Zeebe client written in Go.

The idea of using workflows and their service tasks in order to provide a service allows using each service task independently as a microservice. Those microservices fulfill only a single task. Therefore, microservices can get reused in different workflows. Thus, when using microservices for development, each task gets implemented only once. Furthermore, all microservice can get implemented or changed independently. When improving the code of one microservice, all workflows benefit. Additionally, implementing small tasks that can get distributed on different servers, allows scaling each microservice individually which makes scaling the service easily possible.

For saving the metadata of the apps, CouchDB was used. When using a document-based database, this allowed saving the data in JSON without defining the structure of the database beforehand. Furthermore, CouchDB is fault-tolerant and is made for handling varying traffic. For querying, CouchDB views allow retrieving necessary parts of documents fast and, when implemented, grouped by a key that gets defined when implementing the view. Therefore, CouchDB is well suited for a scalable service.

At the beginning of the implementation, a problem with the used Play Scraper was fixed. With the Play Scraper, it was not possible to get a list of applications in all details. Only the data that appears in search results in the Google Play Store was returned. Fixing this issue allows using this feature and also helped to understand how the Play Scraper works. A new version of the Play Scraper with the implemented fix was released soon after.

When beginning with the implementation of workflows and service tasks, small workflows got implemented first. With this, it was discovered how to implement each technology. Every workflow got tested individually. However, the service tasks were all implemented in the same place and got included in every test. Including the services allows accessing the service task implementations from every workflow and not having to copy them. Furthermore, when changing the implementation of a service task, the changes are getting applied to all workflows directly.

After having created all workflows that are needed for the concept, a user interface was implemented. When creating an interface, testing the workflows in combination was possible. Furthermore, the use case of analyzing the metadata of apps could get demonstrated. The user interface was created as a web interface in Go. For creating the views, HTML templates were used, and dynamic content got added using JavaScript. Implementing the user interface in Go allowed using the same Zeebe client as before. Therefore, parts that were written earlier for testing the workflows could get reused. However, when creating the user interface some of the workflows needed to get revised, i.e., workflows were split to prevent letting them wait for user input. Doing this, it was prevented that workflows are not getting finished; when a user does act differently than it was planned, e.g., the user does not want to save an app after searching for a keyword. For allowing the users to rate the apps, a survey based on MARS was implemented. Adding the data of this survey as an expert rating allows enabling therapists to choose which app is best to treat a specific disease.

In summary, the analysis service implemented in this thesis is suitable for showing the use case of using a distributed workflow-based service for analyzing the metadata of mobile applications.

7.2 Future Work

When implementing the analysis service, it was clear that it is not possible to implement a service that is more than a proof of concept. Thus, the focus of this thesis was creating a prototype with workflows, service tasks, and a user interface that shows most aspects of the concept.

Therefore, the services were not distributed to different servers, and the service tasks were not started by workers multiple times. However, distributing the services to different servers is possible without any significant changes. The same applies for letting the same jobs multiple times to distribute the workload.

Furthermore, as the app store, only the Google Play Store got implemented. Therefore, no apps from the Apple App Store can get searched, saved, and analyzed by now. Adding more stores to the implementation would mean to add new workflows and service tasks.

Another aspect that should get implemented is user accounts. For the prototype implemented, only one user per user role was implemented with a hard-coded username and password. Instead of using these hard-coded user credentials, the user data should get encrypted and saved to the database. Also, with this, the registration and user administration can get implemented.

When going on to work with this service, further analysis could get implemented to give a better overview of the apps. For this, mostly, the user interface would need to get extended. However, implementing more analysis even later when already using the service is not a problem, since the app metadata is saved with all details to the database.

A further idea that could not get implemented is to follow apps from a specific point in time. Following apps from a point in time would mean, to show the analysis data on the page of such a followed app from this point. Showing the analysis, not from the beginning, would allow seeing changes after this point more prominent. Therefore, this could be interesting for therapists that use an app for treatments.

Since the analysis service was implemented as a prototype, not every possible user

entries were tested. Additionally, when testing the service Zeebe, the database and the play scraper were always up and answering directly. Therefore, the implemented service needs to get tested to work as expected in all error situations.

Finally, a problem when using a scraper to get metadata from the Google Play Store, is that the app store changes the interface once in a while. As a result of a changed interface, the play scraper cannot get the metadata from the app store as before. For solving issues as a changed interface, the play scraper gets updated, mostly in hours after the app store changed. However, to make use of these changes, the play scraper needs to get updated. When using the service in public, such changes should get registered and, in the best case, solved automatically.

A Software Used and Its Versions

In this appendix, the software that was used for creating the implementation and its versions are getting stated.

For writing code in JavaScript, which was necessary to fix an issue in the Play Scraper (Section 5.1) an IDE was very helpful. The same applies for writing code in Go for the implementation of the Zeebe client (Section 5.2) and the UI (Section 5.3). Additionally, software was used to run all services that were needed. The Google Play API was used via npm. Here it was necessary to update the used play scraper version to make use of the in Section 5.1 implemented fix. Zeebe, the Zeebe simple monitor, and CouchDB were used with Docker.

Software	Version	Used For
WebStorm ¹	2019.3.1	Developing JavaScript
GoLand ²	2020.1.2	Developing Go
Zeebe Modeler ³	0.9.1	Modeling BPMN 2.0 workflows
google-play-api ⁴	1.0.7	Implementing the play scraper
google-play-scraper ⁵	7.1.3	Scraping the Google Play Store
Zeebe ⁶	0.23.1	Orchestrating workflows
Zeebe Simple Monitor	0.18.0	Debugging workflow implementation
CouchDB ⁷	3.1.0	Saving data persistently

¹<https://www.jetbrains.com/webstorm/>

²<https://www.jetbrains.com/go/>

³<https://github.com/zeebe-io/zeebe-modeler>

⁴<https://github.com/facundooolano/google-play-api>

⁵<https://github.com/facundooolano/google-play-scraper>

⁶<https://github.com/zeebe-io/zeebe-docker-compose>

⁷<https://couchdb.apache.org/>

Bibliography

- [1] Rakesh Agrawal et al. "The Claremont Report on Database Research". In: *ACM Sigmod Record* 37.3 (2008), pp. 9–19. DOI: 10.1145/1462571.1462573. URL: <https://doi.org/10.1145/1462571.1462573>.
- [2] Hannes Ametsreiter. *Smartphone-Markt: Konjunktur und Trends*. https://www.bitkom.org/sites/default/files/2020-02/bitkom-pressekonferenz-smartphone-markt-20-02-2020-prasentation_final.pdf. (last accessed: Jun, 19th, 2020). 2020.
- [3] J Chris Anderson, Jan Lehnardt, and Noah Slater. *Couchdb: The Definitive Guide*. " O'Reilly Media, Inc.", 2010. ISBN: 978-0-596-15589-6.
- [4] André B Bondi. "Characteristics of Scalability and Their Impact on Performance". In: *Proceedings of the 2nd international workshop on Software and performance*. 2000, pp. 195–203. DOI: 10.1145/350391.350432. URL: <https://dl.acm.org/doi/10.1145/350391.350432>.
- [5] Kevin Brennan et al. *A Guide to the Business Analysis Body of Knowledge*. 2nd ed. IIBA - International Institute of Business Analysis, 2009. ISBN: 978-0-9811292-1-1.
- [6] Martin C Brown. *Getting Started with CouchDB: Extreme Scalability at Your Fingertips*. " O'Reilly Media, Inc.", 2012. ISBN: 978-1-449-30755-4.
- [7] Andrea Burattin. "Process Mining Techniques in Business Environments". In: *volume 207 of Lecture Notes in Business Information Processing*. Springer, 2015.
- [8] Mario Castro Contreras. *Go Design Patterns*. Packt Publishing Ltd, 2017. ISBN: 978-1-78646-620-4.

- [9] Remco M Dijkman, Marlon Dumas, and Chun Ouyang. “Semantics and Analysis of Business Process Models in BPMN”. In: *Information and Software technology* 50.12 (2008), pp. 1281–1294. DOI: 10.1016/j.infsof.2008.02.006. URL: <https://doi.org/10.1016/j.infsof.2008.02.006>.
- [10] Philipp Dörzenbach. “Konzeption und Realisierung einer webbasierten Anwendung zur systematischen Bewertung medizinischer und psychologischer Anwendungen”. Bachelor thesis. Ulm University, 2019. URL: <http://dbis.eprints.uni-ulm.de/1797/>.
- [11] Nicola Dragoni et al. “Microservices: Yesterday, Today, and Tomorrow”. In: *Present and ulterior software engineering*. Springer, 2017, pp. 195–216. DOI: 10.1007/978-3-319-67425-4_12. URL: https://doi.org/10.1007/978-3-319-67425-4_12.
- [12] Ben Evans. *Java, the Legend: Past, Present, and Future*. O’Reilly Media, 2015. ISBN: 978-1491934678.
- [13] Ben Evans and David Flanagan. *Java in a Nutshell: A Desktop Quick Reference*. 7th ed. O’Reilly Media, 2019. ISBN: 978-1492037255.
- [14] Jürgen Galler. *Vom Geschäftsprozeßmodell zum Workflow-Modell*. Springer-Verlag, 1997. ISBN: 978-3-322-90848-3. DOI: 10.1007/978-3-322-90847-6.
- [15] Camunda Services GmbH. *A Workflow Engine for Microservices Orchestration*. <https://zeebe.io/>. (last accessed: Jun, 19th, 2020). 2020.
- [16] Camunda Services GmbH. *Microservices and BPM*. May 2017. URL: <https://camunda.com/de/learn/whitepapers/microservices-and-bpm/>.
- [17] Camunda Services GmbH. *Zeebe Documentation*. <https://docs.zeebe.io/>. (last accessed: Jun, 19th, 2020). 2020.
- [18] Jan Gottschick et al. “Microservices”. In: *ÖFIT-Trendschau: Öffentliche Informationstechnologie in der digitalisierten Gesellschaft*. Berlin: Kompetenzzentrum Öffentliche IT, 2019.
- [19] Object Management Group. “Business Process Model and Notation (BPMN)”. Version 2.0.2. In: *OMG Specification, Object Management Group* (2014).
- [20] Christine Hennings and Cornelius Herstatt. *Belief Elicitation Study: Identifying Salient Beliefs of Patients Towards the Use of Mhealth*. Tech. rep. Working Paper, 2019. DOI: 10419/192994. URL: <https://doi.org/10419/192994>.

- [21] Bradley Holt. *Scaling Couchdb: Replication, Clustering, and Administration*. " O'Reilly Media, Inc.", 2011. ISBN: 978-1-4493-0343-3.
- [22] Kevin Kline, Daniel Kline, and Brand Hunt. *SQL in a Nutshell: A Desktop Quick Reference Guide*. " O'Reilly Media, Inc.", 2008. ISBN: 9781565927445.
- [23] Mehdi Maoui et al. *Continuous API Management: Making the Right Decisions in an Evolving Landscape*. O'Reilly Media, Incorporated, 2018. ISBN: 978-1492043553.
- [24] Tom Marrs. *Json at Work: Practical Data Integration for the Web*. " O'Reilly Media, Inc.", 2017. ISBN: 978-1449358327.
- [25] Russ McKendrick and Scott Gallagher. *Mastering Docker: Unlock New Opportunities Using Docker's Most Advanced Features*. Packt Publishing Ltd, 2018. ISBN: 978-1789616606.
- [26] Ingo Melzer et al. *Service-orientierte Architekturen mit Web Services: Konzepte-Standards-Praxis*. 4th ed. Elsevier, Spektrum, Akad. Verlag, 2010. ISBN: 978-3-8274-2549-2.
- [27] I. J. Mojica Ruiz et al. "Examining the Rating System Used in Mobile-App Stores". In: *IEEE Software* 33.6 (2016), pp. 86–92. DOI: 10.1109/MS.2015.56.
- [28] Sam Newman. *Building Microservices: Designing Fine-Grained Systems*. "O'Reilly Media, Inc.", 2015. ISBN: 978-1-491-95035-7.
- [29] Roy Oberhauser and Sebastian Stigler. "Microflows: Enabling Agile Business Process Modeling to Orchestrate Semantically-Annotated Microservices". In: *Proceedings of the Seventh International Symposium on Business Modeling and Software Design (BMSD 2017)*. 2017, pp. 19–28. ISBN: 978-989-758-238-7. DOI: 10.5220/0006527100190028. URL: <https://doi.org/10.5220/0006527100190028>.
- [30] Facundo Olano et al. *Google Play API*. <https://github.com/facundoolano/google-play-api>. (last accessed: Jun, 19th, 2020). 2020.
- [31] Facundo Olano et al. *Google Play Scraper*. <https://github.com/facundoolano/google-play-scraper>. (last accessed: Jun, 19th, 2020). 2020.

- [32] Luc Perkins, Eric Redmond, and Jim Wilson. *Seven Databases in Seven Weeks: A Guide to Modern Databases and the Nosql Movement*. Pragmatic Bookshelf, 2018. ISBN: 978-1680502534.
- [33] Aaron Ploetz et al. *Seven Nosql Databases in a Week: Get up and Running With the Fundamentals and Functionalities of Seven of the Most Popular Nosql Databases*. Packt Publishing Ltd, 2018. ISBN: 978-1787127142.
- [34] Pethuru Raj and Ganesh Chandra Deka. *A Deep Dive into NoSQL Databases: The Use Cases and Applications*. Academic Press, 2018. ISBN: 978-0-12-813786-4.
- [35] Ramda. *Practical functional Javascript*. <https://github.com/Ramda/ramda>. (last accessed: Jun, 19th, 2020). Dec. 2019.
- [36] Mario Scheliga. *CouchDB kurz & gut*. O'Reilly Germany, 2010. ISBN: 978-3-89721-559-7.
- [37] Vincent Smith. *Go Web Scraping Quick Start Guide: Implement the Power of Go to Scrape and Crawl Data From the Web*. Packt Publishing Ltd, 2019. ISBN: 978-1789615708.
- [38] Michael Stach et al. "Mobile Health App Database - A Repository for Quality Ratings of mHealth Apps". In: *33rd IEEE CBMS International Symposium on Computer-Based Medical Systems (CBMS)*. Rochester, MN, USA: IEEE Computer Society Press, July 2020.
- [39] Michael Stach et al. "Technical Challenges of a Mobile Application Supporting Intersession Processes in Psychotherapy". In: *The 17th International Conference on Mobile Systems and Pervasive Computing (MobiSPC)*. Procedia Computer Science. Leuven, Belgium: Elsevier Science, Aug. 2020.
- [40] Michael Stach et al. "Towards a Beacon-based Situational Prioritization Framework for Process-Aware Information Systems". In: *Procedia Computer Science* 134 (2018), pp. 153 –160. ISSN: 1877-0509. DOI: <https://doi.org/10.1016/j.procs.2018.07.156>. URL: <http://www.sciencedirect.com/science/article/pii/S1877050918311190>.
- [41] Mihalīs Tsoukalos. *Mastering Go: Create Golang Production Applications Using Network Libraries, Concurrency, Machine Learning, and Advanced Data Structures*. Packt Publishing Ltd, 2019. ISBN: 978-1838559335.

Name: Jörn Hofschlaeger

Matrikelnummer: 791591

Erklärung

Ich erkläre, dass ich die Arbeit selbständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den

Jörn Hofschlaeger