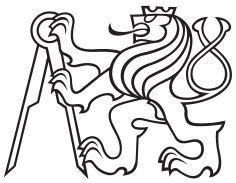
Master Thesis



Czech Technical University in Prague

F3

Faculty of Electrical Engineering
Department of Computers

# The Architecture Transformation of FelSight Faculty Application

Bc. Adam Kohout

Supervisor: Ing. Jan Zídek
Field of study: Open Informatics
Subfield: Software Engineering
May 2022

# ZADÁNÍ DIPLOMOVÉ PRÁCE

## I. OSOBNÍ A STUDIJNÍ ÚDAJE

Příjmení: **Kohout** Jméno: **Adam** Osobní číslo: **474670**

Fakulta/ústav: **Fakulta elektrotechnická**

Zadávající katedra/ústav: **Katedra počítačů**

Studijní program: **Otevřená informatika**

Specializace: **Softwarové inženýrství**

## II. ÚDAJE K DIPLOMOVÉ PRÁCI

Název diplomové práce:

**Transformace architektury aplikace FelSight**

Název diplomové práce anglicky:

**The architecture transformation of FelSight faculty application**

Pokyny pro vypracování:

1) Seznamte se s problematikou softwarových architektur v oblasti softwarového inženýrství a definujte tento pojem. Popište jejich praktický význam a motivaci pro použití v rámci vývoje aplikací.
2) Proveďte rešerši existujících příkladů architektur často používaných v praxi a porovnejte je.
3) Analyzujte problém transformace architektury existující aplikace včetně motivace a možných úskalí.
4) Popište zvolenou fakultní aplikaci pro podporu výuky a proveďte analýzu stávající monolitické architektury. Uveďte motivaci pro její transformaci a navrhněte novou architekturu, do které má být původní architektura převedena.
5) Součástí práce bude i přesný postup transformace, včetně jednotlivých kroků.
6) Implementujte PoC (proof of concept), který ověří vhodnost řešení. Popište potenciální praktické důsledky pro budoucí vývoj v rámci tohoto řešení.

Seznam doporučené literatury:

[1] ISO/IEC/IEEE 42010:2011. Systems and software engineering — Architecture description. Switzerland: IEEE, 2011. 10.1109/IEEESTD.2011.6129467. Dostupné také z: https://ieeexplore.ieee.org/servlet/opac?punumber=6129465
[2] BASS, Len, Paul CLEMENTS a Rick KAZMAN. Software architecture in practice. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN 978-032-1815-736.
[3] RICHARDS, Mark. Software Architecture Patterns. Sebastopol, CA: O'Reilly, 2015. ISBN 978-1-491-92424-2.
[4] BUCCHIARONE, Antonio, Nicola DRAGONI, Schahram DUSTDAR, Stephan T. LARSEN a Manuel MAZZARA. From Monolithic to Microservices: An Experience Report from the Banking Domain. IEEE Software. 2018, 35(3), 50-55. ISSN 0740-7459. Dostupné z: doi:10.1109/MS.2018.2141026

Jméno a pracoviště vedoucí(ho) diplomové práce:

**Ing. Jan Zídek Centrum znalostního managementu**

Jméno a pracoviště druhé(ho) vedoucí(ho) nebo konzultanta(ky) diplomové práce:

Datum zadání diplomové práce: **12.01.2022** Termín odevzdání diplomové práce: **20.05.2022**

Platnost zadání diplomové práce: **30.09.2023**

| Ing. Jan Zídek<br>podpis vedoucí(ho) práce | podpis vedoucí(ho) ústavu/katedry | prof. Mgr. Petr Páta, Ph.D.<br>podpis děkana(ky) |
|---|---|---|

## III. PŘEVZETÍ ZADÁNÍ

Diplomant bere na vědomí, že je povinen vypracovat diplomovou práci samostatně, bez cizí pomoci, s výjimkou poskytnutých konzultací.
Seznam použité literatury, jiných pramenů a jmen konzultantů je třeba uvést v diplomové práci.

. _____
Datum převzetí zadání

_____
Podpis studenta

# Acknowledgements

I want to express my sincere thanks to my supervisor Ing. Jan Zídek for his help by giving me valuable advice throughout the writing of this thesis. I would also like to acknowledge the consultation assistance provided by the architecture team of the Center of Knowledge Management headed by Bc. Tomáš Malinkovič. Lastly, I extend my appreciation for the continuous support given by my family.


# Declaration

I hereby declare that I have completed this thesis on my own and that all the used sources are included in the list of references, in accordance with the *Methodological instructions on ethical principles in the preparation of university theses*.

In Prague, May 20, 2022

Prohlašuji, že jsem předloženou práci vypracoval samostatně a že jsem uvedl veškeré použité informační zdroje v souladu s *Metodickým pokynem o dodržování etických principů při přípravě vysokoškolských závěrečných prací*.

V Praze dne 20. 5. 2022

.....
Bc. Adam Kohout

# Abstract

The thesis covers the topic of software architectures and the problem of transforming the architecture of an existing application. In the research part, the topic is introduced together with its history and motivating factors. Then, several examples of common architectures in practice are presented. Research also includes criteria and methods for designing software architecture, attributes that characterize the quality of architecture, and the architecture transformation process. The practical part of the thesis covers the architectural transformation of FelSight, the faculty application for students and teachers of FEE. This part includes a detailed description of the current monolithic architecture and structure of the application. The subsequent analysis identifies significant features of FelSight. The final chapter proposes the migration process of a chosen feature into a separate microservice. The feasibility of the approach is eventually evaluated and demonstrated in a proof of concept (POC) implementation.



**Keywords:** software architecture, architectural style, monolithic architecture, FelSight application, microservices

**Supervisor:** Ing. Jan Zídek



# Abstrakt

Diplomová práce se zabývá softwarovými architekturami a problematikou transformace architektury existující aplikace. V rešeršní části je toto téma nejprve představeno a jsou uvedeny i historické okolnosti včetně motivace. Dále je představeno několik příkadů architektur, které se běžně v praxi aplikují. Rešerše se také zabývá kritérii a metodami návrhu architektur, atributy popisující kvalitativní aspekty architektur, a procesem transformace architektury. Praktická část práce řeší transformaci architektury aplikace FelSight – fakultní aplikace určené pro studenty a vyučující FEL. Tato část obsahuje podrobnější popis stávající monolitické architektury aplikace a její struktury. Následuje analýza, v rámci které jsou identifikovány významné funkcionality, které mohou uživatelé FelSightu používat. Poslední kapitola představuje postup pro migraci vybrané funkcionality do samostatné mikroslužby. Proveditelnost postupu je nakonec vyhodnocena and demonstrována v rámci proof of concept (POC) implementace.



**Klíčová slova:** softwarová architektura, architektonický styl, monolitická architektura, aplikace FelSight, mikroslužby

**Překlad názvu:** Transformace architektury aplikace FelSight

# Contents

# Figures

# Tables

# Part I

# Theoretical Part

# Chapter 1
# Introduction

## 1.1 Preface

Today's software, especially in the enterprise world, is becoming more complex. It must not only fulfill all specified functional requirements – i.e. what the system can do – but also comply with qualitative requirements such as security. Designing, developing, and maintaining applications while satisfying all of these demands is not an easy task. Moreover, as an application grows in complexity, the need for an organized structure arises. This is necessary, especially in cases where the application is expected to have multiple functional components, distinguished by their purpose, that need to communicate mutually. As a result, the architecture of the application is established (either explicitly, or implicitly).

## 1.2 Motivation

Being concerned with software architectures is important because it helps developers grasp the overall complexity of the application. Although this might not be apparent or essential for trivial applications with a small codebase, the ability to structure software meaningfully is useful and universal.

Software architecture also has an influence on multiple aspects of a software project. It should help in communication among stakeholders as a special kind of language. This is because it describes the structure and behavior of the application without going into the technical details of the implementation.

Modern software evolves at a high rate and applications are required to accommodate the changes and additions required by customers. However, older applications might not be suitable for such accommodation. The root cause of the issue could be the legacy architecture that the application has been using since its inception. For example, it may no longer be possible to easily manage the complexity of the application, and incorporating further additions can become increasingly difficult. This could imply the need for a restructuring in the form of an architectural transformation, e.g. to break down the application's domain into smaller sections that are easier to comprehend and organize.

## 1.3 Thesis Structure

The first part of the thesis introduces software architectures together with several definitions of this term. It briefly connects the topic with its history and mentions the main motivation points. Next, two terms, *architectural style* and *design pattern*, are described together and the main difference between them is explained. Then, several examples of architectural styles are provided. Design patterns are not, however, further examined, as they are not the focus of this thesis.

The next chapter examines the problem of designing a software architecture. This includes the decision criteria, important questions to ask, and two methods that can be used to design architectures. The chapter is concluded with some examples of software quality attributes – properties that are useful for defining and measuring the quality of a designed architecture.

The thesis then continues with an analysis of the task of transforming an existing architecture into another one. First, several typical reasons for migrating an architecture are mentioned. In particular, this chapter focuses primarily on decomposing a monolithic architecture. Therefore, three approaches that can be used as a guide for this process are introduced. The last part of the chapter mentions some general recommendations for migrating architectures.

The following practical part of the thesis is concerned with the problem of the architectural transformation of FelSight, the faculty web application for students and teachers of the Faculty of Electrical Engineering at the Czech Technical University. The introductory chapter briefly introduces the application. Then its current monolithic architecture solution is introduced. In addition, the main modules and the build and deployment processes of the application are described. It is also explained why architecture is worthy of transformation.

In the second chapter of the practical part, the analysis and design of a new architecture are carried out. These activities are based on the idea of *decomposition*. Individual business areas of the application are identified and associated based on user actions. The result of the analysis is a collection of independent units – potential candidates for microservices.

The analysis is followed by the implementation of the proof of concept (POC), covered in the final chapter. The purpose of the POC is to demonstrate the migration process of a smaller business area of FelSight. Therefore, one of the candidates in the collection is chosen. Subsequently, the steps of the migration process are presented. Each of the steps is individually explained in detail with encompassing activities. POC results are evaluated by discussing how the presented solution affects team collaboration. Part of the discussion is a description of the expected workflow when the solution is applied in practice. The chapter is concluded with a brief discussion of the topic of security with a focus on the authentication of microservices. In this part, a suitable authentication model is suggested for potential use in the future FelSight microservice architecture.

# Chapter 2

# Software Architectures

First, it is important to define the term *software architecture* and clarify the meaning of this term in the field of software development and its motivation. There are multiple architectures, and they can be distinguished from each other by different characteristics, such as structure, or number of parts they are comprised of. A few of the examples that can often be encountered in practice are introduced in the last part of this chapter.

## 2.1 Definition

The ISO/IEC/IEEE 42010 standard, which deals with descriptions of architectures in the field of software engineering, defines the term *architecture* as follows:

> *"<system> fundamental concepts or properties of a system in its environment embodied in its elements, relationships, and in the principles of its design and evolution"* [1]

On the basis of this definition, architecture is simply a collection of characteristics of a given system altogether with its components and relationships. Furthermore, the evolution and design techniques used in the development are also considered important.

The term is perceived similarly by the group of three authors Len Bass, Paul Clements, and Rick Kazman in their book *Software Architectures in Practice*. They describe the software architecture of a system as a set of structured elements whose existence in the system should be taken into account. These structures are again made up of smaller software components and are identified by their characteristics and relationships among them. [2]

For a software architect, this term can also represent some blueprint and the so-called *roadmap* for the development of a new system. Under these formulations, we can imagine the said system structure (i.e. architectonic styles, see Section 2.4), its characteristics (for example, reliability, performance, or security), but also conventions and principles established in a project team. Conventions determine restrictions and rules that a resulting system should comply with (e.g. in a layered architecture, only neighboring layers are

allowed to communicate). On the other hand, principles serve as a guide to achieve the required characteristics of the system. [3]

To answer the question of why we should care about software architectures, it is useful to describe how this concept was actually born and how people started to get acquainted with it in the first place.

## 2.2 History

In the late 1950s of the last century, common, repetitive patterns in machine codes started to emerge and programmers realized that these patterns can be defined by a series of mathematical symbols. Until this point, programs were written by storing individual machine instructions manually in memory. However, this discovery of abstraction made it possible to automate this process and generate low-level code on behalf of the programmer. It also significantly simplified software development, but also opened the door to writing more complicated programs in a modularized and organized manner. As a result, the first high-level programming languages began to rise. [4]

In the meantime, the analogy with abstracting all of the repetitive patterns away started to find its purpose among system architects. Similarly, different methods of systematic organizing were developed with the motivation of hiding the implementation within the components of the system. [4]

One of the first references about the architecture of software was mentioned by Ian P. Sharp at the NATO Conference on Software Engineering Techniques in 1969. Sharp emphasized that software should not only be described by its functional specifications (i.e. the functions a system must perform, as defined by the ISO/IEC/IEEE 24765 standard [5]) but also by how it was designed. [6]

It was not until the 1990s that the topic of software architectures started to catch on as a discipline on its own. Different types of notation for describing architectures of software were invented and new job positions for software architects started to emerge. Today, the importance of software architectures is widely acknowledged. [6]

## 2.3 Motivation

As mentioned in the previous section, describing an architecture of software can help in conceiving unnecessary implementation details, which is useful for the validation of a whole system at a much higher level. There are, however, many more motivational factors, and they can be considered beneficial from both technical and business (or management) viewpoints. An example is given for each of the sectors in the following two paragraphs.

From a technical standpoint, designing an architecture not only includes capturing individual components but also predicting the system's qualities and specifying its implementation constraints. Another part of this process is the fact that it documents early decisions that are difficult to change

in the future [2]. This relates to evolution, which represents an adaptation of a system to new conditions. However, it should be noted that violations of the existing architecture and insensitivity to the architecture are the main causes of a system's resistance to potential changes. [7]

Specifying an architecture should also be useful for the business sector of a project. Given that complexities and technicalities are omitted at this level, this form of the specification is still decently comprehensible, especially for non-technical people. Thus, it helps to improve communication among all people involved in the project – i.e. the stakeholders. [2]

## 2.4 Architectural Styles vs. Design Patterns

The above mentioned definitions are rather abstract. Therefore, it is relevant to explain what software architectures practically mean when they are implemented in a real-world scenario.

In practice, there are two terms closely related to the design of systems, *architectural styles* and *design patterns*. Although analogously similar, they are not identical.

Architectural styles are concerned with the organization of components within the system. When designing a system, architects typically utilize multiple styles (or their usable aspects) to achieve the attributes required by the customer. These styles can be divided into several groups based on their specific usages. For example, when considering the *structure* of an application, one can opt for the *layered architecture* (described in Subsection 2.5.1) where parts are separated based on their functional purpose. [8, 9]

Design patterns aim to address problems that occur repetitively in software. This is also a characteristic of architectural styles. However, the main difference is that design patterns are concerned with much smaller parts of the system. Commonly, they are used within one or multiple styles on a program/process level. An example of a design pattern is *Facade pattern* whose purpose is to provide a single interface for executing multiple operations that would otherwise have to be executed individually by the client. [10]

In terms of designing and developing systems, both concepts complement each other. However, this thesis is concerned with the design at the system level rather than with the design at the process level. For this reason, the design patterns are not discussed in the following parts.

## 2.5 Common Examples of Architectures

There are a significant number of solutions for common problems in the field of software architectures. These styles can be used as a guide when designing, if not implementing them directly in a new system.

This section introduces several frequently used types of architecture.

### 2.5.1 Layered Architecture

One of the most widely used types is *layered architecture*, which is given by the fact that it reflects conventional communication concepts in traditional systems such as web applications.

The structure of this architecture is quite straightforward. As the name suggests, individual components are depicted as horizontal layers positioned on top of each other. Every layer has its own purpose and typically communicates only with the neighboring layers.

Using the aforementioned web application as an example, its structure would, in most practical cases, comprise of four layers with their own roles:

1. *Presentation layer.* Represents an interface for handling user actions.
2. *Business layer.* Logic and behavior are usually driven by business rules.
3. *Persistence layer.* Responsible for establishing communication with the database(s).
4. *Database layer.* Responsible for storing data. Includes databases of any type.

The communication flow together with the layers is shown in Figure 2.1. The flow is typically initiated by a request that originates from the user. The request is then transmitted via individual layers, processed by them based on their internal implementation, and passed on to the next layer. It is important to note that, in this case, the request cannot skip any layer of the architecture (which is labeled with the word "CLOSED" for every layer). However, it is not a required property of this architecture type.

All the information mentioned in this section, including the figure itself, was taken from the book *Software Architecture Patterns* by Mark Richards. [11]

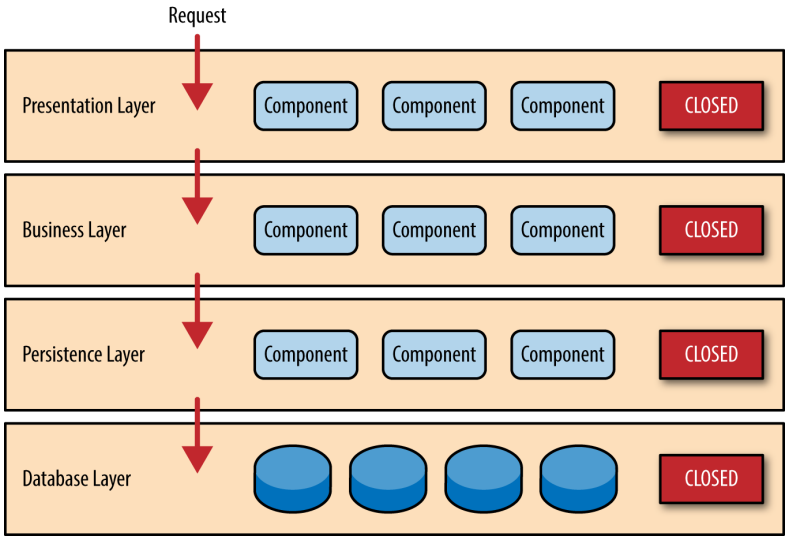



**Figure 2.1:** Layered architecture with four layers. [11]

### 2.5.2 Service-Based Architecture

Another, conceptually different, example is an architecture whose basic building blocks are called *services*. Such an architecture is called *service-based*.

Two of the most popular service-based architecture subtypes are *service-oriented architecture*, abbreviated SOA, and *microservices architecture*. Both have some unique characteristics; however, the main trait they have in common is that the components are usually distributed as separate deployable units. This distinguishes them from the layered architecture, where most of the layers are part of one unit (even though they are still separated into their own objects by their functions).

Given that the services run independently in their own separate environments, it is necessary to define a specific communication protocol among them so that they can exchange data via a network. For this need, there are standardized and widely adopted solutions based on the HTTP protocol, such as Representational State Transfer (REST) or Simple Object Access Protocol (SOAP).

When it comes to naming at least one difference between SOA and microservices, one of the most prominent ones would be related to component sharing. SOA aims for extensive sharing across multiple services, whereas microservice architecture is characterized by having rather a separate instance of dependency for each service making them more autonomous. A good example would be a database: In Figure 2.2 there are four services that are called by the user interface. They are connected to one master database that is shared by them. In microservices architecture, each of the services would be connected to their own separate one, i.e. four services result in four databases.

The book *Microservices vs. Service-Oriented Architecture* by the author Mark Richards [12] was used as the source of information for this particular architecture example.

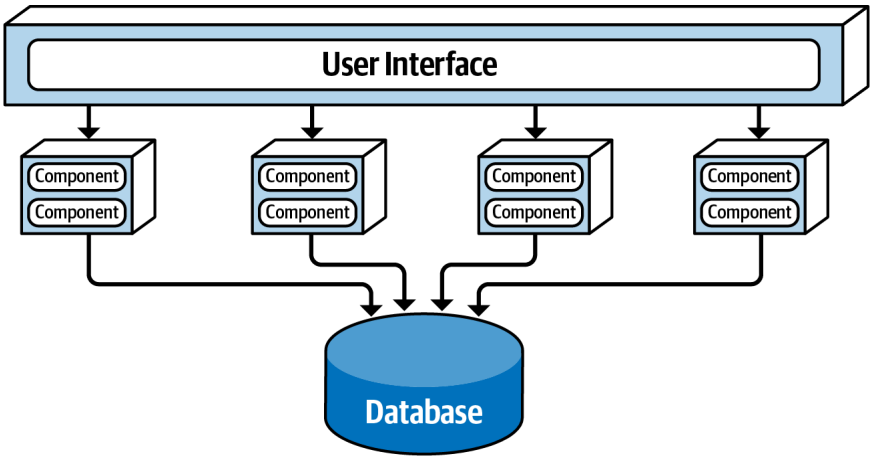



**Figure 2.2:** Structure of a simple service-based architecture (SOA type). [3]

### 2.5.3 Event-Driven Architecture

The third architecture type example is *event-driven architecture*. This style addresses how *messaging* is carried out among individual components. In particular, it describes how the *events* occurring in the system are handled by individual components. There are two basic variants based on the topology of the architecture, mediator, and broker topologies. [11, 8]

The event-based architecture with the mediator topology consists of four component types, namely *event queue*, *event mediator*, *event channel* and *event processor*. First, when an event (e.g. a request) is generated by the client, it is sent to the event queue, where it waits for further processing. The event mediator then receives the event via the queue and generates new events that represent operations needed to be performed based on the original event. These are communicated individually and asynchronously with event processors via event channels. The entire communication process is depicted in Figure 2.3. [11]

The broker topology, on the other hand, typically makes use of decentralized event processing. In particular, the event mediator in the previous variant is replaced by *broker*. The events are then distributed directly across the processors through the broker with the contained event channels. [11]

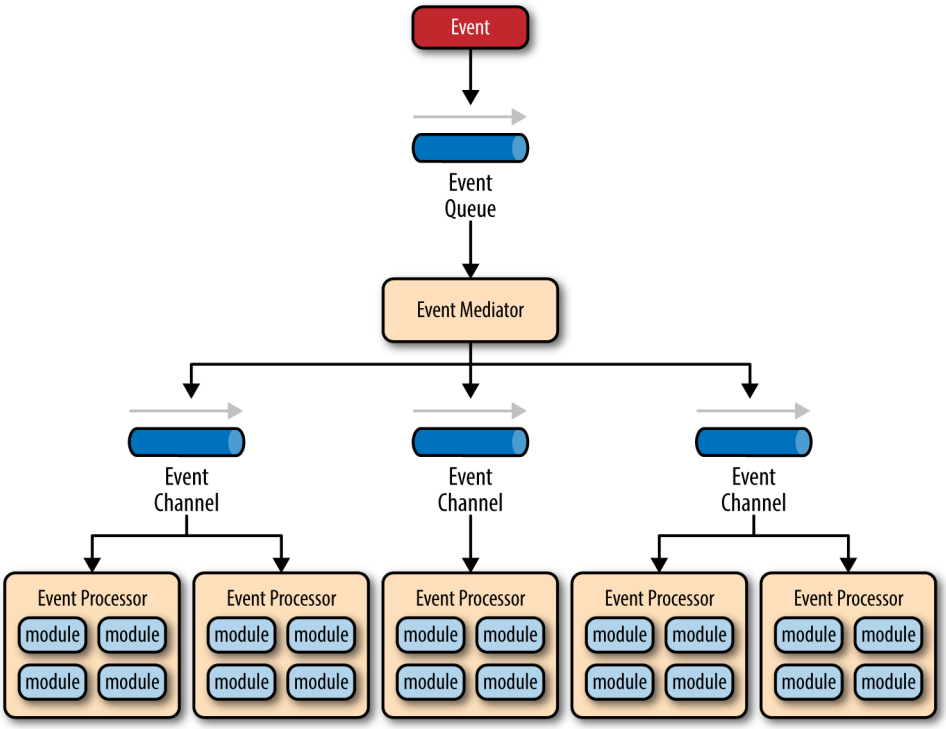



**Figure 2.3:** Components of an event-driven architecture with the mediator topology. [11]

### 2.5.4 Pipeline Architecture

Finally, the last example is *pipeline architecture*, also called *pipe-and-filters architecture*.

As the alternative name suggests, the components of this architecture are of two principal types, *pipes* and *filters*. The structure is rather straightforward: The filters are joined together, the pipes being the communication channel between them. In common scenarios, pipes are one-way channels and have one source and one target point, i.e. output data from one filter flows to another filter as input data. This is shown in Figure 2.4. Filters act as independent components without any state, which means that input data is processed and output data of the process are forwarded to the next filter.

A filter can be one of the four types – producer, transformer, tester, or consumer. The producer is the starting point of the data flow. It does not accept any data; it only outputs data and initiates the whole process. Any intermediate filter in the chain can either be transformer, or tester. The transformer optionally changes the received data and directs it to another filter. The tester accepts the data and optionally outputs the data according to one or more tested criteria. At the terminating point, there is the consumer that only accepts data as the final result of the whole process. Should the result be persisted for later use, it can be saved e.g. into a database.

The great advantage of this architecture is the simplicity and modularity of the solution. However, it is not well suited for scalable and elastic applications due to its monolithic properties.

All the information mentioned in this example comes from the book *Fundamentals of software architecture: an engineering approach* by Mark Richards and Neal Ford [3].

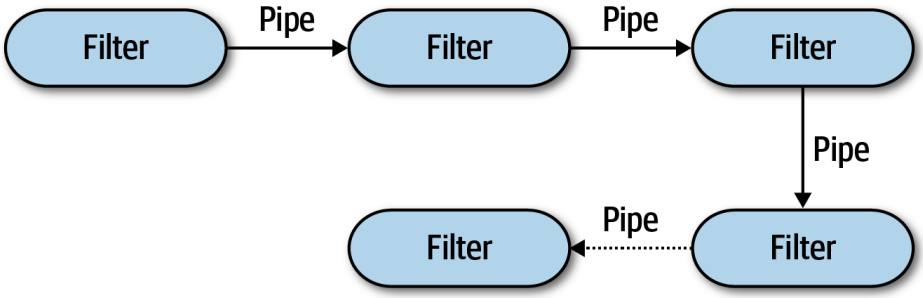



**Figure 2.4:** A simple structure of the pipeline architecture with pipes and filters. [3]

# Chapter 3

# Designing a Software Architecture

When making decisions regarding a system's architecture, it is essential to consider numerous aspects. This chapter addresses them and introduces general recommendations for designing the architecture of a system and choosing an appropriate architectural style.

## 3.1 General Criteria of the Decision Process

There are several factors to consider when designing a new architecture. They include solid understanding of the project domain, specification of data architecture, team and internal process knowledge and external factors. [3]

### 3.1.1 The Domain

It is essential that an architect becomes familiar with the domain covered by the project. By *domain* we mean the subject area for which the system is built, i.e. the scope of the problem the software is designed to solve. [3, 13]

Knowledge of the domain is important because it determines business needs and it is the core reason the software is developed in the first place. Being acquainted with it means understanding the problem and thus creating a proper solution for it in the form of software. The discipline concerned with designing software with primary focus on the domain is called *domain-driven design* (*DDD*). It is discussed in detail in Subsection 3.3.2. [13]

### 3.1.2 Data Architecture

For software, it is a common scenario to work with data (in an arbitrary form). In addition to software architectures, there are also *data architectures*. In essence, this type of architecture is concerned with the way in which data are stored and used. Similarly to software architectures, data architectures provide a high-level view of the system. Although this discipline is not the main focus of the thesis, it is still highly relevant. An architect must be familiar with it to ensure proper use and manipulation of data within the system (e.g. database structure). [3, 14]

### 3.1.3 Project Team and Internal Process Knowledge

Another factor that can influence the decision (and its consequences) while designing software architectures are internal processes in the project team. Team members should be aware of the fact that the established project management practices and methodologies can also have significant impact on the ability to use particular architectural styles. [3, 2]

Two different project management models/techniques – *Waterfall* and *Agile* – are a good example. The Waterfall model can be visualized as a one-way process flow through individual project phases, which take place only once. On the other hand, the concept of Agile project management is based on iterations in which every project phase is repeated. This means that the software is frequently released in smaller parts, and it is more flexible in terms of potential future changes. On the contrary, the Waterfall model specifies everything from the beginning without the possibility of easily making changes in the future, and this might not be ideal for the evolution of the architecture. [3, 2]

### 3.1.4 External Factors

There are also influential factors coming from the external environment, that is outside the project team. One of the examples could be deciding what cloud provider to use. The main factors in this case are typically price, the range of features and customer support. Another incentive can show up within the organization when it is, for example, acquired by another company and the team is forced to choose specific software and services, usually owned or promoted by the acquiring company. [3]

## 3.2 Decisions to Make

Additionally, an architect must make important decisions that are based on the criteria mentioned in the previous subsection. The most important questions to be asked are [3]:

- Should the architecture be monolithic, or distributed?
- Where the data should be located?
- What technology stack should be used?

Note that the list summarizes a few examples of important decisions that have an influence on the future development of the architecture. In practice, the list would include many more questions.

The following subsections describe the listed questions in detail.

### 3.2.1 Monolithic, or Distributed Architecture

An architect should assess whether the quality attributes required to be met are valid for the whole system or only for some of its parts. The answer to this problem could also guide them to answering the question of whether to choose an architecture that is its own part, or is divided into multiple pieces. The former – the monolithic architecture – could be suitable for a unified set of characteristics, the latter could be better for multiple groups of characteristics. [3]

### 3.2.2 Data Location

As already mentioned in Subsection 3.1.2, the architect should also be concerned with data storage and with its usage. This problem logically follows from the previous question. In the monolithic architecture, the data would be typically stored in a (relational) database. Then the application is usually able to connect to it directly. If, however, the architecture is distributed, the data would be accessible via independent components. It is the architect's task to decide which components should have this responsibility and how the data should be accessed. [3]

### 3.2.3 Technology Stack

In the end, any architecture is implemented in the chosen technology. Although technology should remain independent of architecture, it may still have an impact on quality attributes, development process, and documentation. Real-world examples show that the technology choice can be driven by its cost (e.g. introduction of cheaper technology), by its effectiveness or the features (e.g. component-based technology allowing code generation directly from the specification). [15]

It must be noted that architects/developers might not be able to decide whatsoever due to earlier decisions made by someone else. However, if such decision is possible, the architect should be aware of possibilities available to them and they should consider its advantages and drawbacks and also its compatibility with the existing technology currently used. [2]

## 3.3 Design Methodologies

Although the concrete architecture design process is unique based on the given situation, having a universal strategy on how to approach this problem might be helpful. This section presents two methods that architects can use to help them along the way.

### 3.3.1 Attribute-Driven Design

Introduced in 2000, originally named *Architecture Based Design* and later renamed to *Attribute-Driven Design* (*ADD*), this method works with the assump-

tion that designing architectures is not straightforward because of the detail lacking in business requirements in the beginning. It is intended for building architectures at a high level of abstraction in early stages of development. This allows for uncertainty in the requirements which are yet to be elaborated in the future. [16, 17]

In 2006, the method and its steps were revised to make it clearer and easier for architects to use. The core procedure itself, however, remains unchanged. The method uses the idea of recursive *decomposition* and works with a list of requirements and constraints as input: functional and quality attributes. In summary, the steps of the method are as follows [16, 17, 18, 2]:

1. **Choose one element of the system to be designed.**
2. **For the chosen element, find all attributes with a potential impact on the architecture.** Typically, these attributes are related to modifiability, performance, availability and security. [2]
3. **For the chosen element, create a design solution satisfying the attributes identified in the previous step.** This is the essential step of the method. It includes choosing a pattern, the allocation of responsibilities to individual child elements and defining their interfaces. [18]
4. **Test that the attributes were satisfied, refine them, if needed, and select remaining requirements and constraints as the input to the next iteration.** The refinement here refers to assignment of the requirements to the potential child elements in further decompositions. [18]
5. **Repeat steps 1–4 until all requirements and constraints have been satisfied.**

The general process of ADD is shown in Figure 3.1. In every iteration, a (sub)system is split into several parts and these are split (decomposed) again in further iterations. During the process, *software templates* at each level are created. Every software template defines behavior and responsibilities of a software element of a given type and interaction with the rest of the system. Software templates' purpose is to provide reusable parts in the system. A common example of a software template is a logging service. [16]

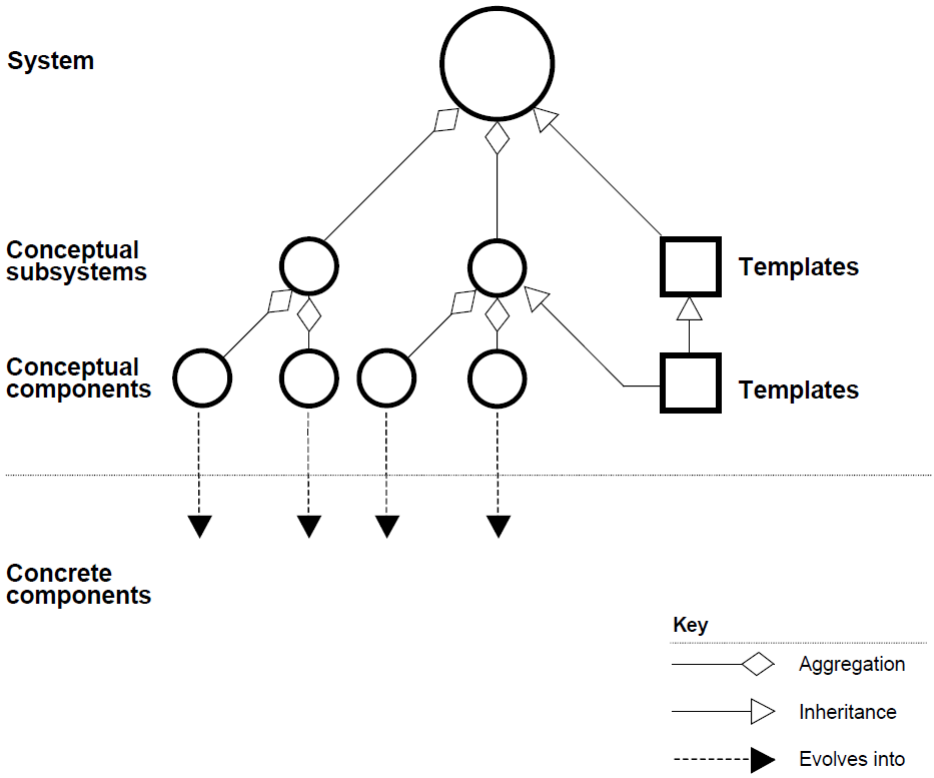



**Figure 3.1:** Decomposition in ADD. [16]

### 3.3.2 Domain-Driven Design

The second design method, Domain-Driven Design (DDD), is based on a different concept. It stresses the importance of business domain as the core reason for developing a software product. The premise is that in order to come up with a proper solution to a problem, one must have a solid understanding of the problem first. This is necessary especially when the domain for which the implemented solution implemented is complex. [13]

The central idea of DDD is the domain model, which serves as a guide for both analysts and developers. To make it realizable in practice, a shared language that is understandable to all must be established, the so-called *Ubiquitous Language*. This language encompasses the unified terminology describing a given domain and helps everyone across the team communicate thanks to explicitly defined terms. These ideas grouped together are known as *Model-Driven Design*. [13]

DDD also acknowledges the fact that maintaining a single model is difficult – especially in complex projects – and it addresses the need for distinguishing areas containing similar concepts from other areas within the model. These areas are called *bounded contexts* and are usually defined by their responsibilities. In the e-commerce example, shown in Figure 3.2, a product (domain) can be referred to in multiple contexts (subdomain), i.e. in pricing context

(e.g. prices of the product for different types of customers) or inventory context (e.g. stock information about the product). In practice, this means that the product is defined multiple times but in different contexts based on the responsibilities they are assigned, as shown in Figure 3.3. [13] Given the resemblance, this concept could be utilized – for example – in a microservices architecture.

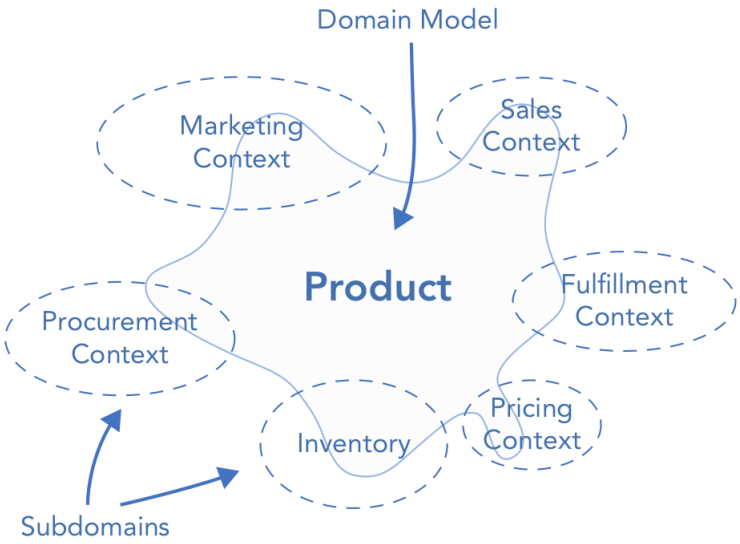



**Figure 3.2:** Identifying bounded contexts in the domain. [13]

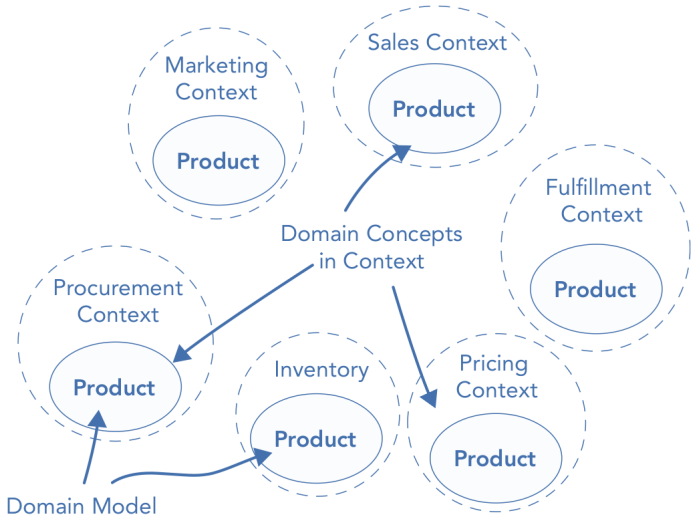



**Figure 3.3:** Separated bounded contexts with their own models. [13]

## 3.4 Software Quality Attributes

When developing a system, functional requirements / attributes are usually taken into consideration thoroughly. They define what the system can do in terms of its features. Nevertheless, poor performance, maintainability, or scalability are common reasons for redeveloping a system. These examples are the second type of properties, collectively called *quality attributes*. [2]

The ISO/IEC/IEEE 24765 standard defines the term *quality attribute* as *"feature or characteristic that affects an item's quality"* or *"requirement that specifies the degree of an attribute that affects the quality that the system or software must possess"*. [5] Satisfying particular quality attributes is as important as satisfying the functional ones and they should not be neglected because they can be influenced by the architecture. For this reason, it is essential to consider them during the architecture design process. [2, 19]

This section introduces some of the common quality attributes.

### 3.4.1 Availability

Availability is an ability of a system to *"perform its required function at an agreed instant or over an agreed period time"*. [5]

This metric is usually expressed as a percentage of the time the system is available to the user, including only the period during which the system *should* be operational. This is important because it does not include planned unavailability, such as system maintenance. It is a common requirement specified in service-level agreement (SLA) documents. [2, 5, 19]

For example, as a part of the Firebase service, Google specifies hosting and realtime database availability in the SLA as follows (as of December 26, 2021):

> *Firebase will use commercially reasonable efforts to make Firebase available with a Monthly Uptime Percentage (defined below) of at least 99.95%, in each case during any monthly billing cycle (the "Service Commitment").* [20]

### 3.4.2 Modifiability

Whether it is adding a few novel features or migrating to the new technology, software is always dynamic and changing. Modifiability quality attribute determines how easily a system can be changed without introducing defects. In order to analyze them, an architect should know what in the system might change, what probability that the change will take place is, who will be affected by it (who is responsible for implementation of the changes) and what the cost of the change will be. Answering to these questions helps with the estimation of the risk and cost of hypothetical changes. The cost includes both money and time. [2, 5]

There are three primary techniques for increasing modifiability of the system that help reduce the cost of the change:

- Decrease size of a component.
- Reduce coupling between components.
- Increase cohesion of a component.

Decreasing the size of a component typically means splitting it into two smaller parts. The reasoning behind this action is the assumption that the modification of the larger component will be costly due to the multiple responsibilities it has. The purpose of reducing coupling between two components is rather intuitive: If component A subjected to change is loosely coupled to component B, component B is less likely to also be noticeably affected by the change. Increasing the cohesion of components in a system is related to assigning responsibilities to them. Two different responsibilities should be covered by two different components. This prevents the responsibilities from affecting each other by the change. [2]

### 3.4.3 Performance

Another important metric to care about is performance of the system, in particular its response times and throughput. Performance is crucial especially in web-based systems where users initiate up to millions of requests in a short time span via their web browser. In this scenario, the required performance can be expressed as a constraint e.g. requests the system can process in a minute. [2, 19]

Common strategies for improving system performance are controlled demand for resources and resource management. Examples of controlled resource demand techniques include event prioritization, overhead reduction, and resource efficiency increase (by using efficient algorithms). Appropriate resource management models include techniques like increasing the number of resources, concurrent processing, caching, or resource scheduling. [2]

### 3.4.4 Testability

In the ISO/IEC/IEEE 24765 standard, testability is generally defined as *"extent to which an objective and feasible test can be designed to determine whether a requirement is met"*. [5]

Testing is one of the main parts of the typical project life cycle and helps verify functionality and identify system faults. The testability quality attribute describes the ease with which a fault in the system can be discovered by a test. The better the testability, the sooner the fault is discovered. [2]

One of the main tactics for ensuring better testability is system state observation and control. Common approaches of this tactic are recording the state causing the fault in order to recreate it, isolating the tested instance of the system from the real environment (usually achieved by virtualization) and using assertions in different part of code in order to test expected values against actual ones. [2]

### 3.4.5 Security

Security is a rather complex topic that deals with the challenge of protecting data in the system from unauthorized access. The system must also be able to provide the data to authorized users and other systems. [2]

Security of systems is characterized by three core properties:

- Confidentiality,
- Integrity,
- Availability.

Confidentiality refers to the fact that data in the system are protected from unauthorized access. Integrity is the ability of the system to prevent data from unauthorized tampering and the ability to detect such manipulation. Finally, availability describes the ability of the system to be available to provide its functionality for legitimate use despite a potentially ongoing attack. [2]

Security covers much more, however, it is not the major topic of this thesis.

# Chapter 4

# Software Architecture Transformation Process

This chapter explores the task of using the existing software architecture to convert it into a different one. It puts emphasis especially on the case of converting a monolithic architecture to SOA or microservices architecture.

The chapter begins by mentioning several motivational factors for migrating an existing architecture to a different one. Then it presents three approaches that can be combined together to proceed with the migration (i.e. decomposition) of monolithic architecture. Finally, the chapter is concluded with several general recommendations for migration of the architecture.

## 4.1 Motivation

The need to modify the architecture of an existing system or replace it with an entirely new one can be motivated by multiple factors [3]:

- **Migrating to newer technology.** Today's technology evolves at a great pace. Keeping up with the latest or at least still relatively new trends might be beneficial for an organization. However, potential benefits should first be analyzed before making technological changes.
- **Emerging of new possibilities.** Relating to the previous bullet, the organization should also consider all the options and tools available to achieve a goal. A widely adopted example is Docker – developers and architects became aware of its advantages and started migrating their applications to virtual environments, i.e. containers.
- **Volatility in the domain.** The project domain is also subject to future changes. Most often, this is given by the business' evolution, and while it does not necessarily mean restructuring the whole architecture, the application can be significantly affected by it.
- **Experience.** Incentives that motivate architectural change can also be driven by observations from the past. These driving observations are often negative (i.e. bad experience) and shape an architect's view of styles

and their suitability in different scenarios. As a result, an architect also reflects on their previous experiences when designing a new architecture.

- **External stimuli.** Motives can also be external and do not have to relate to the development itself. For example, the choice of tools and migration can be constrained by licenses bought by the organization in the past.

There is no universally applicable solution to every problem. Before proceeding, it is crucial to have the decision rationally justified – i.e. *why* the change should take place. In addition, all possible benefits and drawbacks associated with the change should be considered.

After thinking everything through thoroughly, one can start preparing for the change. The rest of this chapter considers the case of transforming a monolithic application to SOA/microservices architecture.

## 4.2 Monolith Decomposition Approaches

Transforming a monolithic application into microservices implies the need for analysis of the existing architecture and identification of its parts mapped and eventually decomposed into individual microservices. Several patterns can be used to decompose a monolithic architecture.

### 4.2.1 Strangler Pattern

Introduced by the software engineer Martin Fowler [21], the *strangler fig application* is a way of safely rewriting an existing application in a gradual manner, one part at a time.

The name of the pattern refers to parasitic strangler fig species which seeds in other tree's branches. The fig grows slowly and wraps around the host tree until it reaches the ground where it roots. The host tree is eventually killed, while the fig becomes an independent tree on its own. [22]

The same happens when the pattern is applied in software engineering. The existing system acts as a fundamental support for the new architecture. As the new architecture progressively develops and grows in size, it "wraps" the original application architecture. Eventually, the old architecture is replaced by the new one in its entirety. [22]

Strangler pattern consists of three main steps, as shown in Figure 4.1:

1. Identify part of the old architecture to map to the new architecture.
2. Map the part to the new architecture.
3. Redirect all calls of the part from the old architecture to the new architecture.

The process of identifying a part to be migrated always depends on the context and is based on the architect's assessment. For this purpose, DDD, mentioned in 3.3.2, can be used. [22]

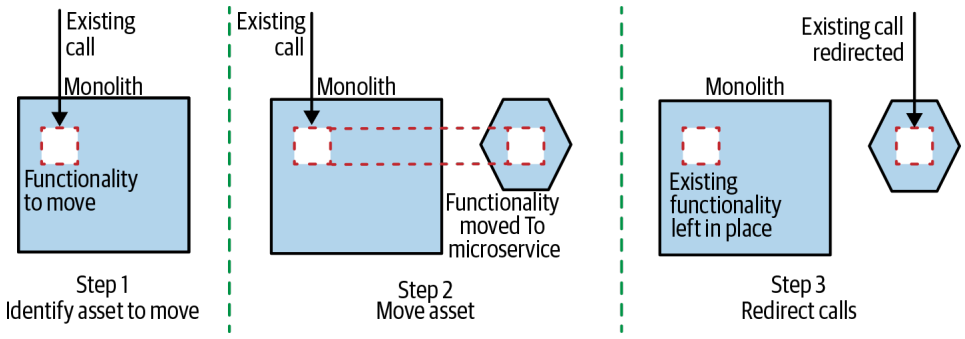



**Figure 4.1:** Basic steps of strangler pattern. [22]

Mapping the identified part means implementing it in the new architecture (e.g a microservice). Specifically, this could mean copying or rewriting the existing code. [22]

Finally, after finishing the new implementation, existing calls and dependencies can be redirected from the old architecture to the new one. [22]

The main advantage of this approach lies in its flexibility. During the migration, both architectures coexist side by side making the system functional during the whole process. Therefore, migration can be done not only in an incremental manner, but also can be paused or arbitrarily reversed by redirecting calls back to the old architecture. [22]

### 4.2.2 UI Composition Pattern

The concept of incremental migration can also be applied to the user interface (UI), the user interacts with directly, e.g. in a web browser. The *UI composition* pattern can be used for the purpose of developing and testing new visuals or it can be utilized alongside the strangler pattern, where parts of a web portal are redirected one by one to the newly developed microservices. [22]

In the case of UI, the composition can be of the following two types [22]:

- Page composition,
- Widget composition.

In the page composition, the migrated unit is a whole web page. A good use case is the introduction of new visuals to users. If a web application is made up of multiple web pages, one (or some of them) can be redesigned while the rest of the portal remains intact. This can be done for less important sections of the web portal where a possible failure would not affect users negatively, and the changes can be easily rolled back. [22]

In widget composition, the migrated unit is part of a single web page. The use case is analogous to the one in page composition. The general idea here is that a page can also be split into multiple independent parts. For example, a web-based news outlet can display on its home page sections related to different topics – politics, sport, culture, traveling, etc. Originally, all sections used

one monolithic service to obtain the appropriate data. During migration, individual sections can start redirecting their calls one by one to their own respective microservices. Again, this can be done incrementally, which helps developers migrate the whole infrastructure smoothly and carefully. [22]

### 4.2.3 Branch by Abstraction Pattern

Unfortunately, the strangler fig pattern does not work well in cases where the inbound calls come from within the existing system. Given the example illustrated in Figure 4.2, if we tried to migrate the component *User Notifications* to its own microservice, redirecting the incoming calls made by the components *Invoicing* and *Payroll* would require modification of the monolith (which could be an unwanted disruptive action). [22]

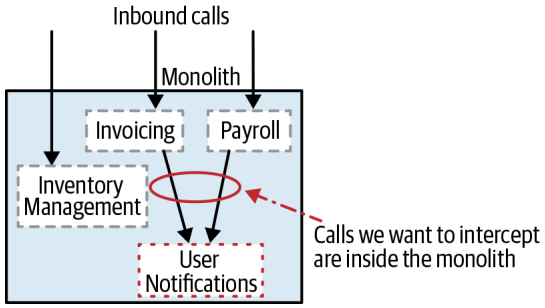



**Figure 4.2:** Inbound calls to the migrated component are coming from within the monolith itself. [22]

To overcome this problem, the *branch by abstraction* pattern can be used. It is based on having two implementations (old and new) temporarily during the migration. Then, by using abstraction, it is possible to easily switch between them. [22]

The monolith still has to be modified, but the modifications are not disruptive, and the behavior is not changed until the new implementation is ready to be switched over. [22]

The pattern consists of five steps [22]:

1. Create an abstraction of the migrated functionality.
2. For all clients using the migrated functionality, switch their calls to the newly created abstraction.
3. Create a new implementation of the abstraction.
4. Switch the abstraction to use the newly created implementation.
5. Remove the old implementation.

In the first step, an abstraction of the operations of the migrated functionality is created. This abstraction is then implemented by existing functionality. The purpose of this step is to create a generic interface for any client that

uses the migrated functionality without the need to rely on a specific implementation. The switch of all client components (*Invoicing* and *Payroll*) from implementation to abstraction is made in the second step. Both steps are shown in Figure 4.3. [22]

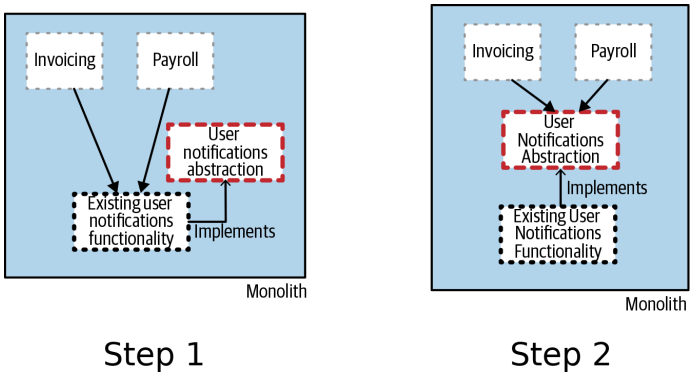



**Figure 4.3:** Steps 1 and 2 of *branch by abstraction* pattern. Adapted from [22]

The third step is concerned with the creation of a separate reworked implementation using the new microservice. This step is done independently of other developers who work on the old functionality that exists alongside the new implementation. After this step, the clients still use the old implementation, while the new implementation is still being worked on. Therefore, no changes in the behavior of the existing system have yet been made. Only when the reworked functionality is considered finished, a switch of the abstraction over to the new implementation can be done in the next, fourth, step. Once the switch has been made, all clients can also start using it via the abstraction. The process done in steps 3 and 4 is depicted in Figure 4.4. [22]

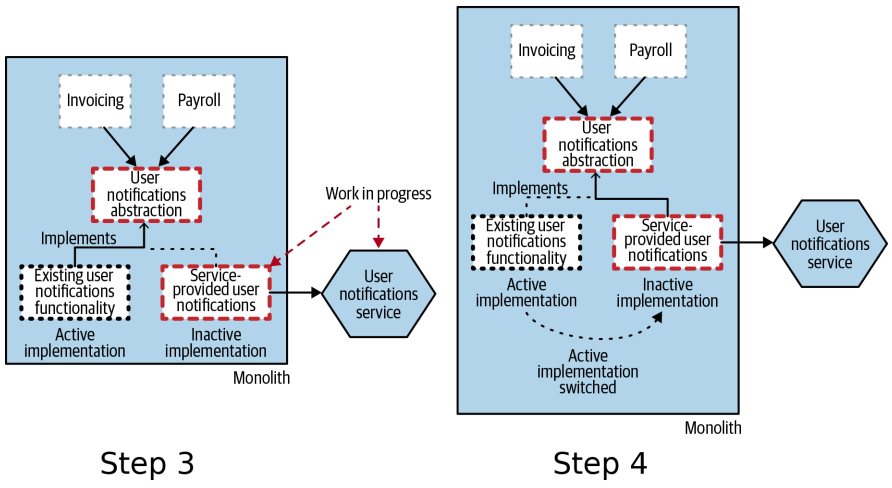



**Figure 4.4:** Steps 3 a 4 of *branch by abstraction* pattern. Adapted from [22]

At this point, it is possible to freely switch between both implementations and make any rollback if necessary.

Ultimately, when the new implementation is considered ready, the old implementation can be removed. This is shown in Figure 4.5 as the fifth and final step of the branch by abstraction pattern. [22]

Additionally, the abstraction can also be removed as an optional sixth step. [22]

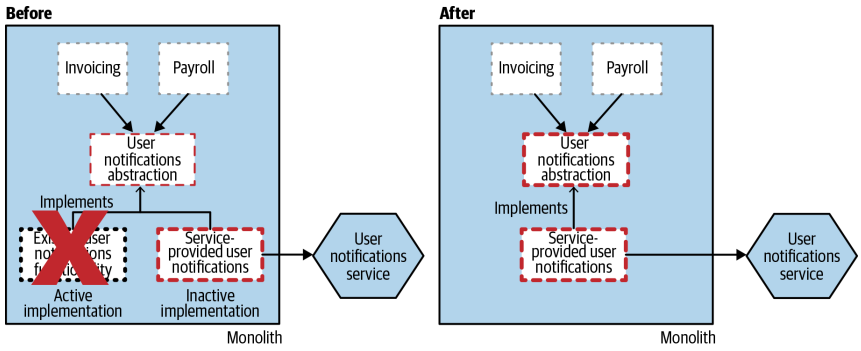



**Figure 4.5:** Final step of *branch by abstraction* pattern. Adapted from [22]

## 4.3 General Recommendations

When migrating to SOA or microservices-based architecture, it is advisable to follow some general principles. This section summarizes them.

- **There is no universal way for migrating architectures.** Every decision carries its own trade-offs, and it is up to the architect to consider the possible advantages and drawbacks. Different approaches can be suitable for different scenarios and depend on the current context. [22]
- **During the migration, try to avoid making changes in the current behavior.** New functions and bug fixes should be performed after the migration is finished. This is motivated by making the rollbacks from the new architecture to the old architecture as smooth as possible. If the new architecture incorporated new features and/or bug fixes, a rollback could potentially take these features away from customers and reintroduce the bugs. [22]
- **Implement log aggregation.** It is advisable to aggregate logs of different services in one place. This can profoundly improve any future debugging. A commonly used implementation is the so-called *ELK stack* which includes the combination of three components – Elasticsearch, Logstash, and Kibana – allowing collection and visualization of the logs. [22]

# Part II

# Practical Part

# Chapter 5

# FelSight Application

The first chapter of the practical part introduces the FelSight application, which is the subject of architectural transformation. Furthermore, the project of the application's current implementation is analyzed. The analysis includes a description of the existing architecture, the structure of the project, and the technology used. In the final part, the motivation for refactoring of the existing project is presented.

## 5.1 Application Introduction

FelSight is a web application designed primarily for students and teachers of the Faculty of Electrical Engineering (FEE) of the Czech Technical University in Prague. The application is being developed in the Center of Knowledge Management at the CTU and is available at `https://felsight.fel.cvut.cz/welcome.xhtml`.

It serves as a support tool for its users during studies and offers a variety of features. The most prominent feature is the page with timetables, as shown in Figure 5.1. Here, the user can view their enrolled courses in the current semester together with their own events and tasks.

The set of features also includes viewing time availability of rooms that can be used for studying, semester overview with public events, special mode for customizing timetable, and much more.

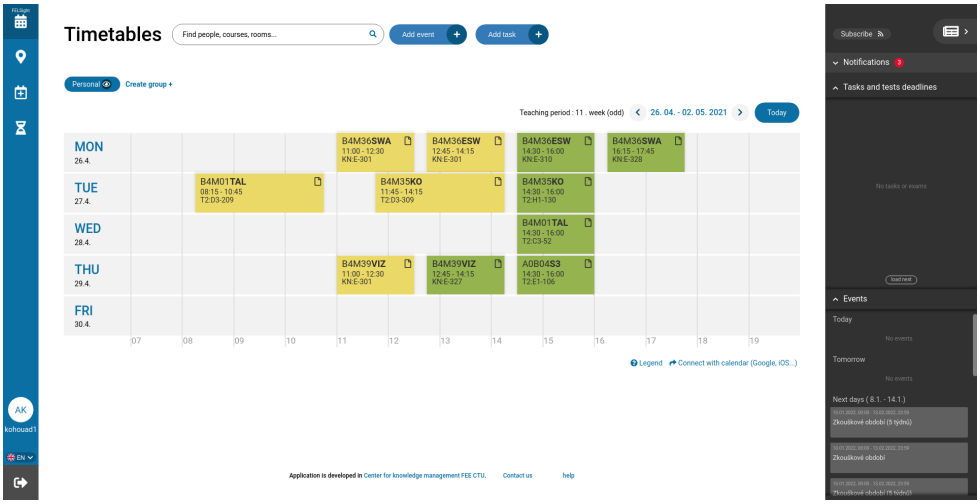



**Figure 5.1:** *Timetables* page in FelSight.

## 5.2 Original Architecture

Being the typical web application, the architecture of FelSight follows common layered pattern, similar to the one depicted in Figure 2.1.

### 5.2.1 Presentation Layer

The presentation layer represents the front-end of the application. In the case of FelSight, here, the front-end is a collection of web pages.

As the user interacts with the page and performs different actions, requests are initiated and sent to the server for further handling by the business layer.

### 5.2.2 Business Layer

The requests sent from the front-end are handled by its designated invoked method. The method contains logic that defines how the request should be processed.

Common scenarios of request processing include actions such as communication with the database, invocation of a remote API via REST, or simply updating the inner state of the server. Usually, these actions are combined within one request.

### 5.2.3 Persistence and Database Layers

In cases where communication with the database (i.e. the database layer) is performed, the persistence layer is used. It ensures connectivity to the database and data mapping between this layer and the database layer.

The mapping is necessary because of the two different representations on each side. In the database layer, data are represented as tables, and in the persistence layer, entity objects (Java objects) are used for this purpose.

Finally, the persistence layer also provides an interface for operations such as reading, creating, updating, and deleting data stored in the database. This creates an abstraction for a developer without the need to use tools such as SQL to manipulate the data in the database directly.

## 5.3 Original Project Structure

The original project consists of three modules – the web module, the EJB module, and the EAR module – as shown in Figure 5.2. This section describes each of these sections together with the technology used within them.

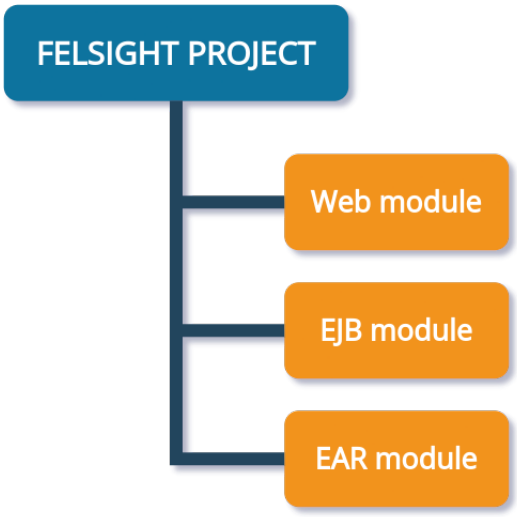



**Figure 5.2:** Structure of the original FelSight project with three modules.

### 5.3.1 Web Module

First of the modules is the *web module*. Its purpose is to provide HTML, CSS and JS files for interpretation by the web browser on the client's side.

However, because the data displayed on the web page are unique to each user, HTML files must be generated first. In FelSight, this is accomplished on the server by JSF (JavaServer Faces), the framework for building modular web pages using XHTML (Extensible HTML) files based on custom XML tags (syntactically similar to HTML tags).

An XHTML file serves as a template for a whole web page or just a part of it – a component. When the user requests a page via the web browser, the related XHTML files are evaluated into a resulting HTML file which is subsequently sent to client. The important part of the framework are *Java beans*, ordinary Java classes or interfaces. Beans are accessible from XHTML files and can be used to call their methods to retrieve data and use them in places in the template where the methods were called.

In terms of CSS files, the FelSight project uses LESS, a CSS preprocessor which allows a developer to use advanced syntax and features that are not part

of the original CSS specification. LESS files are compiled by the preprocessor into equivalent CSS files that a web browser can interpret.

Regarding JavaScript, to reduce the number of files sent over the network, the files are merged into one. This single file is then sent to the client.

Beans can hold the state related to the currently logged user and can implement its own logic. However, it is common for the request to require interaction with the database or with a remote service to retrieve the data. Such logic should not be implemented inside the beans, as they should handle logic related to front-end only, like updating the mentioned state. Any other logic (i.e. closer to the back-end) should be separated. This is what the EJB module, which provides complex functionalities and interfaces, is used for.

### 5.3.2 EJB Module

The second – and the largest – module is the *EJB module*. EJB (standing for Enterprise Java Beans) is a way of building server applications in Java. One of the fundamental features are the components that allow developers to modularize the application. The components can be composed into a larger one with the help of *dependency injection*, a mechanism of instantiating specifically annotated fields of a class with the correct implementation. This process and the lifetime of the components, also known as *beans*, is managed by the container on behalf of a developer.

EJB module encompasses server-related functions which are provided to the web module via its designated methods. Two of the most significant functions are persistence layer and a collection of REST API clients for accessing data of selected remote services needed to display on FelSight's front-end.

As already mentioned in the Subsection 5.2.3, persistence layer mediates communication with the database and exposes a simple-to-use interface to the rest of the code. In the case of FelSight, this is achieved by using the Java Persistence API standard (JPA) together with the EclipseLink provider. In order to work with the data retrieved from the database, it is mapped to corresponding Java entity classes (typically, a table in the database corresponds to one Java entity class with the columns mapped to individual fields of the entity class).

FelSight accumulates and displays data to the user from other sources than the database. As a matter of fact, majority of the education-related data (e.g. courses, semester enrollments etc.) is originally retrieved from remote services and subsequently saved in the database or put in the cache for later access. Such services are, for example, KOSapi (CTU's student information system data – KOS) or Sirius API (data reflecting exceptions in timetables such as the national holidays). The data is retrieved via the REST API by a dedicated client in the EJB module.

In addition to the two previous features, the module also provides more functionalities such as sending email or running periodic jobs to update the database. However, these features will not be further elaborated on.

### 5.3.3 EAR Module

The only purpose of the last module is to package the first two modules, the web module and the EJB module. The rationale behind this process is to create a single file for easy deployment on the application server. The resulting file is also known as *enterprise application archive*, thus being abbreviated as *EAR*.

EAR module itself does not contain any code and only has a file with a Gradle task definition for the application's build process, described in the following section.

## 5.4 Build and Deployment

This section summarizes how the original FelSight project is built and deployed in its runtime environment.

### 5.4.1 Build Process

Before deploying the application, it should first be built from the source code. To simplify the process of building the whole application, the project uses Gradle as its build system. As indicated in Subsection 5.3.3, the result of the build process is one archive (EAR) file that encapsulates the compiled web and EJB modules.

The process starts by running the Gradle task defined in the EAR module in the file `build.gradle`. The web and EJB modules also contain their own `build.gradle` file with their dependencies and tasks and are built recursively.

After the build process, the resulting EAR file contains both compiled modules in a form of compressed archives, with WAR and JAR extensions for the web and EJB modules, respectively. WAR file contains XHTML files together with the compiled and bundled LESS and JS files and bean classes with libraries. The result of the compilation of the EJB module is the mentioned JAR file with the library dependencies located in the separate `lib` directory. The general structure of the generated EAR archive can be seen in Figure 5.3.

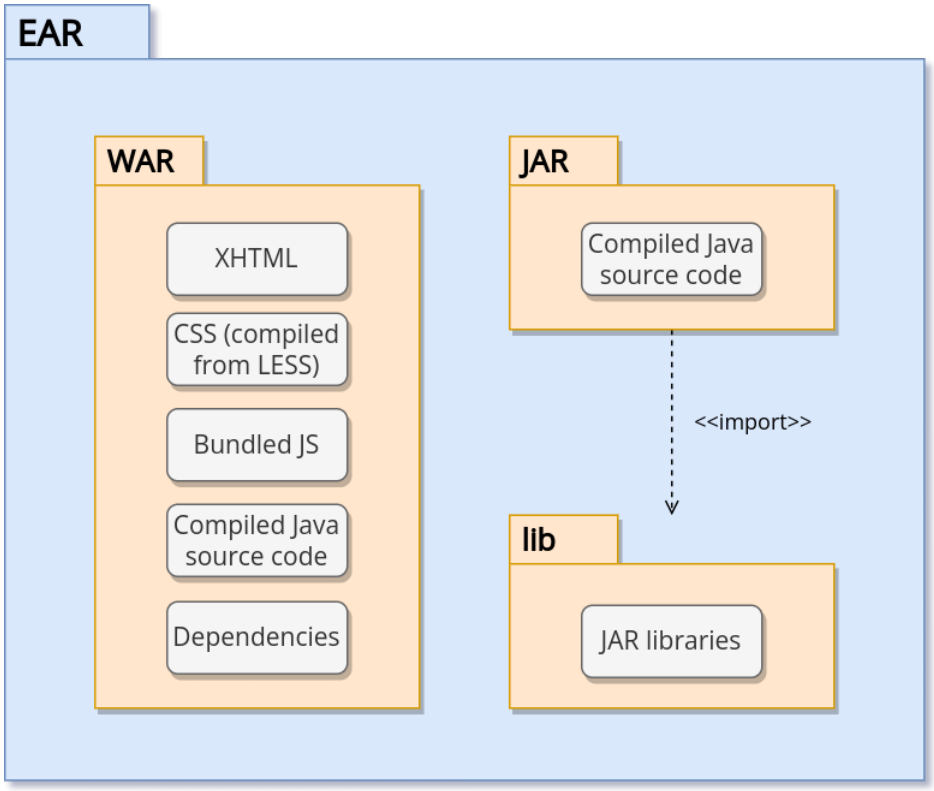



**Figure 5.3:** EAR structure overview.

### 5.4.2 Application Server and Database

After building the entire application, it can be deployed. In the case of FelSight, the application runs on Payara application server.

Apart from hosting applications, the application server, where the EAR is run, also takes care of other responsibilities. For example, it uses the JDBC interface to communicate with the database and manages pools of reusable connections to conserve resources. It is also possible to define custom JNDI resources. The application can then look up a given resource based on its ID dynamically and use it during the run-time.

Additionally, in order for the application to startup correctly, a running instance of PostgreSQL database server is expected to be available. In the Payara administration console, the proper credentials for database access must be configured.

### 5.4.3 Topology

In summary, the deployment schema of the original project could be imagined as three core parts connected in a linear topology, as illustrated in Figure 5.4. This corresponds to the layered pattern, each layer being delineated by the dotted boundaries.

The client's web browser (presentation layer) communicates with the appli-

cation server (business layer) via HTTP. The application, packaged inside the EAR file and hosted on the server, receives and processes incoming requests.

Access to the database is carried out between the database server and the application server only – the client is not participating in this communication directly in any way. Connection is achieved by JDBC. Furthermore, table data fetched from the database are mapped onto entity objects in the application. This is achieved by the persistence layer located in the EJB module, which also provides a convenient interface for executing operations over the database.

Note that the diagram does not illustrate the physical devices on which the browser and both servers are run. In reality, the application and database server could run either on different machines, or they could share one. For simplicity, remote services called by the application are left out of the diagram.

## 5.5 Motivation for Architecture Transformation

Although there is a particular separation of concerns given by the layered architectural pattern, the layers themselves incorporate a significant amount of code, making the architecture effectively monolithic.

This is true especially in the case of the business layer. Being composed of many business areas, the domain of FelSight is complex, and due to the lack of separation of those areas, the project results in a lot of clustered code. Demarcating the identified parts of the application would decompose the project into smaller parts, making it easier to handle the overall complexity.

Another example of tight coupling that can be observed in the current architecture, as shown in Figure 5.4, is the fact that the persistence layer is actually part of the EJB module. This means that – from the point of code structure – it is also a part of business layer.

Decomposing the application into logical pieces is one of the fundamental steps in the process of designing new microservice architecture which is covered in the next chapter.

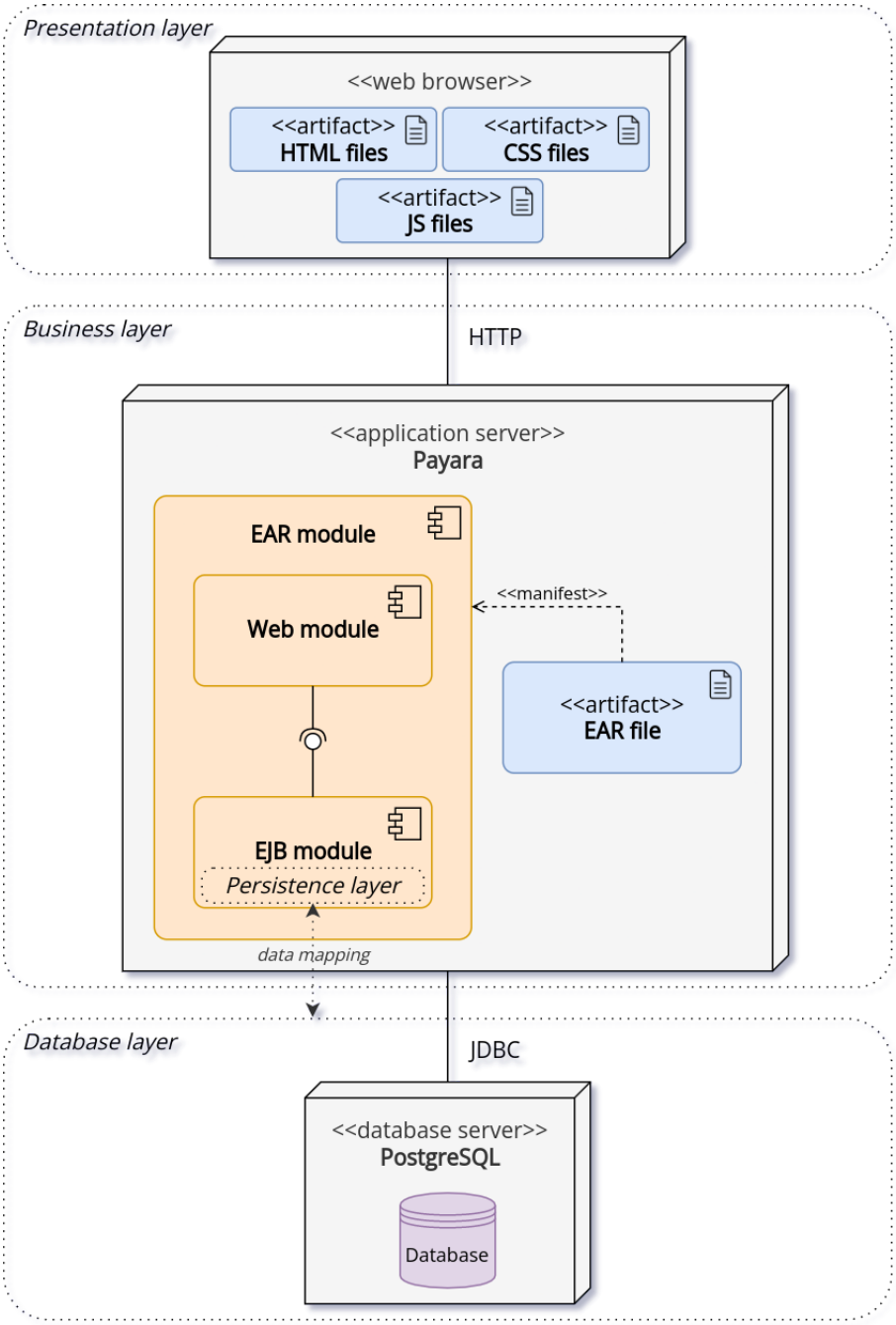



**Figure 5.4:** Deployment diagram of the original project.

# Chapter 6

# Analysis and Design of the New Architecture

After the introduction of the application's current deployment configuration and the motivation for its modification, it is possible to proceed to the details of the transformation.

## 6.1 The Vision

Before starting with analysis and design, it is important to have a general idea of the current application's state and the final result of the transformation. Having a clear picture of the future helps to describe the fundamental differences between As-Is and To-Be states influencing the surrounding environment, e.g. stakeholders. Moreover, it should introduce justifications as to *why* the change should take place at all.

### 6.1.1 Transformation Approach

Regarding As-Is state, FelSight is a monolithic application with a single codebase. Reiterating the main point of Section 5.5, one of the present difficulties of this architecture is the large amount of code concentrated in one place. This makes usability, readability, and the overall organization of the codebase difficult. Having this problem suggests the need for an architectural change.

Following the previous paragraph, the general idea is to identify individual business functions and treat them as independent components. Therefore, the general approach to transformation is to view the application as a collection of independently deployable components that communicate with each other. The configuration of components, described in the previous paragraph, can then be regarded as the To-Be state.

Furthermore, the functionality of each component should be limited to its designated feature only. This improves the scalability of the component. Furthermore, changes made to one component have little or no influence on other components of the architecture. [23]

### 6.1.2 Aspects Influenced by the Transformation

It is important to mention that the eventual transformation process also influences three aspects, equivalent to the following questions:

- Where is the application logic located?
- Where is the application's data stored?
- Who is assigned a particular part of the application?

#### Application Logic and Data Storage

In the current (As-Is) state, the application's logic and data are stored in one place; logic related to individual features is part of a single program (running process), and the data are stored in a single database (server). On the other hand, when transformed into independent parts, logic and data are related to their assigned features only and are physically isolated from each other (To-Be). This is depicted in Figure 6.1 with blue and violet elements. Eventually, logic related to a given feature only uses data from that identical feature. The components of this configuration communicate with each other via a dedicated protocol (typically REST API) while still not interfering with others due to physical isolation of code.

#### Team Work

The third aspect affected by the change is the team that works on the project. When working on a single codebase, project team members should be assigned tasks so that collisions with other members' tasks are minimized. This might not be an easy task, and in some cases collisions are unavoidable anyway. However, with the features distinctively separated from each other, assigning tasks to members is transparent and imposes no conflicts, again due to the physical separation of code.

This makes a strong case for developers working on front-end and back-end simultaneously (related to the same feature). In the As-Is state, the FelSight application couples its front-end code with its back-end code (given by the nature of JSF). While there is some separation provided by bean classes between front-end and back-end, code for both sides is still contained within one project, allowing anyone to change anything unnecessarily. If front-end and back-end parts are physically separated (including the back-end itself being split into smaller pieces), it will be possible to assign team members to individual components, and thus define the scope of their responsibilities bounded by the component itself. This is illustrated in Figure 6.1 with yellow elements.

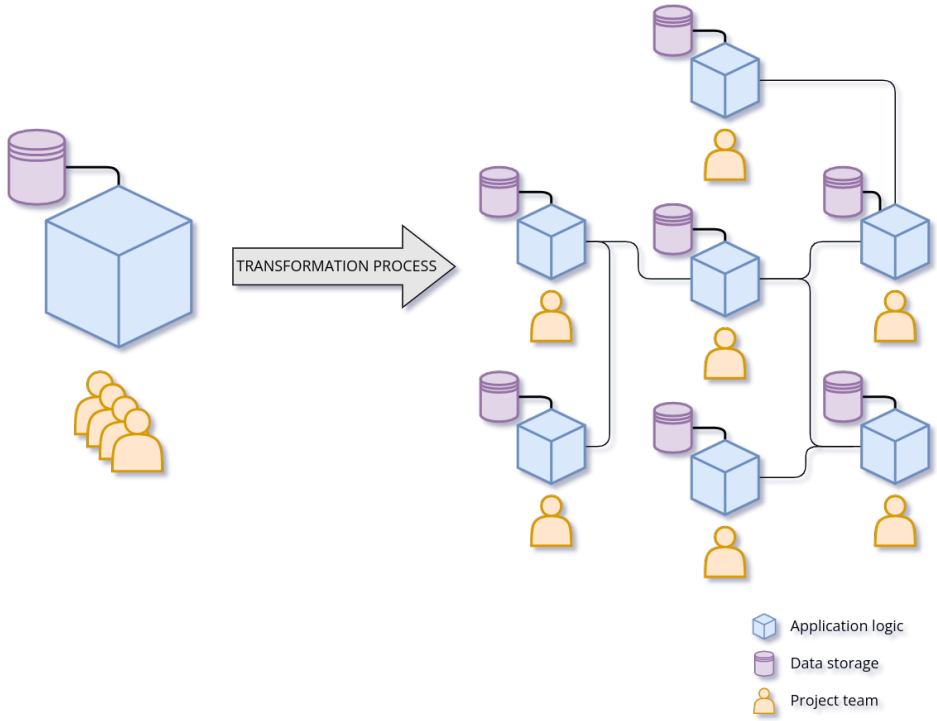



**Figure 6.1:** Schematic illustration of As-Is and To-Be states of FelSight's architecture.

### 6.1.3 Summary

In summary, the vision of transforming FelSight's architecture is based on its feature-wise breakdown. This carries the ability to prevent code expansion in one place and make development much more flexible and structured.

The transformation influences three aspects – application logic, data storage, and teamwork. The first two aspects are tightly connected, as the application logic works with data that bring added value.

Finally, teamwork is also strongly influenced by the process. In the As-Is state, team members work together on a single code base where any member can be affected by any other member by conflicting changes. In the To-Be state, however, teamwork is spread across multiple components (different applications), preventing any conflicting changes by different members and making it easier to define the responsibilities of each member that are scoped by the corresponding feature.

## 6.2 Transformation Process Outline

Given that FelSight is a complex application, the entire process of completely transforming its monolithic architecture into a new architecture will span a longer period of time.

To ensure that the process is systematic, it is broken down into two phases.

### 6.2.1 Phase 1

The first phase is focused on the analysis and identification of individual components. Following the previous section, the identification is based on the features of FelSight.

The components identified in this phase are candidates for independent deployable units. These units are expected to be part of the final microservice architecture. In addition, one or multiple components could serve the purpose of other applications developed in the Center of Knowledge Management at the CTU.

The identification and analysis are covered mainly by the practical part, while also supported by the POC project, being the part of the thesis evaluating the analysis results.

The steps of the first phase are described in Section 6.3.

### 6.2.2 Phase 2

In the second phase, the transformation itself takes place. It requires several actions and decisions to be performed for *each* potential microservice. In general, they include the scope definition of the service (functionality, operations), design of the service's database model, development of the service, including its REST API, and migration of any existing data from the original FelSight database to the database of the service, if necessary.

This thesis will not cover this phase because of its extensiveness. Therefore, it is estimated that this phase will continue for a more extended period of time, possibly years, until the transformation process is finished.

## 6.3 Analysis Steps

Being based on a feature-wise breakdown into individual components, the initial analysis consists of the following general steps:

1. Identify existing business features directly usable by users.
2. Describe actions for each identified business feature.
3. Identify relations between actions and entities used by business features.
4. Based on the identified relations in the previous step, find candidates for services. If there is an entity used by more than one service (has multiple relations), it is a candidate for being its own separate service.

## 6.4 Feature Overview

This section summarizes the features, FelSight users can use. A feature might not necessarily have to be a candidate for a future microservice.

### 6.4.1 Events and Tasks

Events are one of the key features of FelSight. Based on the user's ability to interact with an event, they can be regarded as external or custom events.

External events are usually public events held by the university or important points in time during an academic year. These events themselves can only be read and cannot be modified in any way by the user.

Custom events, on the other hand, can be created, read, modified, and deleted by the user they are tied to.

Tasks are a special type of event. Similar to custom events, they can also be managed by users. Both external and custom events can be viewed in the timeline, sidebar, and in the *Semester overview* section.

### 6.4.2 Groups

Users of FelSight can create groups and invite other selected users. The purpose of this feature is to allow users (e.g. classmates) to share timetables with each other. Moreover, users in a group can create shared events (seen only by members of a group the shared event is a part of) as a way of arranging private meetings (e.g. for the purpose of working on a school project together).

Invited users must first accept the invitation if they want to be part of the group to which they were invited, or they can choose to decline the invitation.

Any user can create a group. However, only group owners can modify or delete a group they originally created.

### 6.4.3 Searching

The searching feature is spread over multiple sections of FelSight and includes a variety of entities that can be queried, namely people, courses, rooms, and groups.

For particular entities, specific filters can be used. For example, in the advanced course search, courses can be filtered by semester, time period, number of credits, department, and teacher.

### 6.4.4 Rooms

Rooms represent another important type of entity in FelSight. They are used mainly in combination with courses and the *Study rooms* section.

Study rooms can be used to view rooms available to students for self-studying during the times when no teaching takes place (according to the official schedule). Users can also see whether a room is currently occupied due to an ongoing teaching session. This makes it easier to find a vacant room without the need to look up the official teaching schedule for that room.

### 6.4.5 Timetables

Timetables are one of the core features of FelSight, along with events. They make use of some other features already mentioned above. In FelSight, there are two sections related to timetables (excluding the *Study rooms* section, which – in terms of functionality – is related to rooms, rather than timetables).

Users can use the *Timetables* section to manage custom views of (shared) events and courses. Views are created based on different criteria, such as rooms, people, and groups. Multiple views can be combined together or hidden on demand. The default view includes students' enrolled courses and events arbitrarily added by the user.

The second section is *Timetable Creation Mode*. Students can use it to search for any course and add it to their timetable alongside their enrolled courses in their default view. Adding courses, however, does not have the effect of official enrollment in the course and is relevant only for viewing the courses in FelSight timetables.

### 6.4.6 Notifications

Notifications are another example of a cross-cutting feature of FelSight. Their purpose is to notify the user of a wide range of different events that occur throughout the application. A notification can be an alert about an upcoming exam, a group invitation, a public event, or a modification of an event, task, or group.

Thus, notifications must be based on some action related to a certain entity.

### 6.4.7 Building Plans

FelSight also incorporates a section with the plans of three locations where teaching at FEE takes place. The feature is realized by including an external front-end component to a page within FelSight. The component communicates with its dedicated external service separated from the FelSight application logic.

### 6.4.8 Moodle Evaluation (Grades)

One of the latest FelSight features is the integration of faculty Moodle grades. This feature allows students to view the grades given by teachers through the official faculty Moodle website directly inside FelSight. Grades are read-only and cannot be created, nor modified in any way within the application. The feature already exists as its own microservice.

### 6.4.9 Food Menu

Finally, FelSight users can also view daily menus for different CTU canteens. Data are read-only as well and are also provided by a separate service, similar to the Moodle Evaluation service.

## 6.5 Actions, Entities and Relations

Having identified all important business features, the next two steps involve summarizing all user actions and tracking all entities directly affected by those actions.

The results of the two steps are summarized in Table 6.1. This table is described in detail in Subsection 6.5.3.

### 6.5.1 Actions

Actions are based on the features identified in Section 6.4 and represent operations on data that can be initiated by users in the application.

Typically, actions are one of the four basic operations on data (i.e. entities) – *create*, *read*, *update* and *delete* – abbreviated as *CRUD*. Users, however, do not always have permission to perform all four types of operations. In fact, a considerable amount of operations in FelSight are read-only.

Important user actions are shown in the left column of Table 6.1.

### 6.5.2 Entities

In most cases, features correlate with entities of the same or similar name. Typical examples include events, rooms, or groups. However, it does not imply that an action is always isolated from other entities.

A large number of entities are connected to other entities. For example, a custom event can have a group (shared event), a person, or a room assigned to it. For this reason, performing an operation on the original entity (the custom event) *could* affect an entity linked to it (e.g. when assigning a group to the custom event).

Nevertheless, most entities originate from external sources. Usually, these entities are read via APIs like KOSapi or Sirius API in bulk and are saved by FelSight in the local database for quick access. Information from these two sources is often used in combination with other entities in FelSight, but it is never modified per se. This is because FelSight does not allow such modifications, and it would not be reasonable to modify the entity in the first place (as its information comes from the external system). For example – from the FelSight user's view – it would not make sense to alter names of courses or codes of rooms.

The relevant entities are shown in the upper part of Table 6.1.

### 6.5.3 CRUD Matrix

Due to the high connectivity of the entities, actions often affect more than one entity through transitivity. Using the example with custom event creation, since a custom event in FelSight can have a group, a room, or a person assigned to it, these additional entities must be retrieved first so that the user can, e.g. choose a proper value from the available list.

Each action can influence any number of available entities. In order to visualize all existing relations among actions and entities, a tool called *CRUD matrix* was chosen.

Proposed in *Information Systems Planning Guide* and published by IBM in 1978, the CRUD matrix is a table that enumerates entities and operations (originally called data classes and processes) used in a system. Entities and operations are arranged in horizontal and vertical directions, respectively. To indicate a relation between an entity and an action, one or more of the letters *C*, *R*, *U* or *D* can be placed in the appropriate cell of the matrix. Originally, in IBM's publication *Information Systems Planning Guide*, only the letters *C* and *U* would be used. *C* would indicate the data *creation* by a process, while *U* would indicate *usage* of the data by a process. [24]

Nevertheless, all four letters are rather used today (with the same meanings, as explained above in Subsection 6.5.1). CRUD matrix for the actions and entities identified in the FelSight application is represented by Table 6.1.

For example, when creating a group, its members must be chosen. Therefore, it is necessary to retrieve the information about them contained by *Person* entity. Furthermore, when a group is created, a notification about the group invitation is created and sent to all members.

Additionally, the following facts about the matrix shown below must be noted:

- A task is a type of an event with a course assigned to it. In order to highlight this fact, it is represented as a separate row in the matrix.
- The matrix includes only the most important and relevant entities. In FelSight, however, the number of entities used is much larger.
- Actions related to the Food Menu feature are omitted, as its entities (e.g. Canteen) are not related in any way to other entities in FelSight. However, it will still be considered in the final architecture configuration.

| **Action** | **Entity** | | | | | |
|---|---|---|---|---|---|---|
| | Event | Group | Room | Person | Course | Notification |
| CRUD a custom event | CRUD | R | R | R | | C |
| CRUD a task | CRUD | | | | R | C |
| Read an external event | R | | | | | |
| CRUD a group | | CRUD | | R | | C |
| Read a room | | | R | | | |
| Searching | | R | R | R | R | |
| Managing timetables | R | R | R | R | R | |
| RU a notification | R | R | | | | RU |
| Read course grades | | | | | R | |

**Table 6.1:** CRUD matrix.

## 6.6 The Proposed FelSight Microservice Architecture

The final step of the analysis is to find candidates for components in the resulting microservice architecture. The previously designed Table 6.1 can be used for this.

In this case, the chosen approach involves a summary of features (or, more precisely, actions relating to particular features) using a given entity. If an entity is used by more than one feature (i.e. it is used by any other feature than by its own corresponding feature), it should be chosen as a candidate for a separate service.

The reasoning behind this approach is that an entity that is used repeatedly in different contexts should be extracted into its own service, as it can then be used independently by any other service that requires it as a dependency. As a result, this increases the scalability and reusability of the individual components in the system and also eliminates the need to reimplement the functionality every time it is required by any service.

After highlighting the actions of individual features, as shown in Table 6.2, it can be seen that every entity is used by more than one feature. For example, *Person* entity is used by event (green), group (blue), searching (purple), and timetable management (red) features.

| **Action** | **Entity** | | | | | |
|---|---|---|---|---|---|---|
| | Event | Group | Room | Person | Course | Notification |
| CRUD a custom event | CRUD | R | R | R | | C |
| CRUD a task | CRUD | | | | R | C |
| Read an external event | R | | | | | |
| CRUD a group | | CRUD | | R | | C |
| Read a room | | | R | | | |
| Searching | | R | R | R | R | |
| Managing timetables | R | R | R | R | R | |
| RU a notification | R | R | | | | RU |
| Read course grades | | | | | R | |

**Table 6.2:** CRUD matrix with features distinguished by colors.

This yields the following entity-based candidates: event, group, room, person, course, and notification services.

In addition, there is also a special case of a feature-based candidate – timetable management. It is related to many entities. However, there is no entity such as "Timetable". This is because a timetable is a collection of different entities linked together. Since it does not belong to any other service, it should exist as a service on its own.

After including the already existing services – i.e. Moodle Evaluation and Food Menu services – the full list of candidates is following:

- Event Service,
- Group Service,

- Room Service,
- Person Service,
- Course Service,
- Timetable Service,
- Notification Service,
- Moodle Evaluation Service,
- Food Menu Service.

It is important to note that a feature does not necessarily have to be transformed into its own corresponding microservice. An example of this case is the Searching feature, which consists only of read-only operations on different entities (which can be implemented by individual entity-based services) and does not provide any additional functionality.

The generic schema in Figure 6.2 illustrates the dependencies of the individual services enumerated above. For example, *Event Service* requires data from *Group Service*, if the event is *shared* in a group created by a user in FelSight. *Food Menu Service* represents the special case in which there is no relation (in terms of data) to other services, since the data originate only from its own subdomain.

All services form a particular cluster of FelSight back-end microservices, each covering a specific part of the application's domain. The services are invoked by a separate front-end application (as depicted in Figure 6.2). However – in contrast to back-end services – it is not decomposed into individual business areas and is de facto represented by one service.

In the following chapter, one of the services on the list is chosen and its implementation is demonstrated in the three-step migration process.

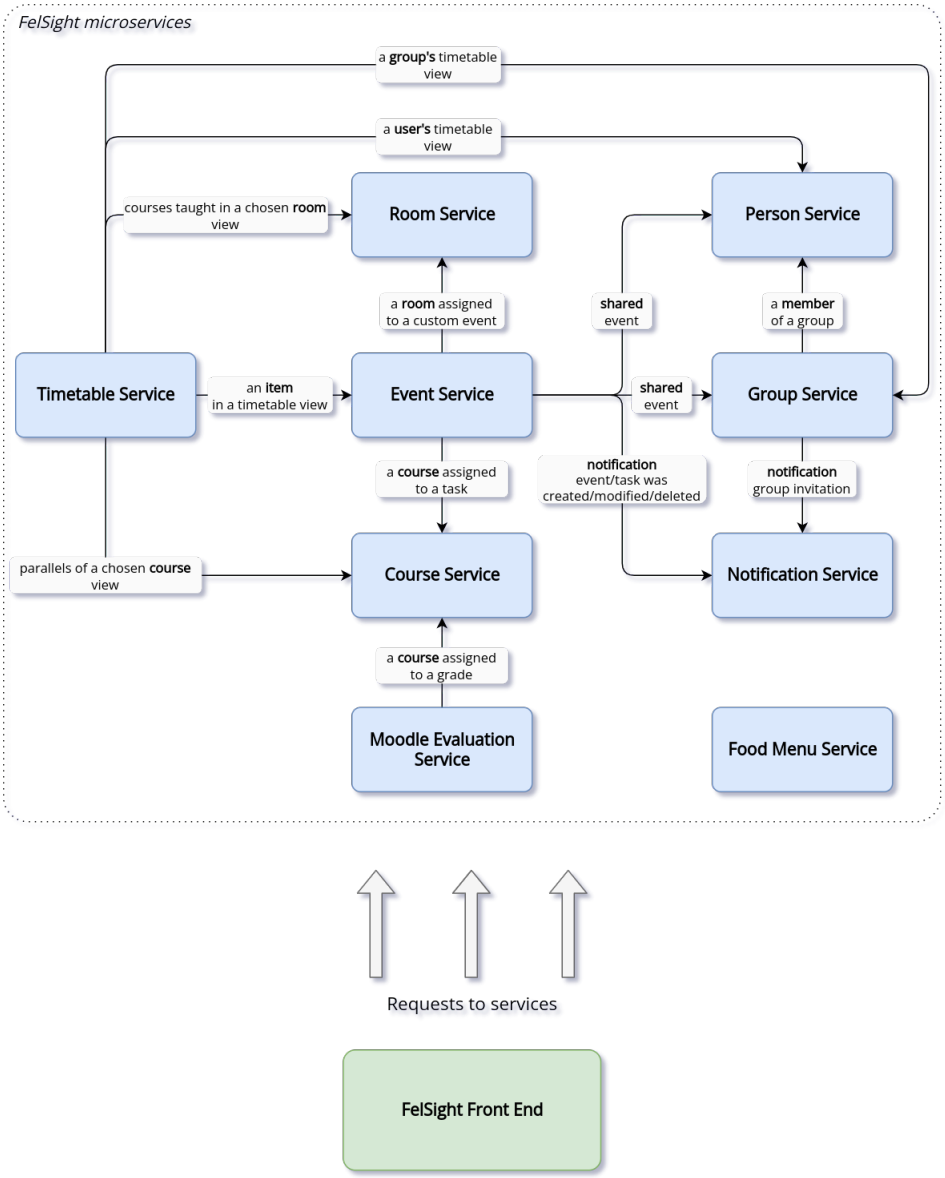



**Figure 6.2:** FelSight services and their dependencies.

# Chapter 7

# POC Implementation

The final chapter covers the POC implementation. First, the purpose and scope of the POC are outlined, together with the steps taken during the process. The last part of the chapter presents a possible future development workflow associated with the approach proposed by the POC. Finally, the chapter discusses possible security approaches in the future development of the FelSight microservice architecture.

## 7.1 Purpose and Scope of the POC

The POC aims to demonstrate the migration process of a minor part of the original architecture.

On a higher level, this means choosing one of the components identified in Section 6.6, implementing its microservice, and linking it to the original monolithic application. This approach supports incremental migration (based on the strangler pattern described in Subsection 4.2.1).

Following the previous paragraph, two outputs result from the implementation:

- Microservice, implemented for the chosen feature.
- Modified original FelSight project, connected to the implemented microservice, using its exposed interface.

The newly implemented microservice provides primary data on the CTU rooms. The data retrieved from the service are read-only, keeping implementation simple while still demonstrating the migration process that applies to other identified services.

The rest of this chapter discusses the details of microservice implementation and its integration with the original monolithic project.

## 7.2 Migration Process

The methodology proposed in this POC implementation consists of several essential steps. The process was designed with an emphasis on the automation

and distribution of reusable artifacts. This allows smoother integration with other projects, specifically front-end and other services.

The steps can be briefly described as follows:

1. Define API of the service.
2. Implement desired functionality of the service.
3. Integrate the service with the monolith via REST API client.

It should be noted that the third step is mainly for the purpose of demonstrating how the service, implemented in the second step, can be used by other applications. It is assumed that in the future the front-end of FelSight will be rewritten with the usage of modern front-end frameworks or libraries (most likely React), which is not covered by the thesis. Therefore, an existing code is modified to demonstrate the communication between the front-end and back-end.

Each of the three steps is described in detail in the following sections. Screenshots accompanying second and third steps can be found in Appendix C.

### 7.2.1 API Definition

Before starting with the implementation of any service, it is crucial to have an idea of the operations and data that it should provide.

The service is expected to provide data represented in JSON data format. This widely used text-based format is convenient for transferring data provided by REST API services.

For an accurate description of the service's interface, the *OpenAPI Specification* is used. This standard provides a universal and human-readable way of describing APIs. Additionally, by being language-agnostic, it is possible to define expected functionality before choosing the target programming language. Finally, thanks to its universality, OpenAPI Specification is perfect for use with code generators (POC makes use of this possibility). [25]

#### Operations

In the case of the room data service, its API specification (in YAML format) defines two read-only operations. The first operation returns a list of rooms, whereas the second operation can be used to find data about a specific room, identified by the room code.

The definition of the first operation is shown in Figure 7.1. It is identified by the URL path `/rooms` and GET HTTP method. In addition to the descriptive properties `tags`, `operationId` and `parameters`, the operation also accepts three optional parameters. These can be used for filtering and paging purposes (in order to limit potentially unnecessary data sent over the network per request).

Most importantly, each operation must specify the nature of its response. OpenAPI Specification enforces this by the obligatory `responses` property.

This property allows enumerating all possible HTTP response status codes together with the data schemas that the operation can return. [25]

```yaml
paths:
  /rooms:
    get:
      tags:
        - room
      operationId: getRooms
      summary: Returns a collection of rooms
      parameters:
        - $ref: '#/components/parameters/limitQueryParam'
        - $ref: '#/components/parameters/offsetQueryParam'
        - name: forSelfStudy
          in: query
          description: Whether the rooms returned are available to students
          required: false
          schema:
            type: boolean
          example: true
      responses:
        200:
          description: OK
          content:
            application/json:
              schema:
                $ref: '#/components/schemas/RoomArray'
```

**Figure 7.1:** Extract from the OpenAPI specification file – operation for retrieving a list of rooms.

### Data Schemas

One of the essential features of OpenAPI Specification is the definition of data schemas that can be reused repeatedly throughout the specification file. Schemas are defined in a separate `schemas` section and can be referenced from other places in the file by `$ref`.

For a successful response, the service is expected to return a list of rooms. Here, a successful response refers to an HTTP "200 OK" response, defined by the 200 property in the `responses` section in Figure 7.1. Data should be returned in JSON format and the schema used in the response is `RoomArray`.

As shown in Figure 7.2, `RoomArray` is a user-defined schema and represents an array of `Room` objects (also defined as a separate schema). A `Room` is defined as an object with attributes such as code, name, and locality. Some attributes also refer to other schemas (defined in the same specification file); however, these will not be covered.

After creating the API specification file, it can be used with code generators. This is covered in the next step.

```yaml
Room:
  type: object
  properties:
    capacityForExamining:
      type: integer
    capacityForTeaching:
      type: integer
    code:
      type: string
    locality:
      type: string
    forSelfStudy:
      type: boolean
    name:
      $ref: '#/components/schemas/LanguageField'
    division:
      $ref: '#/components/schemas/Division'
    availability:
      $ref: '#/components/schemas/AvailabilityArray'
    access:
      type: string
    type:
      type: string
RoomArray:
  type: array
  items:
    $ref: '#/components/schemas/Room'
```

**Figure 7.2:** Extract from the OpenAPI specification file – definitions of `Room` and `RoomArray` schemas.

### 7.2.2 Service Implementation

The second step of the migration process is the implementation of the new service. It is the most time-consuming step, despite the fact that it primarily involves moving existing logic. This is due to the fact that it is required to define and implement the service's API logic, identify all sources of the data and integrate them with the service, and determine whether the service needs its own dedicated data storage.

In the case of FelSight architecture migration, decisions regarding these aspects are mainly based on the current implementation of the monolith. Additionally, the technologies used in the implementation of the service are

discussed along the way. The basis of the implementation of the room service is a Spring Boot application, based on Spring, the popular Java framework.

However, given that the original application was implemented in Java EE, there are some differences that have to be taken into account during the migration process. Annotations are a good example – e.g. `@Inject` used in Java EE shall be replaced with `@Autowired` in Spring.

For quick project configuration, the web tool *Spring Initializr* was used to generate all necessary baseline files. [26] The configuration can be seen in Figure C.1.

### Controller Layer

In order to expose RESTful service's operations, the controller layer must be defined. This implies defining all said operations, including their parameters and responses, right in the code. However, in this case, the specification file created in the first step is used to generate these definitions (also known as *stubs*) which are subsequently imported into the code. Generating server stubs from the specification allows developers to include functional REST endpoints without the need to write them manually in the target programming language. The generated stub for the operation `getRooms` (defined by the specification, as shown in Figure 7.1) can be seen in Figure C.2.

In the room service project, the generated stubs are used to include a custom logic via polymorphism – inheritance, in particular. Specifically, the logic of the method `getRooms` is overridden by implementing the generated interface `RoomApi` (the method `getRooms` is part of), as shown in Figure C.3. This approach also influences the project structure and the build process, discussed below.

### Data Sources

It is common for microservices to communicate with other external services and use their data. In FelSight, room data is aggregated from two different external sources – KOSapi (whose documentation is available from [27]) and a CTU webpage with a list of rooms, available to student for study purposes beyond the officially scheduled teaching sessions (available from [28]).

Therefore, the room service project implements HTTP clients for the invocation of these two services and aggregates the retrieved data. The data are then provided to applications calling the service through operations at the controller layer.

### Database and Resource Updaters

In a simple scenario, data could always be obtained from external sources and returned within each call. However, making HTTP requests to external services during every incoming request imposes additional overhead and lengthens the overall processing time of the request. Furthermore, such

HTTP requests are unnecessary when the data do not change frequently and the application is not required to provide real-time data.

For this reason, the back-end of FelSight stores the data in its own database and sends it to a client every time a particular web page is visited. However, the database does have to be periodically updated in order to provide fresh data. Therefore, FelSight employs components called *updaters*, responsible for periodic updating of stored data. Updaters are activated every day, preferably during low-traffic times (e.g. after midnight), to retrieve fresh data from external sources and subsequently use them to keep entries in the database up-to-date. The same pattern applies to the corresponding service that uses the PostgreSQL database to store room data. The updater model described above is depicted in Figure 7.3.

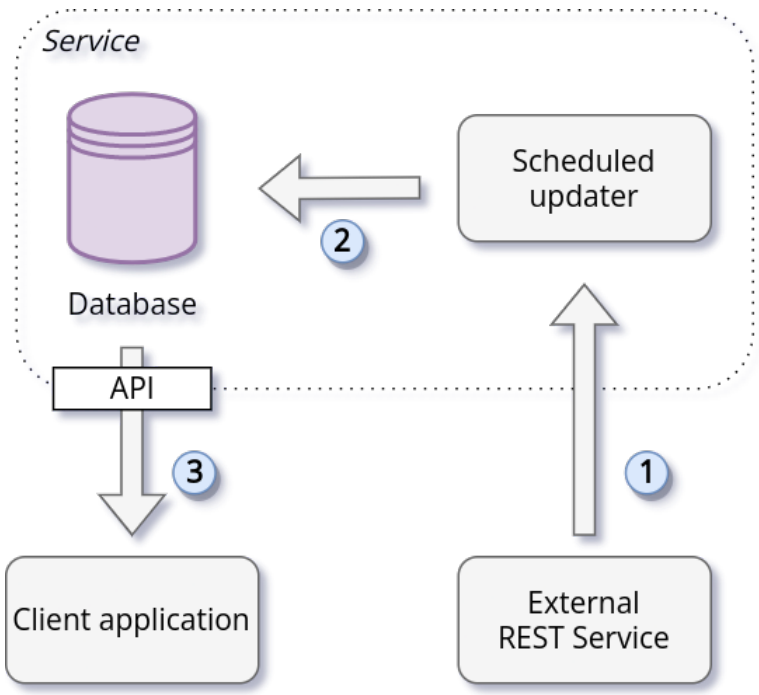



**Figure 7.3:** Updater model used by FelSight.

#### Project Structure

Regarding the structure, the project consists of two core modules – `app` and `specification`.

The purpose of the `specification` package is to separate the generation of server and client code from the rest of the project. Initially, the package contains a single OpenAPI specification file `latest.yaml`, parts of which are shown in figures 7.1 and 7.2. The Gradle build file in this package contains custom task definitions that can be used to generate server and client source code.

The project is configured in such a way that the `app` module depends on the `specification` module. When the project is built, Gradle first generates the server source code from the specification by running the Gradle task `generateApiJavaServer`. The task uses the *OpenAPI Generator Gradle Plugin* (available from [29]) which provides granulated configuration of the generator. For example, it is possible define name of the packages with the generated sources or name patterns of Java classes. The task definition is depicted in Figure C.9. In the next step, the generated code is compiled and packaged into a JAR file. Finally, the `app` module, using the JAR file as its dependency, is built. The entire build process of the project is illustrated in Figure 7.4.

The `app` module contains multiple base packages that contain the source code of the service. The most significant packages include `db` (database entities and operations using Spring Data JPA), `external` (the two HTTP client implementations discussed in Section Data Sources), `rest` (controller layer), and `update` (updaters). The module can be built, packaged into a single JAR file, and eventually deployed.

### 7.2.3 Integration with the Monolith

Following the implementation of the service is the step of its integration with the original project. In short, integration with the existing project means switching data sources; instead of reading from its large database, the monolith will retrieve room data from the running service via HTTP requests.

#### Service Client Library

The straightforward solution would be to implement an HTTP client with the help of a Java library for making HTTP requests. This solution, however, has two drawbacks.

First, it requires the implementation of methods to call individual operations exposed by the service together with all their parameters and response objects defined by the specification. As a result, a considerable amount of boilerplate code is created in the process. The second disadvantage emerges when a microservice is used by more than one application. The client has to be reimplemented in the source code of each application using the service.

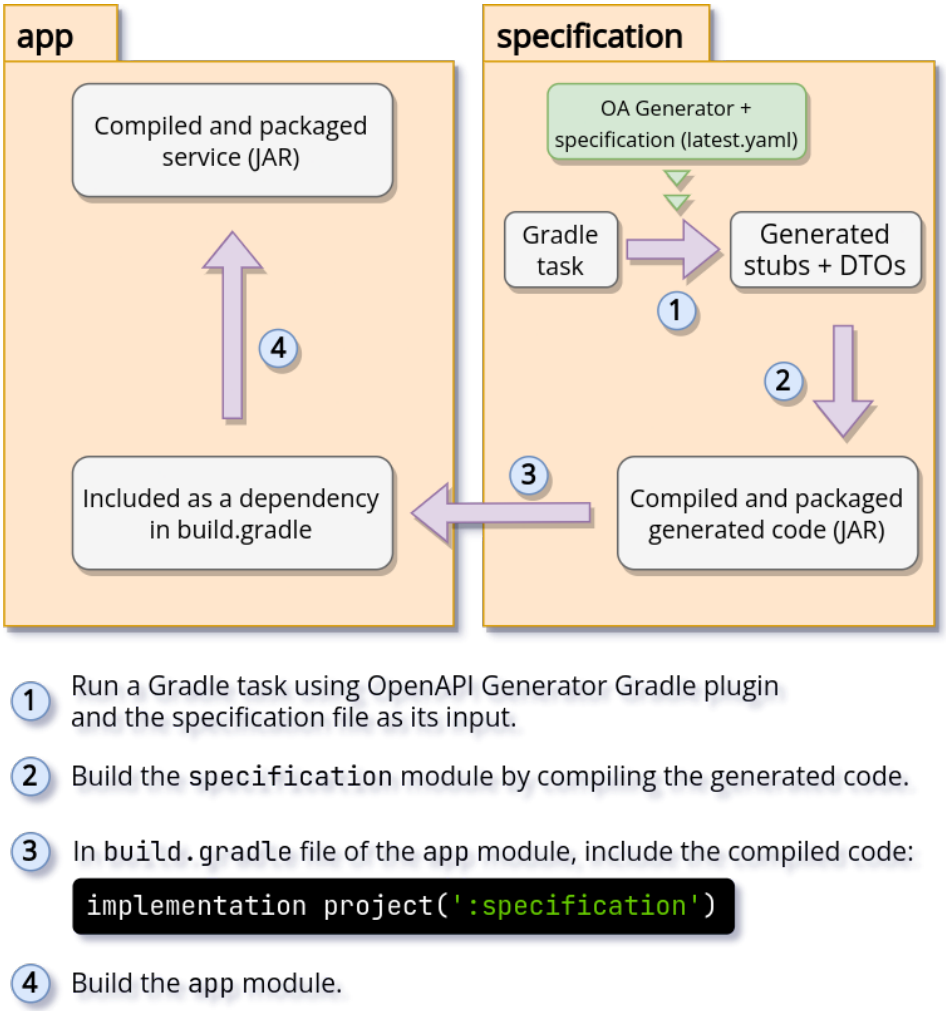



**Figure 7.4:** The process of building the microservice project.

Both drawbacks can be eliminated by having a separate client library for the service. Every application that intends using the service can download it and use it as a dependency. The library hides the details of the implementation of the HTTP client in the interface that it provides.

The client library for the room service is generated from the specification, similarly to the server part. Furthermore, the generated code is packaged and published in the GitLab package registry, so it can be downloaded by other applications. All of these actions are performed by the GitLab pipeline. Both the package registry and the pipeline are part of the same GitLab repository, where the room service project is stored. This process is part of the proposed development workflow, discussed in Section 7.3.

### Call Redirection

The final part of the migration process presented as part of the POC illustrates how the previously implemented service can be called from an application. For these purposes, the code from the original FelSight project is used. However, this requires that existing calls be redirected to the running instance of the room service.

In the original project, the room data is currently retrieved directly from the database using the class `RoomSession`. The new implementation is provided by the class `RoomServiceClient` using the generated client library code. The source code for this class is shown in Figure C.4. The class extends the generated `RoomApi` class providing the method `getRooms` with three parameters, as defined in the specification file (Figure 7.1).

While it is possible to change the implementation class where needed (by using `RoomSession` or `RoomServiceClient` directly as a type of variable), the better approach is to redirect the call by using the *branch by abstraction* pattern, discussed in Subsection 4.2.3. The classes now implement a new `RoomClient` interface, declaring a method with one or more methods originating from the old implementation – `RoomSession`. The code where the methods are called then refers to the interface, rather than to the implementation classes.

Figure 7.5 illustrates how the class model is altered when the *branch by abstraction* pattern is applied. With this model, it is possible to switch between implementations without the need to modify the type of the variable.

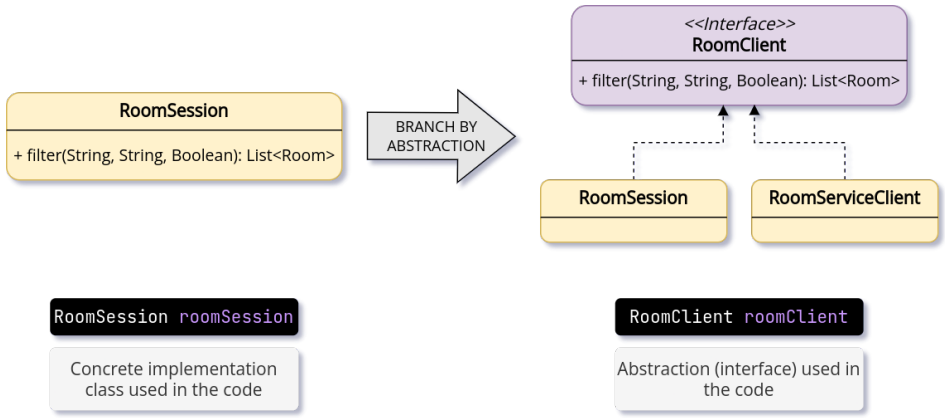



**Figure 7.5:** Changes made in code by applying the *branch by abstraction* pattern.

The abstracted method here is called `filter`, accepts three arguments and returns a list of `Room` entities. To minimize the amount of modification of the old code, the method declaration at the interface level must be identical to that in the original implementation class (`RoomSession`).

The new implementation class (`RoomServiceClient`) needs to be adapted to the existing declaration (now moved to the interface), i.e. it must accept the identical set of arguments and must have the same return type. This fact might be troublesome, since the room data retrieved from the service

are represented by a different class – a DTO class, generated as a part of the client library – that cannot be used as a return value. The DTO Java class generated by the OpenAPI Generator is shown in Figure C.5.

The DTO class has to be mapped to the target class first. To avoid having to manually implement the attribute mappings, the tool *MapStruct* was used. This library generates the necessary implementation for method signatures and proper annotations provided by the user.

Figure C.6 shows two interface methods for converting a `RoomDTO` object to a `Room` object and for converting lists of objects of these two types. The implementation class, generated by *MapStruct*, is then used when calling the mapping methods on the interface. The `RoomServiceClient` class can now the return the compatible type declared in the `RoomClient` interface by calling the mapper method `fromDto` on the interface `RoomMapper` (Figure C.4).

This last step of integrating the new service with the monolith successfully completes the migration process of a minor business feature.

## 7.3 Development Workflow

With the migration process fully covered, it is useful to describe the possible development workflow in the development team.

The core part of the workflow is the specification file; it is the single source of truth when it comes to documenting APIs. The main advantage of this approach is centralization. Furthermore, when used in combination with code generation tools and versioning supported by pipelines and package registries, the workflow brings another benefit of smoother incorporation of changes. GitLab, where the source code of the original FelSight project is hosted, offers both options for setting up pipelines and managing package registries (which are also employed in the workflow).

### 7.3.1 CI Pipeline and the GitLab Package Registry

GitLab pipelines allow developers to automate a wide range of actions to support continuous integration and deployment (CI/CD) practices. A common example is an automated compilation and a test run whenever code changes are pushed to the repository by a developer. One of the advantages is better detection of errors in early development phases. GitLab Package Registry provides means of sharing various packages, such as Maven or NPM packages. Other projects with access to the registry can then download the published packages and use them as dependencies. [30, 31]

The GitLab CI/CD pipeline for the room microservice project is defined in the file `.gitlab-ci.yml` (as shown in Figure C.7) and consists of four stages in the determined order: *Build App*, *Generate Client*, *Build Client*, and *Publish Client*.

During the stage *Build App*, the `app` module is built using the command `gradle clean build` that launches the process depicted in Figure 7.4.

The subsequent steps typically involve running tests and deploying the application; however, these actions are omitted for simplicity. After successfully building the application, the Java client library is generated during the stage *Generate Client*, using the task `generateApiJavaClient`. The generated code must also be compiled; this is ensured by the *Build Client* by running the `build` command in the previously generated directory. Finally, during stage *Publish Client*, the artifact (JAR package) is published to the GitLab Package Registry, ready to be downloaded by other projects. The package registry to push the artifact to is defined in the `init_repo.gradle` file that is run as an initialization script. The contents of the file are shown in Figure C.8.

### 7.3.2 Practical Example

To give a practical example, assume that there are two actors in the workflow; a developer of a service (*Developer A*) and a developer of a client application using the service (*Developer B*). The client application can be a React application or another Java Spring Boot application. The typical steps (also shown in Figure 7.6) of the workflow could be as follows:

1. *Developer A* updates the API and increments its version in the specification file.
2. *Developer A* pushes the changes to the GitLab repository.
3. Pipeline defining several consecutive jobs is launched (after the changes have been reviewed, approved and merged to a designated branch). The pipeline performs the following actions, all of which must pass in order for the pipeline to finish successfully:
    a. Builds the project and runs tests.
    b. Deploys the service with the new API changes.
    c. Generates and builds one or multiple client libraries for the service.
    d. Publishes the built client libraries to the GitLab Package Registry of the same repository. The libraries are added to the registry in the form of packages with the incremented version.
4. *Developer B* wants to update their client application to support the new version of the service. In the dependency list of the application, they increment the version of the API client to be used.
5. The client application downloads the new version of the client library from the GitLab Package Registry.

When *Developer A* makes changes to the API – i.e. edits an existing endpoint or adds a new one with additional schemas – *Developer B* does not have to manually update the HTTP client in their application, nor do they have to create any DTO objects. Thus, for consumers, adapting to API changes is much more convenient. Finally, this approach also benefits from versioning, making it easier to roll back unwanted changes.

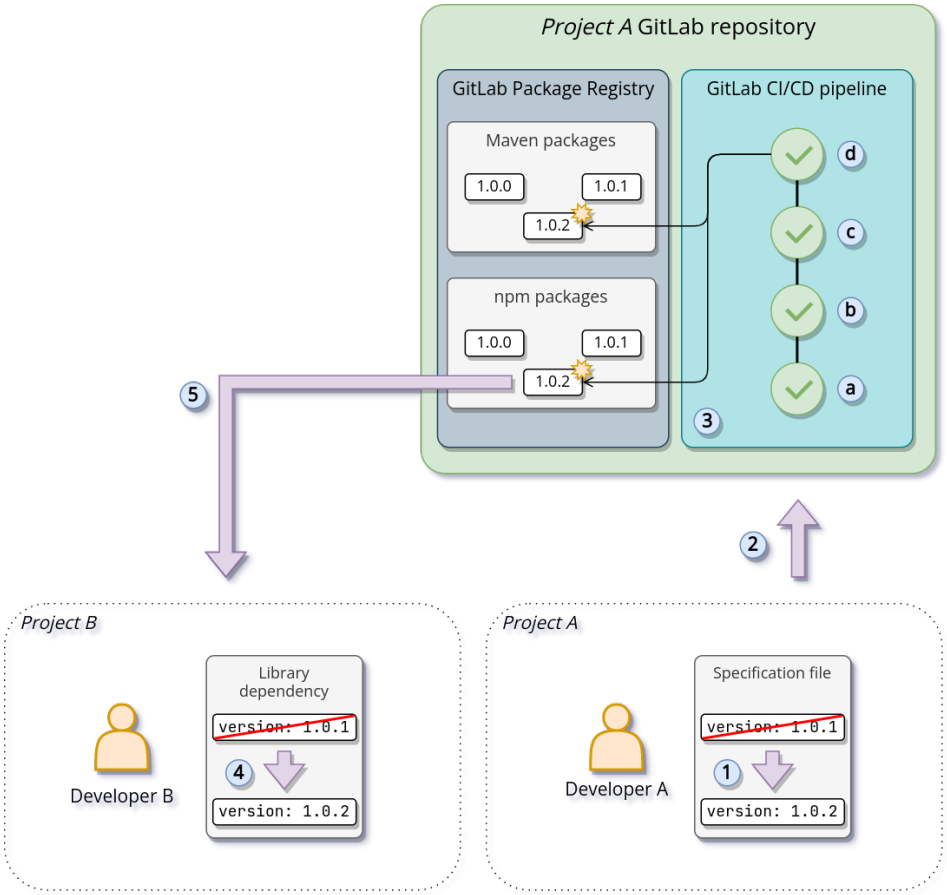



**Figure 7.6:** Development workflow of the proposed solution.

## 7.4 Security

The last issue that should be addressed is the security of the future FelSight microservice architecture. There are numerous related topics that could be covered in detail. However, the main focus of this last section is on the authentication of communication.

In general, any communication with a service should be authenticated. This is to check authenticity of the party initiating the connection (i.e. a front-end application or another service). One of the approaches used at FEE is the authentication with Keycloak.

Keycloak is an identity and access management tool. Its goal is to make modern web applications easier to secure. Since Keycloak handles multiple authentication-related mechanisms (e.g. storing credentials), developers do not have to implement them on their own. [32]

If a front-end application wants to communicate with a back-end service, it needs to fetch the required tokens from Keycloak first. Therefore, the user is redirected to the Keycloak login page to enter their username and password. The user is then redirected back to the original application. The front-end

now sends a request containing tokens (retrieved from Keycloak as a result of a successful login) to the back-end service. Subsequently, the service can verify the received tokens by sending them to Keycloak, and permit the request. [32]

In the future, this approach to authentication between the front-end application and the back-end services in combination with Keycloak is expected to be adopted by FelSight.

## 7.5 Solution Summary and Evaluation

The migration process proposed in the POC can be used in an incremental migration of FelSight's monolithic architecture, i.e. one business area/feature at a time. The key concept is the OpenAPI Specification file that defines the communication contract between the microservice and its clients. In addition, the specification file is used in the OpenAPI Generator for generating server method stubs and client libraries for a selected target language.

Comprising of three main steps, the migration of a selected business feature from the FelSight monolith starts with the design of the new microservice's API. In the specification file, operations (corresponding to individual endpoints) and schemas are defined. The next step involves the implementation of the service. Multiple aspects have to be considered; however, it makes sense to unify them across all FelSight services, if possible, to establish cohesive conventions and a unified technology stack. Nevertheless, determining whether a database is needed and identifying data sources is service-specific and depends on the nature of the data. The final step is to integrate the new service into the existing codebase. Integration, in fact, means replacing the old data source the monolith was originally using. This was shown to be feasible with the help of the *branch by abstraction* pattern, which requires minimal modification of the legacy code.

The solution also incorporates the use of GitLab features – CI/CD pipelines and the GitLab Package Registry. In combination with the generation of source code, the process allows smoother development owing to reusable client libraries, shared among projects.

On a final note, the actions in the last step do not necessarily have to be applied in the actual migration process. In the future, it is anticipated that the front-end of FelSight will be rewritten using one of the popular front-end libraries – React. The integration step would then involve creating a new codebase, rather than modifying the existing code.

# Chapter 8

# Conclusion

The thesis covered the topic of software architectures. In particular, it focused on the architectural transformation of FelSight, the faculty application for students and teachers of the Faculty of Electrical Engineering at the Czech Technical University.

In the introductory chapter, the topic of software architectures was briefly introduced, together with its motivation for its usage in software development.

The second chapter presented several definitions of this term, briefly mentioned its history, and then its beneficial factors, further motivating its purpose. The chapter also compared the terms *architectural style* and *design pattern* and explained the main difference between them. Finally, four examples of architecture types, commonly used in practice, with accompanying illustrations were described.

The next chapter looked at the software architecture design. In particular, it addressed criteria to consider, decisions to make, and questions to ask when designing architecture. Next, two existing methods supporting the design process were presented. These methods can be used by the architect to analyze a larger system and break it down into smaller parts. The last part of the chapter presents attributes that describe the quality of software. Five examples of such attributes, which are typically part of system requirements, were mentioned.

In the last chapter of the theoretical part of the thesis, the problem of transforming an existing software architecture into a new one was covered. It was stressed that this task should be rationally motivated and justified. In this context, several practical reasons for migrating an architecture were mentioned. The chapter then enumerated three mutually related practical patterns that can be used as a guide for decomposing a monolithic architecture. These patterns aim to make the migration process incremental and safer by allowing a rollback. Lastly, general recommendations for the migration of an architecture are mentioned.

The subsequent practical part was introduced with a brief description of FelSight, the application that was the subject of the architectural transformation. Afterward, the current architecture and project structure of this application were described. Moreover, the process of building and deploying was explained together with the roles of individual modules that the existing

application consists of. The final part presents motives as to why FelSight's current architectural solution is worthy of structural change.

The following chapter is concerned with the analysis and design of the application's new architecture. This process stems from the application's principal issue that it is comprised of multiple business areas. This means that any addition of new potential features in the future increases the complexity of the codebase. Therefore, the need for isolation of individual components arises. In practice, this requirement implies the breakdown of FelSight's monolithic architecture into smaller independent units, which is well-suited for the microservices architecture. Based on listing individual business areas (or features) and use cases associated with them using the CRUD matrix, the analysis yielded a collection of microservices candidates.

In the final chapter of the practical part, the POC implementation was covered. One of the microservice candidates was chosen to practically demonstrate the migration process of a small part of the application. Initially, the steps of the migration process were described and the subsequent parts of the chapter were elaborated in detail. The results of the POC implementation have shown that the proposed process steps allow for incremental migration from FelSight's current monolithic architecture to microservices. The solution makes use of an OpenAPI Specification file for the centralized definition of the microservice's API. The file also serves as an input to the OpenAPI Generator used during the process. Finally, the topic of security with a focus on authentication was briefly discussed. The model involves Keycloak, an identity and access management tool suitable for authentication of front-end applications and REST API services.

The architectural migration of FelSight is a task whose completion might take several years. In conclusion, the POC implementation presented a foundation that could be used for the future migration process.

# Bibliography


1. ISO/IEC/IEEE Systems and software engineering – Architecture description. *ISO/IEC/IEEE 42010:2011(E) (Revision of ISO/IEC 42010:2007 and IEEE Std 1471-2000)* [online]. 2011 [visited on 2021-10-30]. Available from DOI: 10.1109/IEEESTD.2011.6129467.
2. BASS, Len; CLEMENTS, Paul; KAZMAN, Rick. *Software architecture in practice*. 3rd ed. Upper Saddle River, NJ: Addison-Wesley, 2013. ISBN 978-032-1815-736.
3. RICHARDS, Mark; FORD, Neal. *Fundamentals of software architecture: an engineering approach*. First Edition. Beijing: O'Reilly, 2020. ISBN 978-1-492-04345-4.
4. GARLAN, David; SHAW, Mary. *An Introduction to Software Architecture*. Pittsburgh, PA, 1994. Available also from: http://sunnyday.mit.edu/16.355/intro_softarch.pdf. Technical Report CMU-CS-94-166. Carnegie Mellon University, School of Computer Science.
5. ISO/IEC/IEEE International Standard - Systems and software engineering– Vocabulary. *ISO/IEC/IEEE 24765:2017(E)* [online]. 2017 [visited on 2021-10-30]. Available from DOI: 10.1109/IEEESTD.2017.8016712.
6. KRUCHTEN, P.; OBBINK, H.; STAFFORD, J. The Past, Present, and Future for Software Architecture. *IEEE Software* [online]. 2006, vol. 23, no. 2, pp. 22–30 [visited on 2021-10-30]. ISSN 0740-7459. Available from DOI: 10.1109/MS.2006.59.
7. PERRY, Dewayne E.; WOLF, Alexander L. Foundations for the Study of Software Architecture. *SIGSOFT Softw. Eng. Notes* [online]. 1992, vol. 17, no. 4, pp. 40–52 [visited on 2021-11-01]. ISSN 0163-5948. Available from DOI: 10.1145/141874.141884.
8. SHARMA, Anubha; KUMAR, Manoj; AGARWAL, Sonali. A Complete Survey on Software Architectural Styles and Patterns. *Procedia Computer Science*. 2015, vol. 70, pp. 16–28. ISSN 18770509. Available from DOI: 10.1016/j.procs.2015.10.019.

9. SHAW, M.; CLEMENTS, P. A field guide to boxology: preliminary classification of architectural styles for software systems. *Proceedings Twenty-First Annual International Computer Software and Applications Conference (COMPSAC'97)*. 1997, pp. 6–13. ISBN 0-8186-8105-5. Available from DOI: 10.1109/CMPSAC.1997.624691.

10. MONROE, R.T.; KOMPANEK, A.; MELTON, R.; GARLAN, D. Architectural styles, design patterns, and objects. *IEEE Software*. 1997, vol. 14, no. 1, pp. 43–52. ISSN 07407459. Available from DOI: 10.1109/52.566427.

11. RICHARDS, Mark. *Software Architecture Patterns*. First Edition. Sebastopol, CA: O'Reilly, 2015. ISBN 978-1-491-92424-2.

12. RICHARDS, Mark. *Microservices vs. Service-Oriented Architecture*. First Edition. Sebastopol, CA: O'Reilly, 2016. ISBN 978-1-491-94161-4.

13. MILLETT, Scott; TUNE, Nick. *Patterns, Principles, and Practices of Domain-Driven Design*. 1st Edition. Canada: Wrox, 2015. ISBN 978-1-118-71470-6.

14. INMON, William H.; LINSTEDT, Daniel. *Data architecture: a primer for the data scientist : big data, data warehouse and data vault*. 1st Edition. Amsterdam: Academic Press is an imprint of Elsevier, 2015. ISBN 978-0-12-802044-9.

15. MUSTAPIC, G.; WALL, A.; NORSTROM, C.; CRNKOVIC, I.; SANDSTROM, K.; FROBERG, J.; ANDERSSON, J. Real world influences on software architecture - interviews with industrial system experts [online]. 2004, pp. 101–111 [visited on 2021-12-13]. Available from DOI: 10.1109/WICSA.2004.1310694.

16. BACHMANN, Felix; BASS, Len; CHASTEK, Gary; DONOHOE, Patrick; PERUZZI, Fabio. The Architecture Based Design Method [online]. 2000, p. 61 [visited on 2021-12-14]. Available from: https://www.researchgate.net/publication/235088008_The_Architecture_Based_Design_Method.

17. BACHMANN, Felix; BASS, Len. Introduction to the Attribute Driven Design Method [online]. 2001, pp. 745–746 [visited on 2021-12-14]. ISBN 0769510507. Available from: https://dl.acm.org/doi/10.5555/381473.381623.

18. WOJCIK, Rob; BACHMANN, Felix; BASS, Len; CLEMENTS, Paul; MERSON, Paulo; NORD, Robert; WOOD, William. Attribute-Driven Design (ADD), Version 2.0 [online]. 2006, no. CMU/SEI-2006-TR-023 [visited on 2021-12-14]. Available from: http://resources.sei.cmu.edu/library/asset-view.cfm?AssetID=8147.

19. GOMAA, Hassan. *Software Modeling and Design: UML, Use Cases, Patterns, and Software Architectures*. 1st Edition. Cambridge University Press, 2011. ISBN 978-0-521-76414-8.

20. *Service Level Agreement for Hosting and Realtime Database* [online]. 2020 [visited on 2021-12-26]. Available from: `https://firebase.google.com/terms/service-level-agreement`.

21. FOWLER, Martin. *StranglerFigApplication* [online]. 2000 [visited on 2021-12-28]. Available from: `https://martinfowler.com/bliki/StranglerFigApplication.html`.

22. NEWMAN, Sam. *Monolith to microservices: evolutionary patterns to transform your monolith*. First Edition. Beijing: O'Reilly, 2019. ISBN 978-149-2047-841.

23. BUCCHIARONE, Antonio; DRAGONI, Nicola; DUSTDAR, Schahram; LARSEN, Stephan T.; MAZZARA, Manuel. From Monolithic to Microservices: An Experience Report from the Banking Domain. *IEEE Software*. 2018, vol. 35, no. 3, pp. 50–55. ISSN 0740-7459. Available from DOI: `10.1109/MS.2018.2141026`.

24. IBM. *Business Systems Planning - Information Systems Planning Guide*. Second Edition. New York: IBM, 1978.

25. MILLER, Darrel; WHITLOCK, Jeremy; GARDINER, Marsh; RALPHSON, Mike; RATOVSKY, Ron; SARID, Uri. *OpenAPI Specification v3.1.0* [online]. Linux Foundation, 2021 [visited on 2022-04-20]. Available from: `https://spec.openapis.org/oas/v3.1.0`.

26. *Spring Initializr* [online] [visited on 2022-04-03]. Available from: `https://start.spring.io/`.

27. *KOSapi* [online]. Jakub Jirůtka, 2015 [visited on 2022-05-07]. Available from: `https://kosapi.fit.cvut.cz/projects/kosapi/wiki`.

28. *Seznam učeben k využití pro studenty mimo plánovaný rozvrh* [online]. České vysoké učení technické - Fakulta elektrotechnická [visited on 2022-05-07]. Available from: `https://fel.cvut.cz/cz/education/studovny-samostudium.html`.

29. *Gradle - Plugin: org.openapi.generator* [online]. Jim Schubert, 2022 [visited on 2022-05-07]. Available from: `https://plugins.gradle.org/plugin/org.openapi.generator`.

30. *CI/CD pipelines* [online]. GitLab [visited on 2022-05-02]. Available from: `https://docs.gitlab.com/ee/ci/pipelines/`.

31. *Package Registry* [online]. GitLab [visited on 2022-05-02]. Available from: `https://docs.gitlab.com/ee/user/packages/package_registry/`.

32. THORGERSEN, Stian; SILVA, Pedro Igor. *Keycloak - identity and access management for modern applications: harness the power of Keycloak, OpenID Connect, and OAuth 2.0 protocols to secure applications*. Birmingham: Packt, 2021. ISBN 978-1-80056-249-3.

# Appendices

# Appendix A

## Nomenclature

| | |
|---|---|
| (X)HTML | (Extensible) HyperText Markup Language |
| ADD | Attribute-Driven Design |
| API | Application Programming Interface |
| CI/CD | Continuous Integration / Continuous Deployment |
| CRUD | Create, Read, Update, Delete |
| CSS | Cascading Style Sheets |
| DDD | Domain-Driven Design |
| DTO | Data Transfer Object |
| EAR | Enterprise application Archive |
| EJB | Enterprise Java Beans |
| FEE | Faculty of Electrical Engineering |
| HTTP | Hypertext Transfer Protocol |
| IEC | International Electrotechnical Commission |
| IEEE | Institute of Electrical and Electronics Engineers |
| ISO | International Organization for Standardization |
| JAR | Java Archive |
| Java EE | Java Enterprise Edition |
| JDBC | Java Database Connectivity |
| JNDI | Java Naming and Directory Interface |
| JPA | Java Persistence API |
| JS | JavaScript |

| | |
|---|---|
| JSF | JavaServer Faces, newly Jakarta Server Faces |
| JSON | JavaScript Object Notation |
| KOS | Komponenta studium |
| LESS | Leaner Style Sheets |
| NATO | North Atlantic Treaty Organization |
| NPM | Node Package Manager |
| POC | Proof of Concept |
| REST | Representational State Transfer |
| SLA | Service-Level Agreement |
| SOA | Service-Oriented Architecture |
| SOAP | Simple Object Access Protocol |
| SQL | Structured Query Language |
| UI | User Interface |
| WAR | Web application Archive |
| XML | Extensible Markup Language |

# Appendix B

# Software Used in the Implementation

## B.1 Technologies

- Java 17 (Azul Zulu OpenJDK, build 17.0.3+7-LTS)
- Gradle 7.4.1
- Spring Boot 2.6.6
  - spring-boot-starter
  - spring-boot-starter-web
  - spring-boot-starter-webflux
  - spring-boot-starter-oauth2-client
  - spring-boot-starter-data-jpa
  - spring-boot-starter-validation
- OpenAPI Generator – org.openapi.generator 5.3.0 (Gradle plugin)
- PostgreSQL 12.7 + PostgreSQL Driver 42.3.3
- Flyway 8.0.5
- Lombok 6.4.1
- MapStruct 1.4.2.Final
- Other dependencies
  - io.swagger:swagger-annotations 1.6.5
  - org.openapitools:jackson-databind-nullable 0.2.2
  - com.fasterxml.jackson.dataformat:jackson-dataformat-xml 2.13.2
  - javax:javaee-api 8.0.1
  - javax.xml.bind:jaxb-api 2.3.1
  - javax.activation:activation 1.1
  - org.glassfish.jaxb:jaxb-runtime 2.3.0
  - org.jsoup:jsoup 1.14.3

## B.2 Tools

- IntelliJ IDEA 2022.1
- Spring Initializr
- Postman 9.16.0
- GitLab CI Pipeline
- GitLab Package Registry

# Appendix C

## Screenshots

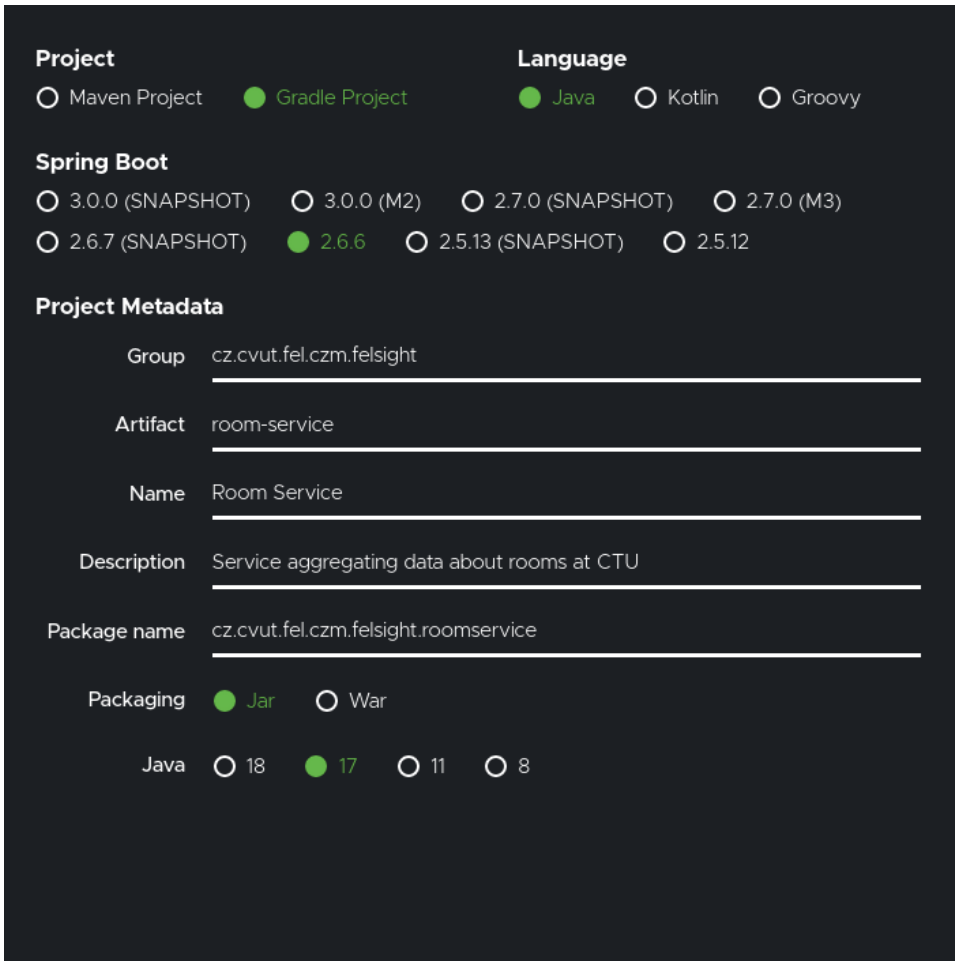



**Figure C.1:** Microservice project starting configuration in the *Spring Initializr* tool.

```java
/**
 * GET /rooms : Returns a collection of rooms
 *
 * @param limit Maximum number of records to return (optional)
 * @param offset Number of records to skip. To be used together with the &#39;limit&#39; param for paging.
 * @param forSelfStudy Whether the rooms returned are available to students for self-study purposes beyond
 * @return OK (status code 200)
 */
@ApiOperation(value = "Returns a collection of rooms", nickname = "getRooms", notes = "", response = RoomD
@ApiResponses(value = {
    @ApiResponse(code = 200, message = "OK", response = RoomDTO.class, responseContainer = "List") })
@RequestMapping(
    method = RequestMethod.GET,
    value = "/rooms",
    produces = { "application/json" }
)
default ResponseEntity<List<RoomDTO>> _getRooms(@ApiParam(value = "Maximum number of records to return") @
  skip. To be used together with the 'limit' param for paging.") @Valid @RequestParam(value = "offset", req
  self-study purposes beyond the official teaching schedule") @Valid @RequestParam(value = "forSelfStudy",
    return getRooms(limit, offset, forSelfStudy);
}

// Override this method
default  ResponseEntity<List<RoomDTO>> getRooms(Integer limit, Integer offset, Boolean forSelfStudy) {
    getRequest().ifPresent(request -> {
        for (MediaType mediaType: MediaType.parseMediaTypes(request.getHeader( headerName: "Accept"))) {
            if (mediaType.isCompatibleWith(MediaType.valueOf("application/json"))) {
                String exampleString = "{ \"division\" : { \"code\" : \"code\", \"divisionType\" : \"divis
                  \"en\" } }, \"code\" : \"code\", \"access\" : \"access\", \"forSelfStudy\" : true, \"loca
                  \"availability\" : [ { \"from\" : \"from\", \"to\" : \"to\" }, { \"from\" : \"from\", \"t
                ApiUtil.setExampleResponse(request,  contentType: "application/json", exampleString);
                break;
            }
        }
    });
    return new ResponseEntity<>(HttpStatus.NOT_IMPLEMENTED);

}
```

**Figure C.2:** Server stub `getRooms` generated by the `spring` OpenAPI Generator.

```java
@RestController
@RequiredArgsConstructor
public class RoomController implements RoomsApi {

    private final RoomService roomService;
    private final RoomDTOMapper roomDTOMapper;

    @Override
    public ResponseEntity<List<RoomDTO>> getRooms(Integer limit, Integer offset, Boolean forSelfStudy) {
        return ResponseEntity.ok(
                roomDTOMapper.toDto(
                        roomService.getRooms(
                                limit != null ? limit : 20,
                                offset != null ? offset : 0,
                                forSelfStudy
                        )
                )
        );
    }
```

**Figure C.3:** Override of the method stub `getRooms`, originally declared in the generated class `RoomApi` (Figure C.2).

```java
@Named("roomServiceClient")
public class RoomServiceClient extends RoomApi implements RoomClient, Serializable {

    private static final Logger logger = Logger.getLogger(RoomServiceClient.class.getName());

    @Inject
    private RoomMapper roomMapper;

    @PostConstruct
    private void init() {
        getApiClient()
                .setBasePath("http://localhost:8081"); // For local testing purposes
    }

    @Override
    public List<Room> filter(String name, String location, Boolean available) {
        List<Room> rooms = new ArrayList<>();
        try {
            List<RoomDTO> roomsFromService = getRooms( limit: 20,  offset: 0,  forSelfStudy: true);
            rooms.addAll(roomMapper.fromDto(roomsFromService));
        } catch (ApiException e) {
            logger.log(Level.SEVERE,  msg: "Error occurred while retrieving self-study rooms", e);
        }

        return rooms;
    }
}
```

**Figure C.4:** `RoomServiceClient` Java class with the new implementation retrieving data from the microservice.

```java
@javax.annotation.Generated(value = "org.openapitools.codegen.languages.SpringCodegen")
public class RoomDTO   {
  @JsonProperty("capacityForExamining")
  private Integer capacityForExamining;

  @JsonProperty("capacityForTeaching")
  private Integer capacityForTeaching;

  @JsonProperty("code")
  private String code;

  @JsonProperty("locality")
  private String locality;

  @JsonProperty("forSelfStudy")
  private Boolean forSelfStudy;

  @JsonProperty("name")
  private LanguageFieldDTO name;

  @JsonProperty("division")
  private DivisionDTO division;

  @JsonProperty("availability")
  @Valid
  private List<AvailabilityDTO> availability = null;

  @JsonProperty("access")
  private String access;

  @JsonProperty("type")
  private String type;
```

**Figure C.5:** DTO Java class generated by the `spring` OpenAPI Generator.

```java
@Mapper(componentModel = "jsr330", uses = {
        LanguageFieldMapper.class, DivisionMapper.class, RoomAvailabilityMapper.class
})
public interface RoomMapper {

    @Mapping(source = "forSelfStudy", target = "availableForStudents")
    @Mapping(source = "availability", target = "roomAvailability")
    @Mapping(target = "validity", ignore = true)
    @Mapping(target = "id", ignore = true)
    @Mapping(target = "transliteration", ignore = true)
    @Mapping(target = "address", ignore = true)
    @Mapping(target = "parallels", ignore = true)
    @Mapping(target = "roomOccupancy", ignore = true)
    @Mapping(target = "access", ignore = true)
    Room fromDto(RoomDTO roomDTO);

    List<Room> fromDto(List<RoomDTO> roomDTO);

    @AfterMapping
    default void afterMapping(@MappingTarget Room room, RoomDTO roomDTO) {
        try {
            room.setAccess(RoomAccessEnum.valueOf(roomDTO.getAccess()));
        } catch (IllegalArgumentException e) {
            room.setAccess(null);
        }
    }

}
```

**Figure C.6:** Interface that is used by *MapStruct* as an input to generate mapper implementations.

```yaml
image: registry.gitlab.fel.cvut.cz:443/czm/infrastructure/ci-images/gradle7_jdk17_cs:latest
stages:
  - Build App
  - Generate Client
  - Build Client
  - Publish Client

variables:
  GENERATED_PROJECT_DIR: "specification/build/generated/java-client"

cache:
  paths:
    - specification/build/generated/

build app:
  allow_failure: false
  stage: Build App
  when: always
  script:
    - gradle clean build

generate client:
  allow_failure: false
  stage: Generate Client
  when: manual
  script:
    - gradle specification:clean specification:generateApiJavaClient

build client:
  allow_failure: false
  stage: Build Client
  when: on_success
  script:
    - gradle -p $GENERATED_PROJECT_DIR build

publish client:
  allow_failure: false
  stage: Publish Client
  when: on_success
  script:
    - cp init_repo.gradle $GENERATED_PROJECT_DIR
    - gradle --init-script init_repo.gradle -p $GENERATED_PROJECT_DIR publish
```

**Figure C.7:** The GitLab CI/CD pipeline configuration file.

```
allprojects {
    apply plugin: 'java'
    apply plugin: 'maven-publish'

    final DEPLOY_TOKEN_PROPERTY :String  = "roomServiceDeployToken"
    final DEPLOY_TOKEN_ENV_VARIABLE :String  = "DEPLOY_TOKEN"

    publishing.repositories {
        maven {
            url "https://gitlab.fel.cvut.cz/api/v4/projects/23790/packages/maven"
            credentials(HttpHeaderCredentials) {
                name = "Deploy-Token"
                if (project.hasProperty("$DEPLOY_TOKEN_PROPERTY")) { // from ~/.gradle/gradle.properties
                    println "Property with deploy token found - will be used for authentication"
                    value = project.property("$DEPLOY_TOKEN_PROPERTY")
                } else if (System.getenv("$DEPLOY_TOKEN_ENV_VARIABLE")) {
                    println "Environment variable with deploy token found - will be used for authentication
                    value = System.getenv("$DEPLOY_TOKEN_ENV_VARIABLE")
                } else {
                    throw new GradleException("Property '${DEPLOY_TOKEN_PROPERTY}' or environment variable
                }
            }
            authentication {
                header(HttpHeaderAuthentication)
            }
        }
    }
}
```

**Figure C.8:** `init_repo.gradle` initialization script with the repository configuration.

```
// A custom definition of a task that generates server code.
task generateApiJavaServer(type: GenerateTask) {
    generatorName = "spring"
    inputSpec = "${projectDir}/src/main/resources/specifications/latest.yaml".toString()
    apiPackage = "${group}.${restPackage}.api".toString()
    modelPackage = "${group}.${restPackage}.dto".toString()
    modelNameSuffix = "DTO"
    outputDir = "${generatedJavaServerDir}".toString()
    configOptions = [
            artifactId: "${javaServerArtifactId}",
            groupId: "${group}"
    ]
    additionalProperties = [
            interfaceOnly: "true",
            hideGenerationTimestamp: "true",
            delegatePattern: "true"
    ]
}
```

**Figure C.9:** Gradle task definition for the server stub generation from the specification file.

# Appendix D

## Contents of the Attached CD

- `felsight_integration.zip` ..... Microservice integration files
  - `README.txt` ..... Description of the contents
- `room_service.zip` ..... Microservice project
  - `app` ..... Microservice source code module
  - `specification` ..... REST API specification module
  - `README.md` ..... Installation manual (Markdown)
- `The_Architecture_Transformation_of_FelSight_Faculty_Application.pdf` Master thesis in PDF format