University of São Paulo
Institute of Mathematics and Statistics
Bachelor of Computer Science

Improving modifiability in microservices architectures

A reference implementation for a remote work platform in software startups

Thiago Guerrero Balera

Final Essay

mac 499 — Capstone Project

Supervisor: Prof. Dr. Eduardo Martins Guerra
Co-supervisor: MSc. João Francisco Lino Daniel

São Paulo
2023

Ficha catalográfica elaborada com dados inseridos pelo(a) autor(a)
Biblioteca Carlos Benjamin de Lyra
Instituto de Matemática e Estatística
Universidade de São Paulo

Balera, Thiago Guerrero
Improving modifiability in microservices architectures: A reference implementation for a remote work platform in software startups / Thiago Guerrero Balera; orientador, Prof. Dr. Eduardo Martins Guerra; coorientador, MSc. João Francisco Lino Daniel. - São Paulo, 2023.
50 p.: il.

Trabalho de Conclusão de Curso (Graduação) - Ciência da Computação / Instituto de Matemática e Estatística / Universidade de São Paulo.
Bibliografia

1. Microservices. 2. Modifiability. 3. Software Architecture. 4. Design Patterns. I. Martins Guerra, Prof. Dr. Eduardo. II. Título.

Bibliotecárias do Serviço de Informação e Biblioteca
Carlos Benjamin de Lyra do IME-USP, responsáveis pela
estrutura de catalogação da publicação de acordo com a AACR2:
Maria Lúcia Ribeiro CRB-8/2766; Stela do Nascimento Madruga CRB 8/7534.

Code never lies, comments sometimes do.
— Ron Jeffries

Acknowledgement

I would like to express my sincere gratitude to João Daniel for our remarkable partnership over the past five years of my academic journey. João has not only been my supervisor during this work, but also a reference, coworker, teacher, and friend. His guidance was essential in my journey towards the software engineering field.

Additionally, I would like to extend my gratitude to Prof. Dr. Zara Isaa Abud, my calculus I professor, who truly believed in my potential during my freshman semester. I am also thankful for the outstanding computing professors, Prof. Dr. Carlos Eduardo Ferreira and Prof. Dr. Daniel Macedo Batista. Finally, my appreciation goes to my closer friends.

Resumo

Thiago Guerrero Balera. **Melhorando a modificabilidade em arquiteturas de microsserviços: *Uma implementação de referência para uma plataforma de trabalho remoto em startups de software***. Monografia (Bacharelado). Instituto de Matemática e Estatística, Universidade de São Paulo, São Paulo, 2023.

Nesta monografia, investigamos desafios de modificabilidade dentro da Arquitetura de Microserviços (MSA), com foco em um estudo de caso no projeto Digi-Dojo, uma plataforma para equipes remotas de startups de software. O estudo identifica os principais problemas de modificabilidade, como duplicação do modelos de dados e problemas de arquitetura interna, e avalia soluções, incluindo os padrões de Objeto de Transferência de Dados (DTO), Arquitetura Limpa e Replicação de Dados. Os resultados revelam melhorias significativas na modificabilidade e integração do sistema.

Palavras-chave: Microsserviços, Modificabilidade, Arquitetura de Software, Padrões de Design.

Abstract

Thiago Guerrero Balera. **Improving modifiability in microservices architectures: *A reference implementation for a remote work platform in software startups***. Capstone Project Report (Bachelor). Institute of Mathematics and Statistics, University of São Paulo, São Paulo, 2023.

In this thesis, we investigate modifiability challenges within Microservices Architecture (MSA), focusing on a case study at the Digi-Dojo project, a platform for remote software startup teams. The study identifies key modifiability issues, like data model duplication and internal architecture problems, and evaluates solutions including the Data Transfer Object (DTO), Clean Architecture, and Data Replication patterns. The results reveal significant improvements in modifiability and system integration.

Keywords: Microservices,Modifiability,Software Architecture,Design Patterns.

List of Figures

List of Programs

Contents

Introduction

Software architecture is an area that studies important concepts and practices to ensure the quality and success of a software system, hence improving its chances to succeed as a project. A good architecture allows you to have a broad view on a software system and its future evolution (Bass *et al.*, 1997). However, a software developed without a good architectural plan might be hard to keep up during its evolution. A poor architecture is a major cause of impediment for developers (Fowler, 2019).

Modifiability is a crucial quality attribute in software systems when it comes to be ready for future evolution, as it allows for more flexibility to use new frameworks, libraries, design patterns, programming languages, data sources, and so on (Bogner *et al.*, 2019). It is also closely related to other quality attributes such as maintainability, scalability, and testability. Though, designing for high modifiability is a huge challenge in Software Engineering, as it requires anticipating future changes and designing the system in a way that makes those changes easy to implement (Bogner *et al.*, 2019).

This thesis presents a comprehensive study on improving modifiability in microservices architectures. Based in the Digi-Dojo project, it identifies critical issues related to existing challenges, explores practical solutions, and offers a comparative study of some patterns.

Chapter 1 is going to cover some concepts essential to this thesis. Following, chapter 2 is going to retake the proposal, highlighting what changed. Chapter 3 is going to explain the Digi-Dojo project. Finally, chapter 4 presents an extensive content, picturing the issues, challenges, and solutions explored on this work.

Chapter 1

Literature Review

This chapter will cover some essential concepts in Software Architecture that are going to be important to understand this work.

1.1 Modifiability

The **Modifiability** is a quality attribute of the software architecture that measures the cost of making changes and reflects the ease with which a software system can accommodate modifications (Northrop, 2004). This attribute captures the cost in time and money of making a change, including the extent to which it affects other functions or quality attributes (Bass *et al.*, 1997).

A change can occur in any aspect of a system, such as internal functions, infrastructure, environment (dependencies, protocols, etc.), quality attributes (performance, reliability, etc.), capacity (Transactions Per Second (TPS) and limits), and new features/modification (Bass *et al.*, 1997). Thus, changes can be made by developers, installers, and even end users.

The view on modifiability we will work with consists of two types of costs: the cost of preparing the system for changes and the cost of making changes. In other words, modifiability measures the ability of a system to easily incorporate a new solution as an alternative to existing options and to easily address new capabilities that the system did not previously support.

1.2 Microservices Architecture

The **Microservices Architecture (MSA)** is a software architectural style in which the system is composed by independently deployable, highly cohesive units – called microservices – that typically communicate via HTTP (Newman, 2015). Microservices are modeled around a specific business domain, encapsulating it via one or more well-defined network endpoints in order to expose it to other microservices or to external clients, making them a form of distributed system (Newman, 2020).

MSA offers several advantages, such as loosely coupled services, technology-agnosticism, autonomous decision-making, distributed responsibility and evolution between teams (as per Conway's Law), distributed blast radius, and independent risk assessments (Newman, 2020). For these reasons, it is often a suitable choice for teams, enabling them to manage complex systems and large engineering staff in smaller parts, improving scalability, robustness, modifiability, and other quality attributes.

Although the MSA has become very popular, it is not the best solution for all systems. It significantly increases the complexity of common tasks such as designing, infrastructure setup, and communication (mainly due to the network burden) (Newman, 2015). However, Those difficulties can be mitigated by using microservices patterns.

1.3 Microservices Pattern Language

A Pattern Language (PL) is an organized and coherent set of patterns, each of which describes a problem and the core of a solution that can be used in many scenarios (Alexander *et al.*, 1977). The **Microservices Pattern Language** is a specific type of PL which describes microservices' problems and their solutions. One of the well-known MSA PL embraced by the community is presented in Richardson, 2018, and is also available on *microservices.io*[1]. This pattern language encompasses a range of valuable patterns, some of which include:

- **API Gateway** provides an abstraction of the system APIs to their clients, making the microservices' system to look like it is a single component system.
- **Database per service** keeps each microservice's persistent data private and accessible only via its API. A microservice's transactions only involve its own database.
- **Event sourcing** persists the state of a business entity as a sequence of state-changing events. Whenever the state of a business entity changes, a new event is appended to the list of events. The application reconstructs an entity's current state by replaying the events.
- **Saga** implements each business transaction that spans multiple microservices as a saga. A saga is a sequence of local transactions. Each local transaction updates the database and publishes a message or event to trigger the next local transaction in the saga. If a local transaction fails because it violates a business rule, then the saga executes a series of compensating transactions that undo the changes that were made by the preceding local transactions.

1.4 Spring Boot

Spring is a popular open-source Java framework for building web applications. It simplifies the process of developing applications by providing a comprehensive set of tools and features that make it easier to build, maintain, and enhance the functionality of the application. Some key aspects of the Spring framework include: inversion of control,

[1] https://microservices.io/patterns/

dependency injection, integration with other technologies, and modularization (Bawiskar *et al.*, 2012).

Spring Boot is a Java-based platform which simplifies the creation and deployment of stand-alone, production-ready Spring applications with minimal configuration (Singhal, 2022). It provides a flexible and efficient way to develop web applications, offering features such as: auto-configuration, simplified dependency management, and RESTful web services.

It is often used in combination with other technologies, like JPA for persistence, Junit and Mockito for testing, SLF4J for logging, Jackson for parsing, and more, as highlighted by Larsson, 2019. While Spring Boot is well-suited for a broad range of scenarios, the primary utilization of this framework on this work lies in the context of a microservice with Spring Cloud, making the integration with cloud capacities smooth.

1.5 Apache Kafka

Apache Kafka is an open-source distributed event streaming used for building real-time data pipelines and streaming applications. It is designed to handle high volumes of data and provides a scalable, fault-tolerant, and durable messaging system for applications. Kafka is based on a publish-subscribe model, where producers publish messages to topics, and consumers subscribe to those topics to receive the messages (Hubli and Jaiswal, 2023).

Apache Kafka is widely used in various industries, such as finance, healthcare, and e-commerce, for use cases such as log aggregation, real-time analytics, and stream processing (Hubli and Jaiswal, 2023).

Chapter 2

Proposal

2.1 Motivation

For large applications, a trending choice is Microservices Architecture (MSA). When confronting Modifiability and MSA, there is a convergence of factors. MSA emerged from a context of a wide range of techniques and technologies, and of limitations on the monolith (Newman, 2015). In a monolith, all responsibilities are bundled together into a single code-base, even when following a style like Layered Architecture, leading to difficulties in achieving the scale of modern software demands of large development teams and vast set of responsibilities (Richards, 2015). In other words, the nature of monolith limits Modifiability, leading professionals to adopt MSA to mitigate these limitations.

However, merely adopting an MSA-based architecture for a system does not immediately guarantee Modifiability. The ultimate goal with MSA is to be able to develop a new microservice and have it automatically incorporated to the rest of the system – the same goes for changes into an existing microservice. When that is not a reality, often a new feature in one microservice requires changes to others, resulting in additional work.

2.2 Goals

In this thesis, our objective is to investigate the impact of modifications in MSAs and propose effective solutions to address some of the issues identified. Thus, the Research Question (RQ) is:

RQ1: *"How do the professionals deal with modifications in their MSA-based systems, in terms of practices, challenges, and solutions?"*

2.3 Methodology

2.3.1 Modifications

The diagram in the figure 2.1 serves to elucidate the initially proposed methodology in order to help leading the logical sequence towards the final version.

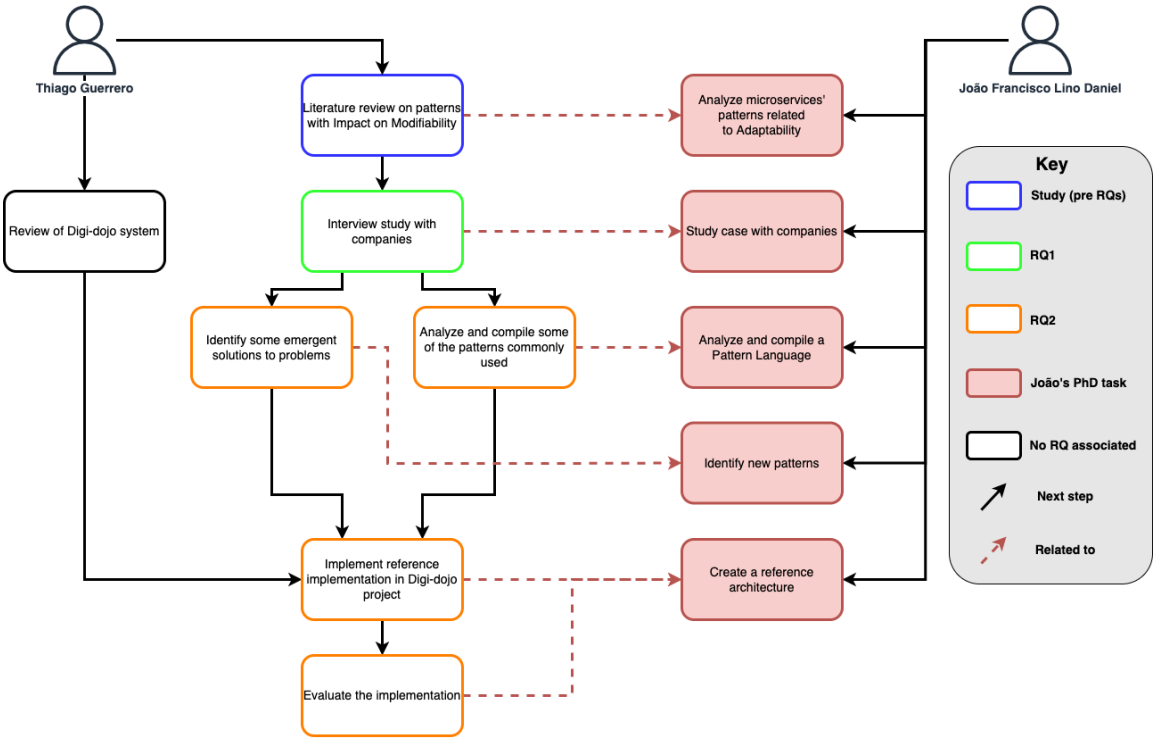

Figure 2.1: *Illustration of the steps (tasks) in the initial proposed methodology of this work*

Initially, our plan was to conduct an interview study, in particular with companies that use microservices daily. The primary objective was to systematically identify and categorize the challenges faced by professionals in terms of Modifiability in MSA. Though, time constraints led to the adjustment of priorities, and the interview task was omitted from the final methodology, deferred for future exploration.

The following tasks were planned to involve two main efforts: compiling some of the patterns commonly used to address these challenges, and identifying some emergent solutions. They were also removed from the final methodology.

2.3.2 Final methodology

The diagram in the figure 2.2 illustrates the final version of the methodology.

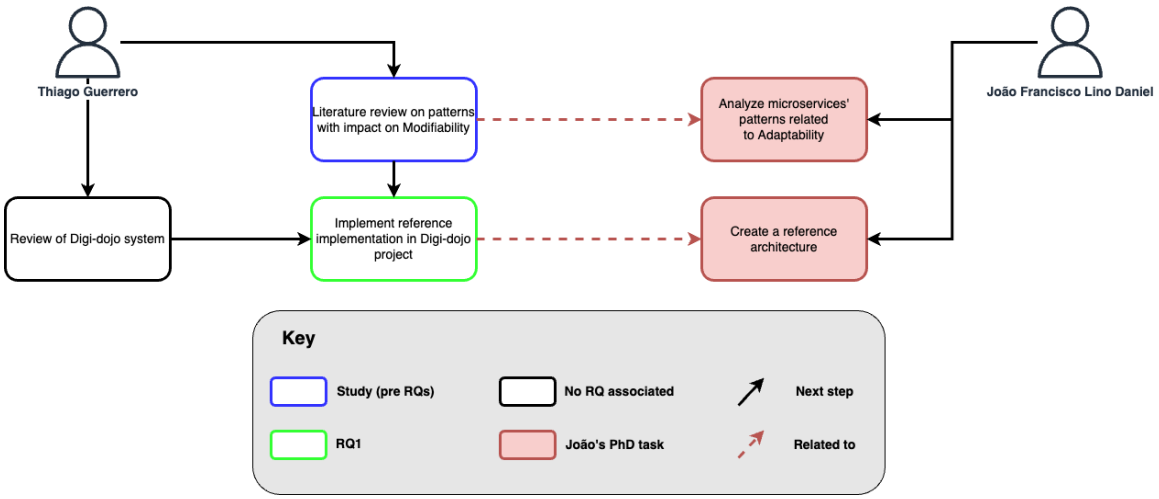

Figure 2.2: *Illustration of the steps (tasks) in the final methodology of this work*

The first thing to notice is that the work is related to João Francisco Lino Daniel's PhD work. In the next paragraphs, we will define the tasks and explain how they relate to João's work.

In the preliminary stage prior to defining the RQs, our aim was to do a literature review about a selection of patterns that enhance and promote modifiability at different levels of architecture. To achieve this goal, we used Daniel *et al.*, 2023. Authored by João, this article is dedicated to curating a set of established patterns. The work includes 22 patterns, structured among six categories, and classified according to their influence on three dimensions: *Service Adaptability*, the ability of improving a service's behavior without changing its interface; *System Extensibility*, the capacity of the system to receive new microservice units; and *Microservice Extensibility*, the capacity of one single microservice unit to incorporate new services.

In parallel with the preliminary stage, we conducted a review of the microservices developed in the 76250A - Software Architecture discipline, offered at the Free University of Bozen-Bolzano (Unibz) in the second semester of 2022/2023[1] by Eduardo Martins Guerra, with João as teacher's assistant. In this discipline, second-year students developed three distinct microservices – each one responsible for a set of entities – as part of the Startup Digi-Dojo project. This code served as the starting point for the subsequent tasks.

Finally, our plan involved developing a reference implementation within the Digi-Dojo project. Since the compiled patterns from the interview study were out of scope in the final methodology, we adjusted our approach to incorporate insights from the literature review and leverage the author's professional experience. The author has approximately 4 years of expertise in software architecture as a software engineer. Using the three microservices as the starting point, our objective was to implement, compare and analyze patterns aimed to improve modifiability, focusing on making the project easier for future integrations while identifying the associated costs to do that.

[1] The semester started on February 27th and ended on July 8th

Chapter 3

The Digi-Dojo project

3.1 What is the Digi-Dojo project?

The Digi-dojo project is a platform designed to support development teams in Software Startups that work remotely. The goal is to provide the interactive and creative environment of a startup for these teams. One of the focus of this project is the integration of existing Software Engineer tools with the platform, which has a microservices-based architecture (Wang *et al.*, 2022).

3.2 First prototype

As previously mentioned, the first prototype was implemented between February and July. Throughout this period, the class was divided into three groups of students, denoted as A, B, and C. Each group was assigned to develop a distinct microservice along with its associated entities. The resulting microservices were as follows:

- **StartupsAndUsers:** Microservice developed by team A responsible for managing startups, users, and team members
- **VirtualSpaces:** Microservice developed by team B responsible for managing virtual places and notes
- **TasksAndCalendars:** Microservice developed by team C responsible for managing tasks and calendar events

3.2.1 Domain

The prototype comprehend the following entities: startups, users, team members, virtual places, notes, tasks, and calendar events. Startups and users represent their respective concepts in the real life. A user can be a team member of multiple startups. A virtual place belongs to a startup and can be one of the three types: a personal desk, a meeting room, or a board. A user can be in a place. A note can be added by a user to a place. A task or

a calendar event can be created and must be linked to either a user or a place. The UML diagram in the figure 3.1 illustrates this domain.

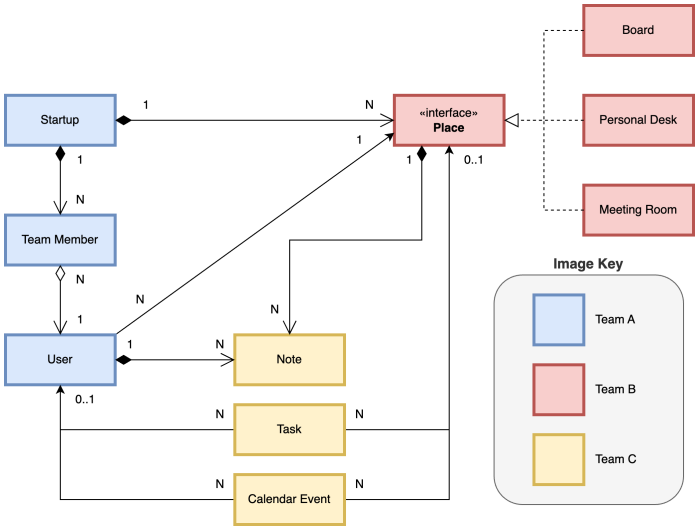

Figure 3.1: *Digi-Dojo prototype UML diagram*

The figure 3.2 illustrates the same domain at a higher level of abstraction.

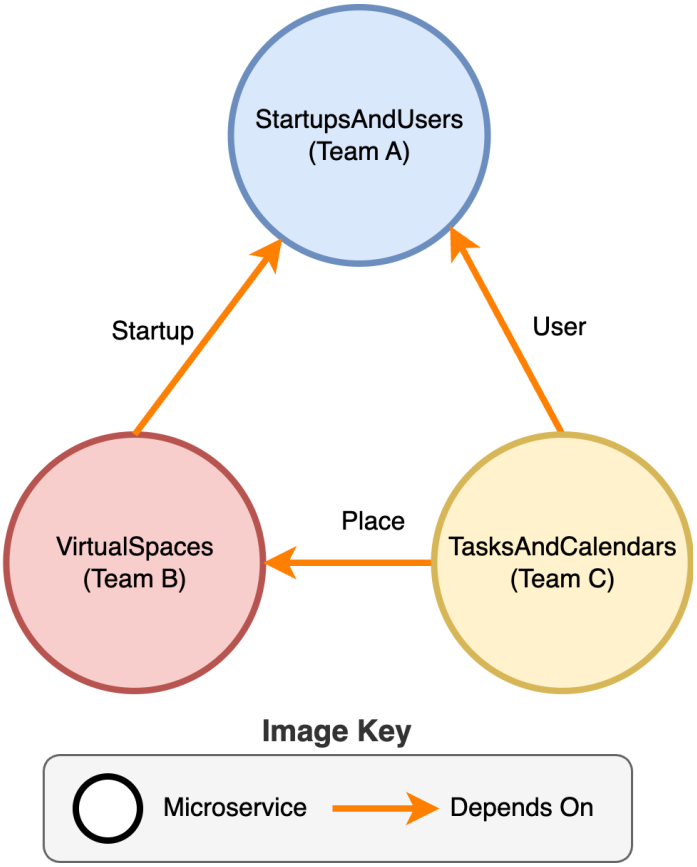

Figure 3.2: *Dependency between the microservices in the Digi-Dojo prototype*

3.2.2 Internal architecture

An overview of the internal architecture of each microservice is illustrated in the figure 3.3.

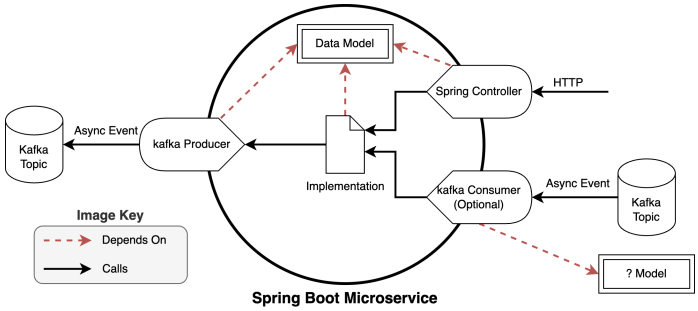

Figure 3.3: *Microservice internal architecture diagram*

The internal architecture is composed by a Spring Boot application with two entry points and one event writer.

The first entry point is a Spring controller responsible to implement a REST API to handle synchronous HTTP requests. This API serves as the main entry point for the microservice, being responsible for handling requests from clients of the Digi-Dojo system.

Both the second entry point and the event writer are the implementations of the event sourcing pattern. This pattern is responsible for the asynchronous communication throughout the system by utilizing events, ensuring consistency across the states of the business entities.

The second entry point is a Kafka consumer, responsible for listening to specific topics in order to receive relevant events. Unusual scenarios may arise where a microservice, such as the StartupsAndUsers microservice, has no dependency on these events. Consequently, the Kafka consumer becomes an optional component, not mandatory for all microservices.

Lastly, the event writer is a Kafka producer, responsible for sending relevant events associated to the entities owned by the microservice. These events are intended to be consumed by dependent microservices.

All these components rely on models that represent the business entities and the associated events and requests.

3.2.3 External architecture

The figure 3.4 represents each microservice and its integrations within the system.

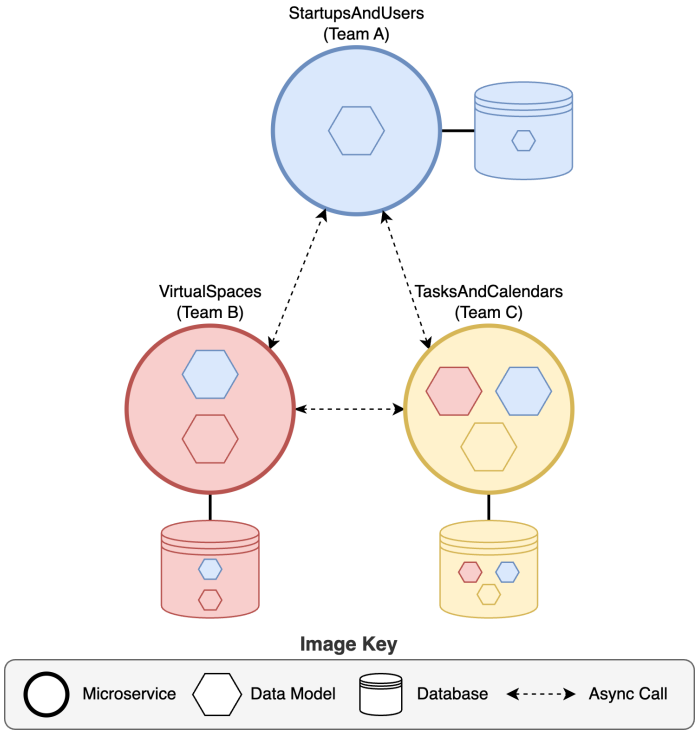

Figure 3.4: *Digi-Dojo prototype high level diagram*

At a first glance, it is crucial to elucidate that the frontend component of the prototype is omitted in this thesis. Thus, the communication with the actual client is beyond the scope of this work, given its inherent complexity inherited from the usage of the MSA, which is not the primary focus of the work here.

The prototype consists of the aforementioned three microservices, each communicating asynchronously with one another solely through events. Additionally, each microservice maintains its own database, adhering to the database-per-service pattern.

The models mentioned in the previous section become even more evident in this context. Each microservice rely on maintaining a local copy of the model representing the business entities it depends on. Additionally, this model is used to replicate the data within the database.

Chapter 4

Results

4.1 Spotting the symptoms

The first step towards the reference implementation involved spotting symptoms of a lack of modifiability in the Digi-Dojo prototype. This step is essential to identify real issues inside the application before proposing effective solutions.

4.1.1 Data model duplication

Referencing section 3.2.3, it becomes evident that the local duplication of the data model is unconventional. figure 4.1 illustrates this symptom, with a specific focus on the data model of the StartupsAndUsers microservice.

Besides being unusual, this practice introduces a dangerous failure point within the application, where a modification to an internal data model needs to be communicated to all dependent teams. The risk associated with this practice escalate significantly as the number of teams increases, being unscalable.

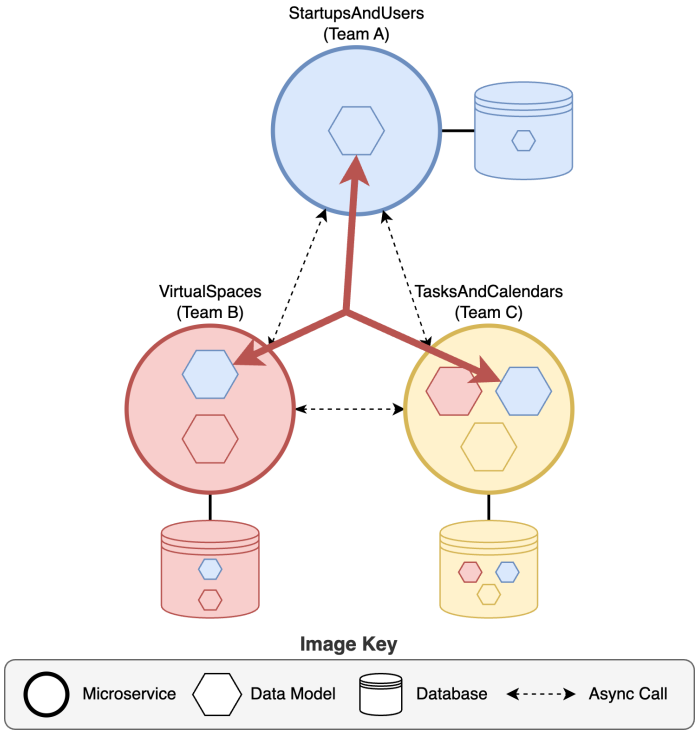

Figure 4.1: *Diagram spotting the data model from StartupsAndUsers being duplicated in the other microservices*

4.1.2 Data duplication

Beyond the data model duplication, section 3.2.3 also highlights another unconventional practice: duplicate the data owned by a microservice throughout the entire application. Figure 4.2 illustrates this symptom, with a specific focus on the data of the StartupsAndUsers microservice.

While replicating the data of another microservice is a known solution, it is uncommon to observe such replication across all microservices. This practice can introduce unnecessary complexity to the application in certain scenarios, demanding additional effort for maintenance.

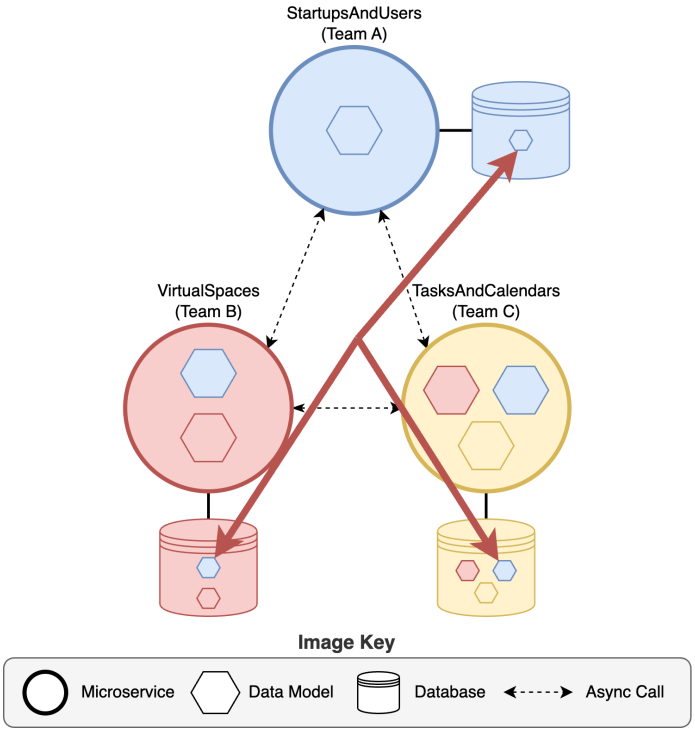

Figure 4.2: *Diagram spotting the data from StartupsAndUsers being duplicated in the other microservices*

4.1.3 Bad internal structure

An initial step towards understanding a microservice comprehends exploring its internal structure, such as folders, code, tests, and so on. However, in the Digi-Dojo prototype, the internal structure introduced confusion due to unusual and inconsistent practices. Figure 4.3 exemplifies one of these unusual practices: the duplication of the Kafka configuration across entities owned by the StartupsAndUsers microservice. Figure 4.4 illustrates an inconsistent practice: hidden domain violation of the Place and User entities nested under the "common" folder in the TasksAndCalendars microservice.

Even though these practices rarely introduce failure points, a poorly designed internal structure can induce to errors during the development and maintenance of the system.

Figure 4.3: *Internal structure of the StartupsAndUsers microservice focusing on the duplication of the Kafka configuration file*

Figure 4.4: *Internal structure of the TasksAndCalendars microservice focusing on the hidden domain violation of the Place and User entities*

4.2 Shared data model

This section is going to dive deep into the symptom related to the data model duplication.

4.2.1 Understanding the issue

Upon closer investigation, we could trace the symptoms back to a specific challenge inherited from the utilization of MSA: how to properly establish the connection between an entity owned by the microservice and another owned externally. Consequently, we were able to identify two main issues associated with this challenge.

The first issue presupposes a prior knowledge of the external model of the entity. Program 4.1 demonstrates the logic of the Kafka consumer in the TasksAndCalendars microservice, responsible for listening to events related to the creation of a new user. The entire method relies on an unstructured format of the listened event, exposing failure points on the application.

The second one involves an attempt to own the external entity within the microservice. For instance, in the TasksAndCalendars microservice, the User entity is not only stored within the database, but its life cycle is also managed by the microservice through local controllers and requests, violating the domain owned by the StartupsAndUsers microservice.

Program 4.1 Code snippet from the TasksAndCalendars microservice related to the Kafka consumer responsible for the User business entity.

```
1  @Component
2  public class UserConsumer {
3     private final CRUDUser crudUser;
4
5     @KafkaListener(topics = "user.created", groupId = "00")
6     public void consumeUserCreatedEvent(String jsonMessage) {
7        try {
8           JSONObject jsonObject = new JSONObject(jsonMessage);
9           JSONObject payload = jsonObject.getJSONObject("payload");
10
11          String idAsString = payload.getString("id");
12          String userName = payload.getString("name");
13
14          crudUser.createUser(userName);
15          System.out.println(userName);
16       } catch (Exception e) {
17          System.err.println("Error processing user created event: " + e.
                getMessage());
18       }
19    }
20 }
```

4.2.2 DTO pattern

Fowler, 2003 defines Data Transfer Objects (DTOs) as objects designed to carry data between processes in order to reduce the number of methods calls. Fowler explains that this is accomplished by consolidating multiple parameters into a single call. In essence, a DTO is an instance of a class that defines these parameters, serving solely for transfer purposes and devoid of any business logic.

Additionally, DTOs also serve as an excellent mechanism to encapsulate the serialization logic into a single point, keeping the data exchange format stable and isolated from the internal data structures changes. This allows both components to change independently. The figure 4.5 illustrates an example of how the communication can be done using the DTO pattern.

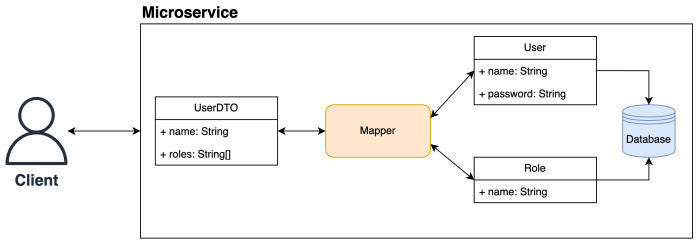

Figure 4.5: *Client <-> Microservice example communication using the DTO pattern*

4.2.3 The shared package

To address the first issue, the mechanism provided by the DTO pattern turned out to be a valuable solution by providing the necessary isolation to prevent the exposure of those failure points.

However, in the context of MSA, the client defined in figure 4.5 extends beyond the end-user through a frontend application; it also encompasses other microservices. In this scenario, ensuring a single source of truth and maintaining a consistent library becomes crucial for dependent microservices to rely on. In the context of the Digi-Dojo prototype, a unified package, named DigiDojoSharedModel, was created to serve as the library responsible for the DTOs and serialization logic of requests, entities, and events across all microservices.

The figure 4.6 compares the original implementation with the new one utilizing the shared package. The Spring controller, Kafka producer, and the Kafka consumer have transitioned from relying on an undefined model or local copy of the model to rely on a shared component, maintaining the data model solely for internal structures.

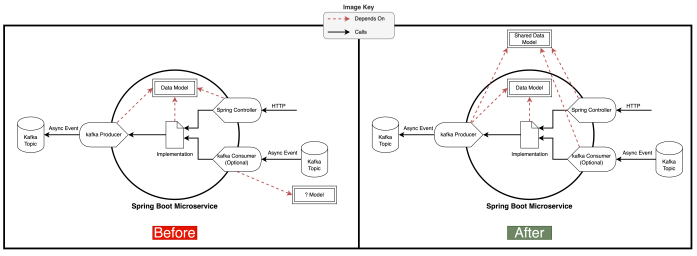

Figure 4.6: *Microservice internal architecture diagram showing before and after the refactor to use the shared package*

The figure 4.7 performs a similar comparison, but focusing on the external architecture. The visualization effectively highlights the removal of the local copy of the model from the microservices.

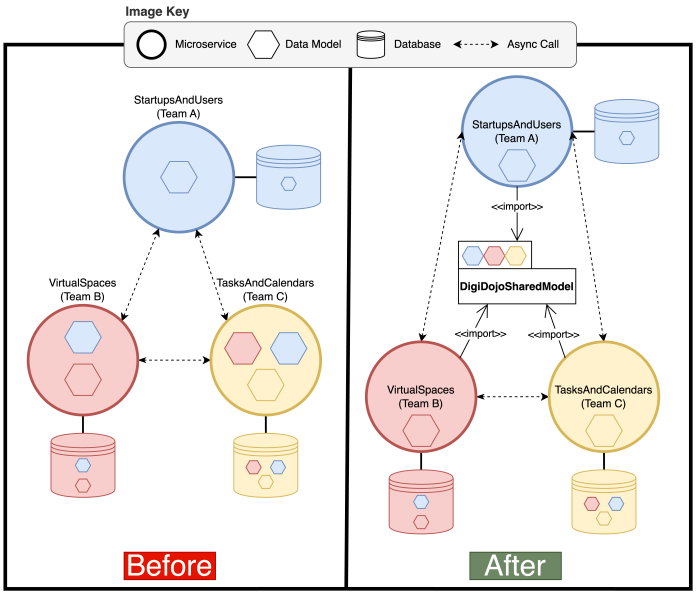

Figure 4.7: *Digi-Dojo prototype high level diagram showing before and after the refactor to use the shared package*

Finally, program 4.2 illustrates the updated implementation of the Kafka consumer in the TasksAndCalendars microservice that, in contrast with program 4.1, evidences an improved serialization logic with a well-structured model.

4.2.4 The domain violation

While the shared package doesn't completely resolve the second issue, it provides the needed consistency to discourage developers from violating the domain by having the essential models in a shared library. Concerning the local ownership of the lifecycle, the adopted approach involved removing this logic, as the integration between the microservices was appropriately adjusted (More details on this can be found in section 4.5). Finally, regarding the storage of entities within the database, this isn't necessarily an issue, as the next section will delve deeper.

4.2.5 Trade-offs

The DTO pattern evidently stands out as a robust solution to the challenge of properly establishing the connection between the microservices' domains. It ensures the needed attributes of consistency, isolation, and modifiability crucial for an MSA, as covered by the preceding subsections. Another notable advantage of the DTO pattern is its positive impact on system integration, contributing to cleaner code and reducing the risk of errors.

On the other hand, its inherent complexity poses challenges in terms of maintenance. Providing shared models requires each microservice owner to take ownership of them, managing its life cycle and ensuring that breaking changes will not impact the multiple clients. Therefore, there's value in a naive implementation that exposes the internal data model rather than maintaining DTOs, particularly in scenarios where this complexity can be avoided.

Program 4.2 Code snippet from the TasksAndCalendars microservice related to the new implementation of the Kafka consumer responsible for the User business entity.

```
1  import it.unibz.digidojo.sharedmodel.dto.UserDTO;
2  import it.unibz.digidojo.sharedmodel.event.user.UserCreatedEvent;
3
4  @Slf4j
5  @Component
6  public class UserConsumer extends BaseConsumer {
7     private static final String EVENT_TYPE_PATTERN = "USER_.+";
8     public static final RecordFilterStrategy<String, String> filterStrategy =
          generateFilterStrategy(EVENT_TYPE_PATTERN);
9     private final CRUDUser crudUser;
10
11    @KafkaListener(
12          topics = KafkaConfig.CREATE_TOPIC,
13          filter = "#{@userConsumer.filterStrategy}"
14    )
15    public void consumeUserCreatedEvent(String jsonMessage) {
16       try {
17          log.info("Processing a new Kafka event. jsonMessage={%s}",
                jsonMessage);
18          UserCreatedEvent event = marshaller.unmarshal(jsonMessage,
                UserCreatedEvent.class);
19          UserDTO user = event.user();
20          String username = user.name();
21
22          crudUser.createUser(username);
23       } catch (Exception e) {
24          log.error("Error processing user created event: " + e.getMessage());
25       }
26    }
27 }
```

4.3 Data replication VS Synchronous integration

This section is going to dive deep into the symptom related to the data duplication.

4.3.1 Understanding the issue

Delving deeper into the symptoms, we could trace them back to a sub-challenge of the one presented in section 4.2.1: how to properly handle the data of an entity owned externally. The difference between these challenges is subtle, and they cannot be entirely isolated from each other.

Consequently, the utilization of the data replication pattern is also linked to the attempt to internalize the external entity within the microservice. Furthermore, the widespread use of this pattern demonstrated an inability in handling this data effectively, rather than a deliberate utilization for the sake of the pattern's advantages.

4.3.2 The comparison

There is no single pattern that stands out as the definitive solution to address this challenge. As previously mentioned, data replication is a valid solution and should not be discarded. Consequently, the chosen approach involved a comparison and analysis of two distinct patterns: data replication and synchronous integration.

This comparison was implemented in the TasksAndCalendars microservice, involving two different entities. Data replication was retained for the Place entity, whereas synchronous integration was implemented to the User entity. Figure 4.8 elucidates how each pattern works. While data replication stores a copy of the data inside the database, synchronous integration calls the microservice owner of the entity – in the case of the User entity, the StartupsAndUsers microservice – to fetch the most recent necessary data, without storing it locally.

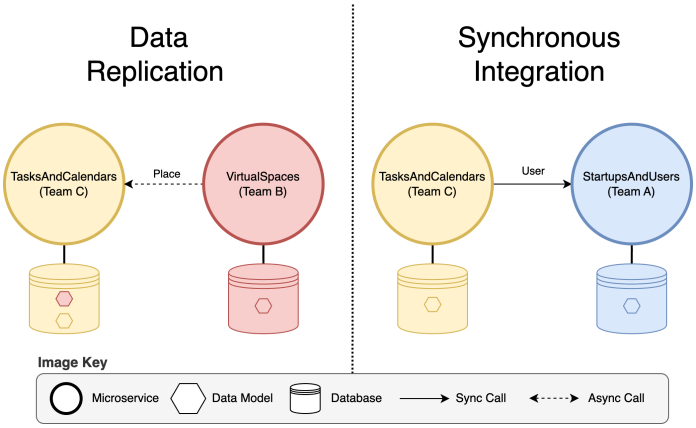

Figure 4.8: *Data replication and synchronous integration patterns explained*

Figure 4.9 illustrates the updated external architecture in contrast to the one presented in figure 4.7.

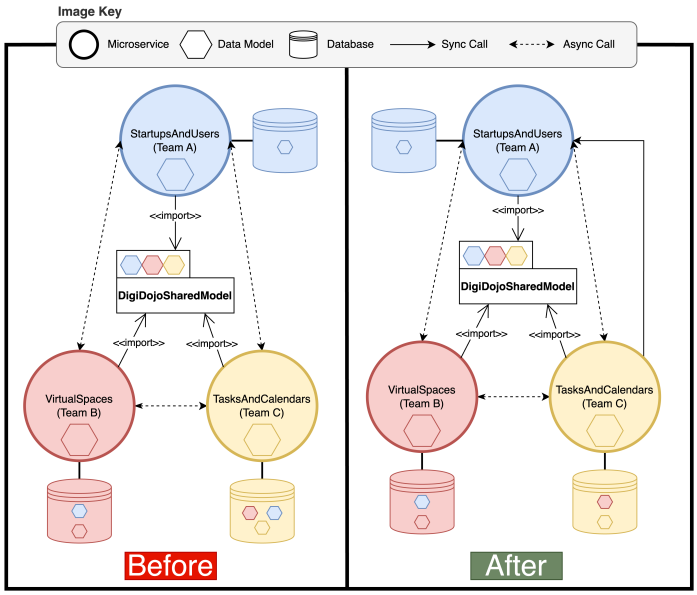

Figure 4.9: *Digi-Dojo prototype high level diagram showing before and after the refactor to compare the patterns*

4.3.3 Data replication pattern

The data replication pattern presents a valuable mechanism for a variety of use cases. The first and most straightforward utilization is in high-performance scenarios. The pattern can function as a service-level cache for microservices constrained by this scenario, effectively avoiding the overhead of network requests and improving availability (Selvi and Anbuselvi, 2018). However, given that the Digi-Dojo project does not relate closely with this requirement, we opted not to delve deeper into this aspect.

Another employment, although less discussed, is for flexibility requirements. In an ideal architecture, each microservice operates within its specific domain, enabling diverse interpretations and modelings of the same entity based on the unique characteristics of each domain. In this context, data replication provides the flexibility to restructure the model stored within the database for each service (Selvi and Anbuselvi, 2018). While this scenario is not necessarily distant from the Digi-Dojo project, the prototype comprehends only initial entities, making it premature to address such specific needs at this stage.

However, the application of this pattern has highlighted an interesting advantage for microservices implemented using an object-oriented programming language. Object mapping are techniques employed on this type of language to transform data stored in the database into in-memory objects for manipulation during the microservice execution (Störl *et al.*, 2015). Using the data replication pattern together with an object mapper potentializes the behavior of this practice. Taking as example the program 4.3, it exemplifies an object-relational mapping (ORM), responsible for automatically maps relational data into an in-memory instance of the Task entity. In this instance, there's a populated reference of a Place entity that was built out of the box by the ORM, because the places are being replicated within the relational database.

Program 4.3 Code snippet from the TasksAndCalendars microservice related to the Task entity model (without the association to the user)

```
1   @Data
2   @Entity
3   @NoArgsConstructor
4   @AllArgsConstructor
5   public class Task {
6      @Id
7      @GeneratedValue(strategy = GenerationType.IDENTITY)
8      private Long id;
9
10     private String title;
11
12     private String description;
13
14     private Status status;
15
16     @ManyToOne(cascade - CascadeType.REMOVE)
17     private Place place;
18
19     private List<String> tags;
20  }
```

4.3.4 Synchronous integration pattern

The synchronous integration pattern offers a more conventional mechanism, requiring less complexity to maintain. Its principal advantage lies in the assurance of consistently receiving the most up-to-date data. With this pattern, microservices are guaranteed to fetch the latest information available, being only subjected to the eventual consistency of the microservice that owns the data.

On the other hand, compared to the data replication pattern, integrating with object mapping techniques becomes much more challenging. Revisiting program 4.3 as an example, let's consider adding the missing association with the User entity. Since it doesn't exist in the database, the ORM is going to be unable to build a populated reference, unlike the place reference. Additionally, some SQL mechanisms, such as foreign key constraints, cannot be relied upon to ensure the consistency of relationships. For example, in the event of a user deletion, the microservice must actively listen to this event and update the database; no automatic processes are in place. Program 4.4 illustrates the updated Task model, containing only a reference to the primary key of the associated user, mirroring the SQL modeling. To maintain the consistency in the described scenario where a user in deleted, the microservice needs to modify the user identifier field to be nullable or adopt a representation of the absence of a user, relying on a customized constraint to ensure that the task has at least one owner entity remaining, as represented by the *@HasAtLeastOneOwnerconstraint* annotation.

Program 4.4 Code snippet from the TasksAndCalendars microservice related to the Task entity model

```
1   @Data
2   @Entity
3   @HasAtLeastOneOwnerConstraint
4   @NoArgsConstructor
5   @AllArgsConstructor
6   public class Task {
7      @Id
8      @GeneratedValue(strategy = GenerationType.IDENTITY)
9      private Long id;
10
11     private String title;
12
13     private String description;
14
15     private Status status;
16
17     private Long userId;
18
19     @ManyToOne(cascade - CascadeType.REMOVE)
20     private Place place;
21
22     private List<String> tags;
23  }
```

4.3.5 Trade-offs

While the data replication pattern supplies notable benefits in performance, low-coupling, and fault tolerance across multiple architecture scenarios, it also presents an enormous potential risk that must be carefully considered. The advantages are evident, yet it is crucial to analyze the escalating complexity associated with data ownership, infrastructure, and modeling.

Although the automated integrations with object mapping techniques may create an impression of simplicity, the effort required to ensure the consistency of the replicated data significantly amplifies the overall complexity. Consequently, inexperienced developers may lead themselves towards a pitfall when deciding to adopt this pattern, as evidenced in the Digi-Dojo prototype.

Conversely, the data replication pattern serves as an excellent mechanism for enhancing modifiability, offering isolation and flexibility throughout the application. It facilitates the reuse of domain components across multiple business logics, contributing to improved adaptability and maintainability.

Meanwhile, the synchronous integration pattern offers a simple and straightforward alternative at the cost of increased coupling between microservices and reduced fault tolerance. It's essential to understand that coupling is not always a negative effect, in certain scenarios, it fosters a better domain isolation and improved consistency across the application. However, this pattern introduces complexity in effectively managing

relationships between internal and external entities.

In certain instances, an approach aligned more closely with the monolith architecture can help in keeping this complexity manageable, such as opting for a shared database instead of following the database per service pattern.

4.4 Domain-centric style VS Classic style

This section is going to dive deep into the symptom related to the bad internal structure.

4.4.1 Understanding the issue

Directly from the figures 4.3 and 4.4, it is possible to identify an attempt to use the clean architecture pattern using the domains of the microservice as the main division.

Clean Architecture is a software design concept introduced by Martin, 2017. It emphasizes the separation of concerns and the independence of the application's business logic (the "domain" folder) from the infrastructure (the "application" folder). The architecture is designed with various layers, such as the Entities, Use Cases, and Interface Adapters, to ensure the separation of concerns and the independence of the frameworks and tools. The main goal of Clean Architecture is to create a system that is independent of the delivery mechanisms, databases, frameworks, and other external details, allowing it to be simple, testable, and flexible.

After analyzing the symptoms illustrated in figure 4.3, we could identify that all Kafka configurations were identical. This observation led us to trace the root cause back to a specific challenge arising from the utilization of the clean architecture: how to properly share artifacts across domains.

Regarding the symptoms manifested in figure 4.4, we correlated them with a consequential side effect from the domain violation issue exposed in section 4.2.1. As a result, incorporating this issue into the natural structure of the clean architecture becomes more challenging, given that it was not originally intended to be part of the design.

4.4.2 Shared kernel

In the context of Domain-Driven Design (DDD), a shared kernel refers to a set of shared domain concepts, rules, and language that is understood and agreed upon by all stakeholders involved in the software development process. By focusing on this shared kernel, DDD aims to facilitate better communication, reduce misunderstandings, and improve the overall design and implementation of the software system (Evans, 2003).

To tackle the observed challenge, we opted to adapt the shared artifacts into a shared kernel, enabling multiple domains to benefit from them properly. This adaptation provides an effective mechanism for supporting new domains while preserving the necessary isolation between them.

4.4.3 The comparison

Taking advantage of the identified issues associated with the clean architecture pattern, for experimentation purposes, we chose to implement and compare two styles of this pattern: the domain-centric style and the classic style.

The domain-centric style represents the current adopted style. It properly isolates the application's business logic from the infrastructure by dividing the internal structure into distinct domains. The shared kernel was implemented following this style in the StartupsAndUsers microservice.

In contrast, the classic style represents a more direct implementation of the clean architecture. Instead of dividing the internal structure, it solely isolates the application's business logic from the infrastructure. On this style, the division by domains only happens inside the business logic itself. Following this approach, the shared kernel was implemented in the TasksAndCalendars microservice.

4.4.4 Domain-centric style

Figure 4.10 provides a visual representation of the updated internal structure of the StartupsAndUsers microservice. As this was the current style, the core structure has been retained. The shared kernel was refactored into the "common" folder, following a straightforward implementation, since the only shared artifact was the Kafka configuration. However, this implementation highlighted the pattern effectiveness as a mechanism for reusability while maintaining the necessary flexibility for extension. For example, the mentioned figure demonstrates the shared kernel with generic Kafka consumer and producer implementations, whereas the Startup domain introduces its specific adaptations of these components.

In the domain-centric style, the shared kernel is treated similar to a new domain within the microservice, residing in its own designated folder with its own implementations. This approach enhances clarity, facilitating the identification of artifacts within this component and ensuring their proper isolation from the other domains.

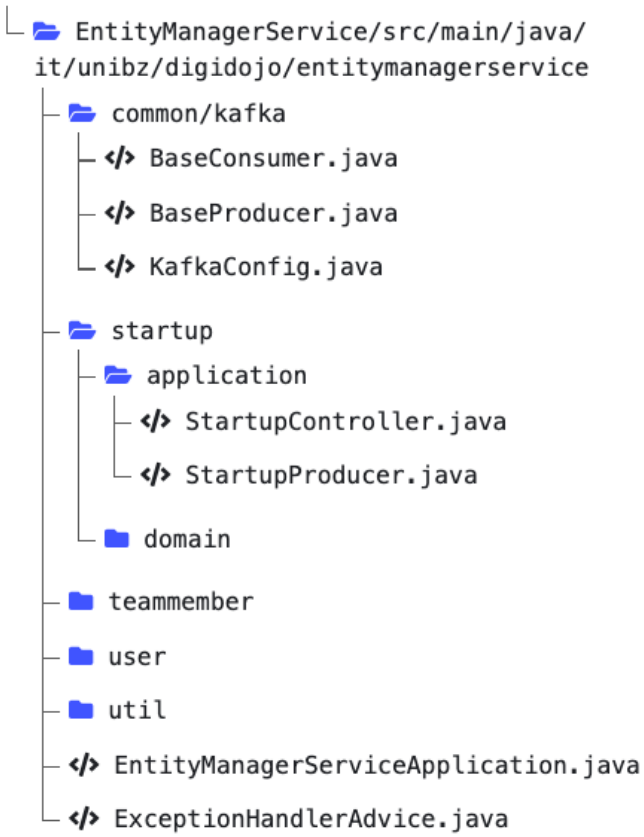

Figure 4.10: *Updated internal structure of the StartupsAndUsers microservice (renamed to EntityManagerService) focusing on the shared kernel for Kafka artifacts*

4.4.5 Classic style

Figure 4.11 produces a visual representation of the updated internal structure of the TasksAndCalendars microservice. The first noticeable change is the adoption of an entirely new format, with the shared kernel spread across the structure without a centralized location, unlike the previous style. Consequently, there is an impression of the shared kernel' s absence, primarily rooted by the decision not to divide the structure by domain. While this choice sacrifices the isolation of the shared kernel into a distinct component, it offers greater flexibility.

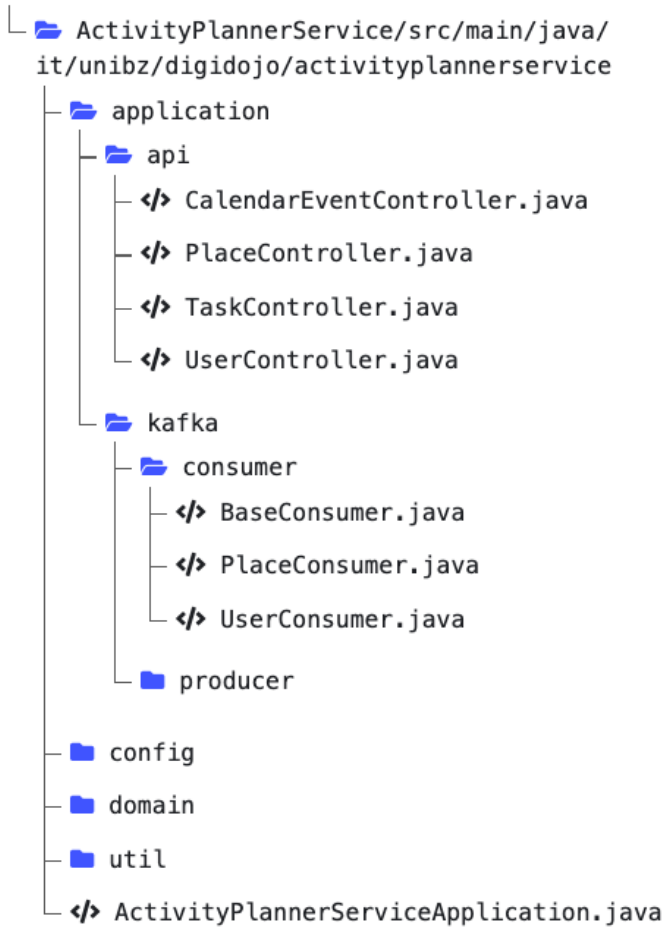

Figure 4.11: *Updated internal structure of the TasksAndCalendars microservice (renamed to ActivityPlannerService) focusing on the shared kernel for Kafka artifacts*

Moreover, this style presents a notable advantage in terms of visibility. A glance allows a comprehensive overview of both the entire infrastructure – nested under the "application" folder – and the business logic – arranged by purpose under the "domain" folder. This organization facilitates a quick and holistic understanding of the implementation.

4.4.6 Trade-offs

Regardless of the chosen implementation style of the clean architecture, this pattern promotes a robust domain and application segregation, increasing the modifiability by minimizing the need for extensive adaptations to accommodate new changes. However, the primary trade-off lies in the experience of the team responsible for the microservice.

The domain-centric style supplies a more intuitive visualization of the domains, enhancing the clarity of specific contexts and contributing to an overall clean structure. Nevertheless, maintaining this structure over the long term requires a certain level of experience to ensure its continued quality. Inexperienced developers may be prone to duplicating artifacts due to a lack of comprehensive visualization of the broader structure.

Conversely, the classic style serves as an alternative to enhance visibility. It also fosters improved sharing of artifacts by providing a centralized location for them. On the other hand, this style might offer an overwhelming amount of information, potentially presenting too much visibility. Considering the right balance between clarity and complexity is crucial when considering the classic style.

4.5 Additional work done

In order to be able to properly perform the aforementioned analysis, several additional tasks were needed. The following list itemizes these endeavors:

- **Kafka integration:** Searched and configured a free Kafka provider for the application. Due to the limits of the free tier, certain adaptations, such as implementing a filter logic, were necessary.
- **Containerization:** implemented Docker containers using Docker Compose to ensure replicability, isolation, and reliability in the application environment.
- **Microservices naming:** Renamed the microservices from TasksAndCalendars to ActivityPlannerService and from StartupsAndUsers to EntityManagerService, aiming to enhance semantic clarity.
- **Integration tests:** Introduced a prototype (formerly referred to as a tracer bullet) of an ideal integration test.
- **General improvements:** Implemented several enhancements, including addition of logging, parsing improvement, configuration of Checkstyle, Spotbugs, and code coverage for code quality checks, refactoring towards the REST standards, remodeling certain business entities, and more.

Chapter 5

Conclusion

5.1 Recap

This thesis marks a significant contribution to the field of software architecture, with a specific focus on enhancing the modifiability quality attribute in Microservices Architecture (MSA). Its relevance is further amplified by its practical application in a remote work platform, demonstrating its alignment with current trends in the software industry.

A key achievement of this study is the identification and resolution of a fundamental modifiability challenge: how to properly establish the connection between an entity owned by the microservice and another owned externally. The implementation of the DTO pattern and the shared data model emerged as pivotal solutions, significantly augmenting system robustness and flexibility. This approach effectively overcomes traditional barriers, offering a comprehensive solution to this challenge.

Another notable aspect of this work is the exploration of a challenge related to properly handle the data of an entity owned externally. The thesis does not commit to a singular approach, but instead provides an insightful comparative analysis of data replication and synchronous integration patterns. This comparative study not only deepens our understanding of these patterns but also provides crucial guidance for system design, particularly in contexts where choosing the optimal approach is crucial for system efficacy.

Finally, the thesis also delves into the challenge of properly sharing artifacts across domains. Leveraging the principles of clean architecture, a shared kernel pattern was proposed and examined in two distinct styles. This investigation yielded valuable insights into the application and benefits of each style, enhancing our understanding of their implementation in the development of new microservices.

An intriguing observation from this study is the direct correlation between modifiability and system complexity. It becomes evident that certain scenarios may not necessitate high modifiability, allowing for simpler yet functional designs. This finding underscores the importance of context in architectural decision-making, advocating for a balanced approach where complexity is only embraced when justified by the requirements.

In conclusion, the practical aspect of these findings within the Digi-Dojo project not

only demonstrates the work's immediate applicability but also sets a strong foundation for future enhancements. The improvements in modifiability achieved through this study lead the way for more adaptable, scalable, and maintainable microservices, ultimately contributing to the success and sustainability of software projects in the dynamic environment of software architecture.

5.2 Future works

Future works may include:

- Explore the impact of the adopted solution to different quality attributes, such as security.
- Explore the Shared Kernel pattern potential.
- Extend the Digi-Dojo implementation to cover more real-world scenarios.

References

[Alexander *et al.* 1977] Christopher Alexander, Sara Ishikawa, and Murray Silverstein. *A Pattern Language: Towns, Buildings, Construction*. Oxford University Press, 1977. ISBN: 0195019199 (cit. on p. 4).

[Bass *et al.* 1997] Len Bass, Paul Clements, and Rick Kazman. *Software Architecture in Practice*. Third Edit. Addison-Wesley, 1997. ISBN: 9780321711502 (cit. on pp. 1, 3).

[Bawiskar *et al.* 2012] Ankur Bawiskar, Prashant Sawant, Vinayak Kankate, and B. B. Meshram. "Spring framework: a companion to javaee". In: 2012. URL: https://api.semanticscholar.org/CorpusID:15705854 (cit. on p. 5).

[Bogner *et al.* 2019] Justus Bogner, Stefan Wagner, and Alfred Zimmermann. "Using architectural modifiability tactics to examine evolution qualities of service- and microservice-based systems". *SICS Software-Intensive Cyber-Physical Systems* 34 (2019). DOI: 10.1007/s00450-019-00402-z (cit. on p. 1).

[Daniel *et al.* 2023] João Francisco Lino Daniel, Xiaofeng Wang, and Eduardo Martins Guerra. "How to design future-ready microservices? analyzing microservice patterns for adaptability". In: *28th European Conference on Pattern Languages of Programs (EuroPLoP 2023)* (Irsee, Germany, July 5–9, 2023). 2023. DOI: 10.1145/3628034.3628046 (cit. on p. 9).

[Evans 2003] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. 1st edition. Addison-Wesley Professional, 2003. ISBN: 9780321125217 (cit. on p. 27).

[Fowler 2003] Martin Fowler. *Patterns of Enterprise Application Architecture*. 1st edition. Addison-Wesley, 2003. ISBN: 9780321127426 (cit. on p. 19).

[Fowler 2019] Martin Fowler. *Software Architecture Guide*. 2019. URL: https://martinfowler.com/architecture (cit. on p. 1).

[Hubli and Jaiswal 2023] Sushmita C. Hubli and Dr. R. C. Jaiswal. "Efficient backend development with spring boot: a comprehensive overview". *International Journal for Research in Applied Science and Engineering Technology* (2023). URL: https://api.semanticscholar.org/CorpusID:265190396 (cit. on p. 5).

[Larsson 2019] Magnus Larsson. “Hands-on microservices with spring boot and spring cloud”. In: 2019. URL: https://api.semanticscholar.org/CorpusID:213222652 (cit. on p. 5).

[Martin 2017] Robert C. Martin. *Clean Architecture: A Craftsman’s Guide to Software Structure and Design*. 1st edition. Pearson, 2017. ISBN: 9780134494166 (cit. on p. 27).

[Newman 2015] Sam Newman. *Building Microservices - Design Fine-Grained Systems*. 2015. URL: http://safaribooksonline.com (cit. on pp. 3, 4, 7).

[Newman 2020] Sam Newman. *Monolith to Microservices - Evolutionary Patterns to Transform Your Monolith*. 2020. URL: http://safaribooksonline.com (cit. on pp. 3, 4).

[Northrop 2004] Linda Northrop. *Achieving Product Qualities Through Software Architecture Practices*. https://resources.sei.cmu.edu/asset_files/Presentation/2004_017_001_22862.pdf. 2004 (cit. on p. 3).

[Richards 2015] Mark Richards. *Software Architecture Patterns - Understanding Common Architecture Patterns and When to Use them*. 2015 (cit. on p. 7).

[Richardson 2018] Chris Richardson. *Microservices Patterns: With examples in Java*. First Edition. Manning, 2018. ISBN: 9781617294549 (cit. on p. 4).

[Selvi and Anbuselvi 2018] S. Selvi and R. Anbuselvi. “An analysis of data replication issues and strategies on cloud storage system”. *International journal of engineering research and technology* 3 (2018). URL: https://api.semanticscholar.org/CorpusID:88476981 (cit. on p. 24).

[Singhal 2022] Ritik Singhal. “Spring boot backend development”. *International Journal of Advanced Research in Science, Communication and Technology* (2022). URL: https://api.semanticscholar.org/CorpusID:249875573 (cit. on p. 5).

[Störl *et al.* 2015] Uta Störl, Thomas Hauf, Meike Klettke, and Stefanie Scherzinger. “Schemaless nosql data stores - object-nosql mappers to the rescue?” In: *Datenbanksysteme für Business, Technologie und Web*. 2015. URL: https://api.semanticscholar.org/CorpusID:1691011 (cit. on p. 24).

[Wang *et al.* 2022] Xiaofeng Wang *et al.* “Startup digi-dojo: a digital space supporting practice and research of startup remote work”. *CEUR Workshop Proceedings* 3316 (2022). URL: https://ceur-ws.org/ (cit. on p. 11).