

Institute of Software Technology

University of Stuttgart
Universitätsstraße 38
D-70569 Stuttgart

Master thesis

**Design, Implementation and
Evaluation of an Application for
Guiding Architectural Refactoring to
Microservices**

Tobias Haller

Course of Study: Software Engineering

Examiner: Prof. Dr. Stefan Wagner

Supervisor: Jonas Fritsch

Commenced: November 9, 2021

Completed: May 9, 2022

Abstract

The microservices architectural style which emerged recently is currently a hot topic and widely discussed in literature. As a result, many companies have become interested in migrating legacy applications to microservices due to factors like, easier scalability, deployment or better flexibility in combining multiple technologies. Nonetheless, finding clear guidance on how to conduct a migration of legacy applications to microservices is a difficult task. Software architects tasked with such a migration are in need of a structured and possible tool assisted guide on selecting an appropriate strategy and refactoring approach. While there are some academic publications conducting Systematic Literature Reviews and Mapping Studies regarding existing approaches for architectural refactoring to microservices, they can only provide a temporary snapshot on the current state of research. This problem should be addressed by a Migration Framework to guide architectural refactoring of monolithic applications towards microservices. Such a framework is currently developed within the ESE group of the Institute of Software Engineering. In order to do this the framework aims to guide users through three main phases helping them with the system comprehension, migration strategy definition and architecture definition of their application. The goal of this thesis is to provide a prototypical implementation and evaluation of a web-based application realizing the migration strategy definition phase of said Migration framework. We specifically choose a web-based application due to easier access to the finished product. In order to achieve this, this thesis is split into three parts. In a first step, an extensible repository should be designed and implemented that hosts existing refactoring approaches based on relevant properties. Following this, a web-based application acting as the application's user interface will be designed to represent the existing architectural refactoring framework. Finally, an empirical evaluation of the developed application is conducted.

Contents

1	Introduction	15
2	Background	19
2.1	Monolithic Applications	19
2.2	Microservices Architecture	20
2.3	Architectural Refactoring Framework	20
3	Related Work	23
4	Methodology	25
4.1	Literature Analysis and Data Model	25
4.2	Requirements Engineering	25
4.3	Stage 1: Extensible Repository	26
4.4	Stage 2: User Interface	26
4.5	Stage 3: Empirical Evaluation	26
5	Extensible Repository for Architectural Refactoring Approaches	27
5.1	Requirements	27
5.2	Data Model	29
5.3	Technology Stack	33
5.4	Design and Implementation	35
6	Web-based User Interface	47
6.1	Requirements	47
6.2	Technology Stack	52
6.3	Design and Implementation	53
6.4	Integration with Back-end and Repository	58
7	Build and Deployment of the Tool	61
7.1	Reverse Proxy	62
7.2	User Interface Application	62
7.3	Extensible Repository Application	63
7.4	Service Orchestration with Docker Compose	63
8	Evaluation of Web-Based Application	67
8.1	Questionnaire Design	67
8.2	Questionnaire Results	70
8.3	Discussion of Results	74
9	Threats to Validity and Limitations	77

10 Conclusion and Outlook	79
Bibliography	81
A Database Tables	85
B Evaluation Questionnaire	89

List of Figures

2.1	The Architectural Refactoring Framework [HF22].	21
5.1	Taxonomy of Service Identification Approaches by Abdellatif et al. [ASM+21]. . .	29
5.2	The Database Data Model of the Extensible Repository as an Entity-Relationship Model using Crow's Foot Notation [PPJ19].	31
6.1	User Interface View to Create a new Refactoring Approach.	54
6.2	Structure of Components and Route Navigation of the Front-End Application. . .	54
7.1	Docker Architecture of the Application.	61
8.1	Histograms of the Participants Demographics.	70
8.2	Box Plot of the Evaluation Results regarding the Tools Structure.	71
8.3	Box Plot of the Evaluation Results regarding the Tools Usability.	73
8.4	Box Plot of the Evaluation Results regarding Potential Future Additional Functionalities of the Tool	73
B.1	Questionnaire Start Section.	89
B.2	Questionnaire Personal Information Section.	90
B.3	Questionnaire Exploration of the Tool Section.	91
B.4	Questionnaire Tool Structure Section.	92
B.5	Questionnaire Tool Usability Section.	93
B.6	Questionnaire Additional Functionality Section.	94

List of Tables

A.1	The Database Table Entries for Domain Artifact Approach Attributes.	85
A.2	The Database Table Entries for Runtime Artifact Approach Attributes.	85
A.3	The Database Table Entries for Model Artifact Approach Attributes.	85
A.4	The Database Table Entries for Executable Approach Attributes.	85
A.5	The Database Table Entries for Quality Approach Attributes.	86
A.6	The Database Table Entries for Direction Approach Attributes.	86
A.7	The Database Table Entries for Automation Level Approach Attributes.	86
A.8	The Database Table Entries for Analysis Type Approach Attributes.	86
A.9	The Database Table Entries for Technique Approach Attributes.	87
A.10	The Database Table Entries for Architecture Approach Attributes.	87
A.11	The Database Table Entries for Service Type Approach Attributes.	87
A.12	The Database Table Entries for Validation Method Approach Attributes.	87
A.13	The Database Table Entries for Tool Support Approach Attributes.	88
A.14	The Database Table Entries for Results Quality Approach Attributes.	88
A.15	The Database Table Entries for Accuracy/Precision Approach Attributes.	88

List of Listings

5.1	Trivial C# Model Class of the Quality Entity.	35
5.2	C# Model Class of the RefactoringApproach Entity.	36
5.3	Manual Configuration of Many-To-Many Relation with Custom Join Table Name.	36
5.4	C# Source Code to Create a new Technique Entry in the Database and its Translated Structured Query Language (SQL) Statement.	37
5.5	C# Source Code to Read and Output Entity from Database and its Translated SQL Statement.	38
5.6	C# Source Code to Delete an existing Technique Entry in the Database and its Translated SQL Statement.	39
5.7	C# ServiceExceptionFilter Class used to Handle Exceptions in Controller Classes.	41
5.8	C# IRecommendationService Interface to Implement Recommendation Algorithms.	42
5.9	Example Configuration Class of Technique Model Class.	45
6.1	Angular Source Code of Permission Service.	58
6.2	Additions to .csproj-File needed to Export the “swagger.json” File.	59
6.3	The JavaScript object notation (JSON) Configuration File used for “ng-openapi-gen” Package.	59
6.4	Example Typescript Source Code showing how Generated Services can be used to Add a new Refactoring Approach.	60
7.1	Dockerfile of the NGINX Proxy.	62
7.2	NGINX Configuration File for the Reverse Proxy Component.	62
7.3	Dockerfile used to Containerize the Angular based User Interface Application.	63
7.4	Dockerfile used to Containerize the .NET 6 based Back-End Application.	64
7.5	Docker Compose File used to Configure the different Application Services.	64

Acronyms

- API** application programming interface. 20
- CRUD** create, read, update and delete. 37
- CSS** Cascading Style Sheets. 52
- DBMS** database management system. 28
- DOM** Document Object Model. 57
- EF Core** Entity Framework Core. 34
- ERM** entity-relationship model. 31
- HTML** HyperText Markup Language. 52
- HTTP/HTTPS** Hypertext Transfer Protocol (Secure). 33
- JSON** JavaScript object notation. 11
- ORM** object-relational mapper. 33
- REST** representational state transfer. 20
- Sass** syntactically awesome style sheets. 52
- SCSS** Sassy CSS. 52
- SDLC** software development life cycle. 23
- SIA** service identification approach. 21
- SQL** Structured Query Language. 11
- URL** uniform resource locator. 40

1 Introduction

While the recently emerged microservices architectural style is widely discussed in literature, it is difficult to find clear guidance on the process of refactoring legacy applications. One of the most discussed aspects in this context is finding the right service granularity to fully leverage the advantages of a Microservices architecture. Software architects facing this challenge are in need of selecting an appropriate strategy and refactoring approach. While a number of academic publications discuss approaches for architectural refactoring to microservices [FBZW19][PMA19], practical guidance for system architects and developers is still missing. They struggle to acquire a structured overview of the underlying techniques and their capabilities based on the current state of research [FBWZ19]. Systematic Literature Reviews and Mapping Studies can only address part of it and always represent a temporary snapshot. The body of research is continuously growing as new approaches and experience reports are added, obsoleting the above-mentioned meta-studies after a short time.

Besides, the existing approaches on microservices refactoring vary significantly regarding profoundness, practicability, quality aspects and other characteristics. In particular, the supported source artifacts, extent of evaluation, considered quality attributes and tool support vary to a great extent. Furthermore, the coverage regarding the entire migration process and level of abstraction differs strongly. This makes it difficult for software architects to overlook and comprehend the existing body of scientific research. Hence, it can be a time-consuming task to figure out if one of the existing approaches is applicable for a given system.

Existing studies [FBZW19][PMA19][SKM20][ASM+21] compare and classify refactoring approaches recently proposed in academic literature in order to solve this problem. In these studies, approaches are classified by attributes such as their underlying decomposition technique and a variety of other properties. However, these classifications do not provide sufficient guidance in practice and leave a lot of additional work and research up to the system architects.

Therefore, the problem should be addressed by a Migration Framework, currently under development in a research project, to guide architectural refactoring of monolithic applications towards microservices. Such a framework is currently developed within the ESE group of the Institute of Software Engineering at the University of Stuttgart. The goal of this architectural refactoring framework is to provide a guide for software architects and developers when migration application from a monolithic architecture to microservices. In order to do this the framework walks users through three main phases helping them with the system comprehension, migration strategy definition and architecture definition of their application. However, to date there exists no implementation of this framework. This master thesis aims to provide a prototypical implementation and evaluation of a web-based application in this regard. In order to realize an implementation and evaluation of this framework, we propose the following research questions, which we will be discussing in this thesis:

RQ 1: How can we design an extensible repository for existing migration and refactoring approaches?

RQ 2: How can a web-based application of an architectural refactoring framework look like?

RQ 3: How do potential users evaluate a prototypical implementation of this framework?

Based on these three research question we define three objectives for this work to answer them. In a first part, an extensible repository should be designed and implemented that hosts existing refactoring approaches based on relevant properties. Here, existing groundwork can be leveraged that was developed in a prior thesis [Gu20]. Important enhancements should offer functionality to easily add new studies and approaches and modify their classification attributes. Design and implementation decisions for this extensible repository will be documented and elaborated in this thesis.

In a second part, the web-based application will be designed to represent the existing refactoring framework. We specifically choose a web-based application due to easier access to the finished product for users and participants of an application evaluation. The application will use the extensible repository that was developed in the first part and act as a user interface for it. The various steps of the framework should follow a common template that standardizes input/output artifacts or decisions made in a certain step. The subsequent implementation will also be documented and discussed in this thesis as well as use appropriate technologies that optimize maintainability aspects of the application.

The third part covers an empirical evaluation of the developed application. To ensure the appropriateness of the implementation and achieve optimal usability, potential users need to be involved early in the development process. System developers and architects may have specific requirements and can provide useful feedback to address such aspects. Therefore, a prototypical implementation of the application will be evaluated in a user study among system developers and architects who are currently or have been earlier involved in microservices migration projects.

In order to answer the defined research questions and fulfill the set goals for this thesis we will discuss the following points:

- Chapter 2: In this chapter we will first present some background information important to the thesis.
- Chapter 3: We will present existing work related to the topic of this thesis and put it into context regarding the current state of research.
- Chapter 4: Here, we will go into more detail regarding the methodology that will be applied throughout this thesis.
- Chapter 5: In this chapter, we will detail the expected requirements, used data model and technologies, as well as design and implementation aspects of the extensible repository.
- Chapter 6: Here, we will similarly to the repository, document important aspects of the web-based user interface such as requirements, design and implementation details and integration with the repository.

-
- Chapter 7: In this chapter, we will go into details about how we can build and deploy the complete application bundle of the applications described in the earlier two chapters.
 - Chapter 8: In this chapter, we will discuss information regarding the conducted user study evaluation of the implemented application, as well as the analysis and discussion of the results
 - Chapter 9: Here, we will cover threats to validity of the user study and limitations regarding the whole thesis.
 - Chapter 10: Finally, in the last chapter we will close out the thesis by summarizing and taking a look at potential future work for the topic of this thesis.

2 Background

In this chapter, we provide background knowledge regarding topics related to the migration of applications to a microservices architecture. We briefly discuss the monolithic architectural style, which is often the initial architecture application want to migrate from. Following this, we will also inform about the microservices architectural style. Furthermore, this chapter will inform about the architectural refactoring framework, the application we will design and implement during this master thesis is based on.

2.1 Monolithic Applications

In an application with a monolithic architecture, all functionalities are contained in a single application. Monolithic software is created with the goal of being self-contained. This architecture is closely coupled, which implies that if one of the components is missing, the program will not be executed or compiled. This of course also means that the monolithic application will be written with the help of one programming language, sometimes even with one framework [SS17].

Even though using a monolithic architecture is the traditional way of developing software, one should not assume that it comes without its use-cases and advantages. Especially for smaller applications, a monolithic architecture should be considered due to its simplicity in regard to implementation and testing. Also, a monolithic application requires less operational overhead since no service or component interfaces and communication paths need to be considered when all components are tightly coupled in a single application [KM19].

However, older monolithic applications have become bigger and bigger over time and therefore suffer from the disadvantages of their monolithic architecture. Firstly, making changes to a monolithic application becomes harder the bigger they become. Moreover, when deploying a monolithic application the whole application will have to be deployed even if only a small part has been changed. This also leads to the whole application needing to be tested in order to ensure that nothing unexpected has broken, resulting in longer time requirements for testing. Also, scaling large monolithic applications becomes a big problem, due to having to scale the complete application instead of only the components which experience big load. This could lead to wasted computing and storage resources, which makes the application less resource efficient [KM19]. Therefore, some applications are better suited for other architectural styles, such as a microservices architecture.

2.2 Microservices Architecture

The term microservices was first primed by Dr. Peter Rodgers, who used the term “Micro-Web-Services” during a presentation on cloud computing in 2005 [Foo21] and later in 2011 the term microservices itself at a conference of software architects [Hab16]. The main characteristic of a microservices architecture is that individual services are held small and focused on their single responsibility. All of these services are decoupled from each other and will have to communicate with each other using e.g. a representational state transfer (REST) application programming interface (API) [SS17].

This concept comes with several advantages, which can be almost directly compared to common disadvantages of monolithic applications. Firstly, due to the nature of small individual service applications, we can independently deploy each one in case of changes. This easy deployment enables one to frequently deploy the microservice and therefore take advantage of concepts such as continuous delivery and similar. Unlike with monolithic applications, this also provides developers with the flexibility to use different frameworks and programming languages for each microservice [MW18]. This also leads to easier scaling for microservices applications, because microservices which experience higher load can be scaled up without affecting the other services of the system [RWW+18].

However, there are also limitations of the microservice architecture. A system utilizing microservices is a complex system since it consists of many applications which have to communicate with each other, often also in a distributed manner. It can also have implications for the performance of a system, because of the mentioned communication between services. A direct local application call is generally faster than other communication methods, especially if internet latency plays a role. Due to this, microservices architecture should not be considered as a universally desired architecture style for any applications [KM19].

2.3 Architectural Refactoring Framework

Many companies are interested in migrating their existing monolithic applications to microservices. However, just planning this migration process requires a lot of time and resources due to a lack of structured guidance. The architectural refactoring framework presented in this chapter aims to provide such a formal guide to software architects and developers.

The architectural refactoring framework will be the basis for the application designed in this thesis and is currently developed within the ESE group of the Institute of Software Engineering. The current version of this framework can be seen in figure 2.1. It consists of three distinct phases, which should help software architects and developers by guiding the migration process in a structured way [HF22]. These phases will be elaborated in the following:

- **Phase 1: System Comprehension:** In phase 1 a comprehensive view of the system to migrate should be created. In cooperation with the most important stakeholders’ business and organizational goals should be defined and important quality aspects of the application should be defined. Additionally, involved software architects want to understand the legacy system and assess if a migration to microservices is even desired in the first place or if

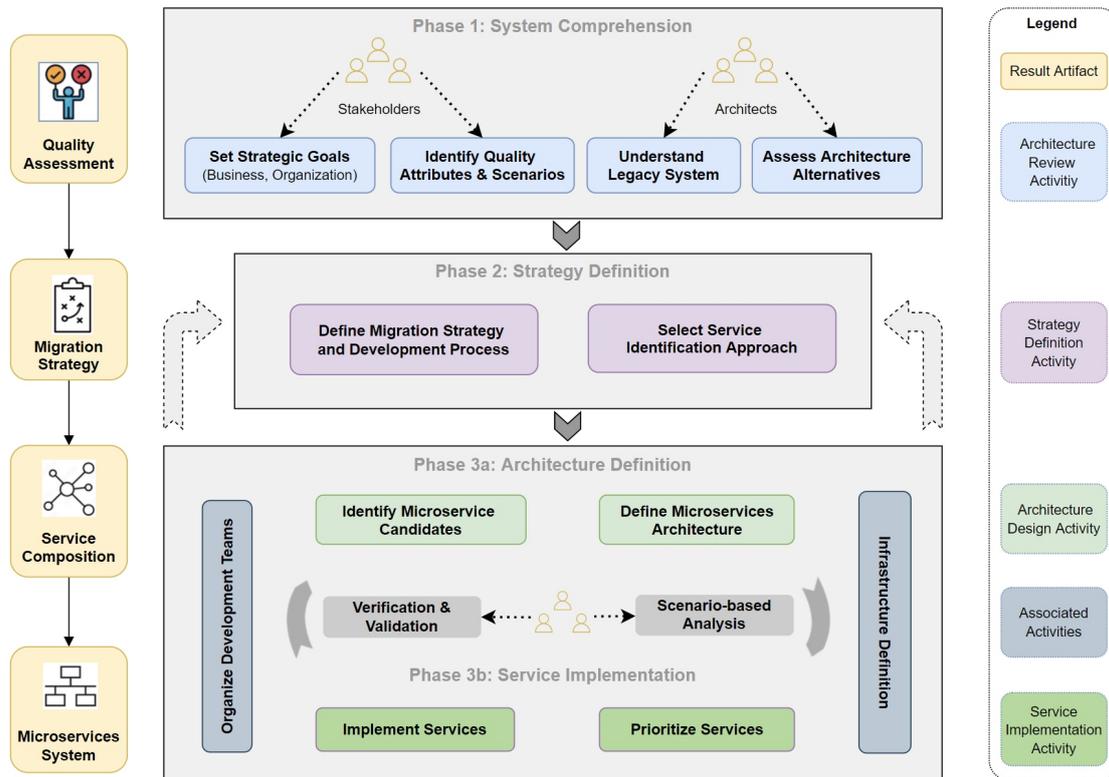


Figure 2.1: The Architectural Refactoring Framework [HF22].

another architecture will ensure better results. In the case where a migration to microservices is intended the mentioned tasks should be done and as a result phase 1 should provide a complete quality assessment of the system to use in the coming phases.

- Phase 2: Strategy Definition:** Phase 2 is the for this thesis most important phase, since most of the implemented functionality relates to it. During this phase, the software architect wants to concretely define which migration strategy they will apply to the migrating software system. Furthermore, the development process will be planned out based on the defined migration strategy. Another important aspect of phase 2 is to select the preferred service identification approach (SIA). After going through this phase, the software architects should have a defined migration strategy with which they can proceed to the next phase.
- Phase 3a: Architecture Definition:** Phase 3 is split into two parts. The first part is about defining the architecture software architects want to migrate their applications to. The identification of microservice candidates and broader definition of the microservices architecture is the main task of this phase. This should result in a desired service composition for the system one wants to migrate. There exists a very tight coupling between phase 3a and phase 3b. This means that during phase 3 as a whole, one can go through both phases multiple times based on insights they could gain during the implementation. Via scenario-based analysis, the defined service composition should be realized in phase 3b.

- **Phase 3b: Service Implementation:** The concrete implementation of the defined services is split into phase 3b. At this stage, the framework can not help software architects anymore, since the implementation is in the hands of the respective development team. However, insights that are gained during the implementation can lead to additional verifications and validations of the service composition of the predecessor phase. This can mean that the system comprehension is adjusted if needed and after the implementation can continue. If the migration is finished, the resulting system should be migrated to microservices architecture.

The application we will design and implement during this thesis is based on this architectural refactoring framework. However, there is additional supplementary work this thesis is based on. These other pieces of literature will be the topic of the next chapter, where we will summarize other related work important to this thesis.

3 Related Work

Since the introduction of microservices as a concept and the realization that many advantages of microservices directly resolve problems of traditional monolithic applications, the topic of migrating such applications has become a widely researched field. Therefore, classifications of such migration approaches and SIAs are interesting pieces of literature as a basis for the data model we will construct in section 5.2 during the thesis. However, as a basis for this thesis, we are interested in very detailed and flexible classifications which can be used to describe many types of refactoring approaches. Unfortunately, many pieces of literature could not provide such a detailed and flexible classification.

One such classification of migration approaches from monolithic applications to microservices comes from Fritzsche et al. [FBZW19]. They compared ten existing refactoring approaches proposed by academic literature and classified them based on their decomposition strategy. Additionally, they worked out a decision guide for these decomposition approaches using requirements of monolithic applications migrating to microservices.

Abdellatif et al. [ASM+21] provide a detailed taxonomy of SIAs in their work. They base their taxonomy on a systematic literature review of 41 SIAs and build a multi-layered classification of these existing approaches. They start at a high-level classification, categorizing a SIA based on its used inputs, approach process, resulting outputs and usability. Following this, they divide each category further into a fine-grained and detailed classification model of SIAs.

Furthermore, Bajaj et al. [BBGG21] propose guidance on which migration technique should be used in the migration process from monolith to microservices. To do this, they examine existing migration approaches and group them based on their software development life cycle (SDLC) artifacts. Additionally, they provide a classification of which SDLC artifacts can be used for either greenfield or brownfield microservices development.

Finally, an important previous work comes from Gu [Gu20] who worked out a classification framework of migration approaches from monolithic applications to microservices in his master thesis. To do so, a systematic literature review was conducted of 31 contributions from 2017 to 2020. Additionally, a tool based on the developed classification was created to guide developers to migration approaches fitting their migration requirements.

In conclusion, even though there is a lot of research regarding migration approaches and sufficient research on the classification of refactoring, service identification or migration approaches, there is currently almost no research done on creating high quality tools which can assist in planning this migration process. Therefore, the tool designed and developed in this thesis will be based on the existing classification research with the goal to guide the architectural refactoring process to microservices of its users.

4 Methodology

The research methodology for this thesis is different for each stage and consists of three different main stages which each correlate to one research question. 1) The design and implementation of the extensible repository and its API, 2) the design and implementation of a prototype web-application acting as the applications' user interface and 3) an empirical evaluation of the developed prototype application.

4.1 Literature Analysis and Data Model

The first step that has to be done before trying to answer any of the defined research questions is researching and deciding upon a classification of migration or refactoring approaches to microservices architecture. In order to decide on such a classification, we analyzed research regarding existing refactoring approaches and further research of already existing classifications of such approaches.

Based on the outcome of this literature analysis, we will then either use one existing classification or create a modified classification. After deciding on a classification we will design a relational data model which we will use to persist and save data regarding refactoring approaches for our application. Further details regarding the refactoring approach classification and the design of the data model will be topic of section 5.2.

4.2 Requirements Engineering

Following the literature analysis and the design of the data model, we will decide on already known requirements in a first requirements engineering phase. The source of these requirements are discussion meetings with the designer of the framework [HF22]. Further requirements can be added in an agile development style during the implementation of the application. For each requirement we will document an identifier, title, description, rationale and priority. Additionally, in some cases we will reference related requirements as dependencies which are prerequisites to realize the requirement or will make it easier in some way to fulfil the requirement. The priority of each requirement will be evaluated on a scale of one (1) to five (5). We consider the application a minimal viable product if all requirements with a priority of 4 or higher are implemented. The following is a brief summary of what each priority means:

Priority 5 Mandatory requirement that must precede all other priorities.

Priority 4 Important requirement, which needs to be implemented to achieve a minimum viable product.

Priority 3 Important requirement, which we want to fulfill with the application, however is not needed to achieve the minimum viable product stage.

Priority 2 Optional requirement, which will improve important quality aspects of the application.

Priority 1 Optional requirement, which we want to fulfill with the application if we have enough time. These requirements are mostly nice to have, but will only be implemented if they take very little time and all other requirements are met.

The complete results of this requirements engineering will be discussed in section 5.1 and section 6.1 for either of the application parts.

4.3 Stage 1: Extensible Repository

After the design of the data model and requirements engineering phase for the repository, a back-end application will be designed and then implemented. All information regarding this requirements engineering is available in chapter 5. Part of the back-end application will be a database reflecting the created data model and the necessary business logic that bridges the gap between the functionality offered in the API and the database. Furthermore, an API to expose endpoints correlating to the functionalities implemented will be part of the back-end application. Needed technologies for this will be decided in this stage and are described in section 5.3.

4.4 Stage 2: User Interface

In the second stage, we will design and implement a web-based application to be used in combination with the previously designed back-end application and extensible repository. Chapter 6 will contain all results regarding this process. This front-end application will act as a user interface for the extensible repository. The application aims to live up to current web development best practices in order to optimize maintainability aspects. As with the repository, the technologies used for this part of the application are defined in this stage and will be elaborated further in section 6.2.

4.5 Stage 3: Empirical Evaluation

In the third and last stage, we will design and conduct a user study of the developed application with the goal to evaluate various aspects of the application. The aspects we are most interested in are the tool's usability, the structure of the user interface, and opinions on potential future features that could be added to the tool. Potential users for this user study include industry contacts of the Empirical Software Engineering group and/or owners of open-source projects that migrated to microservices architecture. This user study will be conducted as a questionnaire in order to reach a higher number of participants and based on related literature [Gil08] [Kro18]. The results of this user study will then be analyzed, and takeaways will potentially result in further changes to the prototype application depending on the severity and estimated time required to implement them.

5 Extensible Repository for Architectural Refactoring Approaches

The most central component of the whole prototype is the extensible repository, which is a knowledge base for existing architectural refactoring approaches to microservices. The extensible repository exists of both a classic repository based on a database holding the refactoring approach data and a back-end application which communicated with this database and provides an API for interacting with the repository. Requirements and expectations are presented in this chapter regarding the extensible repository and its additional application logic. Furthermore, technical aspects such as the data model of the database and the used technology stack are topic of this chapter. The chapter is concluded with the design and implementation aspects regarding the extensible repository.

5.1 Requirements

There are several requirements that have to be taken into account when designing the extensible repository and the back-end application. As already mentioned in the section 4.2 of the methodology chapter 4, these requirements were collected during discussion meetings with the designer of the framework [HF22]. Additional ones were added in an agile way during the implementation of the extensible repository application. They were prioritized following the explanation in section 4.2, where we consider all requirements with priority of four and five necessary so achieve the minimal viable product. This section details these requirements and their rationale.

5.1.1 Data Model and Repository

Repo-R1 Extensible data model

Description The user should be able to extend available refactoring approach attribute options. Therefore, the underlying data model of the repository needs to reflect this by allowing the addition of new attribute options at runtime.

Rationale A static data model is too inflexible in an environment such as microservices where new technologies and refactoring approaches are still actively researched.

Priority 5

Repo-R2 Approach source information

Description The user should be able to see source information about a refactoring approach of the repository. This includes the title of the refactoring approach, the authors, the publication year and optionally a link where the complete publication can be found.

Rationale Since not all information about a refactoring approach of the repository can be summarized shortly, it is important to provide source information about the approach.

Priority 4

Repo-R3 Readable database file

Description A developer or application admin should be able to read the database file of opened in a suitable tool. This means table names are standardized, and additional one-to-one relations can be added to structure related data.

Rationale In case the database file is looked at by a developer or application admin, the database file should be readable, so they can understand the content of the database and more easily find data related to a specific approach they are interested in.

Priority 2

5.1.2 Back-end Application and API

Repo-R4 API to interact with the application

Description The application should provide an API in order to access business functionalities implemented in the back-end.

Rationale The web-based user interface needs a way to interact with the repository.

Priority 4

Repo-R5 Database interaction

Description The back-end application needs to access the repository. It should be possible to use all functionality provided by the database management system (DBMS) such as inserting, querying, updating or deleting data. Additionally, the application needs model classes to internally process database entities and send data from the database through its API.

Rationale To make it possible for the back-end application to provide a connection between the front-end and the repository, it needs to be able to use all database functionalities that are provided by the DBMS. In order to do this, it is also vital that there exist model classes representing database entities in some form in order to communicate the data with the front-end application via the applications API.

Priority 5

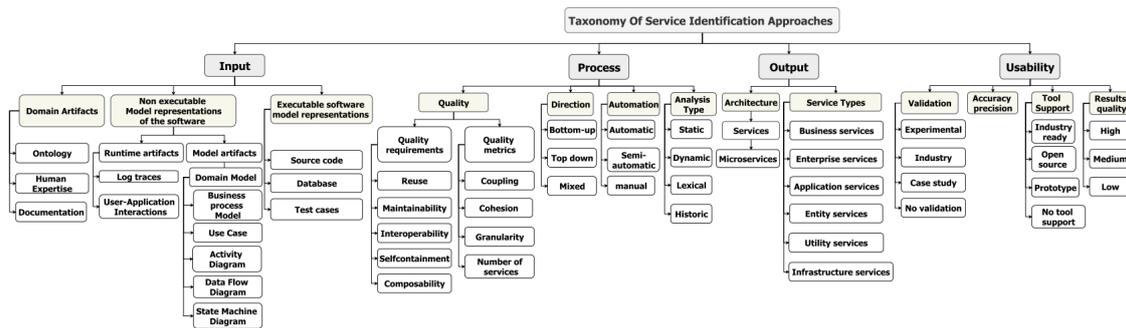


Figure 5.1: Taxonomy of Service Identification Approaches by Abdellatif et al. [ASM+21].

Repo-R6 Data seeding

Description The back-end application should be able to seed data into the database when it is first created.

Rationale This functionality is important in order to make the application usable on its own without the need to roll out a concrete database file that needs to be used in combination with the back-end application.

Priority 3

Dependencies Repo-R5

Repo-R7 Easy adding of new recommendation algorithms

Description The back-end application should provide an easy way to add new recommendation algorithms which can be used aside from the ones developed during this thesis.

Rationale Due to the fact that the concrete recommendation algorithm is not a main focus of this work, we want to provide a way to add new recommendation algorithms without large configuration overhead or the need to define generic interfaces for algorithms.

Priority 4

5.2 Data Model

The data model of the extensible repository is the foundation for the whole back-end application, and in some ways for the whole prototype application. In order to create a fitting data model, we will first define a classification of refactoring approaches which will be the basis for it. Following this, we will describe how the data model is created from the classification and elaborate on the important aspects of said data model. Lastly, we will go into detail about which fields of data we will save for which entities in the database tables.

5.2.1 Refactoring Approach Classification

We base our refactoring approach classification on the taxonomy of SIAs [ASM+21], shown in figure 5.1. Therefore, we also base the description of a refactoring approach on four basic high-level categories, its inputs, outputs, process and usability aspects. Additionally, we will call tables and their data, which describe these four categories *approach attributes*. However, some aspects of this classification are not fit for a direct translation to a data model which we can use for our application. Therefore, and in order to fulfil requirement Repo-R1, we will adjust some aspects of this taxonomy and add some additional aspects.

Inputs

The taxonomy shown in figure 5.1 separated the input category into domain artifacts, non-executables and executables at its highest level. Based on this we will directly adopt the domain artifacts and executables categories in our data model. However, we will split the non-executables category into runtime artifacts and model artifacts, because of the more precise description of their content. Also, since the multi-layered design of the taxonomy is sometimes difficult to translate into a relational data model, it is easier to separate these two tables instead of trying to combine them into a higher level table.

Process

The taxonomy describes a refactoring approach process as a combination of four aspects. These are quality, direction, automation and analysis type, as can be seen in figure 5.1. In our data model we directly adopt the direction, automation and analysis type categories. Since the multi-layered classification of figure 5.1 separates between requirement and metric qualities, we will create a single quality table wherein each quality has a category attribute which is either “requirement” or “metric”. This way one can still differentiate between these two different quality categories, however we don’t require additional database tables. Furthermore, it also enables us to add more quality categories in the future without the effort of creating dedicated database tables for it. Lastly, we extend the process description with an additional attribute, which we will call technique. We think that the process description without the technique attribute lacks descriptive power of how the described migration looks like. We see the most potential for adding new approach attributes to the process category, however an evaluation of further potential attributes to the classification will be considered out of scope for this thesis.

Outputs

When it comes to the outputs of refactoring approaches, we will directly adopt the classification modeled in the taxonomy shown in figure 5.1. Therefore, outputs in our classification and data model also consist of an architecture attribute and a service type.

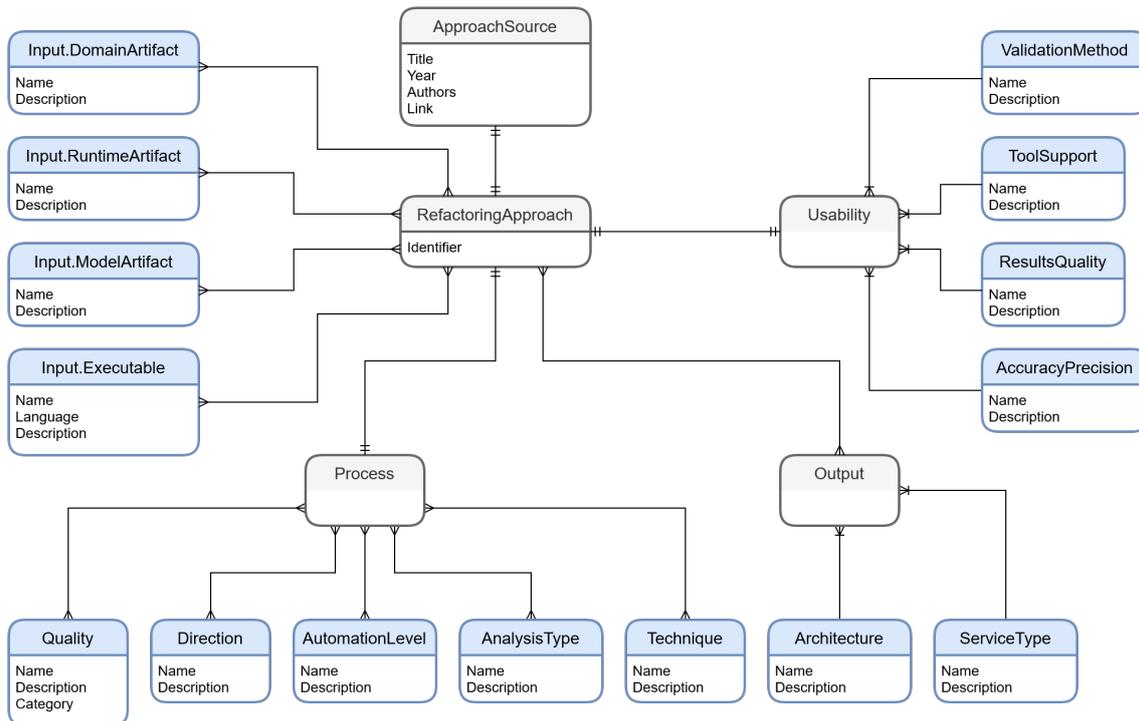


Figure 5.2: The Database Data Model of the Extensible Repository as an Entity-Relationship Model using Crow's Foot Notation [PPJ19].

Usability

Just as with the approach outputs, we will not do any changes to the usability category of refactoring approaches compared to the taxonomy shown in figure 5.1. The usability category exists of four approach attributes, the used validation method of the approach, some information about provided tool support, an evaluation of the results quality and an evaluation of the accuracy/precision of the SIA.

5.2.2 Model Construction

Based on the classification described in section 5.2.1 we will create our relational data model to use in our application's database. An entity-relationship model (ERM) of the complete data model we will be using can be seen in figure 5.2. Approach attributes are colored blue in figure 5.2.

Firstly, even though we try to encapsulate the mentioned four categories of a refactoring approach into separate database tables, this is not possible for the Inputs category. This is due to the fact that we want an approach to have many-to-many relations with each of the possible input attributes. Therefore, we have two different choices to solve this problem. We can save all input entities in one "Input" table of the database and introduce a category field as we did with the process quality. However, this would eliminate the separation of data for each input attribute and would somewhat conflict with requirement Repo-R3 where one could not as easily look up all entities of a specific input attribute. Additionally, we believe that introducing a new input approach attribute can also

easily be done via a new database table due to the chosen technologies for the back-end application and therefore not representing a problem. As a result, we choose to forgo an input table and reflect the nature that domain artifacts, runtime artifacts, model artifacts and executables are inputs in their table name.

The encapsulation into separate database tables works without a problem for the other three classification categories we discussed. The result can be seen in figure 5.2, where we have introduced an “Process”, “Output” and “Usability” table for them.

Lastly, we introduce the “ApproachSource” table as can be seen in figure 5.2 which resolves requirement Repo-R2. Each saved refactoring approach has exactly one such source entity describing metadata of the approach such as its title, authors, publishing year and link to where it can be found.

Table Fields

When it comes to the provided fields of each table, all approach attribute tables have a “Name” and a “description” field. The primary key for all attributes is said “Name” field, except for the input executables. Here, the primary key is a composite key made from the “Name” and the “Language” field.

When it comes to non approach attribute tables, all table’s primary key is an integer ID field which was emitted from the ERM shown in figure 5.2. As an example the primary key of “RefactoringApproach” entities is the “RefactoringApproachId”, the same is equivalent for the other tables. The refactoring approach entity itself has an additional “Identifier” field which is not its primary key but has a unique constraint nonetheless. This Identifier will be set when adding new entities and will be the displayed ID of a refactoring approach, so the database key does not have to be exposed to the application user, but the approach can still be uniquely identified.

Database Content

When looking back at the taxonomy displayed in figure 5.1 we can see that the approach attributes chosen for the data model were all on a classification layer between the four categories and the lowest layers. This is because we will use the provided options of the lowest layer of the classification as *approach attribute options* in our database. Concretely, this means that for each option, there will be a correlating entity in the attribute’s database table. With this and the overall design of the data model, we will successfully fulfil requirement Repo-R1, in which we easily can add additional options in approach attribute tables. The complete list of approach attribute tables and their entities can be seen in appendix A. However, the in the application used descriptions of the entities were omitted due to them being very long in many cases.

5.3 Technology Stack

The back-end application uses some already existing technologies as building blocks for itself. The most important ones will be discussed in this section.

5.3.1 C# and .NET 6

There are various possible programming languages which could be used to implement the back-end application, since they only need to have either native support or third party libraries for implementing REST-APIs. Potential options include Java in combination with Spring Boot or Golang (Go) together with the Gin Web Framework. In comparison to Java and Spring Boot, C# and .NET 6 has less resource consumption and faster Hypertext Transfer Protocol (Secure) (HTTP/HTTPS) response times, making it a more attractive choice [Dha21] [Pre21]. When it comes to Golang, the language is currently vastly younger than C# and therefore missing features like, Generics, Reflection, Lambda expressions, annotations, etc. Also, where .NET 6 already comes as a feature rich framework, Golang need additional libraries for many things, such as an object-relational mapper (ORM) [Yak20]. This makes it so C# and .NET 6 are a preferred choice in comparison to Golang. Additionally, C# and .NET 6 are also a personal preference, since experience with all mentioned options have been made. Therefore, we will be using the C# programming language and .NET 6.

.NET 6

.NET 6 is the latest version of the .NET platform. It is a free and open source developer platform, which also comes with the ability to be used cross-platform for Window, Linux and macOS. Applications for the .NET platform can be written in different programming languages, namely C#, F# and Visual Basic. [Mic22b]. Additionally, .NET 6 already comes with the functionality that is needed to build a REST API without relying on additional libraries.

C#

As mentioned, because we are using the .NET platform we can choose between the programming languages C#, F# and Visual Basic. One of the advantages of the .NET platform is that we could use a combination of all three of these programming options if needed, however, for our application one is sufficient and combing them would only complicate the development without tangible upsides. For our back-end application, we will be using C#. The main reason is that C# is the more modern and better documented programming language compared to Visual Basic and F#. Additionally, most additional technologies and packages we will be using are exclusive to C#.

5.3.2 SQLite

As a way to persist application data, we will use a database. When it comes to databases there are many possible solutions for our application, however we will be using a relational database over a document database, because SQL and therefore many DBMS are used in many large and complex applications, making them some of the most mature and proven software application [SQL22b]. Additionally, many-to-many relations are easily modeled in a relational way but more difficult in a document-based database, since there are multiple ways to do this which would need to be evaluated. Therefore, we will use a relational database over a document-based one. In terms of relation databases there are a couple options, like PostgreSQL, MariaDB, MySQL or SQLite.

For our prototype application, we will be using SQLite because it is the most lightweight and portable options of these. SQLite also doesn't need any installation and can run on almost any device [SQL22c]. When it comes to the limitations of SQLite, the use of it should not cause problems, because the database size will not be very big in this state of the prototype and the features of SQLite are sufficient for our need. Moreover, should the limitations of SQLite become a problem in the future, it will be possible to switch to a feature richer database. This is partly due to Entity Framework Core (EF Core), which will be discussed in the next section, and the fact that switching from a less functional database to a more functional one is easier than the other way around. The reason for this is that SQLite does not support some features provided by other DBMS like, certain types of native queries. Furthermore, SQLite supports fewer data types than most other DBMS which are NULL, INTEGER, REAL, TEXT and BLOB [SQL22a].

5.3.3 Entity Framework Core

EF Core serves in our application as an ORM between C# objects and any of the supported SQL-based databases. Due to this, we will not have to write typical data-access code which normally comes with using any database. [Mic21] As mentioned in the previous section, EF Core enables an easier potential future switch of the DBMS due to the fact that the SQL-query generation is handled by this framework.

5.3.4 Swashbuckle

Swashbuckle is a package providing swagger support to APIs build with ASP.NET Core [Mor22]. It comes with a number of functionalities which will be used during development. It does not have any influence on the production behavior of the application. Via Swashbuckle we can generate an OpenAPI specification conforming description of the API endpoints. This generated file is in the JSON format and in the case of our application named "swagger.json". We also write this generated JSON file to the file system at the start-up of the application, provided we are starting the application outside of production mode. Further details about the usage of this generated JSON file will be discussed in section 6.4, when discussing back-end and front-end integration. Lastly, we can use this generated endpoint description in combination with the SwaggerUI to look at a visual representation of the API endpoints, as well as to easily send test requests to these endpoints for testing purposes.

Listing 5.1 Trivial C# Model Class of the Quality Entity.

```
public class Quality
{
    [Key]
    public string Name { get; set; }
    public string Description { get; set; }
    public QualityCategory Category { get; set; }
}
```

5.4 Design and Implementation

During the development of the back-end application, several design and implementation decisions have to be made. The most important ones as well as some general documentation of how some aspects were realized will be discussed in this section.

5.4.1 Data Model Implementation

Since we use EF Core as an ORM in our back-end application, we will have to implement model classes which can be used by EF Core to create the relational data model described in section 5.2. For this, the data model in figure 5.2 is already designed with this task of creating object-oriented classes in mind. Here, we can translate each entity of the diagram into a respective model class. Attributes of the entities can also be directly adapted into class fields. Therefore, a first trivial translation of for example the “Quality” entity can be seen in listing 5.1 [Smi21].

In case where entities have a single primary key instead of a composite primary key we can declare one field as the key, in the case of the quality class we chose the name field. With model classes implemented like this, we could already create a partial correct database, where we could create tables for each entity. However, these tables would not have any relations between them [Mic21].

In order to add the designed relationships into the model classes, we will have to use one-to-one, one-to-many and many-to-many relations. To add a one-to-one relation between two classes or tables in EF Core we have to reference the object in both classes. Additionally, at least one foreign key is referenced to ensure only one dependent is related to each principal. This concept can be seen in listing 5.2 for e.g. the ApproachSource class. In case of one-to-many relations similar rules apply, but instead of both sides referencing the other class we need to have a list, in the example code of listing 5.2 an `ICollection`. Additionally, we will not need to reference the foreign key. Logically following this, in the case of many-to-many relations, both sides have a `ICollection` of the respective class. We don't need to create model classes for join tables of many-to-many relations because these will be created automatically by EF Core. With these additional relationships defined, a complete and functioning data model can be created from the model classes [Mic21].

In order to create the database file based on our source code, we need to implement a database context class which inherits the `DbContext` class. This context class is a core part of using EF Core, since it is needed to define which model classes we want to create tables of as well as

Listing 5.2 C# Model Class of the RefactoringApproach Entity.

```
public class RefactoringApproach
{
    [Key]
    public int RefactoringApproachId { get; set; }
    public string Identifier { get; set; }

    public int ApproachSourceId { get; set; }
    public ApproachSource ApproachSource { get; set; }

    public ICollection<DomainArtifactInput> DomainArtifactInputs { get; set; }
    public ICollection<RuntimeArtifactInput> RuntimeArtifactInputs { get; set; }
    public ICollection<ModelArtifactInput> ModelArtifactInputs { get; set; }
    public ICollection<ExecutableInput> ExecutableInputs { get; set; }

    public int ApproachProcessId { get; set; }
    public ApproachProcess ApproachProcess { get; set; }

    public ICollection<ApproachOutput> ApproachOutputs { get; set; }

    public int ApproachUsabilityId { get; set; }
    public ApproachUsability ApproachUsability { get; set; }
}
```

Listing 5.3 Manual Configuration of Many-To-Many Relation with Custom Join Table Name.

```
builder.HasMany(left => left.ApproachOutputs)
    .WithMany(right => right.RefactoringApproaches)
    .UsingEntity(join => join.ToTable("JoinTable.Approach.Output"));
```

interacting with the database data, which will be discussed later. Furthermore, we define which SQL-related DBMS we want to use for our database and where this file will be saved to. Lastly, further configurations such as database logging can be done on this context class as well as any configurations or definitions regarding our data model. This is important in cases where one wants to adjust the automatically created data model by making attributes required or unique attributes or other similar configurations. Adding our needed model classes to this database context only requires us to create a `DbSet<>` field of the respective model class.

Since entity and join table names will be based on class names, we want to manually set them to better fit our requirement Repo-R3. In order to do this for entity tables, we can use class attributes such as `[Table('Approach.Process')]` for defining the table name of the “Process” entity [Mic22a]. Adjusting the join tables names is more difficult. Here, we need to modify the `onModelCreating` method of our database context class and manually define and configure the many-to-many relations.

Listing 5.4 C# Source Code to Create a new Technique Entry in the Database and its Translated SQL Statement.

```
var db = new RefactoringApproachContext();
Technique newTechnique = new Technique
{
    Name = "new technique",
    Description = "description of the new technique"
};

db.Techniques.Add(newTechnique);
db.SaveChanges();
```

```
INSERT INTO "Approach.Process.Technique" ("Name", "Description")
VALUES ('new technique', 'description of the new technique');
```

Such an example configuration can be seen in listing 5.3, where we can see how the join table name joining the “RefactoringApproach” and the “Output” entities can be changes via the `UsingEntity()` method.

5.4.2 Business Logic and Database Integration

The majority of the business logic is implemented outside the API controllers. This is the case because we implement a service layer which provides all needed functionalities the REST API exposes. Next to the provision of most business logic, the service layer’s job is also to handle the database integration specified in requirement Repo-R5. Fortunately, due to EF Core this database integration becomes easier than having to write typical data-access code ourselves. However, even if the SQL statement construction itself will be handled by EF Core we still have to tell EF Core what data we want to add, change or query. This is done mostly via the already mentioned `DbContext`.

In our application we need all common create, read, update and delete (CRUD) operations SQL databases provide aside from the update operation. This is because we won’t allow changing approach attribute entities, and the update functionalities related to the approach entity itself are insertions and deletions of join tables. Therefore, we will go over each of the other CRUD operations we use in the application and how they can be handled with the help of EF Core.

Create Operation

There are many entities which can be created in the application with varying degrees of additional involved complexity. The simplest entities to create are approach attributes, such as an approach process Technique.

Listing 5.5 C# Source Code to Read and Output Entity from Database and its Translated SQL Statement.

```
var db = new RefactoringApproachContext();
int outputId = 1;

ApproachOutput? query = db.ApproachOutputs
    .Where(e => e.ApproachOutputId == outputId)
    .Include(e => e.Architecture)
    .Include(e => e.ServiceType)
    .FirstOrDefault();
```

```
SELECT "a"."ApproachOutputId", "a"."ArchitectureName", "a"."ServiceTypeName",
"a0"."Name", "a0"."Description", "a1"."Name", "a1"."Description"
FROM "Approach.Output" AS "a"
INNER JOIN "Approach.Output.Architecture" AS "a0" ON "a"."ArchitectureName" =
"a0"."Name"
INNER JOIN "Approach.Output.ServiceType" AS "a1" ON "a"."ServiceTypeName" =
"a1"."Name"
WHERE "a"."ApproachOutputId" = 1
```

Listing 5.4 shows what the source code to create such an approach attribute can look like and how EF Core created a SQL statement from this code to execute the action. In order to add a new entity we need to call the `Add()` method for the `DbSet` of the entity we want to add with the new entity as its parameter. After adding the entity, we then have to call `SaveChanges()` on the database context object, which will result in executing the in listing 5.4 shown SQL statement.

Obviously, creating a new `RefactoringApproach` in the database requires more additional logic. First, we check if all used approach attributes of an approach that wants to be added already exist in the database. Depending on if the approach attribute already exists, we will either add it if it does not exist or if it does already exist, we query them from the database into an in memory object. After this step, we have a new `RefactoringApproach` object with all in the database existing approach attributes. Now we can add this object to the database, which will automatically create the respective `RefactoringApproach` and `ApproachSource` entries, as well as add entities to join tables for its many-to-many relations.

Read Operation

There are two equally common tasks when reading data from the database on our application. We either want to read all data of a table or we want to read a single specific entity. The source code in listing 5.5 shows what needs to be implemented in order to query an `ApproachOutput` object with the ID "1". Similarly, all entities can be queried using this method with the difference being that we

Listing 5.6 C# Source Code to Delete an existing Technique Entry in the Database and its Translated SQL Statement.

```
var db = new RefactoringApproachContext();
Technique techniqueToDelete = new Technique
{
    Name = "new technique",
    Description = "description of the new technique"
};

db.Techniques.Remove(techniqueToDelete);
db.SaveChanges();
```

```
DELETE FROM "Approach.Process.Technique"
WHERE "Name" = 'new technique';
```

would swap the `FirstOrDefault()` method call for a `ToList()` method call, and we would remove the `Where()` part in the source code since we don't want to only find one specific ID. When the application runs this source code, it will translate it to the SQL statement also seen in listing 5.5.

Delete Operation

Deleting entities from the database is relatively easy on paper. Listing 5.6 show that, similarly to create new entities, in order to delete an entity we only need to call `Remove()` on the `DbSet` of the entity we want to delete with an object representing said entity. This deletion is then also followed by an `SaveChanges()` method call, signaling that the SQL statement can be generated and executed by EF Core.

In practice, the deletion is a bit more complicated for our application. To begin, we don't want to allow deleting approach attributes which are currently related to existing refactoring approach entities. Therefore, we need to check if this is the case before deleting and stop the deletion process if it is. On the flip side, we also don't want to delete approach attributes when refactoring approach entities are deleted. Fortunately, this can be prevented by not cascading on delete, even if it means that we need to delete entities such as the `ApproachSource` manually.

5.4.3 API and Controllers

The final piece of the back-end application is the in requirement Repo-R4 defined controller layer of the API. This layer has two main functions. On the one hand we define which concrete endpoints we want to have in our REST API and on the other hand it provides exception handling and translated exceptions to expected HTTP/HTTPS status codes which are returned.

In our application, we differentiate between five different controllers which provide API endpoints for their respective area of responsibility. The first four controllers are the `ApproachInputController`, the `ApproachOutputController`, the `ApproachProcessController` and the `ApproachUsabilityController`. These controllers function very similarly, with the only difference being the approach attributes they are responsible for. Each controller offers endpoints to create via the HTTP POST method, read via the HTTP GET method and delete via the HTTP DELETE method. As an example, we can retrieve a list of existing domain artifact inputs by sending an HTTP GET request to <http://<api-url>/api/vl/inputs/domain-artifacts>, where “<api-url>” represents a placeholder for whichever uniform resource locator (URL) the API is hosted on. In order to create a new domain artifact input, we will need to send an HTTP POST request to <http://<api-url>/api/vl/inputs/domain-artifacts> with its request body being a JSON representation of the new domain artifact input. Deleting an existing domain artifact input is possible by sending an HTTP DELETE request to <http://<api-url>/api/vl/inputs/domain-artifacts/<name>>, where “<name>” represents the name of the domain artifact input we want to delete. All other approach attributes of each of the mentioned controllers works the same way under a different URL endpoint.

In contrast, the `RefactoringApproachController` provides significantly more endpoints and functionalities. Generally, this controller also provides the already seen functionalities of creating, reading and deleting refactoring approaches. Additionally, we also provide an endpoint to read a single refactoring approach based on a given ID. Updating an existing approach is done through the API via either adding and removing approach attributes, or in the case where only one attribute can be assigned at a given point, we override the current attribute by the desired one. Concretely, for input and process attributes, this means that the `RefactoringApproachController` exposes endpoints to add or remove for each of their attributes. As an example, adding a new domain artifact input to an existing approach can be done by sending an HTTP POST request with the object to <http://<api-url>/api/vl/approaches/<id>/inputs/domain-artifacts>. Removing a domain artifact input can then be done by sending an HTTP DELETE request to the <http://<api-url>/api/vl/approaches/<id>/inputs/domain-artifacts/<name>> endpoint. Similarly, this can be done with approach outputs. However, the difference is that outputs are not approach attributes but a higher level entity consisting of the `Architecture` and `ServiceType` attribute. Therefore, these two approach attributes can not be added to an existing approach separately. Aside from this, the functionality remains the same as for inputs and processes. Furthermore, an existing approach always has exactly one references entity for approach attributes, which falls under the usability aspect. This means that we will not provide an endpoint to remove these attributes from an approach, rather we provide an endpoint which overrides the current referenced entity by the desired one. As an example, changing the validation method of an approach can only be done by sending an HTTP POST request to <http://<api-url>/api/vl/approaches/<id>/usabilities/validation-methods> with the desired validation method in the request body. Lastly, the `RefactoringApproachController` also provides an endpoint at <http://<api-url>/api/vl/approaches/recommendations> for getting approach recommendations based on either an already defined filter preset or a custom configured filter in the request body. Presets can be selected using the “preset” query parameter. This endpoint will always prioritize a selected preset over the configured filter in the request body.

The other task of the controllers of handling runtime exceptions is done by introducing a `ServiceExceptionFilter` that can be seen in listing 5.7, which we apply to all controller classes. This filter implements the `IExceptionFilter` interface that allows us to implement the `OnException()` method, which enabled us to execute our exception handling generically for all controller methods without implementing the needed exception handling for each controller method. In the filter

Listing 5.7 C# ServiceExceptionFilter Class used to Handle Exceptions in Controller Classes.

```

public class ServiceExceptionFilter : IExceptionHandler
{
    public void OnException(ExceptionContext context)
    {
        Console.WriteLine(context.Exception);
        switch (context.Exception)
        {
            case EntityNotFoundException:
                context.Result = new NotFoundObjectResult(context.Exception.Message);
                break;
            case DuplicateElementException:
                context.Result = new ConflictObjectResult(context.Exception.Message);
                break;
            case EntityReferenceException:
                context.Result = new
BadRequestObjectResult(context.Exception.Message);
                break;
            case DbUpdateException:
                context.Result = new
BadRequestObjectResult(context.Exception.InnerException?.Message);
                break;
        }
    }
}

```

we can then define which exception should result in which returned HTTP status code and also adjust the error message if needed. This filter can be enabled for a controller class by adding the “[TypeFilter(typeof(ServiceExceptionFilter))]” class attribute.

5.4.4 Recommendation Logic

One core concept of the in this thesis developed application is the approach recommendation functionality.

Recommendation Filters

In order to give recommendations, we first need to gather information regarding which approach attributes are important for the recommendation. For this, we categorize each approach attribute into three different categories describing how to handle them in regarding to the approaches’ suitability:

Listing 5.8 C# IRecommendationService Interface to Implement Recommendation Algorithms.

```
public interface IRecommendationService
{
    public IEnumerable<ApproachRecommendation> GetApproachRecommendations(
        ApproachRecommendationRequest recommendationRequest,
        int numberOfRecommendations);

    public IEnumerable<ApproachRecommendation> GetApproachRecommendations(
        RecommendationPreset recommendationPreset,
        int numberOfRecommendations);
}
```

- **Include:** The approach should have this property
- **Neutral:** The approach can have this property
- **Exclude:** The approach should not have this property

We then define a recommendation filter as a set of approach attribute information, containing for each attribute which of the three mentioned categories applies for them.

Recommendation Service and Algorithm

Since the concrete recommendation algorithm is not one of the main focuses of this thesis, it is important to provide a way for flexible recommendation algorithm implementation as is required based on requirement “Repo-R7”. For this purpose we provide an IRecommendationService interface which can be seen in listing 5.8. Implemented services of this interface can then be used for the recommendation process in the application. The interface provides two methods for evaluating approach recommendation. One where a recommendation filter is used as a parameter and one where instead a filter preset is given, and the recommendation filter will then be build from it. This interface also provides the possibility to specify the number of recommendations you want to have, which can based on the implementation be sorted by the calculated suitability of each approach. The provided methods then specify that a set of approach recommendation are returned by the algorithm. In the approach, an approach recommendation consists of the entity ID and Identifier of the evaluated approach, the source data of the approach, a given suitability score of the approach and the complete evaluation information for each approach attribute. The approach attribute evaluation can have one of four states. The concrete cases where which evaluation state is picked depends on the implementation of the recommendation algorithm, but for our implementation the following allies:

Algorithm 5.1 Algorithm Calculating the Recommendation Suitability of Refactoring Approaches.

```

procedure EVALUATEAPPROACHSUITABILITY( $\mathcal{A}_{all}, \mathcal{I}_{Rec}$ )
  for all approach  $\in \mathcal{A}_{all}$  do
    attributeCount  $\leftarrow 0$ 
    matchCount  $\leftarrow 0$ 
    neutralCount  $\leftarrow 0$ 
    mismatchCount  $\leftarrow 0$ 
    for all attr  $\in$  approach.attributes do
      attributeInfo  $\leftarrow$  FINDATTRIBUTEINFO( $\mathcal{I}_{Rec}, attr$ )
      attributeCount  $\leftarrow$  attributeCount + 1
      if SHOULDINCLUDEATTRIBUTE(attributeInfo) then
        matchCount  $\leftarrow$  matchCount + 1
      else if SHOULDEXCLUDEATTRIBUTE(attributeInfo) then
        mismatchCount  $\leftarrow$  mismatchCount + 1
      else
        neutralCount  $\leftarrow$  neutralCount + 1
      end if
    end for
    hitCount  $\leftarrow$  matchCount + mismatchCount
    if EVALUATENOTENOUGHINFO(attributeCount, hitCount) then
      approach.suitabilityScore  $\leftarrow -1$ 
    else
      approach.suitabilityScore  $\leftarrow \frac{matchCount \times 100}{attributeCount}$ 
    end if
  end for
end procedure

```

- **Match:** An attribute is evaluated as a match if the attribute is categorized to be included in the recommendation filter and is also part of the approach being evaluated.
- **Neutral:** An attribute is evaluated as a neutral if the attribute is categorized to be neutral in the recommendation filter and is also part of the approach being evaluated.
- **Mismatch:** An attribute is evaluated as a mismatch if the attribute is categorized to be excluded in the recommendation filter and is also part of the approach being evaluated.
- **Error:** An attribute is evaluated as an error under two conditions. First, if the given filter category is not known so if it is nether “Include”, “Neutral” or “Exclude”, or if There are no filter information for the approach attribute is given at all. This is the case since the default value for approach attributes is “Neutral”, so no information is not intended by the application as of the current implementation.

For this thesis, we implemented a simple algorithm for the recommendation service inside a `SimpleRecommendationService` class which implements said `IRecommendationService` interface. The rough idea behind the suitability score calculation of our implementation can be seen in algorithm 5.1, even if as already mentioned the implementation of the service methods does some more things. In order to calculate a suitability score for each approach, we count some attribute

metrics, namely the attribute, match, neutral and mismatch count. Then, based on these counts, we check if we have enough information to calculate a suitability score. This can not happen if the attribute count is zero or if the sum of matched and mismatches is under a defined threshold, in which case the suitability score will be set to “-1”. The threshold in our implementation is currently one, but can be adjusted if needed. If a suitability score can be calculated, we calculate a percentage ratio of matched attributes and the total attribute count.

Filter Presets

As mentioned, we allow recommendation filter presets for commonly used configurations. The application currently provides three different presets based on the approaches input [BBGG21]:

- **New application:** The new application preset aims to find approaches which rely on human expertise and business or use-case modeling.
- **Re-Build:** This filter preset also looks for approaches that involve a combination of human expertise and business or use-case modeling, but in addition software documentation and further models describing the application will also be included here.
- **Re-Factor:** This filter preset look for approaches which rely on runtime artifacts such as log files or executable inputs such as an application’s source code, database files or existing test cases.

These presets are defined via a JSON file describing the filter, which follows the object used in the `IRecommendationService` to describe a custom filter. In the `RecommendationPresetBuilder` these JSON files are then deserialized into a valid object and attributes for which no information is provided in the filter definition will be defaulted to be handles neutral.

5.4.5 Data Seeding of the Database

As determined in requirement Repo-R6 we want to have data seeding in our back-end application. Data seeding is the process of populating a database with an initial set of data. In the application, we differentiate between two different types of data seeding, which also differentiate in how they are implemented.

The first type is the seeding of approach attribute data. To do this, we will utilize the EF Core native way of adjusting the `OnModelCreating()` method of the database context. For better readability of the source code, we will do all additional model configuration in separate configuration classes which implement the `IEntityTypeConfiguration<>` interface and apply each configuration to the model builder of the `OnModelCreating()` method. Listing 5.9 shows how such a configuration looks like for the `Technique` model class. In these configuration classes, we define the data which we want to seed via the `HasData()` method [Vog21]. This will be done for all other approach attributes in the same way. Unfortunately, with this method, we can only define data for one concrete table of the database. This means that in order to seed data for entities in many-to-many relations, we would additionally need to seed that join table of these entities. This means that it becomes difficult to comprehend how, e.g. one complete seeded refactoring approach entity looks like, since the data will be split into multiple different configuration classes.

Listing 5.9 Example Configuration Class of Technique Model Class.

```
public class TechniqueConfiguration : IEntityConfiguration<Technique>
{
    public void Configure(EntityTypeBuilder<Technique> builder)
    {
        builder.HasData(
            new Technique
            {
                Name = "Wrapping",
                Description =
                    "A black-box identification technique that encapsulates the
                    legacy system with a service layer without changing its implementation. The wrapper
                    provides access to the legacy system through a service encapsulation layer that
                    exposes only the functionalities desired by the software architect."
            },
            ...
        );
    }
}
```

Therefore, the second type of seeded data is refactoring approach data. In order to seed refactoring approach data, we will implement a different custom approach. For this we will implement a `DataSeeder` which will run at the end of the applications' startup phase. In the process of seeding, we will first check if there are already refactoring approach entities in the database, which indicates that there is no need to execute the data seeding and the seeding will finish. If the database is empty, we will proceed and read and deserialize the refactoring approach data from a JSON file. Then the service functionality to create new refactoring approaches will be used to create database entities for all approaches of the seed data. If an approach which will be added this way uses an approach attribute which was not already seeded via the conventional EF Core way described earlier, a database entity for this approach will also be created in the process.

As we have seen in this chapter, the extensible repository needs more program logic than a database file and a way to interact with it. Here, the designed data model is most important as the foundation of the extensible repository. We have also shown how vital the designed API and business logic is to the overall functionality of the repository and meeting the defined requirements. Additionally, we explained which technologies we need and use in order to implement the back-end application. In the following chapter, we will similarly discuss the user interface and its role and implementation.

6 Web-based User Interface

The most essential part of the prototype from the user's point of view is the web-based user interface built on top of the extensible repository. In this chapter we will go over requirements the different user have for this user interface, the used technologies and design and implementation aspects. Lastly, we will explain how the integration of the user interface and the extensible repository is done in our application.

6.1 Requirements

This section describes the requirements of the various users for this part of the application prototype. There are two fundamentally different user groups for the application, although it is possible to have users that fall into both categories. The first is the far more common user who uses this interface to look at existing refactoring approaches or recommendations. The requirements for this user will be the subject of the section on the user interface. The other user that serves more of a niche is the administrator user. This user can interact with the extensible repository in different ways, which will be discussed in the section on the administrator interface. Both user interface requirements and administrator interface requirements will be implemented in one web-based application.

The same as for the requirements in section 5.1 the requirement collection was done during interviews with the designer of the architectural refactoring framework [HF22] before starting the implementation as well as during the development process. Additionally, we also consider the user interface as a minimal viable product as soon as all requirements with a priority of four and five were implemented. In the following, we will detail the collected requirements and their rationale.

6.1.1 User Interface

UI-R1 Choose framework phase

Description The user interface should show an image of the current stage of the framework. Additionally, based on this displayed image, the user should be able to choose which phase of the framework they want to engage with, resulting in a navigation to the respective user interface view of the selected phase.

Rationale Since the tool's functionality is based on the architecture refactoring framework, we want to display this framework and familiarize the user with this framework, even if most implemented functionalities are exclusively tied to phase 2 of this framework.

Priority 3

UI-R2 Show UI Mock-ups for the first framework phase

Description When selecting the first phase of the framework, the user interface should show user interface mock-up images of how an implementation could look like.

Rationale Since we allow the user to choose a phase of the framework, we will have to provide some user interface for the phases which are not implemented yet. Therefore, because UI mock-ups for phase 1 already exist, we want to display these, so a user can get a general idea of which functionality is planned for phase 1 of the framework.

Priority 2

Dependencies UI-R1

UI-R3 Provide feedback when selecting the third framework phase

Description When selecting the third phase of the framework, the user interface should show some kind of feedback. Because there are no mock-ups for phase 3, yet this feedback should somehow inform users that the design of phase 3 has not started.

Rationale Since we allow the user to choose a phase of the framework, we will have to provide some user interface for the phases which are not implemented yet. However, unlike phase 1, phase 3 does not have any user interface mock-ups and therefore the feedback can be simpler than for phase 1.

Priority 1

Dependencies UI-R1

UI-R4 List all existing approaches

Description The user interface should be able to display a list of all existing refactoring approaches of the repository.

Rationale Listing all known approaches is a simple way for a user to explore the existing data. It is important to have a way for users which look for specific data or just wish to explore known approaches to access it without having to go through the recommendation functionalities of the application.

Priority 4

Dependencies Repo-R4

UI-R5 Display all information of an approach

Description The user interface should be able to display all information related to a specific refactoring approach.

Rationale There should be a way for users to view all information related to a refactoring approach. This way, only approach identifying information can be listed in list elements where multiple approaches are displayed, and a user will not be overloaded with too much information if they are uninterested in it.

Priority 5

Dependencies Repo-R4

UI-R6 List recommended approaches based on filter

Description The user interface should be able to display a list of recommended approaches. These approaches are based on a defined filter where it is defined whether an approach attribute is wanted, neutral or unwanted. Based on this filter, a recommendation algorithm will calculate the recommendation list.

Rationale The main purpose of phase 2 of the architectural refactoring framework and the application is to help define the migration strategy by showing for the user's use-case interesting refactoring approaches. Therefore, it is required to display these interesting recommendations in the user interface.

Priority 5

Dependencies Repo-R4, UI-R9, UI-R10

UI-R7 Suitability of recommended approaches

Description The user interface should display a suitability value for recommended approaches, based on how suitable they are in regard to the defined recommendation filter.

Rationale The calculation and display of a suitability value for approaches aims to improve understandability of the recommendation process for the user. Additionally, based on this suitability, we can sort the recommendations in order to show the most suitable refactoring approaches first.

Priority 5

UI-R8 Indication of how suitability algorithm works

Description The user interface should indicate how the suitability algorithm works. This can be done by displaying matching and mismatching attributes of each recommended approach in relation to the defined filter.

Rationale Just like the suitability value itself, the indications of how this suitability value is calculated aims to improve understandability for the user.

Priority 3

UI-R9 Configuration of a custom filter

Description The user interface should provide a way for users to define a custom filter where they can select which approach attributes they are interested and uninterested in.

Rationale Since a filter is required in order to suggest refactoring approaches, we need a way for users to define one based on their use-case.

Priority 5

UI-R10 Recommendation filter presets

Description The user interface should provide a way to select already defined preset filters based on which recommendations will be searched.

Rationale There are several possible use-cases where a user does not have an already defined use-case for which they want to search approaches. Therefore, we want to provide some filter presets for some common filter configurations for users that just want to get an overview of existing approaches with rough filtering.

Priority 3

6.1.2 Administrator Interface

UI-R11 Add new approaches

Description The application should provide a way to add new refactoring approaches to the repository.

Rationale The application provides the easiest and best guided way to add new approaches to the repository. Without this way to add new approaches, the only other ways would be directly through an HTTP/HTTPS request to the API which would be a lot more time-consuming or directly through the database which would be even harder.

Priority 5

Dependencies Repo-R4

UI-R12 Edit existing approaches

Description The application should be able to edit and update existing approaches

Rationale There are several use-cases where one would want to edit an existing refactoring approach. Therefore, we want to provide an easy way to do so in the user interface.

Priority 4

Dependencies Repo-R4, Repo-R6, UI-R11, UI-R17

UI-R13 Delete existing approaches

Description The application should provide a way to delete existing approaches.

Rationale In cases where we want to delete existing approaches, the user interface should provide an easy way to so. Deleting through the API is not very difficult when compared to adding new approaches or editing existing ones. Therefore, this requirement does not have a high priority.

Priority 2

Dependencies Repo-R4, Repo-R6, UI-R11, UI-R17

UI-R14 Add new approach attributes

Description The application should provide a way to add new attributes a refactoring approach can have.

Rationale Since the back-end application and the extensible repository is designed to add new attributes for approaches, we want to provide a way in our user interface to do so.

Priority 5

Dependencies Repo-R4

UI-R15 Delete existing approach attributes

Description The application should provide a way to delete existing approach attributes. This is only possible if these attributes are not assigned to any existing approaches.

Rationale Adding new attributes is only one part of an extensible repository. The other part of it is the ability to remove existing attributes. For this, we want to provide a way to do so in the user interface.

Priority 4

Dependencies Repo-R4, Repo-R6, UI-R14

UI-R16 Export existing approaches

Description The application should provide a way to export the list of existing approaches to a file.

Rationale This export is provided to make changed to the data model easier to handle. In this case, an administrator can export all approaches and adjust the exported file to fit a new data model and import them again.

Priority 2

Dependencies Repo-R4

UI-R17 Import approaches from file

Description The application should provide a way to import approaches from a given file.

Rationale This functionality provides an alternative way to add approaches. In cases where a valid approach is already modeled in a file, we can just import the approach instead of having to go through the user interface to do so.

Priority 1

Dependencies Repo-R4

UI-R18 Toggle administrator functionalities

Description The user interface should have a way to toggle administrator functionalities on or off.

Rationale Since the administrator functionalities are implemented and integrated into the user interface, we want to provide a way to toggle administrator specific functionality on and off. This also provides a base for potential future user management, where this toggle can be set based on the privileges of a certain user. Additionally, since we don't want to expose the participants of the user study to admin functionality, this requirement will make creating a user study version much easier.

Priority 3

6.2 Technology Stack

The front-end utilizes a few already existing technologies in order to streamline and simplify development. The concrete used technologies will be discussed in this section.

6.2.1 Angular

There are many possible technologies that we could use when developing the web based user interface. Some considered options include React.js, Vue.js, Angular or, since the extensible repository is written with .NET 6 and C#, Razor-based view templates. However, razor-based view templates were ruled out due to wanting to clearly separate client and server code and functionality. At this point the decision between React, Vue and Angular comes down mostly to personal preference and already existing experience with Angular.

The Angular framework will be the dominant technology we use for our web-based user interface. Here we will be using the most recent version to the date of this thesis, Angular 13. The reason the user interface doesn't require many technologies is due to Angular, which comes with many functionalities that are commonly used during web development, such as routing.

Angular combines three different languages into one framework. The structure of browser views is modeled via HyperText Markup Language (HTML) files serving as templates of how certain components are structured. The styling of different elements is done through a style file, most of the time Cascading Style Sheets (CSS). However, other languages extending CSS such as Sassy CSS (SCSS) or syntactically awesome style sheets (Sass) can also be used for this purpose. In our application, we will be using SCSS because this will be needed when creating custom color themes based on Angular Material, which we do in our application. Therefore, to stay consistent, we use SCSS everywhere even if the extended functionality is not needed in most components. Finally, the functionality of components, services or other functions used is done in Typescript.

6.2.2 Angular Material

In combination with the Angular framework, we will use Angular Material. This is a library of commonly used and pre-styled components that conform to the Material Design specification. These components and Material Design will help us keep our user interface consistent without having to design a separate style for each frequently used element. Also, new elements we need that don't yet exist in Angular material could be more easily designed to look and feel like existing ones, since there is a lot of reference material and the Material Design specification.

6.2.3 Angular Flex-Layout

Angular Flex-Layout is a third party layout API utilizing Flexbox CSS and mediaQuery. The Flex-Layout engine automates the process of applying the appropriate Flexbox CSS to the view hierarchies used in the browser. This removes the traditional complexities and workarounds which one would normally encounter when applying manual layout CSS to their application view. In conclusion, this library vastly simplifies layouting challenges we would encounter when implementing and stylizing the browser view of our application.

6.2.4 ng-openapi-gen

The ng-openapi-gen NPM module enables us to generate model interfaces and web service clients based on a OpenApi 3 specification. As an input, it takes either a JSON or YAML file of a OpenApi specification and produces typescript interfaces based on model classes as well as angular services which provide functionalities in order to send requests to the specified API endpoints. Further details regarding the use of this module will be described in section 6.4.

6.3 Design and Implementation

The look and feel of a user interface is one of the most important aspects for front-end applications. To maximize the usability of the user interface, we followed the material design style guide designed by Google [Goo22]. This was made easier by using the in section 6.2 discussed Angular Material library, which provides pre-built components stylized to conform to Material Design. Figure 6.1 displays how as an example the view to create a new refactoring approach looks like based on this design guidance.

When it comes to the implementation and more technical design, this section, will go over design and implementation details regarding the front-end application. We will go into detail about the structure of the application, routing and navigation logic and design, and other specific functionalities provided by the application.

6.3.1 Home Page and Architectural Refactoring Framework Phases

The complete front-end application structure and its possible routes can be seen in figure 6.2. The web application starts at a home page component, which displays the framework from section 2.3. A redirection to this home page will be done upon opening the web page. Additionally, from here a user can choose which phase of the framework they want to interact with, as is described in requirement UI-R1. Since the application in this thesis mostly focuses on phase 2 of the architectural refactoring framework, most of the applications features will be reached from the phase 2 route as can be seen in figure 6.2. Aside from selecting phase 2, choosing phase 1 or phase 3 will display the functionalities described in the requirements UI-R2 and UI-R3 of section 6.1.

6.3.2 Creating a new Refactoring Approach

Following the selection of phase 2, figure 6.2 shows that there are three routes that can be followed from this point onwards. One of these routes is only available when the administrator interface is enabled, which is the form view to create a new refactoring approach. This creation view utilizes the ApproachFormComponent, which is also used for editing existing approaches for. The concrete view can also be seen in figure 6.1 to demonstrate how the user interface looks as an example. Furthermore, the creation view provides the functionalities described in requirements UI-R11, UI-R14 and UI-R15.

Going through with a successful creation of a new refactoring approach will navigate to a different view and display a detailed view of the created approach, as shown in figure 6.2. On the other hand, canceling the creation process will route the application back to the home page.

6.3.3 List of existing Refactoring Approaches

The second route the phase 2 overview provides is to view a list of all existing approaches. This view contains the functionality for requirement UI-R4 as well as UI-R13 if administrator functionalities are enabled.

There are also multiple possibilities to navigate to different parts of the interface. If the administrator functionality is enabled from this view, one can go to edit one of the listed refactoring approaches. Additionally, selecting an approach will display a more detailed view of it. Lastly, users can go back to the phase 2 overview page and continue differently from there.

6.3.4 Detailed View of existing Refactoring Approach

The detailed refactoring approach view is an essential part of the front-end application. Here all information and approach attributes in combination with their description are displayed for a selected refactoring approach as is specified in requirement UI-R5. Additionally, a viewed approach can also be deleted as requirement UI-R13 describes if administrator functionality is accessible. Lastly, if this view is opened from the recommendation list a suitability score, as described in requirement UI-R7, is displayed as well.

The view provides a back-button which routes to a different point of the application depending on from where this view was navigated to. This means that the button will navigate to the list of refactoring approaches if its view was opened by selecting an approach of the list, or it will go back to the creation view if the view was opened by creating a new refactoring approach. The same is true for the list of recommended refactoring approaches. Additionally, one can switch to the editing page of the viewed refactoring approach.

6.3.5 Editing an existing Refactoring Approach

Editing an existing refactoring approach like creating a new one via the ApproachFormComponent. The difference is that this form is not empty, but filled in with the data of the approach being edited. The editing page provides functionality for requirements UI-R12, UI-R13, UI-R14 and UI-R15.

A successful edit can be saved from here and will not result in navigating to a different route. Also, similarly to the creation page, canceling the editing will result in the loss of any unsaved changed and route back to the application's home page.

6.3.6 Recommendation Overview

The last route which can be followed in phase 2 is the section providing all functionalities related to the recommendation process. The recommendation overview provides the possibility for users to select the defined preset filters described in requirement UI-R10 in order to receive approach recommendations. This recommendation algorithm works for all presets just as was already elaborated in section 5.4.4 and shown in algorithm 5.1.

Selecting such a preset will directly route to the list of recommended refactoring approaches. Alternatively, via a configure-button, users can go to the view where they can define custom recommendation filters. Lastly, a back-button, like is present on many other user interface views, allows users to navigate back to the phase 2 overview.

6.3.7 Custom Recommendation Filter Configuration

One way to apply recommendation filters is by using user defined custom ones. This view provides this functionality as already specified in requirement UI-R9.

After defining a custom filter, users can go forward to the result list of recommended refactoring approaches by clicking on the search-button. Furthermore, this view also provides a back-button which will go back to the recommendation overview.

6.3.8 List of recommended Refactoring Approaches

The last view of the application to discuss is the result list of recommended refactoring approaches after applying a recommendation filter, as was described in requirement UI-R6. Additionally, in the list of recommended refactoring approaches the in requirement UI-R7 described suitability score is displayed. Lastly, every list item is expandable to show a description indicating how the suitability score of the expanded refactoring approach was calculated, therefore complying to requirement UI-R8.

As is the case for most other user interface views, the recommendation list also has a back button which navigates back to the recommendation overview.

6.3.9 Header Toolbar

The whole application has a toolbar header at all times. Here the application name is displayed followed by a home button. This home button can be used from any view of the application and will route back to the home page. These routes are not displayed in figure 6.2 due to the universality of this functionality, which would make the diagram less readable and more cluttered.

Additionally, a settings-button is present in this toolbar which will open the settings dialog of the application. Since the application does not have any user management, this settings button is visible to all users, even if its functionality is exclusively related to administrator functionality. The settings dialog provides functionalities for both the export and import of refactoring approaches, as specified in requirements UI-R16 and UI-R17. Additionally, based on requirement UI-R18, the toggling of the administrator interface is done in this settings dialog.

6.3.10 Administrator Interface toggle

As mentioned in section 6.1 the complete front-end application is split into two sets of functionalities. One set for the functionalities related to the user interface and one set for the functionalities of the administrator interface. As described in requirement UI-R18, it is important to make these functionalities toggleable and adjust the interface, respectively.

From an implementation point of view, all buttons or otherwise clickable elements which provide the router links colored red in figure 6.2 will be made inaccessible. This is done by removing said HTML elements from the HTML Document Object Model (DOM) on the condition that the administrator functionalities are disabled. Listing 6.1 shows how this condition is stored in an angular service named `PermissionService`, which holds the boolean in a subject following the publish-subscribe pattern under the name `_isAdminSubject`. The service is then declared as an injectable service class via the `@Injectable` annotation, as can be seen in listing 6.1, in order to allow all application components to access the service via dependency injection. This will cause the application to create one global instance of the service and ensure that the global state of the publish-subscribe subject is the same throughout the whole application [Ang22]. Then application components can either get if administrator functionality is currently enabled or disabled, enable or disable the functionalities or subscribe to the subject and provide what to do in case the value of the subject changes.

Listing 6.1 Angular Source Code of Permission Service.

```
@Injectable({
  providedIn: 'root'
})
export class PermissionService {
  private _isAdminSubject: BehaviorSubject<boolean> = new BehaviorSubject<boolean>(true);

  get isAdmin(): boolean {
    return this._isAdminSubject.value;
  }

  set isAdmin(value: boolean) {
    this._isAdminSubject.next(value);
  }

  subscribeToIsAdmin(observer: Partial<Observer<boolean>> | undefined): Subscription {
    return this._isAdminSubject.subscribe(observer);
  }
}
```

6.4 Integration with Back-end and Repository

Since the back-end application provides a REST API which the user interface will interact with, the integration of the two applications depends mainly on the web-based front-end application knowing where to access this API and what functions it provides. The question of where the API can be accessed is not difficult to answer for our application, since we define the URL for the back-end in the environment file of the application. Here we use one environment file for development building and one for building the application in production. Knowing the provided functionality of the API is also not difficult. However, if we just reimplement the model classes of the back-end application and write services with functions for all known endpoints, we will reduce some quality aspects of our application such as maintainability. This is due to the fact that all changes to the model classes and API functionalities will have to be implemented and maintained twice. Once in the back-end application and then in the front-end application. Therefore, we use both Swashbuckle and ng-openapi-gen to improve this integration process.

As described in section 5.3 regarding Swashbuckle, we generate and write to file system a file named “swagger.json”, which contains an OpenAPI specification conforming description of the API endpoints. This file is written if the back-end application is started outside of production mode. In order to achieve this, we need to modify the .csproj file with the additions shown in listing 6.2. Here, we use the “swashbuckle.aspnetcore.cli” .NET tool to generate and write the mentioned JSON file. Additionally, we introduce a variable called “GenerateSwagger” in order to deactivate the swagger generation. This will be useful later for running the application in Docker, since we will not have to install the .NET tool. This is preferred, since we don’t want to generate the file in production anyway and this file would be useless in Docker.

Listing 6.2 Additions to .csproj-File needed to Export the “swagger.json” File.

```

...
<PropertyGroup>
  <GenerateSwagger>true</GenerateSwagger>
</PropertyGroup>
...
<Target Name="PostBuild" AfterTargets="PostBuildEvent">
  <Exec Command="dotnet tool restore" Condition="$(GenerateSwagger)" />
  <Exec Command="dotnet swagger tofile --output swagger.json $(OutputPath)$(AssemblyName).dll v1 "
  Condition="$(GenerateSwagger)" />
</Target>
...

```

Listing 6.3 The JSON Configuration File used for “ng-openapi-gen” Package.

```

{
  "$schema": "node_modules/ng-openapi-gen/ng-openapi-gen-schema.json",
  "input": "../ArchitectureRefactoringHelper/Repository/swagger.json",
  "output": "api/repository",
  "ignoreUnusedModels": false
}

```

Now that we have a swagger.json file describing the API endpoints, we can look at using it to automatically generate model classes and API services for the user interface application. We will achieve this with the help of ng-openapi-gen as described in section 6.2. As an input, we take the JSON file and produce typescript interfaces based on our model classes as well as angular services which provide functionalities in order to send requests to the specified API endpoints. The configuration file is shown in listing 6.3. Here we define the used OpenAPI schema, the input file, the output folder for generated files, and we define that even unused models in the “swagger.json” file should result in model classes. Now we can generate with the “npm run openapi-gen” command when inside the user interface project folder. In order to use the generated services, we will only have to import the generated Angular module into the application’s Angular module. Here, we can also configure the URL the API runs at for the generated services.

Listing 6.4 shows how the generated services can now be used in source code. All service methods return observable objects of the returned data to enable asynchronous execution.

In conclusion, we have seen the functionality the user interface provides for both its user and administrators based on the defined requirements and their design and implementation in this chapter. We have discussed technologies which shaped the implementation process and saw how both the user interface and the extensible repository communicate with each other. In the next chapter, we will discuss one way of how the complete application bundle can be build, which we also used to build and deploy the instance for the following application evaluation.

Listing 6.4 Example Typescript Source Code showing how Generated Services can be used to Add a new Refactoring Approach.

```
let newRefactoringApproach: RefactoringApproach;
...
this.refactoringApproachService
  .addRefactoringApproach({
    body: newRefactoringApproach
  })
  .subscribe({
    next: (value: RefactoringApproach) => {
      this.refactoringApproach = value;
      ...
    },
    error: (err) => {
      ...
    }
  });
```

7 Build and Deployment of the Tool

Just implementing the discussed application is not sufficient to also use them comfortably. Especially since we want to host an instance of the implemented application on a web server for the evaluation in chapter 8. Therefore, in order to use the implemented applications, we will first have to build and if needed deploy them. To do this, there are two main ways to build and run the application. It can either be run locally or containerized in Docker. However, local execution is only intended for development purposes, and we will not discuss this way in detail. For further information on running the application locally, consult the README file of the GitHub repository of the developed application.[Hal22]

In order to make the complete application run in Docker, we will have to do several steps. First, we have to set up Docker files for both the .NET 6 back-end and the Angular front-end applications. Then we will set up the reverse proxy NGINX also via a Docker file and a custom configuration file. The job of this reverse proxy will then be to route incoming HTTP/HTTPS traffic to the correct Docker container, which will run either the back-end or front-end application. Lastly, we will set up Docker Compose for orchestrating the three services.[Ram22] The complete Docker architecture can be seen in figure 7.1.

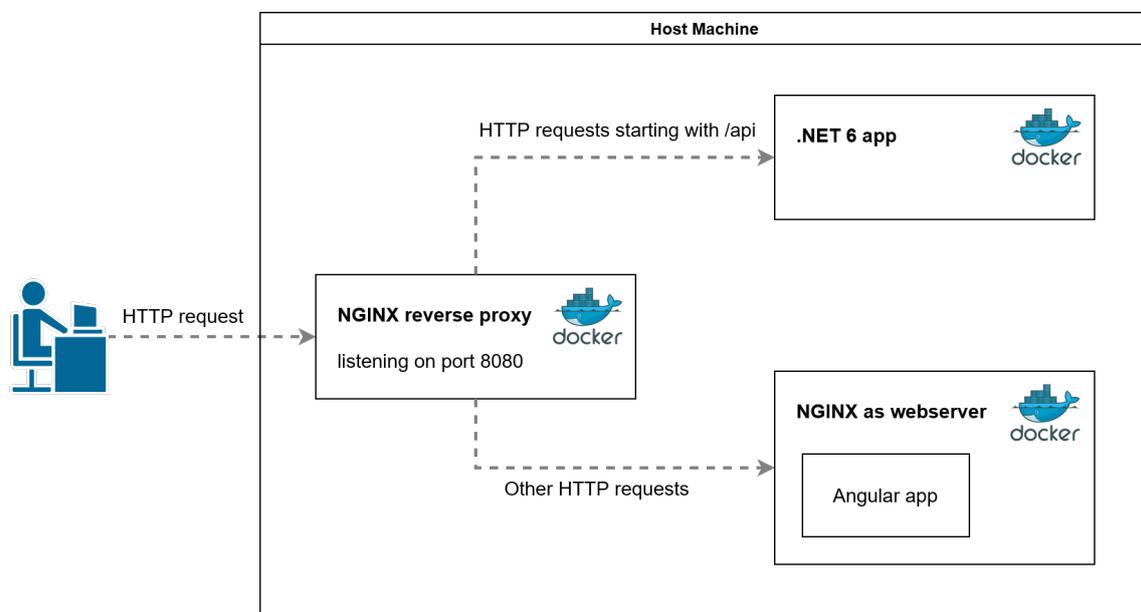


Figure 7.1: Docker Architecture of the Application.

Listing 7.1 Dockerfile of the NGINX Proxy.

```
FROM nginx:alpine
COPY "./conf/default.conf" "/etc/nginx/conf.d/default.conf"
```

Listing 7.2 NGINX Configuration File for the Reverse Proxy Component.

```
upstream frontend {
    server client;
}
upstream backend {
    server api;
}
server {
    listen 80;
    location / {
        proxy_pass http://frontend;
    }
    location /api {
        proxy_pass http://backend;
    }
}
```

7.1 Reverse Proxy

The Docker file for the NGINX proxy is very straight-forward. We only need to pull the base NGINX image and copy our custom configuration into the `/etc/nginx/conf.d` directory. In our custom configuration, we want to configure that based on all incoming HTTP/HTTPS requests, every request that begins with `"/api"` will be routed towards the Docker container where the back-end application runs on. Likewise, all other HTTP/HTTPS request should be routed towards the Docker container with the Angular application.

This functionality can be seen in listing 7.2 where the custom configuration file is shown. Here, we define two upstream fields, one for the front-end application called `"fe"` and one for the back-end application called `"be"`. These reference the services called `"client"` and `"api"` which we will later define in the docker-compose file in section 7.4. The `"server"` attribute is the configuration of the NGINX server itself. Here we define through `"location /api"` where HTTP/HTTPS request beginning with `"/api"` should route to. The same is true for the `"location /"` attribute, which tells the server to route every other request towards the front-end service.[Ram22]

7.2 User Interface Application

Containerizing the user interface application requires us to do two things. First, we need to build the application in the Docker container. As seen in listing 7.3 this mostly consists of installing application dependencies via `npm install` and building the application with the `npm run build`

Listing 7.3 Dockerfile used to Containerize the Angular based User Interface Application.

```

FROM node:alpine AS build
WORKDIR /app

RUN npm install -g @angular/cli

COPY ./package.json .
RUN npm install
COPY . .
RUN npm run build --configuration=production

FROM nginx:alpine AS final
COPY "/nginx/default.conf" "/etc/nginx/conf.d/default.conf"
COPY --from=build /app/dist/ui /usr/share/nginx/html

```

--configuration=production command. Additionally, we have to copy the generated binaries, precisely the index.html and JavaScript files onto the NGINX web server, which takes over the task of hosting these static files. This NGINX instance is a different one from the one we use for the reverse proxy of section 7.1. Furthermore, in order to configure this NGINX web server correctly, we override the web server configuration with a custom configuration. This custom configuration is required in order to have the application’s routing function correctly, since we configure that NGINX should forward all URI requests to the index.html instead of trying to route itself.[Rai18]

7.3 Extensible Repository Application

The containerization of the back-end application requires two basic steps. First we build and publish the solution within a Docker container that provides the .NET 6 SDK which correlate to the “build” and “publish” parts of the Docker file that is shown in listing 7.4. Here, we also set the “GenerateSwagger” flag to false so that we don’t have to install the respective Swashbuckle tool. After this, we copy the generated binaries into a different Docker container, which contains the .NET 6 runtime. This copying and the definition of the entry point can be seed in the “final” part of the Docker file.

7.4 Service Orchestration with Docker Compose

For the orchestration of all the defined Docker containers, we use Docker Compose.[Doc] Docker Compose is a tool that allows us to define and run multi-container Docker applications. A YAML file is used to configure the application’s services, which can be seen in listing 7.5. In the Docker Compose file we define our three services, the proxy service, the “client” service and the “api” service. Since we already defined Docker files for each service, we can now just reference them in the build section of each service. Additionally, we define the ports by which one service can be reached on from outside the container. Lastly, we define that services should always restart in order to let for example a crashed container restart without manual intervention.

7 Build and Deployment of the Tool

Listing 7.4 Dockerfile used to Containerize the .NET 6 based Back-End Application.

```
FROM mcr.microsoft.com/dotnet/aspnet:6.0 AS base
WORKDIR /app

FROM mcr.microsoft.com/dotnet/sdk:6.0 AS build
WORKDIR /src
COPY "Repository/Repository.csproj" "Repository/"
RUN dotnet restore "Repository/Repository.csproj"
COPY . .
WORKDIR "/src/Repository"
RUN dotnet build "Repository.csproj" -c Release -o /app/build-p:GenerateSwagger=false

FROM build AS publish
RUN dotnet publish "Repository.csproj" -c Release -o /app/publish

FROM base AS final
WORKDIR /app
COPY --from=publish /app/publish .
ENTRYPOINT ["dotnet", "Repository.dll"]
```

Listing 7.5 Docker Compose File used to Configure the different Application Services.

```
version: "3"
services:
  proxy:
    build:
      context: ./Proxy
      dockerfile: Dockerfile
    ports:
      - "127.0.0.1:8080:80"
    restart: always
  client:
    build:
      context: ./UI
      dockerfile: Dockerfile
    ports:
      - "9000:80"
    restart: always
  api:
    build:
      context: ./ArchitectureRefactoringHelper
      dockerfile: Dockerfile
    ports:
      - "5000:80"
    restart: always
```

To conclude, in this chapter we have discussed how to create containerized versions of the implemented applications. For this we have used a reverse proxy container routing incoming HTTP/HTTPS traffic to the containerized instances of the back-end and front-end applications. Additionally, we used Docker Compose to orchestrate the setup and communication of these three containers. Now that we are able to build and deploy both of the developed application to any infrastructure we want with the help of Docker, we can deploy and host our application for potential users to evaluate. The design, results and discussion of this evaluation will be the topic of the next chapter.

8 Evaluation of Web-Based Application

In order to answer our third research question, we will conduct a user study of the developed application, where potential users will use the application and evaluate it. This evaluation, as discussed in chapter 4, is done via a questionnaire. In this chapter, discusses the survey and its results. Furthermore, we will discuss the results and try to find further potential improvements to the prototype application. Additionally, the application evaluated via this user study will exclusively provide the user interface functionalities and not any of the administrator interface ones. This is the case because the potential user base for administrator positions is too small at the moment and therefore was evaluated with a smaller of users during the development process of the application.

8.1 Questionnaire Design

The evaluation questionnaire is separated into six different sections. We will discuss the content of each section in addition to explaining the rationale behind the surveys questions and the given tasks the user should solve.

1) Start Section

The start section which can be seen in figure B.1 of appendix B has the title of the questionnaire and a brief description of the created and purpose of it. Additionally, on this start section, participants will have to agree that their information from the survey can be used in this master thesis and future related publications by the University of Stuttgart. This condition has to be accepted before being able to continue and participate in the questionnaire.

2) Personal Information

In the second sections, we will collect a very small amount of personal information about the participants. Here, we don't care about their demographic data, since we don't think that these should have much influence on the perception of the tool. Additionally, we don't plan on comparing how different demographics perceive the tool. In this questionnaire, we are interested how different software architects and developers with experience of microservices evaluate the application. Therefore, we only collect general professional experience of the participants, experience directly with microservices and their current professional role, as can be seen in figure B.2 of appendix B.

3) Exploration of the Tool

The purpose of sections three is to introduce the user to the tool they are going to evaluate in the following sections. Here, we reference where the web-based application can be found via a direct link to a hosted instance of it. Moreover, we present two tasks to the user which we ask them to solve. The purpose of these tasks is to give participants a bit of guidance regarding which parts of the tool they should interact with in order to answer the coming questions. Naturally, participants can explore the tool further if they desire to do so, since solving the tasks only provides the user with a minimal amount of experience with the tool in order to answer the questions of other sections. The complete section can also be found in figure B.3 of appendix B.

For the first task, we want the user to interact with the recommendation presets functionality of the user interface. For this, we ask “How many approaches qualify for a "Re-Factor" activity with a suitability of 100%?”. The intended workflow for this task is that a user selects the second phase of the architectural refactoring framework, then clicks on the button “Find recommended approaches” and clicks on the ”Re-Factor” recommendation preset. After this, the user should be able to see the results-table with the recommended refactoring approaches. Here, the user should see that the correct answer to the first task is “3”.

The second task is designed to use the custom configuration for refactoring approach recommendations. Therefore, we ask “Which approach has a suitability of 100% for the following requirements?”. This question is followed by a description of which options they should include or exclude in the recommendation filter. This description can also be seen in figure B.3 of appendix B. The workflow for this task begins the same as for the first task until the user can select a recommendation preset. Here, the user should click on a button with the text “Configure” on it. This will lead to the view for configuring a custom filter. After selecting the options the way we described in the task, the user can click on the search button, which will result in the same results-table from the first task with different recommendations. Lastly, the user should see that the only approach with a suitability of 100% is the approach “Service Cutter: A Systematic Approach to Service Decomposition” [GKGZ16] with its ID being “11”.

The results of these tasks will be asked in the later tool usability section.

4) Tool Structure

The fourth section is the first sections about the evaluation of the tool. It consists of seven questions regarding the general structure and helpfulness of several aspects of the tool, as can be seen in figure B.4 of appendix B. One of the things we want to evaluate in this section is if the architectural refactoring framework is an understandable way of summarizing common migration activities, and if the user interface functionality that is intended for phase 1 and implemented in phase 2 is even considered helpful in this regard. Additionally, we want to evaluate how helpful the recommendation functionality is. Here, we are interested in both the evaluation of the predefined configuration and the custom configuration of recommendation filters. Lastly, we want to get feedback on the implementation of the recommendation algorithm. We are both interested in if users can comprehend how the recommendation algorithm selects and evaluated recommended approaches, as well as if the by the algorithm calculated suitability percentage is considered helpful

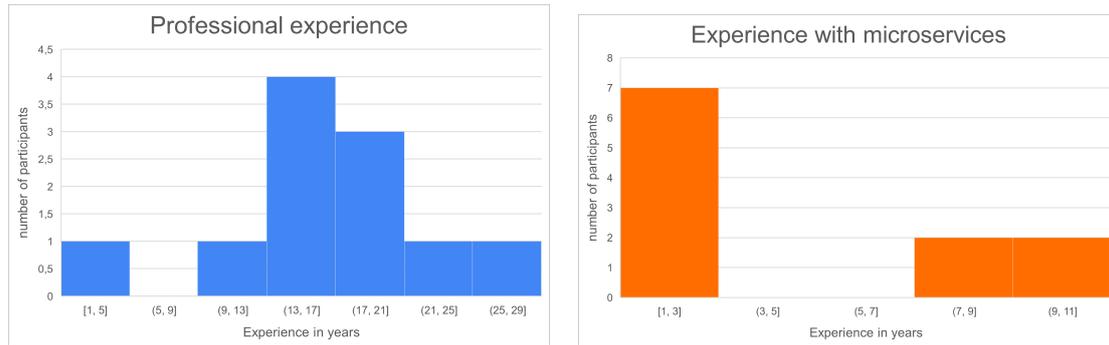
by users. Each answer in this section can be answered on a scale of one to five, where one means that a given aspect has been evaluated as bad and five means that a given aspect has been evaluated as very good. Based on this, the following questions were chosen for this section:

- Is the depicted framework an understandable way of summarizing the common migration activities?
- How helpful do you rate the intended functionality in phase 1 (System Comprehension)?
- How helpful do you rate the functionality in phase 2 (Strategy definition)?
- How helpful do you rate predefined configurations such as “New Application Development”, “Re-Build” and “Re-factor”?
- How helpful do you rate the “Configure” functionality to filter for custom requirements?
- Could you comprehend how the selection of recommended approaches was done by the tool?
- How helpful do you find the calculated percentage on the results page?

5) Tool Usability

Following the tool structure section, we want to look at how users evaluate the usability of the tool. Figure B.5 of appendix B shows the complete usability sections of the evaluation questionnaire. There are several aspects when it comes to usability. One of them is how easy and intuitive it is for users to solve specific tasks. Therefore, part of the usability section are the answers for the in sections 3 defined tasks. In addition to this, we also ask users how easy they perceive each given task to be. This difficulty evaluation is done on a scale of one to five, where one means a task was very easy and five means a task was very difficult. The answers to the tasks and the difficulty questions were optional in case a user didn't solve them. Furthermore, we want to evaluate how intuitive and easy it is for a user to navigate the tool in addition to their overall opinion on the usability of the tool. These questions will also be evaluated on a scale of one to five, with one meaning the navigation and usability were bad and five meaning it was very good. Lastly, we provide a long-answer field for additional recommendations and feedback regarding the tool's usability. The questions we ask in this section were as follows:

- How difficult was it to solve the first task?
- How difficult was it to solve the second task?
- How intuitive and easy is the navigation in the tool?
- How would you rate the overall usability of the tool?
- Do you have additional recommendations or feedback regarding the tool's usability?



(a) Professional Experience of Participants.

(b) Experience with Microservices of Participants.

Figure 8.1: Histograms of the Participants Demographics.

6) Additional Functionality

The last section, which can be seen in figure B.6 of appendix B, is used to get further feedback regarding potential additional functionalities and other feedback that did not fit any questions beforehand. We ask about the following four potential future features specifically:

- Functionality to use system specifications / results of phase 1 as an input for the filter in phase 2
- Functionality to save entered information/progress in form of project
- Multi-user management (Login/Logout/Session Management)
- Functionality to gather user feedback or star-ratings for certain approaches to see other users' opinions

Additionally, we provide a long-answer field where we ask “Which other general features would you like to have implemented in the tool?” in case participants have any specific feature requests already in mind. To finish the questionnaire off, we ask “Do you have other feedback that did not fit to any of the questions above?” for any other opinions that participants want to give.

8.2 Questionnaire Results

In this section, we will present the results of the user study questionnaire and analyze them.

8.2.1 Participant Demographics

There were eleven participants in our user study which filled in the questionnaire. The average participant of the user study has 16.8 years of professional experience, and 4.5 of these years professional experience with microservices. There was one participant with one year of professional experience, which was an unusual outlier, since all other participants had between 13 and 26 years of professional experience. This can also be seen in figure 8.1a. When it comes to experience

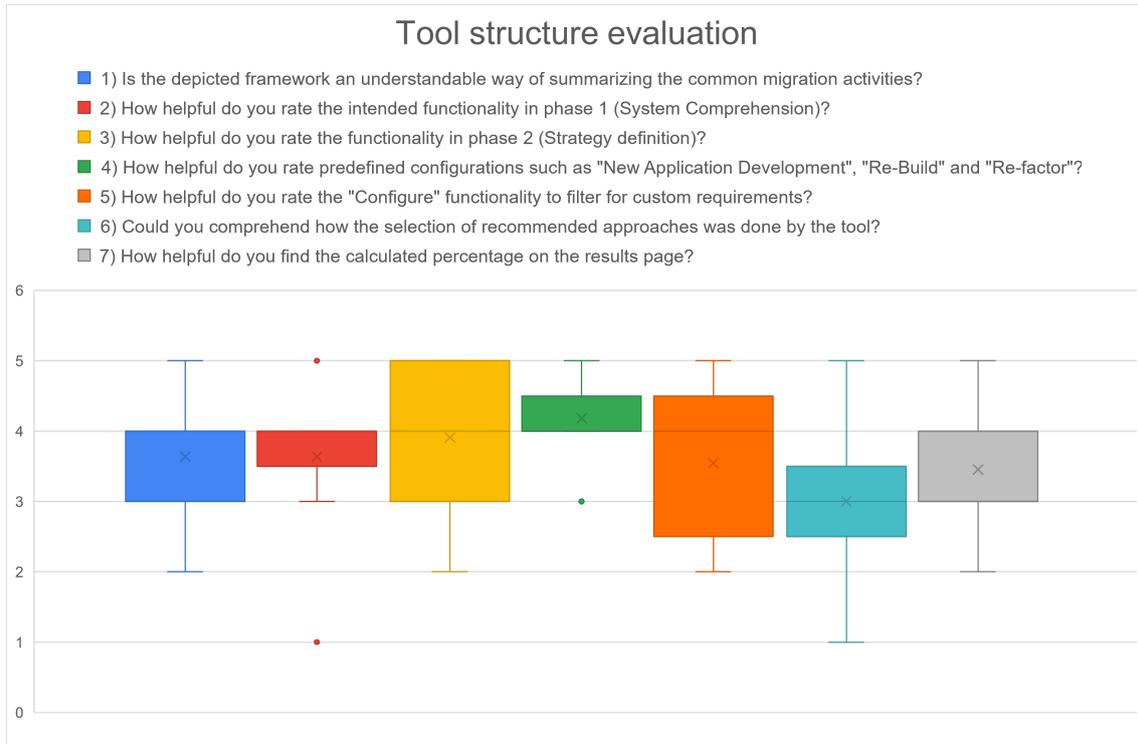


Figure 8.2: Box Plot of the Evaluation Results regarding the Tools Structure.

with microservices, each participant had as a minimum one year and as a maximum 10 years of experience. The distribution of microservices experience can be seen in figure 8.1b and shows that there are two clear distinct groups. One group with four or fewer years of experience with microservices and the other group with seven or more years of experience. The average participant is a software architect, with six out of eleven participants stating it as their professional role. All five other participants had a different professional role. These other roles were software developer, project manager, Chief Information Officer (CIO), researcher and professor.

8.2.2 Tool Structure

The evaluation of the tool's structure can be seen in figure 8.2. When it comes to the helpfulness of the intended phase 1 functionality and the existing phase 2 functionality, phase 2 was rated slightly more helpful with an average rating of "3.9" compared to phase 1 which was rated at "3.6".

The recommendation filter preset functionality was evaluated the highest, with "4.2". Compared to this, there were differing opinions on the helpfulness of custom filter configuration, since their standard deviation was the highest of any question asked in this questionnaire at 1.16. The functionality itself was rated with an average of "3.5".

Understanding how the recommendation algorithm evaluates the suitability of refactoring approaches was rated the worst of the tool structure questions with a rating of “3”. Also, with a standard deviation of 1.04, one of the highest in the survey, opinions seem to vary greatly depending on the participant. On the flip side, the usefulness of this approach suitability concept was evaluated higher at “3.5”.

8.2.3 Tool Usability

The first part of the tool usability section is the task evaluation. Of the eleven participants, seven participants answered the questions related to the first task. For the first task, six out of seven participants, which relates to 85.71%, were able to solve it correctly. Additionally, everyone except one person evaluated the difficulty of task 1 as “1” on a scale of one to five, with the only person not doing so evaluation it as a “2”. This results in an average difficulty of task 1 of “1.14”.

One of the participants who answered the questions for the second task will not be included in the list of participants who attempted the second task. This is the case because they did not solve the task correctly and answered with the approach ID “42” even though the highest approach ID is 18. Additionally, they did not evaluate the difficulty of the task. Therefore, we conclude that the task was deliberately miss-answered, and we will therefore not include them in the evaluation of task 2. This means there are six participants who attempted to solve task 2 and four out of these six participants or 66.67% solved the task correctly. However, one person who we count towards a correct solve couldn’t find the approach ID and answered the correct approach title instead. The average difficulty of task 2 was evaluated as “2” overall, interestingly when we only consider participants who solved the task correctly the average difficulty is higher at “2.5”. The highest answer difficulty evaluation was “4” and the lowest was “1”.

The second part of the tool usability evaluation is an assessment of both how intuitive the navigation is and the general usability. Figure 8.3 shows the evaluation of participants. Both the navigation and the usability were evaluated identically by nine out of eleven participants. For the other two participants, the navigation was marginally better than the usability. The navigation intuitiveness was evaluated at an average of “3.7” and the usability with an average of “3.5”.

There was a lot of additional feedback regarding the usability of the tool. The most important feedback which was mentioned by three different participants is that they had a hard time understanding the approach attribute options. It was stated that the used terms should be described in more detail than how they are at the moment. Additionally, regarding an easier understandability of the user interface, it was suggested that explanation videos could be used to explain the functionality of the framework phases or other functionality. In the same area, a brief explanation of each approach in text form was suggested to be useful.

8.2.4 Additional Features

Figure 8.4 shows the evaluation of the four additional features we suggested for the application. The feature evaluated as the most helpful one was functionality which allows users to save and maybe also load entered information or progress. It has an average rating of “4.2”. Both the feature to use

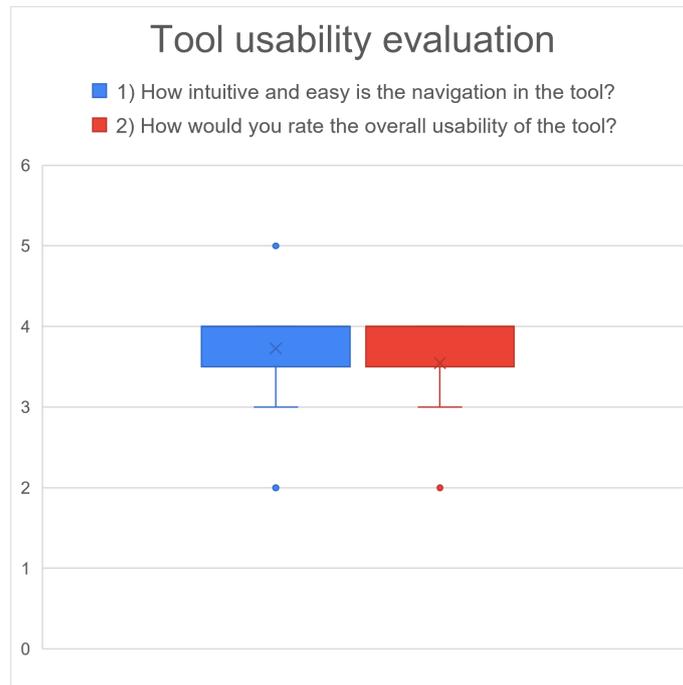


Figure 8.3: Box Plot of the Evaluation Results regarding the Tools Usability.

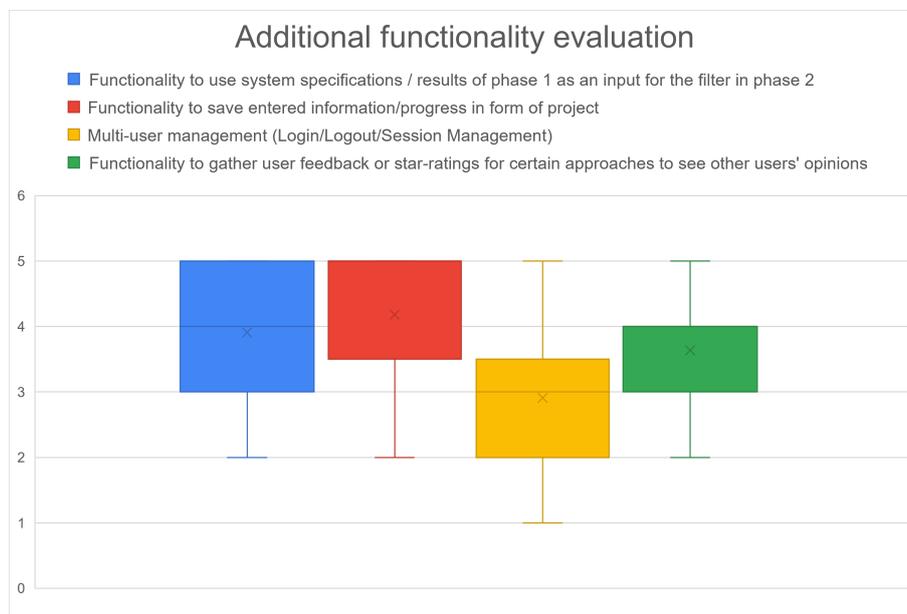


Figure 8.4: Box Plot of the Evaluation Results regarding Potential Future Additional Functionalities of the Tool

results of phase 1 of the framework as input for phase 2 and the feature where user feedback and opinions to approaches can be shared, were evaluated as useful with an average of “3.9” and “3.6”. In comparison, user management was rated the least useful with an average of “2.9”.

When it comes to additional features, which were not listed in the questionnaire, all text answers, which were two answers, were related to more information about the specific refactoring approaches. Some examples were availability of commercial support or size of supported code cases. Further, in other questions, which provided a free form text answer, information regarding the domain of the approach and the size of the team which created it was mentioned as interesting information by one participant. Also, two participants suggested extending the approach list with approaches from more sources aside academic papers like books, articles, blog posts or industry practices.

8.2.5 Other Feedback

There were two types of feedback which were not related to the above-mentioned topics. The first one was regarding the overall usefulness of the tool, aside from how it was realized and implemented. Here, one participant generally does not believe that the migration decision process can be modelled within a tool. Another participant mentioned that approaches linking to articles behind a paywall reduces the value of the tool greatly.

The other topic is the usability of the questionnaire. Only one participant gave feedback in this regard, however, they had difficulty navigating and answering the questionnaire. The reason for this is that they perceived no way of jumping back and forth between the questionnaire and tool, which made solving the tasks difficult for this participant.

8.3 Discussion of Results

In this chapter, we take a look at the analysis of the results and discuss their implications. This includes whether participants found the functionality of the architectural refactoring framework and its implementation useful, the usability of the tool, and which further improvements to the tool should be made in order to improve it.

8.3.1 Participant Demographics

As seen in the results, there are two groups of participants when it comes to their experience with microservices. However, since the total number of participants is eleven, we believe there can be no strong statement made when comparing these two. This is the case because such a comparison would compare a group of seven participants and a group of four. We deem these numbers too low to have much significance, and therefore will try to discuss general trends for the complete group of participants.

8.3.2 Tool Structure

When it comes to the helpfulness of the overall tool, a few people questioned how helpful or if it is helpful at all. However, the majority of participants found the tool overall to be decently useful, even if it needs a couple of improvements. Therefore, it seems that there is a market for a tool such as the one developed in this thesis.

Users exceptionally liked the provided recommendation filter preset functionality. The reason for this is that it takes less time and knowledge about the tool and refactoring approaches as a whole to get suggestions for your use-case. Further, it takes over the task of interpreting the importance of individual approach attribute options for common use-cases. Based on this, it might be a good idea to allow users to import these presets into the custom filter configuration so that they can continue from there. This could allow users to iterate upon the given presets and modify them to more properly fit their individual use-case. This is also reflected by the worse evaluation of the custom configuration functionality, which could also be improved through such a functionality. Based on the participants' feedback, one of the biggest problems of the custom configuration is the understanding of the displayed options of approach attributes. It is not completely clear if participants could not understand the options because they could not find how to show their description in the user interface, if said description was not elaborative enough or the way of explaining options with the help of a description was not liked. We think that improving the overall usability of the tool could also result in a better understanding of these options. Anyhow, improving the description of options is always a goal to increase the tool understandability, and therefore it might make sense to extend descriptions with links to more information.

Regarding the recommendation algorithm, the expectation was that users might have difficulty comprehending how exactly it works. The results confirm this expectation somewhat in that the comprehension answers ranged from user could not at all comprehend how the algorithm worked to they could fully comprehend the algorithm. However, since the overall helpfulness of the suitability was considered high, we think this feature might be worth iterating on more. This could be done by a better recommendation result description of each approach. Here ratios and percentages of the matches and mismatches in relation to the amount attributes could be displayed, e.g. if there were four matches input options out of six total the approach has we could display "4/6 (67%)" next to the input header. Another idea could be displaying the concrete suitability calculation at the bottom of the description window.

8.3.3 Tool Usability

Even though the overall usability of the tool was evaluated as good, there can still be a lot of improvement in this regard. A few specific participants has a hard time understanding how to use the tool. Most of them could figure it out through the descriptions of the user interface and time. Additionally, the tasks given in the user study were considered rather easy, even the most difficult use-case which was described in task 2 was considered generally easy. This, together with the information on how users evaluated the navigation of the tool, leads us to believe that the usability of the tool is in a good spot. However, some usability improvements would still be interesting since they would also help other shortcoming of the current application. One such usability feature is the introduction to some sort of functional explanation or tutorial. The most trivial way to do so would be through embedded videos explaining functionality and ideas in the sport where this functionality

occurs. Another idea could be a tutorial overlay which runs at the first usage of the web-page until it is closed by the user. This tutorial overlay could then also be reused through a hoverable button, again displaying the overlay in case of users wanting to reevaluate what it explained.

8.3.4 Additional Features

Additional features which we provided as options in the questionnaire are not intended to be implemented during this thesis, but as an evaluation of potential future features. The functionality to use the results of phase 1 as an input for phase 2 is already a likely future feature regardless of the participants answers, since the goal is to eventually provide tool support for the complete architectural refactoring framework. Even though, participants still evaluated this feature as helpful. The feature of saving information or progress during the tool usage is an almost universally liked feature and therefore should take a high priority. The multi-user management was not seen as very helpful, which makes sense in terms that it is hard to associate any user value from it due to the fact that features relying on this feature were not describes as such. On the contrary, this could be seen as a negative since the user might need to create an account for the tool before using it, introducing an additional barrier of entry. Furthermore, this impression is strengthened by the evaluation of the user feedback or start-rating feature for refactoring approaches. This feature was evaluated higher than the user management, even if the implementation could rely on said user management. Of course, this feedback functionality could be implemented anonymously, but this introduced different challenges.

An additional user requested feature which was mentioned was more information about refactoring approaches. If this additional mentioned information can fit into some already existing approach attribute or not will have to be evaluated in the future. Another user requested feature is the diversification of refactoring approach sources. This makes sense at a point where most important and well known approaches were added to the knowledge base of this tool, but since this is not the case at the moment this will also have to happen in the future. One small feature request was to add the search button in all tabs of the custom filter configuration user interface. This makes sense since we don't force users to look at each tab and allow them to jump to the final step right away. Since this is a very small and easy to implement change, we will implement it during the processing time of this thesis.

8.3.5 Other Feedback

Lastly, regarding problems one participant had when answering the questionnaire, it is not clear how we could provide a way of jumping between a questionnaire in one browser tab and the tool in a different one. The only way this is a problem is the participant did not utilize the browser tab functionality and tried to do both in one browser tab. We conclude, that this is hardly something we could improve. However, something we do think we could improve when looking at the fact that 63.64% for task 1 and 54.55% for task 2 of the participants tried to solve the given tasks, is that we should put the tasks and their evaluation next to each other, so participants don't have to write them down somewhere else before filling them in at a later point of the questionnaire.

9 Threats to Validity and Limitations

In this chapter we will address some threats to validity for the user study we conducted as well as general limitations regarding the complete thesis.

One general limitation to this thesis is a lack of previous research regarding the topic of this thesis. This limitation was already raised in chapter 3 and therefore represents a limitation of this thesis. Currently, there is limited research regarding the classification of refactoring approaches and guidance during the migration process. Additionally, there is no research regarding a tool assisted migration process. Therefore, a lot of concepts of this work had to be designed from the ground up.

Aside from this, there are some limitations and threats to validity which are specific related to the application evaluation conducted in chapter 8. These will be discussed in the following:

1. The first limitation regarding the conducted evaluation is the limited access to application evaluation data. The reason for this is that the developed application is an expert application, meaning that from the beginning potential users are software architects or developers with knowledge of microservices which are interested in the topic of application migration to microservices architecture. Generally, there might be a decent amount of people who fall under this category, however finding and reaching them is very difficult since it is such a specific target group.
2. As a result of the first limitation, the sample size of participants for the user study can be considered a limitation, with eleven participants. This amount might be too small to accurately conclude general trends and opinions on the in this thesis developed application. We think eleven participants is enough to get a general view on opinions and perceived limitations of the applications. However, as a result, we will generalize the results to the complete group of software architects and developers.
3. A potential threat to the external validity of the evaluation is the volunteer bias, which comes from the fact that the only feedback we got for our evaluation were volunteering participants and therefore makes it more difficult to make generalizations regarding the participants feedbacks. We don't think this bias holds too much weight for this thesis, since the participant group was already very specific and generalizations towards a broader group of software architects or developers were not made or intended in this thesis.
4. Another potential threat to external validity is a selection bias in the sense that the only feedback we collected was from participants finishing the survey. However, this is unavoidable and can only be counteracted by making answering the survey as accessible as possible. One example of how we tried to make the survey more accessible is making it so answering the provided tasks optional, so participants with less time can have an easier time participating.

10 Conclusion and Outlook

In this thesis we set out to answer three research questions related to the design, implementation and evaluation of an application for guiding architectural refactoring to microservices.

First, we asked research question 1: How can we design an extensible repository for existing migration and refactoring approaches? In order to answer this research question, we wanted to solve the task of designing and developing an extensible repository hosting existing refactoring approaches. During this thesis, we proposed that such an extensible repository needs more program logic than a regular repository or database with a fitting data model. Therefore, we designed a classical service application, which acts as a back-end to the complete application, tasked with bridging the database and user exposed functionality via back-end business logic. This back-end utilized commonly used service oriented architecture concepts such as a REST API to interact with it. Furthermore, we propose a modern technology stack to solve the development of the back-end application and fulfill defined requirements. The additional program logic in combination with a robust and flexible data-model, created from a research-based classification of refactoring approaches, was used to build the defined extensible repository.

The next research question was research question 2 where we asked: How can a web-based application of an architectural refactoring framework look like? The task designed to answer this research question was to create a fitting web-based user interface acting as the front-end part of the developed application. Based on defined requirements, we proposed one front-end application which provides functionality for the two different user bases of the application. We identified that aside from regular users using the tool for guidance in the migration process, we also need to provide functionality for potential administrator users which can utilize the flexibility of the created data model. These administrator users needed ways to extend the repository via adding new approaches or extend the possibilities of approaches can be described. Since we integrated the functionalities for both user groups into one front-end, we also provided a way to change which functionalities users have access to in the front-end application. For the implementation of the user interface we proposed, as already with the extensible repository, a modern and heavily supported and documented technology stack. Additionally, we described how the integration of both back-end and front-end applications can be realized.

Before tackling the last research question of this thesis, we discussed how the build and deployment of the application can be realized. Therefore, we proposed containerization via Docker to make building the application easy and independent on available technologies on potential host machines. We designed a container architecture which utilized a proxy to decide which of the two application parts incoming requests are routed to. This designed architecture was then used in the coming task to host the application for evaluation.

Lastly, we conducted an evaluation of the developed application in order to answer the third and last research question, which asked: How do potential users evaluate a prototypical implementation of this framework?. Participants of this evaluation were industry contacts of the Empirical Software

Engineering group, which resulted in participant with on average 16.8 years of professional experience and 4.5 of these years professional experience with microservices, of which the vast majority were software architects. The application which was evaluated excluded administrator functionality and was already hosted in order to reduce the barrier of entrance for participants. Based on the evaluation feedback, we concluded that users see usability aspects of the prototype generally positive. Similarly, participants evaluated the implemented functionality of the application as useful with potential for improvement. On the other hand, even though the recommendation functionality of the application was helpful, it was hard for participants to understand how it worked.

In conclusion, this work has shown that the current research of refactoring approach classifications can not be translated directly into a helpful application. In order to create helpful applications, one needs to carefully evaluate the research and make practical adjustments in this regard. Additionally, we showed how after such adjustments, tool support for the architectural refactoring framework shown in this thesis becomes more feasible if sufficient time is used for the design and implementation. Therefore, tool support may also be desirable for other research topics related to the migration to microservices architecture.

Outlook

There are many potential areas of this thesis which were out of scope and could be researched in more detail.

Since the implemented application mostly covers phase 2 of the architectural refactoring framework, which was a basis for this work, implementing the other phases and integrating them in a congruent and complementing way should be considered in the future. This way, an evaluation of the complete framework and how the application realizes it could be done and improve its helpfulness for potential users.

Additionally, since the recommendation algorithm proposed in this work is very rudimentary, additional research on how more complex and sophisticated algorithms could influence user perception of the helpfulness of recommendations could be considered.

Lastly, further work on adjustments to the classifications of refactoring approaches and data model could be considered. Bases for this could be some feedback provided by participants of the evaluation in this thesis.

Bibliography

- [Ang22] Angular. *Introduction to services and dependency injection*. 2022. URL: <https://angular.io/guide/architecture-services> (cit. on p. 57).
- [ASM+21] M. Abdellatif, A. Shatnawi, H. Mili, N. Moha, G. E. Boussaidi, G. Hecht, J. Privat, Y.-G. Guéhéneuc. “A taxonomy of service identification approaches for legacy software systems modernization”. In: *Journal of Systems and Software* 173 (2021), p. 110868. ISSN: 0164-1212. DOI: <https://doi.org/10.1016/j.jss.2020.110868>. URL: <https://www.sciencedirect.com/science/article/pii/S0164121220302582> (cit. on pp. 15, 23, 29, 30).
- [BBGG21] D. Bajaj, U. Bharti, A. Goel, S. Gupta. “A Prescriptive Model for Migration to Microservices Based on SDLC Artifacts”. In: *Journal of Web Engineering* (2021), pp. 817–852 (cit. on pp. 23, 44).
- [Dha21] H. K. Dhalla. “A Performance Comparison of RESTful Applications Implemented in Spring Boot Java and MS. NET Core”. In: *Journal of Physics: Conference Series*. Vol. 1933. 1. IOP Publishing, 2021, p. 012041 (cit. on p. 33).
- [Doc] Docker. *Overview of Docker Compose*. URL: <https://docs.docker.com/compose/> (cit. on p. 63).
- [FBWZ19] J. Fritzsich, J. Bogner, S. Wagner, A. Zimmermann. “Microservices Migration in Industry: Intentions, Strategies, and Challenges”. In: *2019 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. 2019, pp. 481–490. DOI: [10.1109/ICSME.2019.00081](https://doi.org/10.1109/ICSME.2019.00081) (cit. on p. 15).
- [FBZW19] J. Fritzsich, J. Bogner, A. Zimmermann, S. Wagner. “From Monolith to Microservices: A Classification of Refactoring Approaches”. In: *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment*. Ed. by J.-M. Bruel, M. Mazzara, B. Meyer. Cham: Springer International Publishing, 2019, pp. 128–141 (cit. on pp. 15, 23).
- [Foo21] K. D. Foote. *A Brief History of Microservices*. 2021. URL: <https://www.dataversity.net/a-brief-history-of-microservices/> (cit. on p. 20).
- [Gil08] B. Gillham. *Developing a questionnaire*. A&C Black, 2008 (cit. on p. 26).
- [GKGZ16] M. Gysel, L. Kölbener, W. Giersche, O. Zimmermann. “Service Cutter: A Systematic Approach to Service Decomposition”. In: *Service-Oriented and Cloud Computing*. Ed. by M. Aiello, E. B. Johnsen, S. Dustdar, I. Georgievski. Cham: Springer International Publishing, 2016, pp. 185–200. ISBN: 978-3-319-44482-6 (cit. on p. 68).
- [Goo22] Google. *Material Design*. 2022. URL: <https://material.io/design> (cit. on p. 53).
- [Gu20] Q. Gu. “A meta-approach to guide architectural refactoring from monolithic applications to microservices”. M.S. thesis. 2020 (cit. on pp. 16, 23).

- [Hab16] O. Habib. *A Quick Primer on Microservices*. 2016. URL: <https://dzone.com/articles/a-quick-primer-on-microservices> (cit. on p. 20).
- [Hal22] T. Haller. *GitHub repository of the Architecture Refactoring Helper*. 2022. URL: <https://github.com/T-Haller/architecture-refactoring-helper> (cit. on p. 61).
- [HF22] T. Haller, J. Fritzsich. *Interviews with Jonas Fritzsich regarding the architectural refactoring framework*. 2022 (cit. on pp. 20, 21, 25, 27, 47).
- [KM19] J. Kazanavičius, D. Mažeika. “Migrating Legacy Software to Microservices Architecture”. In: *2019 Open Conference of Electrical, Electronic and Information Sciences (eStream)*. 2019, pp. 1–5. DOI: [10.1109/eStream.2019.8732170](https://doi.org/10.1109/eStream.2019.8732170) (cit. on pp. 19, 20).
- [Kro18] J. A. Krosnick. “Questionnaire design”. In: *The Palgrave handbook of survey research*. Springer, 2018, pp. 439–455 (cit. on p. 26).
- [Mic21] Microsoft. *Entity Framework Core*. 2021. URL: <https://docs.microsoft.com/en-us/ef/core/> (cit. on pp. 34, 35).
- [Mic22a] Microsoft. *Attributes (C#)*. 2022. URL: <https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/attributes/> (cit. on p. 36).
- [Mic22b] Microsoft. *What is .NET*. 2022. URL: <https://dotnet.microsoft.com/en-us/learn/dotnet/what-is-dotnet> (cit. on p. 33).
- [Mor22] R. Morris. *Swashbuckle.AspNetCore*. 2022. URL: <https://github.com/domaindrivendev/Swashbuckle.AspNetCore> (cit. on p. 34).
- [MW18] B. Mayer, R. Weinreich. “An Approach to Extract the Architecture of Microservice-Based Software Systems”. In: *2018 IEEE Symposium on Service-Oriented System Engineering (SOSE)*. 2018, pp. 21–30. DOI: [10.1109/SOSE.2018.00012](https://doi.org/10.1109/SOSE.2018.00012) (cit. on p. 20).
- [PMA19] F. Ponce, G. Márquez, H. Astudillo. “Migrating from monolithic architecture to microservices: A Rapid Review”. In: *2019 38th International Conference of the Chilean Computer Science Society (SCCC)*. 2019, pp. 1–7. DOI: [10.1109/sccc49216.2019.8966423](https://doi.org/10.1109/sccc49216.2019.8966423) (cit. on p. 15).
- [PPJ19] I. Puja, P. Poscic, D. Jaksic. “Overview and Comparison of Several relational Database Modelling Methodologies and Notations”. In: *2019 42nd International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*. 2019, pp. 1641–1646. DOI: [10.23919/MIPRO.2019.8756667](https://doi.org/10.23919/MIPRO.2019.8756667) (cit. on p. 31).
- [Pre21] P. Prema. *Spring Boot vs ASP.NET Core: A Showdown*. 2021. URL: <https://medium.com/@putuprema/spring-boot-vs-asp-net-core-a-showdown-1d38b89c6c2d> (cit. on p. 33).
- [Rai18] S. Rainville. *How to configure NGINX for Angular and ReactJS*. 2018. URL: <https://www.serverlab.ca/tutorials/linux/web-servers-linux/how-to-configure-nginx-for-angular-and-reactjs/> (cit. on p. 63).
- [Ram22] Ram. *Building a Full-Stack Container Setup with ASP.NET Core and Docker Compose*. 2022. URL: <https://referbruv.com/blog/posts/dockerizing-multiple-services-integrating-angular-with-aspnetcore-api-via-docker-compose> (cit. on pp. 61, 62).

- [RWW+18] Z. Ren, W. Wang, G. Wu, C. Gao, W. Chen, J. Wei, T. Huang. “Migrating Web Applications from Monolithic Structure to Microservices Architecture”. In: *Proceedings of the Tenth Asia-Pacific Symposium on Internetware*. Internetware ’18. Beijing, China: Association for Computing Machinery, 2018. ISBN: 9781450365901. DOI: 10.1145/3275219.3275230. URL: <https://doi.org/10.1145/3275219.3275230> (cit. on p. 20).
- [SKM20] C. Schröer, F. Kruse, J. Marx Gómez. “A Qualitative Literature Review on Microservices Identification Approaches”. In: *Service-Oriented Computing*. Ed. by S. Dustdar. Cham: Springer International Publishing, 2020, pp. 151–168. ISBN: 978-3-030-64846-6 (cit. on p. 15).
- [Smi21] J. P. Smith. *Entity Framework core in action*. Simon and Schuster, 2021 (cit. on p. 35).
- [SQL22a] SQLite. *Datatypes In SQLite*. 2022. URL: <https://www.sqlite.org/datatype3.html> (cit. on p. 34).
- [SQL22b] SQLite. *Most Widely Deployed and Used Database Engine*. 2022. URL: <https://sqlite.org/mostdeployed.html> (cit. on p. 34).
- [SQL22c] SQLite. *SQLite is a Self Contained System*. 2022. URL: <https://sqlite.org/selfcontained.html> (cit. on p. 34).
- [SS17] Sarita, S. Sebastian. “Transform Monolith into Microservices using Docker”. In: *2017 International Conference on Computing, Communication, Control and Automation (ICCUBEA)*. 2017, pp. 1–5. DOI: 10.1109/ICCUBEA.2017.8463820 (cit. on pp. 19, 20).
- [Vog21] E. Vogel. “Seeding Data”. In: *Beginning Entity Framework Core 5: From Novice to Professional*. Berkeley, CA: Apress, 2021, pp. 57–64. ISBN: 978-1-4842-6882-7. DOI: 10.1007/978-1-4842-6882-7_6. URL: https://doi.org/10.1007/978-1-4842-6882-7_6 (cit. on p. 44).
- [Yak20] A. Yakunin. *Go vs C#, Part 3: Compiler, Runtime, Type System, Modules, and Everything Else*. 2020. URL: <https://medium.com/servicetitan-engineering/go-vs-c-part-3-compiler-runtime-type-system-modules-and-everything-else-faa423dddb34> (cit. on p. 33).

All links were last followed on May 5, 2022.

A Database Tables

Approach.Input.DomainArtifact	
Name	Description
Documentation	-
Human expertise	-
Ontology	-

Table A.1: The Database Table Entries for Domain Artifact Approach Attributes.

Approach.Input.RuntimeArtifact	
Name	Description
Log traces	-
User-Application interactions	-

Table A.2: The Database Table Entries for Runtime Artifact Approach Attributes.

Approach.Input.ModelArtifact	
Name	Description
Business process model	-
Use case model	-
Activity diagram	-
Data flow diagram	-
State machine diagram	-

Table A.3: The Database Table Entries for Model Artifact Approach Attributes.

Approach.Input.Executable		
Name	Language	Description
Source code	No specification	-
Database file	No specification	-
Test cases	No specification	-

Table A.4: The Database Table Entries for Executable Approach Attributes.

Approach.Process.Quality		
Name	Description	Category
Reuse	-	Requirement
Maintainability	-	Requirement
Interoperability	-	Requirement
Self-containment	-	Requirement
Composability	-	Requirement
Coupling	-	Metric
Cohesion	-	Metric
Granularity	-	Metric
Number of services	-	Metric

Table A.5: The Database Table Entries for Quality Approach Attributes.

Approach.Process.Direction	
Name	Description
Bottom-up	-
Top-down	-
Mixed	-

Table A.6: The Database Table Entries for Direction Approach Attributes.

Approach.Process.AutomationLevel	
Name	Description
Automatic	-
Semi-automatic	-
Manual	-

Table A.7: The Database Table Entries for Automation Level Approach Attributes.

Approach.Process.AnalysisType	
Name	Description
Static	-
Dynamic	-
Lexical	-
Historic	-

Table A.8: The Database Table Entries for Analysis Type Approach Attributes.

Approach.Process.Technique	
Name	Description
Wrapping	-
Genetic algorithm	-
Formal concept analysis	-
Clustering	-
Custom heuristics	-
General guidelines	-

Table A.9: The Database Table Entries for Technique Approach Attributes.

Approach.Output.Architecture	
Name	Description
Services	-
Microservices	-

Table A.10: The Database Table Entries for Architecture Approach Attributes.

Approach.Output.ServiceType	
Name	Description
Business services	-
Enterprise services	-
Application services	-
Entity services	-
Utility services	-
Infrastructure services	-
No specification	-

Table A.11: The Database Table Entries for Service Type Approach Attributes.

Approach.Usability.ValidationMethod	
Name	Description
Experiment	-
Industry	-
Case study	-
No validation	-

Table A.12: The Database Table Entries for Validation Method Approach Attributes.

Approach.Usability.ToolSupport	
Name	Description
Industry ready	-
Open source	-
Prototype	-
No tool support	-

Table A.13: The Database Table Entries for Tool Support Approach Attributes.

Approach.Usability.ResultsQuality	
Name	Description
High	-
Medium	-
Low	-
Not available	-

Table A.14: The Database Table Entries for Results Quality Approach Attributes.

Approach.Usability.AccuracyPrecision	
Name	Description
High	-
Medium	-
Low	-
Not available	-

Table A.15: The Database Table Entries for Accuracy/Precision Approach Attributes.

B Evaluation Questionnaire

Evaluation Questionnaire - Architecture Refactoring Helper

This questionnaire is about the web-based tool "Architecture Refactoring Helper" which is part of the master thesis "Design, Implementation and Evaluation of an Application for Guiding Architectural Refactoring to Microservices" by Tobias Haller. The tool tries to offer guidance in the migration to microservices architecture via a repository of existing approaches.

*Required

1. Hereby, I agree that my information from this survey can be used as anonymized data in the master thesis "Design, Implementation and Evaluation of an Application for Guiding Architectural Refactoring to Microservices" as well as in future related publications by the University of Stuttgart. *

Tick all that apply.

I agree

Figure B.1: Questionnaire Start Section.

Personal Information

2. Professional experience in years *

3. Experience with microservices in year *

4. Current professional role *

Mark only one oval.

Software developer

Software architect

Other: _____

Figure B.2: Questionnaire Personal Information Section.

Exploration of the tool

At this stage you can explore the prototype of the tool, which can be found here:

<https://refactoringhelper.sytes.net>

Please explore the tool first and then go through the evaluation survey.

We prepared two small tasks to help you explore the tool's functionality. Please try to solve them and provide the answers later in the survey.

1) How many approaches qualify for a "Re-Factor" activity with a suitability of 100%?

2) Which approach has a suitability of 100% for the following requirements?

- Input preferences
 - Model artifacts
 - Include "Use case model"
 - Exclude all others
 - Executables
 - Include "Source code (No specification)"
 - Exclude all others
- Process preferences
 - Levels of automation
 - Include "Automatic"
 - Exclude all others
 - Techniques
 - Include "Clustering"
 - Include "Custom heuristics"
 - exclude all others

All other possible requirements should be neutral ("can have").

IMPORTANT NOTE: The tool is in prototype state and does not reflect the final artifact yet. There is only limited functionality available at this point in time. As well, the number of refactoring approaches in the database is only a subset currently available body of research and does not yet provide adequate results. The main focus of this evaluation is the tool's usability and structure. We will analyze your feedback and consider to incorporate it in future versions.

Figure B.3: Questionnaire Exploration of the Tool Section.

Tool Structure

5. 1) Is the depicted framework an understandable way of summarizing the common migration activities? *

Mark only one oval.

	1	2	3	4	5	
Not understandable	<input type="radio"/>	Very understandable				

6. 2) How helpful do you rate the intended functionality in phase 1 (System Comprehension)? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	<input type="radio"/>	Very helpful				

7. 3) How helpful do you rate the functionality in phase 2 (Strategy definition)? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	<input type="radio"/>	Very helpful				

8. 4) How helpful do you rate predefined configurations such as "New Application Development", "Re-Build" and "Re-factor"? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	<input type="radio"/>	Very helpful				

9. 5) How helpful do you rate the "Configure" functionality to filter for custom requirements? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	<input type="radio"/>	Very helpful				

10. 6) Could you comprehend how the selection of recommended approaches was done by the tool? *

Mark only one oval.

	1	2	3	4	5	
I could not	<input type="radio"/>	I fully could				

11. 7) How helpful do you find the calculated percentage on the results page? *

Mark only one oval.

	1	2	3	4	5	
Not helpful	<input type="radio"/>	Very helpful				

Figure B.4: Questionnaire Tool Structure Section.

Tool Usability

1) Please provide the answers to the following two tasks, in case you could solve it

12. How many approaches qualify for a "Re-Factor" activity with a suitability of 100%?

13. Which approach (ID) has a suitability of 100% for the following requirements?

14. 2) How difficult was it to solved the first task? (leave empty if you did not solve)

Mark only one oval.

	1	2	3	4	5	
Very easy	<input type="radio"/>	Very difficult				

15. 3) How difficult was it to solved the second task? (leave empty if you did not solve)

Mark only one oval.

	1	2	3	4	5	
Very easy	<input type="radio"/>	Very difficult				

16. 4) How intuitive and easy is the navigation in the tool? *

Mark only one oval.

	1	2	3	4	5	
Not intuitive	<input type="radio"/>	Very intuitive				

17. 5) How would you rate the overall usability of the tool? *

Mark only one oval.

	1	2	3	4	5	
Not usable	<input type="radio"/>	Very usable				

18. 6) Do you have additional recommendations or feedback regarding the tool's usability?

Figure B.5: Questionnaire Tool Usability Section.

Additional functionality

1) How helpful would you rate the following potential future functionalities (not yet implemented)?

19. Functionality to use system specifications / results of phase 1 as an input for the filter in phase 2 *

Mark only one oval.

1 2 3 4 5

Not helpful Very helpful

20. Functionality to save entered information/progress in form of project *

Mark only one oval.

1 2 3 4 5

Not helpful Very helpful

21. Multi-user management (Login/Logout/Session Management) *

Mark only one oval.

1 2 3 4 5

Not helpful Very helpful

22. Functionality to gather user feedback or star-ratings for certain approaches to see other users' opinions *

Mark only one oval.

1 2 3 4 5

Not helpful Very helpful

23. 2) Which other general features would you like to have implemented in the tool?

24. 3) Do you have other feedback that did not fit to any of the questions above?

Figure B.6: Questionnaire Additional Functionality Section.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

place, date, signature