Vrije Universiteit Amsterdam

Universiteit van Amsterdam

Master Thesis

Automated Data Management for Microservice Architectures

Author: Thijmen J. Kurk (2627603)

1st supervisor: Dr. J. Gordijn
2nd reader: Prof. Dr. R.J. Wieringa

A thesis submitted in fulfillment of the requirements for the joint UvA-VU Master of Science degree in Computer Science

August 19, 2022

"I am the master of my fate, I am the captain of my soul"
from Invictus*, by William Ernest Henley*

Contents

List of Figures

List of Tables

1

Introduction

Nowadays, microservices are the norm when software engineering scalable applications, especially considering that cloud computing and containerization technologies are becoming more prominent (1, 27, 32). Amazon, Netflix, LinkedIn, Spotify, SoundCloud and other companies (11, 52, 55) are actively adopting and evolving architectures in which microservices are deployed. Such architectures typically contain a large number of inter-connected but loosely-coupled services, which collectively form a highly scalable but complex distributed system. The first definition of a *microservice* was introduced by (17) in 2014 as: *'A service that can be automatically and independently be deployed, runs in its own process and communicates using lightweight mechanisms'*. A collection of such services is called a *microservice architecture* (MSA). The services inside a MSA work together towards desired business objects (17), and its composition is partly determined based on the requirements that are tied to these objectives (e.g. performance, usability, and scalability requirements).

The management of data in MSAs is challenging, since it requires coordination between services to perform operations that are otherwise relatively simple in the context of monolith systems (50). A literature study identified three data management challenges in MSAs (Appendix A). Namely, (1) distributed transactions that require the coordination of multiple services, (2) maintaining consistency of data that has relations that spawn different microservices and (3) joining data from separate microservices together efficiently. To the best of our knowledge, we did not find any tractable generic solutions available to resolve these challenges without developers needing to write repetitive code (Appendix A). This thesis we will contribute by designing, implementing and prototyping a solution named DMan towards solving these three challenges(19, 25, 38). DMan implements a transaction system that supports two different protocols for distributed transactions. Additionally, this

solution will introduce a novel layer named data services inside of microservices to handle data consistency and joining for service-crossing relations based on a relational model defined in the domain space of microservices.

The primary audience of this thesis are developers that are either or both interested in data management inside MSAs or looking to use and further develop DMan. The remainder of this thesis is split into seven chapters.

- Chapter 2: Microservice Architectures — provides the reader with the background information required for the rest of the chapters.
- Chapter 3: Problem Statement and Approach — outlines additional context of DMan by describing the requirements, what research methodology is used for this thesis and how to requirements are validated.
- Chapter 4: Design of DMan: A Generic Data Management Solution — describes major design decisions that are of relevance for the reader to understand the implementation.
- Chapter 5: Implementation of DMan: A Generic Data Management Solution — describes how DMan is implemented with the goal of exposing the non-trivial components to interested developers.
- Chapter 6: Validation — outline how the implementation of DMan is validated.
- Chapter 7: Discussion — reflect on the thesis and expose the limitations of DMan.
- Chapter 8: Conclusion — summarize the contributions of this thesis.

2

Microservice Architectures

In this chapter we introduce background information, concepts and ideas necessary to understand this thesis work. The content of this chapter is partly derived from a systematic literature study (Appendix A). The goal of this systematic literature study is twofold: (1) to identify and classify components commonly found in scientific literature used in MSAs and (2) to identify and describe challenges within MSAs regarding data management. The model derived from the literature for the first goal introduces the reader to MSAs, were the results stemming from the second goal introduces the reader to data management between microservices and the challenges thereof. For this thesis, we choose to use the following definition of microservices (Definition 1) and MSAs (Definition 2) both originating from (18), where a process can be either be a process as defined by the operating system (OS) or a collection of threads within an OS process. We use the term service and microservice interchangeably in this thesis. The model created from by the literature study is displayed in Figure 2.1. It defines the type of services and the main components inside a microservice without detailing any topological specifics. The model will now be described in detail.

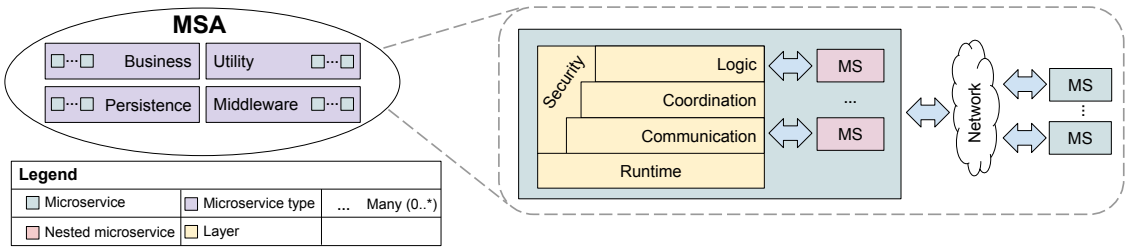

Figure 2.1: Generalized model of microservice architectures

Definition 1 (Microservice, MS) *A microservice is a cohesive independent process interacting via messages.*

Definition 2 (Microservice Architecture, MSA) *A microservice architecture is a distributed application where all its modules are microservices.*

We identified four types of microservices commonly found in MSAs. (i) Business microservices, services that implement logic towards a business objective, (ii) utility microservices, services that provide other microservices with general functionality (e.g., logging, monitoring, circuit breakers, load balancers, service discovery, etc. (41)), (iii) persistence microservices, these services typically provide database management systems (DBMS) or cache systems to other microservices, and finally (iv) middleware services, which facilitate communication between other microservices (e.g., an API gateway, a message broker, a message bus, etc. (18, 41)). An MSA is an interdependent cohesive loosely-coupled composition of these four types of services.

Every microservice can have nested microservices which only the parent microservice is allowed to communicate with. Nested microservices can be colocated on the same node as the parent microservice or hosted on separate nodes (41). The former is generally the case with the sidecar pattern in which nested utility microservices are deployed to abstract away general functionality from the parent, while the latter is often the case with persistence services (36, 41). A microservice consists of 3 layers with one overarching layer being security and one foundational layer being runtime (18, 36, 41, 47). Each layer is now described.

Logic is the top layer inside of microservices and contains the programming needed to implement the necessary business functionality required for a particular service. The structure of both the MSA itself and the logic implemented by microservices in enterprise applications, is typically defined by using Domain Driven Design (DDD) principles (14, 44, 51). In DDD, the problem space of the business is referred to as the domain. The domain can be divided into multiple subdomains, where every subdomain has its own domain model, the scope of which is called a bounded context (13, 41, 42, 49). DDD provides a framework that allows software architects to map business microservices to these contexts (14, 41, 44, 51). Bounded-contexts are mapped using domain models that capture the assigned part of the domain with a collection of objects interconnected with relationships of different cardinalities that reflect real-world entities (14, 41). The business goals of the MSA dictate what properties and actions are attached to each object. Conceptually, the logic of business microservices reflect the domain model to which it is assigned

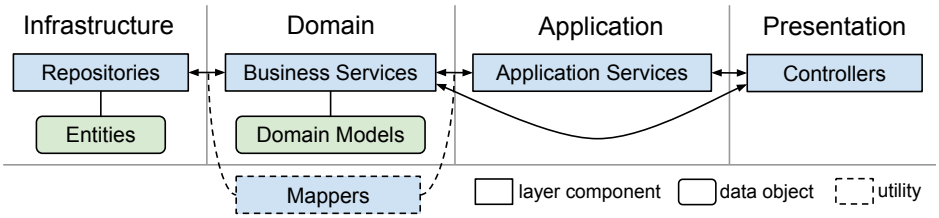

Figure 2.2: The layers commonly used to implement logic in microservices within enterprise applications

and is typically structured using a layered architecture that consists of 4 service layers (14, 16, 44, 51): (1) the user-interface (UI) or presentation layer, (2) the application layer, (3) the domain layer and (4) the infrastructure layer (Figure 2.2). The presentation layer has overlap with the communication layer because there is no UI needed for microservices since typically only data is returned (17). The application layer is a thin layer that does not contain any business rules or knowledge, but only coordinates the flow of the service (e.g. session management) (14). The domain layer is where the objects contained in the bounded-context of the business microservice live, these objects are also called domain models (14). The business services within the domain layer are responsible for applying business logic to the relevant domain models. Finally, the infrastructure layer (or persistence layer) contains repositories that are responsible for retrieving and storing entities that represent the aforementioned objects (or domain models). The translation of entities to domain models and vice versa is done by mappers.

Coordination between services inside an MSA is necessary since it is a distributed system per definition (10, 36, 41). The coordination between services to perform more complex and elaborate functionalities are either choreography or orchestration based (18, 23, 41). Orchestration requires a service (an orchestrator) that directly coordinates with other services to oversee the process required by the functionalities. Whereas, choreography requires an event brokers and asynchronous events with the publish/subscribe pattern to facilitate collaboration (18, 36, 41). In essence, the coordination between microservices is about data management (31) and is required when relations spawn different bounded-contexts. The common challenges of data management within MSAs are described in Section 2.1.

Communication between microservices consists of four parts (47, 50), (i) the network architecture, (ii) the interaction model, (iii) the transportation and finally (iv) the presentation. There are two different types of network architectures in distributed systems (50).

Namely, centralized and decentralized architectures. In a centralized architecture there is typically one central authority that keeps track of state from which it coordinates other nodes in the network (50). Whereas in a decentralized architecture, there is no central state and every node makes its own decisions based on the state of itself and neighboring nodes (50). Centralized systems have the advantage of being relatively simple to implement and debug but are unable to scale beyond a certain point due to hardware restrictions. Decentralized systems typically have the ability to scale by simply deploying more nodes and do not have a singular point of failure. The interaction model between services can be synchronous or asynchronous. Synchronous communication implies a request/response type pattern, meaning that it is vulnerable for services blocking other services while they are waiting for a response (10, 36, 41). Asynchronous communication is non-blocking by definition and often comes hand in hand with event-based architectures (49). However, it requires state management within microservices and typically incurs more communication overhead due to it generally requiring some kind of middleware service, such as a message broker or message bus to facilitate and decouple message delivery (10, 30, 41). Messages between microservices are commonly transported either via the network or through Inter Process Communication (IPC) depending on service locality, performance and scalability requirements (50). The protocol used for transportation should be selected while considering the previously defined parts and can be performed using a variety of protocols (47) (e.g., HTTP, gRPC, XMPP, MQTT, AMQP, etc.). Finally, the presentation of the data entails how data objects are transformed to bytes, such that it can be easily transported using the transportation protocol. This process is called serialization (and its reverse is called deserialization) (41). Common serializers are JSON, XML and Protobuf (41).

Runtime of a microservice consists of the platform on which it is executed, and the technology used to implement the layers above (e.g. the programming langauge used). The industry has evolved such that physical hardware is abstracted away behind many layers of virtualization (41). There are two levels of virtualization relevant to MSAs, (i) Hardware Abstraction Layer (HAL) virtualization (i.e., virtual machines (41)) and (ii) OS-level virtualization (i.e., containers (27, 41)). Microservices are typically deployed in either, or a combination, of these types of virtualized platforms (11). Runtime also specifies the isolation level of the microservices. For example, microservices that run in the same process on the same machine are less isolated than microservices that run on or in separate machines or containers.

Security is a layer that overarches all the previously defined layers and is challenging to implement properly. However, it vital within MSAs, due to the large attack surface

and complicated connections between services (46). There are two approaches to securing distributed systems, either a zero-trust-network or a trust-the-network approach (3, 7). Depending on the security requirements of the MSA, a certain network type should be chosen, since it influences decisions made during designing process of MSAs (3, 7, 18, 46).

2.1 Data Management Challenges

Typically, there is some (minimal) data dependency between the bounded context of services. Therefore, inter-microservice data management (handled by the coordination layer in the model, Figure 2.1) is required, given that MSAs are commonly designed with a database per service pattern (48). According to the **CAP** theorem, in any distributed system which shares **P**artitions of data can only pick one of/attain a balance between the following two properties: **A**vailability or **C**onsistency (4). For example, a highly available MSA cannot be consistent all the time (41). Therefore, depending on the consistency and availability requirements of the MSA, different type of solutions to facilitate inter-microservice data management are desirable. We now describe three common MSA data management challenges found in scientific literature.

2.1.1 Data Consistency

Consider two business microservices, one that keeps track of departments and one that keeps track of employees. Every department can have zero-to-many employees. In a relational DBMS, this type of relation is enforced by a foreign key on employee referring to the department of which the employee is a part of (35). The DBMS enforces policies to keep the database consistent, by for example, cascade deleting employees that are member of a deleted department, or making sure that every employee is member of a valid department. However, this type of enforcement is not trivial if each service has its own (possibly different type of) DBMS (31, 36, 41). Foreign key emulation is currently resolved using ad-hoc implementations due to the lack of a general solution (31).

2.1.2 Distributed Transactions

A transaction is a unit of work (a sequence of operations) that needs to be completed in its entirety or rolled back (28). A distributed transaction is a transaction that involves multiple nodes. Distributed transactions are challenging implement in MSAs, partly due to the use of (possibly polyglot) persistence (10, 31, 41). Transactions typically follow either

the ACID (Atomicity, Consistency, Isolation, Durability) or BASE (Basically Available, Soft state, Eventual consistency) principles (10, 35, 50).

Consider two business microservices, X_1 that keeps track of inventory and X_2 that is responsible for the placement and tracking of orders. When an order is placed, the stock of the product is updated through the inventory service. A product with only one left in the stock is ordered by two users at the same time. Now X_2 receives two requests to check the stock followed by two requests to update the stock. A naive implementation allows the product to be ordered twice, since X_1 and X_2 do not coordinate the transaction. There are two solutions for this problem, depending on the desired properties of the MSA, displayed in Table 2.1.

Saga is pattern in which eventual data consistency can be guaranteed by splitting up inter-microservice transactions into a sequence of compensable (potentially retryable) subtransactions each having the scope of only a single microservice (31, 41). A Saga transaction is successful if each subtransaction has succeeded and can be coordinated using either a choreography or an orchestration based interaction model (41). Choreography interaction models coordinate the execution of subtransactions through events and a central event broker, whereas the orchestration based models use orchestrators (41). Orchestrators are solely responsible for the invocation of the sequence of transaction operations and do not operate on events of other services. If one subtransaction fails, the already succeeded subtransactions are rolled back by executing their respective compensating subtransactions. This makes the MSA eventual consistency and basically available (4).

Two-Phase Commit (2PC) is a pattern in which strong data consistency is guaranteed by orchestrating each operation of the inter-microservice transaction in two phases (31, 41). The transaction orchestrator starts the first phase by requesting each service to prepare the necessary operations and waiting for each to respond. The second phase executes after each service has successfully prepared and makes the operations permanent. The first phase requires each service to lock resources until the transaction is finalized or aborted since otherwise transactional consistency cannot be guaranteed (31, 40). 2PC trades availability for strong data consistency (41).

2.1.3 Data Joining

When a microservice requires data from multiple service, it needs to retrieve and join data from each microservice individually and deal with possible inconsistencies by itself (31). These inconsistencies can form due to, for example, the incorrect implementation of ad-hoc solutions described in Section 2.1.1 or a service failing to return the requested data

Pattern	Interaction Model	Consistency	Availability	Principles
Saga	Orchestration, Choreography	Eventually consistent	Available	BASE
Two-Phase Commit	Orchestration	Consistent	Less available	ACID

Table 2.1: Distributed transaction patterns

(41). Typically, queries that require joins are highly optimized inside of DBMSs. Therefore, performing them manually on data from different services implies steep performance penalties. A common solution to this problem is the Command Query Responsibility Segregation (CQRS) pattern. With CRQS, so-called views (read only copies) of tables are prematurely colocated at microservices that require them to speed up join operations (41). Only the microservice owning the table can write to it. Writes are streamed to the views via events using the publish/subscribe pattern, making them eventually consistent (41).

3

Problem Statement and Approach

We have identified three challenges (C) with data management in MSAs (Chapter 2).

C1 Distributed transactions — Transactions across services are challenging and require ad-hoc solutions or frameworks that require the developer to write repetitive code.

C2 Data consistency — Relations that spawn bounded-contexts of microservices require developers to implement ad-hoc solutions to keep data consistent.

C3 Data joining — Developers manage data joins from multiple services manually using some ad-hoc solutions or patterns like CQRS, requiring them to write repetitive code.

The goal of this thesis is to propose and implement an easy-to-use solution towards solving these challenges named DMan, by following the design, implement and prototyping research methodology (19, 25, 38). The requirements of DMan are partly derived from Chapter 2, the other source of requirements being a specific use-case that uses an MSA.

The use case (**UC**) being a platform with so-called micro-frontends which run in the browser, e.g. using the Module Federation Framework of the Webpack community, and a back-end that provides REST services to the micro-frontend. Such a micro-frontend can be bundled with back-end component(s) and, as a whole, is considered as a plugin. The platform features fat clients (as a substantial amount of the code will run in the browser of the user) and thin servers (restricted to data storage and business logic). Plugins should be hot-pluggable, e.g. it should be possible to (un)install a plugin without restarting the platform. DMan will be used inside the back-end microservices required for the plugins on this platform.

We now draft three requirements (R) to which DMan must adhere.

R1 Provide an API for developers that enables distributed transactions using both Saga or 2PC as underlying technique within an MSA (**C1**).

R2 Automatically guarantee data consistency between bounded-contexts (**C2**).

R3 Manage joins between relations that spawn different bounded-contexts generically and efficiently (**C3**).

R4 Services can be added or removed dynamically from the MSA (hot-pluggable) (**UC**).

R5 Services should not trust code from other services unless explicitly accepted (**UC**).

DMan can be deployed within a MSA and is used by developers to support the development of microservices. Both the developers and the MSA in which DMan is used, impose a set of technical requirements (TR).

TR1 The APIs provided by DMan should be tractable and not require repetitive code (**C1**, **C2**, **C3**).

TR2 The services with the MSA should remain loosely-coupled and remain agnostic to the specifics of other services (**UC**).

TR3 The services within the MSA should be isolated from each other (other services cannot be trusted) (**UC**).

The implementation will be validated (V) in Chapter 6 by using the following techniques:

V1 Implementing end-to-end tests that test core use-cases in different MSAs for **R1**, **R2** and **R3** (22).

V2 Validate **R1**, **R2**, **R3**, **R4**, and **R5** by debugging and prototyping.

V3 Validate **TR1** by illustrating example use cases of the DMan API and verifying that the code required by the API is both tractable and non-repeating.

TR2, **TR3** cannot be validated explicitly, since they are implemented during the design of the system (Chapter 4).

4

Design of DMan: A Generic Data Management Solution

In this chapter, we describe every major design decision, along with its rationale, made during the design phase of DMan. The design decisions are made with the goal of satisfying all the requirements defined in Chapter 3. Chapter 2 described a model in which microservices are dissected into four layers: (1) runtime, (2) communication, (3) coordination and (4) logic. Each of these layers require certain design decisions to be made, and will be used as a guideline for this chapter.

4.1 Network Architecture — Communication

The scalability of our solution is important since our system will be deployed in microservice architectures of arbitrary size (**R4**). Therefore, we have opted for a decentralized network architecture, in which DMan is co-located on the nodes with business microservices. The instances of DMan will form a peer-to-peer system, in which all nodes are equal and the interaction between them is symmetric: meaning that each node will act both as a client and a server at the same time (50). This allows DMan to both naturally scale with the MSA in which it is deployed and maintain a graph of relations that are relevant to each service locally.

4.2 Intercepting Operations — Logic

The operations that flow between the service layers can be classified to be a create, read, update or delete operations (CRUD) (17). The microservice gets a request which flows from the presentation layer to the infrastructure layer and back to return the desired

results to the caller (Figure 4.2). Therefore, each request eventually gets executed by the repositories which performs the appropriate operations on the database. DMan introduces an additional service type within the domain layer named data services, which sit between the repositories and the business services. Originally repository r is responsible for the CRUD operations o invoked by business service b. The newly introduced data service d sits between r and b and translates the operations o to operations on domain model dm and vice versa using mappers. Data services funnels all CRUD operations through a central point named the operation manager. The operation manager provides an interceptor API that components can use to intercept operations. In order to satisfy **R2** and **R3**, DMan needs to be able to intercept these operations and modify their input and output parameters dynamically such that constraints imposed by the relational model (Section 4.3) can be applied to them. DMan utilizes a commonly used pattern for intercepting operations, called the interceptor pattern, where interceptors realize predefined callback interfaces to implement service extensions in a specific manner (43).

4.3 Relational Model Coordination

Each business microservice operates on a subset of objects (or domain models) that are within its bounded-context. These objects have relations with other models that are either inside or outside of its respective bounded-context. We need to capture both of these relation types into a relational model to be able to satisfy **R2** and **R3**. We have chosen to model the service boundary crossing relations as a Directed Acyclic Graph (DAG), since services are hot-pluggable (**R4**). This means that additions to the MSA will be dependent on previously defined services, making the MSA acyclic. To illustrate, in Figure 4.1, we see three services (marked by S) each with their respective domain models (marked by M) forming such a DAG. Directed edges represent boundary crossing relations, while undirected edges represent relationships between models that are inside of the bounded-context of a microservice. Undirected edges and their relevant models are managed by the persistence or infrastructure layer, the directed edges will be managed by DMan. Each edge represents one of the following relationship types, (1) one-to-one (1:1), (2) one-to-many (1:*), (3) many-to-one (*:1) and (4) many-to-many (*:*). The direction of the edges that spawns service boundaries dictate in what service potential join tables reside (21, 29, 33) (i.e. traditional relational databases require *:1 and *:* relation types to be captured using additional tables).

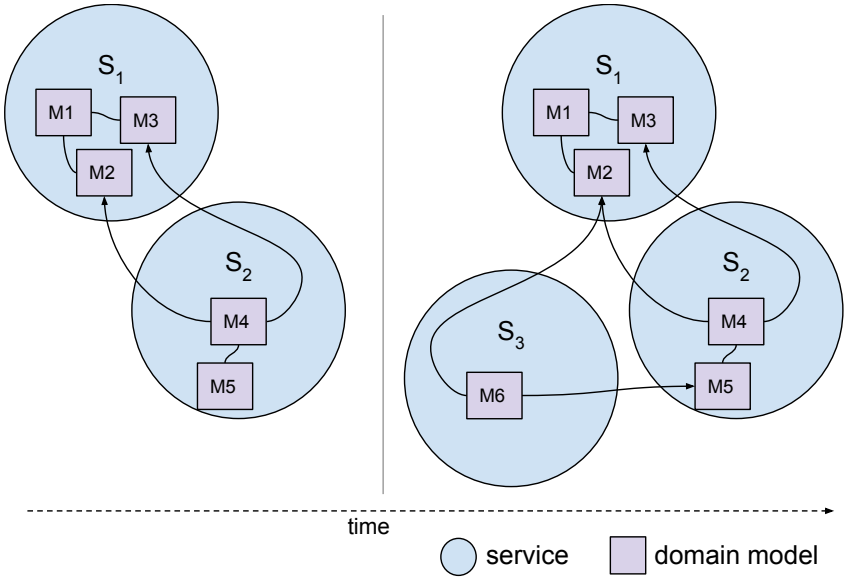

Figure 4.1: The services and the models that exist within them over time

4.4 Definition of the Relational Model Coordination

The implementation of infrastructure layer is typically a task of Object Relational Mapping (ORM) frameworks. In the case of Java, these frameworks provide APIs that facilitate the mapping of Plain Old Java Objects (POJOs or entities) and Data Access Objects (DOAs or repositories) to database scheme metadata and queries respectively (54). Additional metadata on the entities is required in order for ORMs to make the translation to database schemas, this is typically done using annotations on the properties defined on the entities (29). The entities therefore contain all the relational information that is relevant for the bounded-context of the respective microservice. DMan will provide an API similar to that of typical ORMs that allow developers to define the inter-microservice relational metadata on domain models in a simple but intuitive manner (**TR1**), such that a DAG like Figure 4.1 can be constructed. To comply with **TR2**, the microservices need to be agnostic to the fine details of the domain models that lay outside their bounded-contexts. Therefore, minimal information is used to capture such relations (i.e. only the name of the service and the domain model).

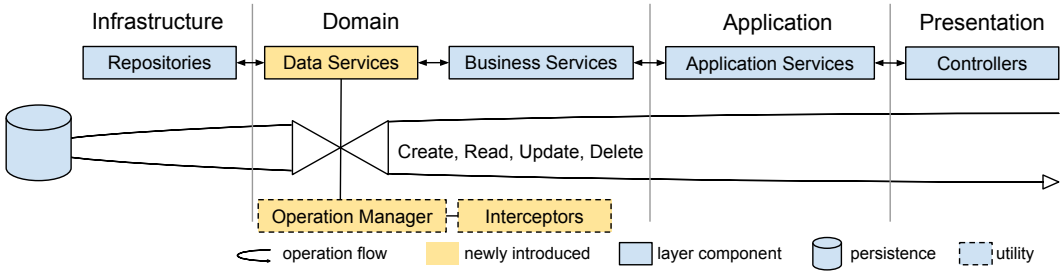

Figure 4.2: The operations that exist within microservices and how they propagate through the service layer

4.5 Distributed Transactions Coordination

We define a distributed transaction to be a collection of transaction operations that is executed sequentially (Section 2.1.2). The transaction operation can either be invoked locally (LO) or remotely (RO) and performs CRUD operations on a subset of the models (M) that are available inside the service (S) in which the operation is executed (Figure 4.3). A transaction is successful if the execution of all operations are successful. If one of the transaction operations in the chain fails, the transaction is rolled back in its entirety, otherwise, all the changes are commited. The interaction model of distributed transactions are generally either orchestration or choreography based (16). Initially, we choose to implement an orchestration interaction model for DMan, since this does not require a separate event broker service in the MSA. The transaction orchestrator is the service that launches the transaction (S_3 in the case of Figure 4.3). As per **R1**, a transaction can have a mixture of 2PC and Saga operations. Saga operations are idempotent by definition and (if necessary) can define a custom rollback or compensating transaction to ensure eventual consistency (17), while 2PC operations utilize the underlying DBMS for transactions. Transactions that are initiated inside another transaction are called nested transactions. The finalization or rollback of nested transactions is deferred to when the root transaction (i.e. the parent transaction that contains all the nested transactions) completes. This allows nested transactions to be rolled back when operations of the parent fail. This is necessary because, for example, consistency updates between services can trigger new transactions (also illustrated in Figure 4.3).

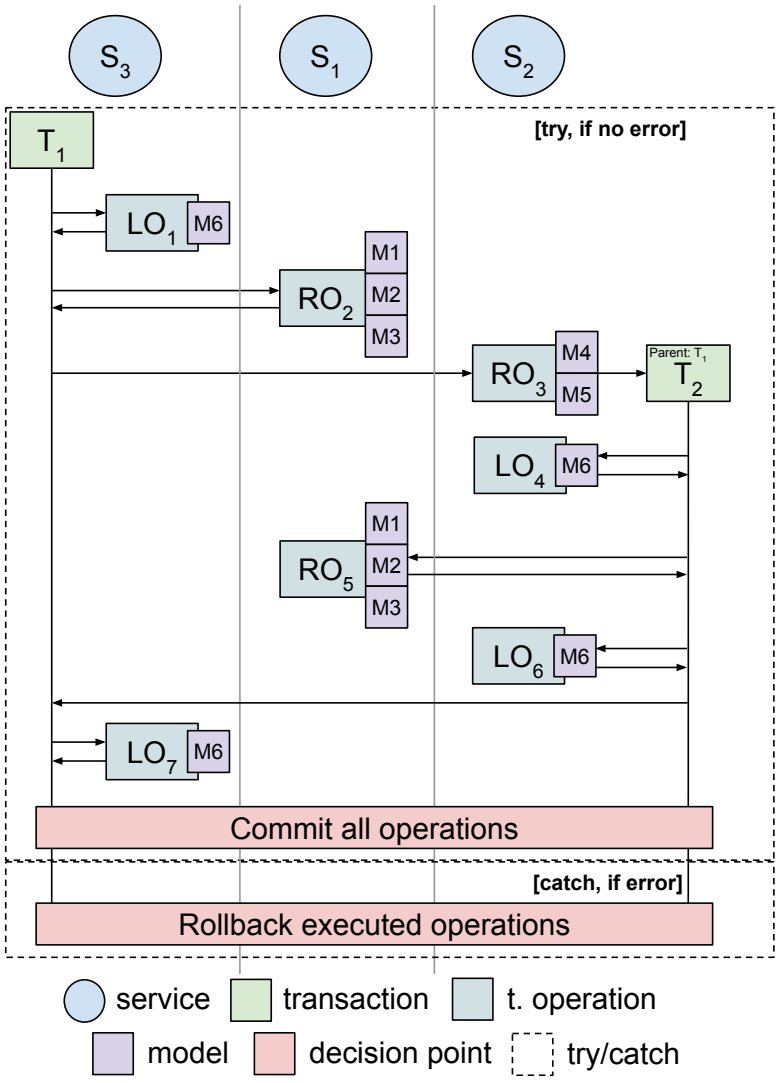

Figure 4.3: The transactions with their respective operations and how they are executed in different service contexts. Note that transaction operation is shortened to t. operation.

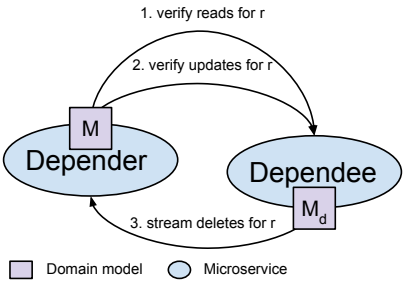

Figure 4.4: How consistency is guaranteed between depender and dependee services

4.6 Security of Custom Distributed Transactions Security

Transaction operations that are defined within DMan are known to all the service in the architecture. Therefore, only the parameters of each transaction operation need to be transferred between the services. Therefore, code that is executed for **C2** and **C3** resides within our trusted library and do not require explicit acceptance (**R5**). However, when developers define transaction operations or views, code must be transferred along with the parameters. DMan allows services to explicitly configure what service can execute custom transaction operations using a simple regex based security policy (**R5**).

4.7 Strategies Coordination

Strategies will implement the logic required for inter-microservice data consistency and joins (**R2**, **R3**). They will utilize the relational model, data services, interceptors, and transactions.

4.7.1 Consistency

The directed relations between services imply that there is a depender and dependee service. For any pair of dependee and depender services in the DAG, there is a model M that is dependent on M_d through relationship r with some cardinality (Figure 4.4). In order to satisfy **R2**, any reference made by a relation on any domain model should be valid. The dependee and depender can transition to both being available from two different states. The first state being that the dependee is unavailable and the depender is available, in which case the dependee cannot respond to any CRUD operations involving model M due to its dependee not being available. The second state being that the dependee is available

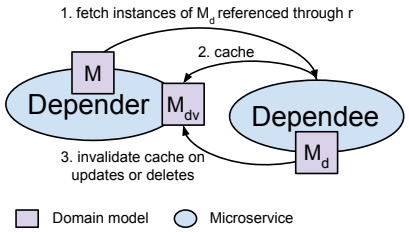

Figure 4.5: How joining is performed between depender and dependee services

and the depender is unavailable, in which case the dependee continues its normal operation. During normal operation, the dependee can delete models on which the depender is dependent through relation r. When both the depender and dependee are online, such operations are streamed to the depender, allowing it to update its dependent models accordingly. However, to validate that all the instances of M still point to valid instances of M_d (i.e. instances of M_d could have been deleted while the depender was offline), the depender needs to verify every instance of M once when it is read. Naturally, this verification also occurs when instances of M are updated to point to different instances of M_d.

There are different integrity policies to resolve inconsistencies, but not all are viable to implement in DMan. For example, a common policy is to restrict the update or deletion of a model when it would violate integrity (35). This policy however cannot be implemented in DMan, due to their not being a guarantee on the availability of the depender service. Viable policies are to delete or null violating models or model fields respectively (35). The developer can configure what consistency policy to use for what model field while defining the relational model.

4.7.2 Joining

For any pair of dependee and depender services in the DAG, there is a model M that is dependent on M_d through relationship r with some cardinality (Figure 4.5). Instances of M can therefore refer to instances of M_d. Fetching these instances from the dependee service whenever they are needed is ineffective (31). To prevent unnecessary reads, DMan will implement the CQRS pattern generically leveraging the relational model to satisfy **R3**. The referenced instances of M_d through relationship r are projected to a view M_{dv} and cached on the depender. This cache is invalidated by the dependee when any updates or deletes are made to the referenced instances of M_d. The view M_{dv} is defined on the depender and only defines the required fields of M_d (**TR2**).

5

Implementation of DMan: A Generic Data Management Solution

In this chapter we present the technical implementation of DMan based on the design specified in Chapter 4. This chapter will expose non-trivial parts of our implementation with the goal of helping developers understand the inner workings of DMan on a technical level. Therefore, a basic understanding of programming concepts are expected from the reader. A common type of diagram to illustrate technical details are class diagrams (39). In summary, class diagrams describe the structure of a system with the use of classes, their properties, methods, and the relationships among them (39). Class, custom, and pseudocode diagrams are used in this chapter to enrich and help visualize complex implementation parts while remaining true to the naming conventions used in the actual code. Implementation details are omitted from diagrams when they do not contribute to the broader concept that is being described. Similar to Chapter 4, the layers in defined in Chapter 2 are used as a guideline for this chapter. The complete code of the implementation can be found on GitHub[1].

5.1 Overview of the Layers

The classes of internal systems in DMan can be grouped by the layers defined in Chapter 2 (Figure 5.1). The runtime layer is omitted from this figure since it cannot be expressed using classes. The communication layer allows the services to communication with external services by exposing contracts that specify callable remote methods. The logic layer contains the data services that sit between the repositories and business services defined

[1] `https://github.com/ThijmenKurk/thesis-code`

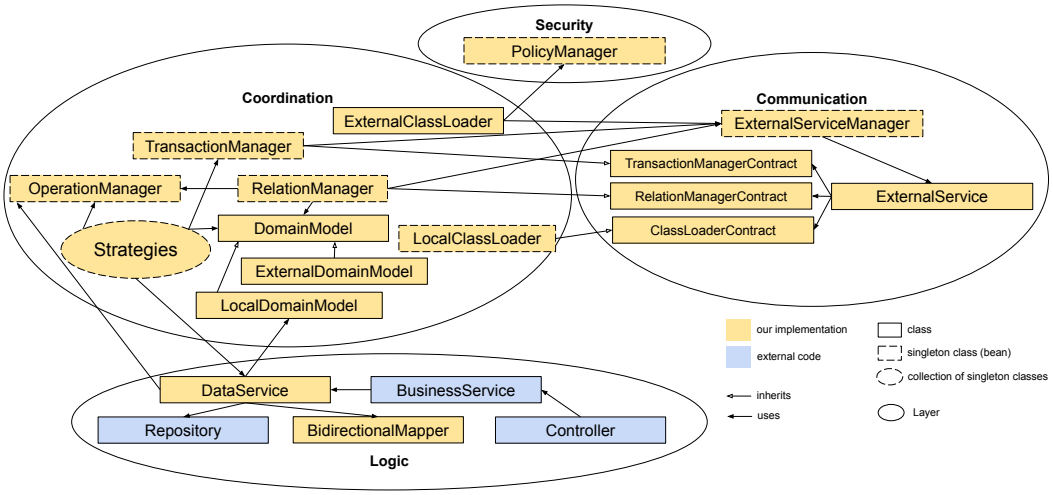

Figure 5.1: The classes of DMan and relations between them grouped by the microservice layers defined in Chapter 2. Note that the uses relation is transitive. The definition of a bean is explained in Section 5.4.2

.

by the microservice developers (4.2). Additionally, the layer contains the mappers that convert entity instances to domain model instances and vice versa. The coordination layer is the core of DMan and is responsible for resolving the data management challenges (**C1**, **C2**, **C3**) and has five internal systems represented by singleton classes.

1. *TransactionManager* — is the transaction orchestrator and keeps track of transaction specific information.

2. *RelationManager* — performs the exchange of relational metadata between services and maintains available relations of local and external domain models (DAG).

3. *OperationManager* — coordinates all the operations that occur within the service (i.e. CRUD operations and relational model update operations).

4. *LocalClassLoader* — facilitates access to classes defined outside DMan to external services.

5. *Strategies* — leverage the *OperationManager*, the *TransactionManager* and the relational model defined by DMan to implement data management requirements based on annotations made by the developer.

Finally, the security layer is responsible for allowing or denying code loaded from external services by the *ExternalClassLoader* based on some policy.

5.2 Language and Framework Runtime

For the implementation of DMan we use Java since it is a widely used and accepted language in enterprise development (10) and because it is used by already existing components of the **UC**. There are many frameworks available in Java that are used to implement microservices (11). We considered two frameworks for the implementation of DMan, namely the OSGi Framework[1] and the Spring Cloud Framework[2]. Ultimately, we choose the Spring Cloud Framework for three reasons: (1) we found that Spring Cloud has a more active community and therefore naturally has more information available about how to use the framework[3], (2) Spring Cloud is widely adopted by the industry (56) and (3) OSGi is designed to run on a single machine (5, 37) providing relatively weak isolation compared to Spring Cloud, which runs microservices in separate processes, containers or machines (17) (**TR3**).

5.3 Execution Form and Platform Runtime

Since DMan will form a peer-to-peer decentralized system (Section 4.1), every microservice needs to run an instance of DMan. We could utilize the sidecar pattern, in which a separate process running DMan is executed next to each microservice. This would make DMan language agnostic. However, it introduces communication overhead and create interoperability issues (i.e. strategies will need to be able to interface with the persistence layer of the microservice directly for 2PC). Therefore, we choose to implement DMan as a library that can be imported by each microservice. The platform of execution is out of scope from DMan since it depends on the requirements of the MSA. However, to facilitate strong isolation between services, DMan is implemented with the assumption that services are executed on or in different machines or containers (**TR3**), making communication only possible through explicit endpoints.

5.4 Spring Cloud Runtime

As stated on the Spring Cloud GitHub, “Spring Cloud provides tools for developers to quickly build some of the common patterns in distributed systems (e.g. configuration management, service discovery, circuit breakers, intelligent routing, micro-proxy, control bus, one-time tokens, global locks, leadership election, distributed sessions, cluster state)”.

[1] `https://github.com/osgi`
[2] `https://github.com/spring-cloud`
[3] `https://trends.google.com/trends/explore?date=today%205-y&q=OSGi,Spring%20Cloud`

Spring Cloud is part of the Spring Framework and uses other components of the Spring Framework internally (9). Spring Boot creates and autoconfigures the aforementioned tools and is used to create stand-alone Spring applications.

5.4.1 Auto-configuration

Spring uses a process called auto-configuration to non-invasively configure Spring applications based on added dependencies (8, 53). For example, if you add the service discovery dependency to a Spring Cloud project, this dependency is automatically configured through this process. DMan leverages this process by providing auto-configurations which automatically initialize the library when it is added as a dependency to a service (**TR1**). The auto-configurations provided by DMan use class path scanning to find classes, by certain criteria, that are defined by the developer outside the library.

5.4.2 Dependency Injection

Dependency injection (DI) is a common software engineering pattern (6) that is a fundamental aspect of Spring (9, 53). DI is a form of Inversion of Control (IoC) (6) and aims to separate the concerns of creating objects and using them with the goal being loosely-coupled components. Within the Spring Framework, the IoC container is represented by the `ApplicationContext` interface. This interface is responsible for creating, configuring and managing the lifecycle of objects that are known as beans inside of Spring. Beans are configured during the auto-configuration phase and are injected into components through a process called autowiring. Autowiring can inject dependencies of components through class fields or constructor arguments (53).

5.4.3 Aspect-Oriented Programming

Aspect-Oriented Programming (AOP) is an extension of Object-Oriented Programming (OOP) which aims to increase modularity by allowing separation of cross-cutting concerns (8). AOP allows, for example, transaction management and security logic, to be developed separately and mingled with functionality at compile time (8). DMan uses AOP proxies to dynamically implement data services and external service defined by the developer (6) (**TR1**).

Annotation(s)	Description
@Builder	`https://projectlombok.org/features/Builder`
@NoArgsConstructor, *@AllArgsConstructor*, *@RequiredArgsConstructor*	`https://projectlombok.org/features/Constructor`
@Getter, *Setter*	`https://projectlombok.org/features/GetterSetter`

Table 5.1: Lombok annotations used by DMan

5.5 Project Lombok Runtime

DMan uses Project Lombok to automatically insert boilerplate code, reducing the amount of repetitive code while increasing code readability[1]. Lombok provides annotations like *@AllArgsConstructor*, *@Getter* and *@Builder* to automatically implement constructors with all fields as parameters, getters for defined fields and builder pattern for classes respectively (34). A brief overview of annotations used by DMan is given in Table 5.5.

5.6 Service Discovery Communication

DMan uses the well-known and battle tested Eureka library from Netflix (26) for service discovery since it is natively supported by Spring Cloud[2]. Microservices can register themselves to the Eureka service discovery server and allow other microservices to find them by name. The service discovery server will remember previously registered microservices, such that microservices can wait on dependencies if they are registered or throw an error if they are not. The primary roll of this utility service is to translate service names into service locations and make communication possible (16).

5.7 Feign Clients Communication

DMan uses feign clients for communication because it is natively supported by Spring Cloud[3] and trivial to use. Feign clients are declarative web service clients and can be used by annotating interface methods with request metadata (9). These interfaces, also known as contracts, get implemented dynamically such that the interface methods translate to calls to remote endpoints automatically. The feign clients are configured to use the Hyptertext Transfer Protocol (HTTP) as the underlying transportation protocol. DMan

[1] `https://github.com/projectlombok/lombok`
[2] `https://spring.io/guides/gs/service-registration-and-discovery/`
[3] `https://cloud.spring.io/spring-cloud-openfeign/reference/html/`

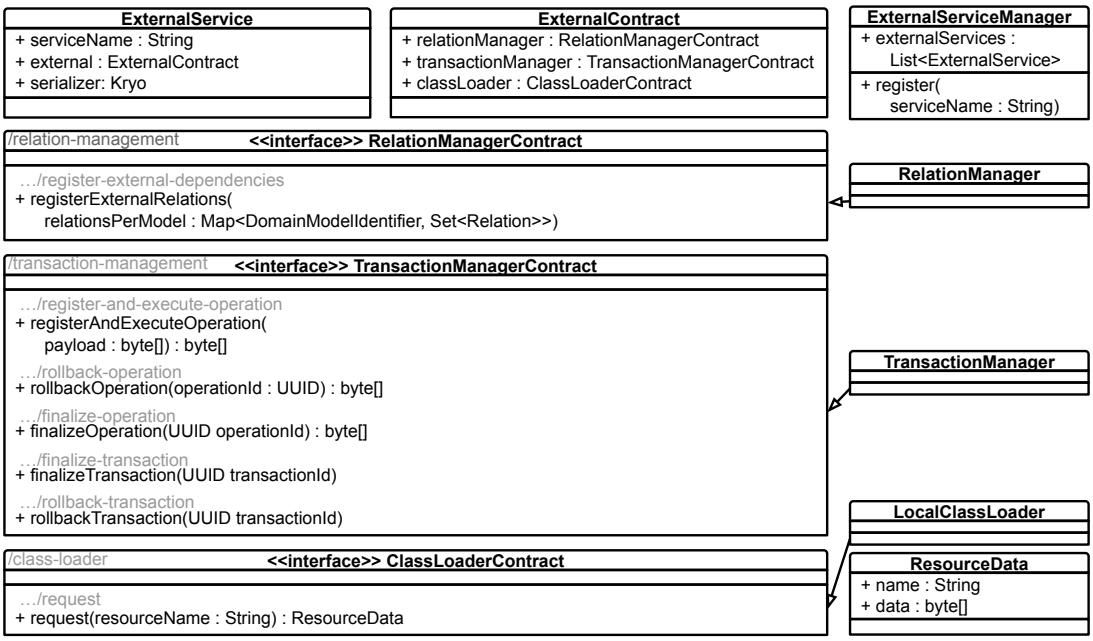

Figure 5.2: Class diagram of the communication layer. The gray text represents the path of the HTTP endpoints made available by DMan.

```
@ImplementExternalService(serviceName = "EMPLOYEE")
public interface EmployeeService extends ExternalService {
}
```

Figure 5.3: An example external service implementation.

hosts the required HTTP endpoints by using the Spring Cloud HTTP server (9). To guarantee that the method signatures of the caller and callee of the endpoint match, the callee implementation of contract simply inherits the contract interface. Input and output parameters of these methods are serialized to JavaScript Object Notation (JSON). The HTTP protocol in combination with JSON was chosen since it widely accepted and used by the industry (56). All the calls between services are performed synchronously to improve the debugability of the entire system.

5.8 External Services Communication

The `ExternalServiceManager` is responsible for providing access to external services to other components (Figure 5.2) and therefore keeps a record of all relevant external services. An `ExternalService` is either defined explicitly by the developer, such that it

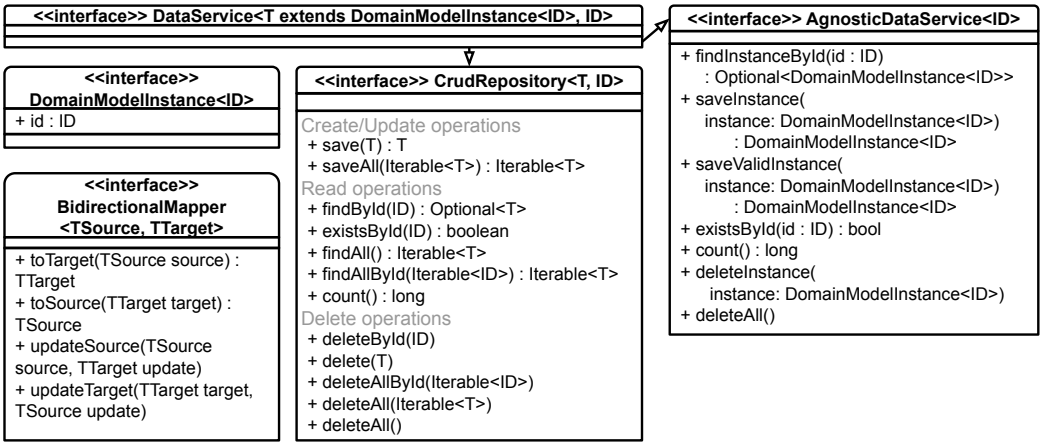

Figure 5.4: Class diagram of the logic layer.

```
@ImplementDataService(
  repositoryClass = DepartmentRepository.class,
  mapperClass = DepartmentMapper.class)
public interface DepartmentDataService extends DataService<
   DepartmentDomainModel, Long> {
}
```

Figure 5.5: An example data service implementation.

can be used while defining relations on domain model instance classes, or automatically inferred based on relational metadata. Either way, the `ExternalService` is created by the `ExternalServiceManager` using the `register` method. Explicitly defining an `ExternalService` can be done by annotating an interface which inherits the `ExternalService` interface with the `@ImplementExternalService` annotation (Figure 5.3). During the auto-configuration process, these annotated interfaces are found using class path scanning and registered to the `ExternalServiceManager`. Explicitly defining the external service results in better code navigability and autocomplete when developers create new relations (Figure 5.6).

5.9 Data Services Logic

DMan proxies the operations necessary for the functionality of the microservice by inserting data services between the repositories and business services. More specifically, a data service serves as a proxy for CRUD operations between repositories and business services

```
@Builder @NoArgsConstructor @Getter @Setter
public class DepartmentDomainModel extends DomainModelInstance
    <Long> {
     private Long id;
     private String name;

     @ManyToOne(
       modelName = "PersonDomainModel",
       service = EmployeeService.class)
     private Long managerId;
}
```

Figure 5.6: An example domain model instance class.

for a specific domain model instance that bidirectionally maps to a specific entity instance. Therefore, every `DataService` inherits the interface `CrudRepository`, since this interface is commonly used by repositories in Spring Cloud (9). This allows business services to interface with data services similarly as to how they would interface with repositories. Domain models, and by extension entities, need to implement the `DomainModelInstance` to make them generically identifiable by DMan. Additionally, every `DataService` inherits the `AgnosticDataService` interface which simply abstracts away the specificity of domain models to allow components to generically interface with them.

Data services need to translate entity instances to domain model instances and vice versa, this functionality is abstracted to the `BidirectionalMapper`. The `BidirectionalMapper` can be implemented manually by the developer for each domain model instance. Alternatively, DMan supports MapStruct[1] and ModelMapper[2] to generate mapper classes automatically (**TR1**). A class diagram of the aforementioned classes can be found in Figure 5.4.

Developers define data services by annotating a interface that inherits the `DataService` interface with the `@ImplementDataService` annotation. Using AOP proxies, this interface is dynamically implemented during auto-configuration. The `@ImplementDataService` allows the developer to specify both the repository from which entity instances are retrieved and what mapper is used by the data service (if no mapper class is specified, ModelMapper is used). An example of a data service defined by a developer can be found in (Figure 5.5).

[1] `https://github.com/mapstruct/mapstruct`
[2] `https://github.com/modelmapper/modelmapper`

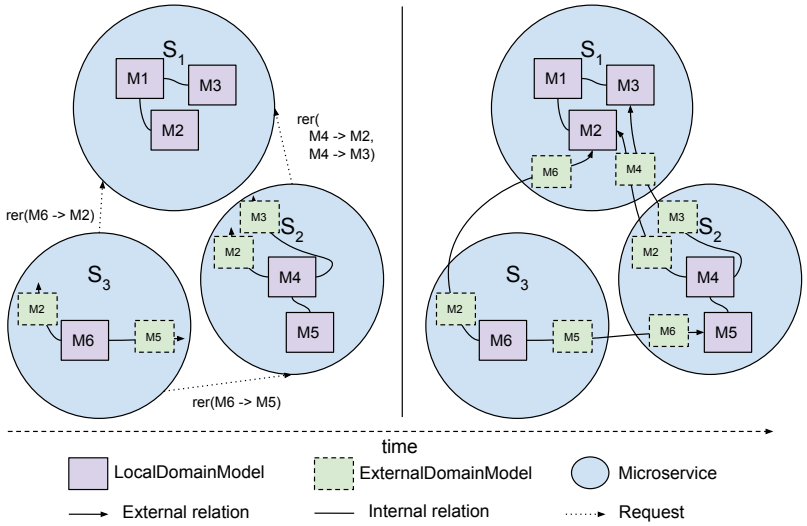

Figure 5.7: Visualization of how relations are propagated to external services

5.10 Relational Model — Coordination

Domain model instances are annotated with relational metadata similarly to JPA entities (Figure 5.6) and inherit the `DomainModelInstance` interface. Developers can use the `@OneToOne`, `OneToMany`, `@ManyToOne` and `@ManyToMany` annotations on the fields of domain model instance classes to specify relational metadata. A `DomainModel` is the materialization of the metadata defined by these annotations in the form of relations. The difference between domain models and domain model instances is that the former contains metadata about the domain model instance while the later contains the actual data as defined by the domain model instance class. Any `DomainModel` can be identified by a `DomainModelIdentifier`. A `DomainModelIdentifier` equals to the name of the owning service and the domain model name. There are two types of `DomainModel`. The first type is a `LocalDomainModel`, which represents a domain model that is defined locally. A `LocalDomainModel` contains service-specific class information (`dataService` and `entityClass`) and represents a domain model instance class. The second type is an `ExternalDomainModel`, which represents a domain model that lays outside the bounded-context of the local service.

During auto-configuration, domain models are materialized from the annotations on domain model instances by the `RelationManager` (Figure 5.7, left segment). After auto-configuration, all external relations are forwarded to the services they reference using

the `registerExternalRelations` (`rer`) method on the `RelationManagerContract` (Figure 5.2). This method purges all relations and domain models that reference the calling service to prevent duplication. Afterwards it adds the relationships to its local domain models and creates the needed external domain models (Figure 5.7, right segment). This provides each service with a complete view of relevant domain models and their relationships. The configuration of the strategies of each `Relation` is captured by `source` on `RelationSource` class by the `RelationManager` during the materialization of metadata (Figure 5.8). Additionally, entity specific properties are captured on `properties` (e.g. is the field nullable in the database or not). The classes mentioned in this section can be found in the upper segment of Figure 5.8.

5.11 Intercetable Operations Coordination

DMan uses operations for actions that are important to strategies. Operations are interceptable modifiable actions. There are two primary classes of operations, (1) relation operations and (2) data operations. Relation operations are executed by the `RelationManager` and occur when relations are added or removed during auto-configuration or by calls to the `RelationManagerContract` from external services. Data operations are executed by data services and are categorized based the distiction made in the class diagram in Figure 5.4. The `OperationManager` runs each operation through a chain of interceptors. Interceptors are called everytime an operation is executed and can be registered dynamically. An interceptor can proceed with the operation by calling `proceed()` on the `OperationExecutionContext`. This function will either call the next interceptor in the interceptor chain or execute the operation when there are no interceptors left to call. An overview of classes relevant to operations is shown in the bottom segment of Figure 5.8.

5.12 Transactions Coordination

This section describes how transactions are implemented in DMan, the reader is advised to use Figure 5.9 complementary to reading. A `Transaction` is a sequence of transaction operations that is executed by the `TransactionManager` on the transaction orchestrator. A `TransactionOperation` has three stages of execution (`TransactionOperationStage`): (1) execute stage, (2) rollback stage, (3) finalize stage. The status of each stage and its corresponding log is explicitly captured by the `operationStatus` and `log` fields on `TransactionOperation` respectively, due to the possibility of a transaction operation being

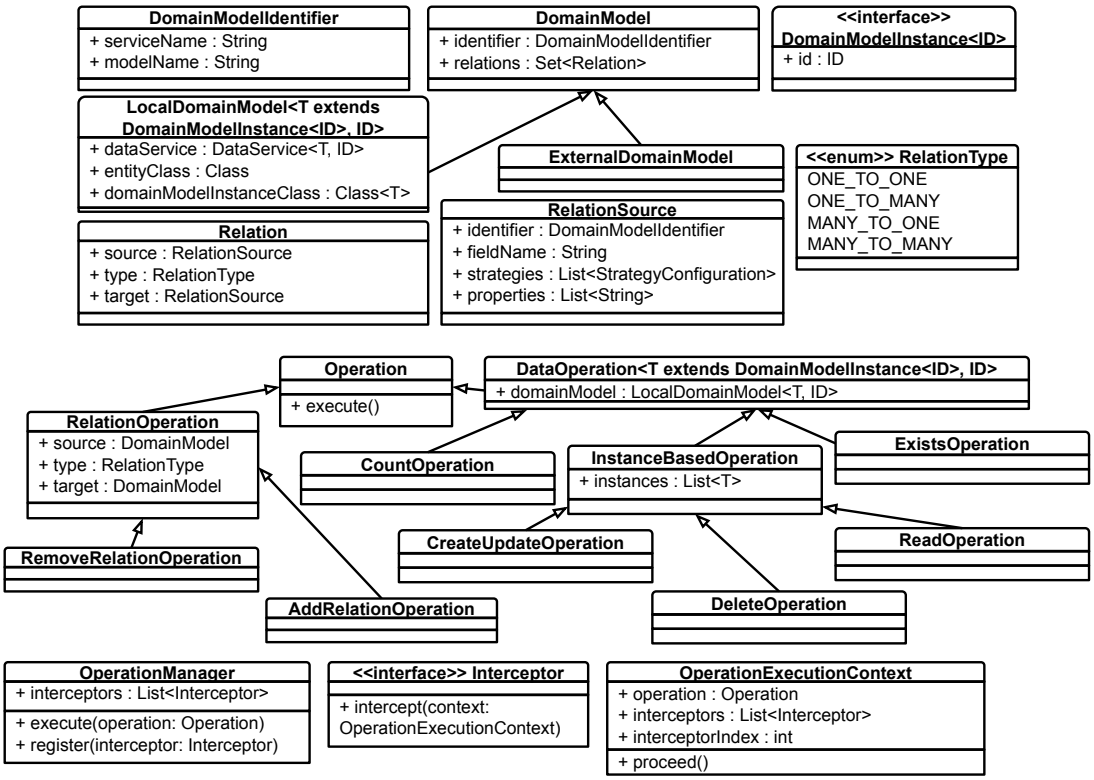

Figure 5.8: Class diagram of the *RelationManager*, relational metadata classes, *OperationManager* and available *Operation* classes.

executed remotely. The transaction manager invokes the `executeOperation()` on every operation in the `Transaction` during the execute stage. If no errors occurred for any transaction operation in the sequence, all transaction operations are finalized (finalize stage). Otherwise, all of the executed transaction operations are rolled back (rollback stage). This is done by the `TransactionManager` using the `finalizeOperation()` and `rollbackOperation()` methods on `TransactionOperation` respectively.

Transaction operations are autowired before executing, allowing transaction operations to access any defined data service, the `PlatformTransactionManager` (15, 33) and any other bean defined in the `ApplicationContext`. The `PlatformTransactionManager` is a bean provided by Spring and allows transaction operations to initiate database transactions to perform 2PC automatically (by overriding `useTwoPhaseCommit()`). Database transactions require transaction operations to be executed on the same thread. Otherwise, changes made by data services by previous operations are invisible to the next operation since changes made during a database transaction are thread-bound and only become final when they are committed. Therefore, the `TransactionManager` keeps a pool of threads (`availableThreads`) and assigns them to root transactions (using `executorMap`) to guarantee the same thread is used for all the operations in the transaction. The root transactions being the UUID of the initial transaction (i.e. the transaction that contains all the other transactions). Additionally, thread allocation allows the `TransactionManager` to manage a thread-local stack of transaction identifiers. This stack is queried when a new transaction is executed. If the stack is non-empty, the new transaction is added as a child to the topmost transaction on the stack, making it a nested transaction. Transaction identifiers are pushed to the thread-local stack at the start of the execute stage and popped from the stack when the execute stage is finished. The rollback or finalization of nested transactions is deferred to when all the operations of the root transaction finish using the `pending` map. When the root transaction reaches either the rollback or finalize stage, all pending child transactions are recursively rolled back or finalized. When not using 2PC, a developer typically defines idempotent execute, rollback and finalize operations by overriding functions manually to implement the Saga pattern.

A `TransactionOperation` can be executed locally (`LocalTransactionOperation`) or remotely on another service (`RemoteTransactionOperation`). When a transaction operation is executed remotely, the parameters (and potentially classes) used by such operation need to be sent to the executing service. DMan uses Kryo, as it is a well-known and fast serialization framework that supports field serialization, to translate a `TransactionOperation` to

bytes and vice versa[1] (57). The default Java serializer is slower and only supports classes that inherit the `Serializable` interface (57). Field serialization allows Kryo to serialize any Java object, including anonymous implementations of classes and the parameters they use. A `TransactionOperation` is implemented anonymously, since this allows any needed variables to be captured within the anonymous class, such that it can be serialized dynamically. Any field that is defined on the `TransactionOperation` or the implementation thereof, is transferred back and forth during the stages of execution. Fields that are marked with the `@Ignore` or `@Autowired` annotations are ignored for transfer. Fields defined on anonymous implementations can be extracted using the `extract` function on the `TransactionOperation` class. The classes defined by DMan are shared between the microservices and therefore only require parameter serialization. Anonymous transactions defined outside DMan by developers require classes to be dynamically loaded on the remote service during deserialization. This is done by implementing a custom class loader (`ExternalClassLoader`), which uses the `LocalClassLoader` of the service invoking the transaction through the `ClassLoaderContract`.

The transaction orchestrator is agnostic to the domain models defined on the remote services. Anonymous transaction operations can interface with the local service using the `LocalService` class, which is available to be autowired before executing every stage of the operation. The `LocalService` class allows transaction operations to interface with data services without knowing the domain model instance or data service class with the method `getAgnosticDataService` by using the `AgnosticDataService` interface (Figure 5.4). An `AgnosticDataService` treats the domain model instance of the underlying `DataService` like a `DomainModelInstance`, making it generic but still allowing reflection to be used on the instances.

Developers can define loosely-coupled domain model views that reflect a domain model on a remote service to directly interface with fields on it. The `ProxyDataService` acquired via the `getProxyDataService` method on `LocalService`, dynamically maps these views onto the domain model instance using ModelMapper at runtime. A developer on the transaction orchestrator simply defines a view by creating a class that inherits the `DomainModelInstanceView` interface and is annotated with the `@View` annotation. This class should contain the fields of interest for the transaction. The `@View` annotation requires the developer to specify the domain model name to which the view maps on the remote service.

[1] `https://github.com/EsotericSoftware/kryo`

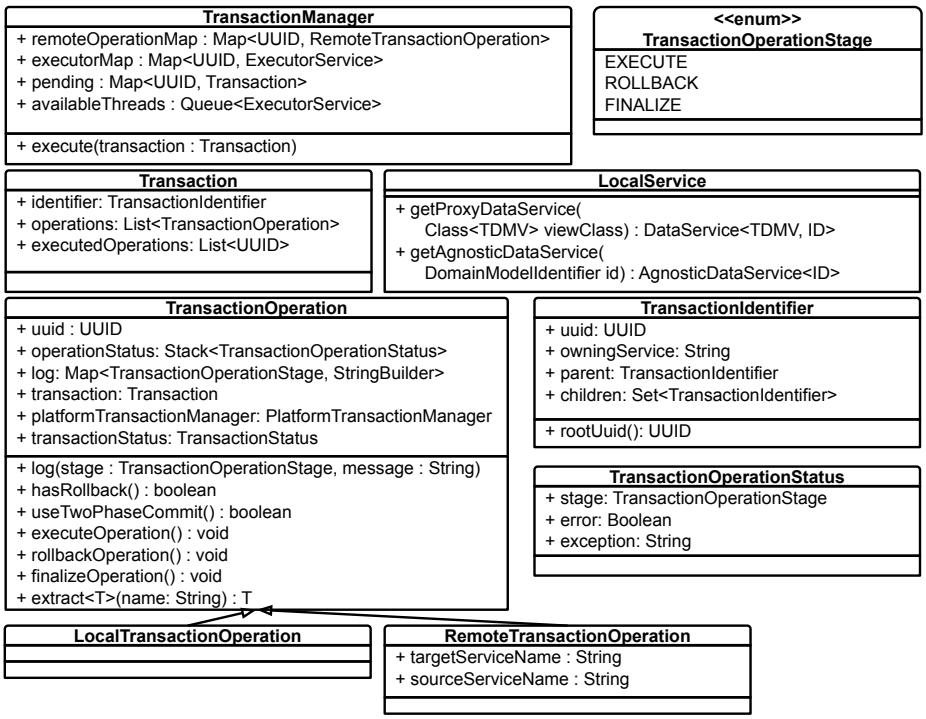

Figure 5.9: Class diagram of the $TransactionManager$, relational metadata classes, $OperationManager$ and available $Operation$ classes. Note that the methods of the $TransactionManagerContract$ defined in Figure 5.2 are omitted on purpose. $TDMV$ stands for $TDomainModelView$.

An example of a very basic transaction can be found in Figure 5.10. This transaction is executed in the context of two microservices, `EmployeeService` and `DepartmentService` . The `EmployeeService` knows nothing about the `DepartmentService`. Using DMan, `DepartmentService` can execute a transaction that contains a transaction operation with arbitrary operations on the domain models defined on `EmployeeService`.

5.13 Transaction Security — Security

Custom transaction operations require the transfer of executable classes to external services. DMan restricts the loading of remote code by `ExternalClassLoader` (Figure 5.1) through the `PolicyManager`. The `PolicyManager` has three different implementations.

1. `AcceptAllPolicyManager` — Accepts any external classes without question.
2. `RegexPolicyBasedManager` — Defines policy with a simple key-value map. The key being the service name and the value being a regex pattern to which the class must match.
3. `ApplicationConfigurationRegexBasedPolicyManager` — Inherits from 2 and applies the same policy but parses the key-value map from `"application.yml"`, which is a Spring configuration file, on the key `"security.allow-remote-transactions-from"`.

5.14 Strategies — Coordination

Strategies use the `OperationManager` to hook into the data services with the goal of modifying operations to implement a certain data management requirement. Strategies can be configured for fields on domain model instances using annotations.

5.14.1 Consistency Strategy

The consistency strategy leverages the relational model and transaction manager to implement data consistency across bounded-contexts. The strategy intercepts the create, read and update operations on the depender service and the delete operation on the dependee service. The depender service needs to verify every domain model instance after startup once, after it is added to a cache. The cache contains the ids of the domain models that

```
@View(domainModelName = "PersonDomainModel") @Getter @Setter
public class PersonView
  implements DomainModelInstanceView<Long> {
    Long id; String lastName;
}
...
var personId = (long) 1;
var transaction = new Transaction(List.of(
    new RemoteTransactionOperation(EmployeeService.class) {
      @Autowired
      LocalService localService;
      @Override
      public boolean useTwoPhaseCommit() { return true; }
      String lastName;
      @Override
      public void executeOperation() {
          var ds = localService.
            getProxyDataService(
                  PersonView.class
            );
          var personView = ds.findById(personId);
          personView.ifPresent(m -> {
            lastName = m.getLastName() });
          ...
      }
  },
  new LocalTransactionOperation() {
      @Autowired
      DepartmentDataService ds;
      @Override
      public boolean useTwoPhaseCommit() { return true; }
      @Override
      protected void executeOperation() {
          var lastName = (String) this.getTransaction().
            getLastExecutedOperation().extract(
                  "lastName"
            );
            ...
      }
  }
));
transactionManager.execute(transaction);
```

Figure 5.10: An example transaction. Note how variables defined outside of the scope of the anonymous definition are captured by the serializer dynamically.

have already been verified. This strategy uses 2PC transaction operations such that possible violation of constraints imposed by the underlying DBMS get enforced properly. We now discuss the depender side of the strategy using pseudocode (Figure 5.11).

The function `onCreateUpdateOrRead` is called by the `OperationManager` everytime a create, update or read operation occurs by any data service. The strategy only needs to operate when there is a configuration for the consistency strategy on a relation. The operation is executed if it is a read operation so that we can interact with the retrieved instances. Next, we select the instances the depender needs to verify with the dependee (`filterInstances`). `filterInstances` filters out instances that are not in the cache if `op` is a read operation, otherwise, all instances are checked. `checkNested` wraps any nested domain models defined as field on `instances` into separate read operations, such that these models also get verified. For example, domain model `ServiceA.ModelA` has a field that references `ServiceA.ModelB`. When `Service.ModelA` is retrieved by a data service, `Service.ModelB` is also retrieved indirectly by the underlying repository. Therefore, we must insert read operations manually since nested fields that reference domain model instances do not fire read operations automatically. `compute` retrieves all the relevant ids of each relation that needs consistency enforced (filtered by `filterRelations`), on the domain model for each instance in `instances`, and batches them together per dependee services in `checksPerService`. The `inverseMap` is used to translate the relevant ids back to the `instances`. Next, we start building the transaction by creating a list of operations (`tops`) that will be executed sequentially by the `TransactionManager`. Every set of ids identified by `compute` is checked on its respective dependee service. Non-existing ids are added to the `invalidIdsMap` of each respective transaction operation. The last transaction operation uses the `inverseMap` with the invalid ids returned by the previous operations to call the `patch` function (Figure 5.12). The function `patch` uses `patches` to resolves the consistency issues based on the policy on the violating field (either null or delete violating field or record). Afterwards, if necessary, the create or update operation is executed, since the instances provided on the operation are now verified. Finally, the transaction is executed. If successful, the results of the patch operation are propagated to the executing operation and valid instances are added to the cache (`process`). The dependee side of the consistency strategy is similar to the depender side but inverted. More specifically, the depender creates a transaction with operations that execute on services that have a relation with the domain model instances that are about to be deleted, and checks if they are referenced on the dependent domain models. Similarly to the `onCreateUpdateOrRead` function, `patch` is used to resolve any inconsistencies.

Integrity policies on fields are checked against the entity field properties at startup. For example, the consistency strategy throws an error if the integrity policy specifies that a field should be nulled when violating consistency constraints while the entity field is non-nullable,

5.14.2 Join Strategy

The developer simply inherits the local domain model to create a local domain model view in which external domain models can be joined using the relational model of the super class. Developers can proceed to define annotated fields that specify details about what underlying field to use. An example of this can be found in Figure 5.14. Now we describe the depender side of the join strategy using pseudocode (Figure 5.13).

The join strategy gets executed every time a data service executes a read operation. Internally, the join uses transaction operations without rollbacks since no create, update or delete operations invoked by this strategy. If this read operation uses an alternative view of the domain model it can get the fields that require external models using the `getJoinFields` function. External model views already in the cache are filtered out by `filterInstances`. Using the function `compute` from the consistency strategy, the relevant ids of external domain models are batched by dependee service such that they can be retrieved using the transaction built by `build`. `process` retrieves the external model views from the cache and applies them to the join fields on the instances. The dependee invalidates the model views in the depender service when they are updated or deleted.

```
OEC = OperationExecutionContext
DMI = DomainModelInstance
IBO = InstanceBasedOperation
func onCreateUpdateOrRead(ctx : OEC):
    op = ctx.operation
    if consistency not configured on any op.domainModel.relations then
        return
    if op.isRead then
        op.proceed()
    dmis = filterInstances(op.instances)
    checkNested(i)
    checksPerService, inverseMap =
     compute(dmis, op, filterRelations(op.domainModel.relations))
    t = build(checksPerService, inverseMap, ctx)
    transactionManager.execute(t)
    process(t, op)
end
func compute(dmis : DMI[], op : IBO, rs: Relation[]):
    dm = op.domainModel; ds = dm.dataService
    checksPerService = Map<String, Map<Relation, Set<ID>>>
    inverseMap = Map<Relation, Map<ID, Set<DMI>>>
    foreach r in rs do
        sN = r.target.identifier.serviceName
        foreach dmi in dmis do
            ids = get(dmi, r)
            if op.isCreateOrUpdate then
                ids = ids - get(ds.findById(dmi), r)
            checksPerService[sN][r].add(ids)
            foreach id in ids do
                inverseMap[r][id].add(dmi)
            end
        end
    end
    return checksPerService, inverseMap
end
func build(checksPerService : Map<...>, inverseMap : Map<...>, ctx : OEC):
    op = ctx.operation
    tops = []
    foreach (sN, checks) in checksPerService do
        top.add(remote transaction operation on sN that returns what ids in
         checks are invalid)
    end
    top.add(local transaction operation that uses patch on the invalid ids found
     by the executed operations and executes ctx.proceed() if op.isRead )
    return Transaction(tops)
end
```

Figure 5.11: Pseudocode of the algorithm behind enforcing consistency.

```
DMI = DomainModelInstance
DID = DomainModelIdentifier
RS = RelationSource
Action = { DELETE, UPDATE, NONE }
func patch(patches : Map<DID, Map<DMI, Map<RS, Set<ID>>>>):
    failed = Map<DID, Map<DMI, Exception>>
    deleted = Map<DID, Set<DMI>>
    actionPlan = Map<DID, Map<DMI, Action>>
    foreach (did, mp) in patches do
        foreach (inst, rp) : modelPatches do
            if actionPlan[did][inst] == DELETE then
                continue;
            mfd = false, mfu = false
            foreach (rs, ids) in rp do
                switch rs.strategies[consistency].policy do
                    case DELETE do
                        mfd = true
                    end
                    case UPDATE do
                        remove(inst, rs, ids)
                        mfu = true
                    end
                end
            end
            actionPlan[did][inst] = mfd ? DELETE : (mfu ? UPDATE : NONE)
        end
    end
    foreach (did, actions) in actionPlan do
        foreach (instance, action) in actions do
            if action == DELETE then
                dataServiceFor(did).delete(instance)
            else if action == UPDATE and not op.isCreateOrUpdate then
                dataserviceFor(did).saveValidInstance(instance)
        end
    end
end
```

Figure 5.12: Pseudocode of the `patch` function.

```
OEC = OperationExecutionContext
DMI = DomainModelInstance
IBO = InstanceBasedOperation
func onRead(ctx : OEC):
    op = ctx.operation
    op.proceed()
    joinFields = getJoinFields(op.domainModel)
    dmis = filterInstances(op.instances)
    rs = joinFields.keys()
    checksPerService, inverseMap = compute(dmis, op, rs)
    t = build(checksPerService, inverseMap, ctx, joinFields)
    transactionManager.execute(t)
    process(t, op, joinFields, inverseMap)
end
func build(checksPerService : Map<...>, inverseMap : Map<...>, ctx : OEC,
 joinFields : Map<Relation, Field>):
    op = ctx.operation
    tops = []
    foreach (sN, checks) in checksPerService do
        top.add(remote transaction operation on sN that retrieves instance view,
          mapped by proxy data services, that are required for the joinFields by
          the ids specified in checks)
    end
    top.add(local transaction operation that puts the retrieved model views in
     cache)
    return Transaction(tops)
end
```

Figure 5.13: Pseudocode of the algorithm behind joining.

```
@Builder @NoArgsConstructor @Getter @Setter
public class DepartmentDomainModel
    extends DomainModelInstanceBase<Long> {
  Long id;

  @ManyToOne(modelName = "PersonDomainModel",
    service = EmployeeService.class)
  @EnsureConsistency(onInconsistency = NULL_VIOLATING_RECORD
     )
  private Long contactId;
}
@Getter @Setter
@View(domainModelName = "PersonDomainModel")
public class PersonView
    implements DomainModelInstanceView<Long> {
  Long id; String lastName;
}
@Getter @Setter
public class DepartmentView
    extends DepartmentDomainModel {
  @Join(propertyName = "contactId")
  PersonView contact;
}
...
departmentDataService.findAll(DepartmentView.class);
```

Figure 5.14: An example of how to use the join strategy and how to define a consistency policy.

6

Validation of DMan

This chapter will describe the validation of the requirements described in Chapter 3. The validation of DMan is split into three sections, conform to the three validation points (**V1**, **V3**, **V2**). The goal of this chapter is to describe how DMan was validated and give developers insight into the inner works of the testing framework.

6.1 End-to-end Testing V1

Testing is one of the primary means of quality assurance (or validation) within software engineering (2). End-to-end is often most the most time-consuming and expensive part of testing, especially in the context of distributed systems (2, 22). DMan needs to be able to support every kind of MSA and therefore any kind of composition of relationships between models. A testing framework was developed to simplify the end-to-end tests of DMan. This framework is described first, afterwards the tests utilizing this framework are described.

6.1.1 Testing Framework

We used the default testing framework of Spring named JUnit to implement the test cases. We have identified three major design decisions while designing end-to-end tests for DMan.

1. At what isolation level should services be ran (e.g. do we host a microservice per machine, per process or per thread)? Since we would like to isolate specific requirements of DMan (**R1**, **R2**, **R3**), we choose to run microservices per thread in the same process to remove the need for service discovery and communication protocols. This allows for better debugability since there is no communication overhead and make the tests faster since it simulates multiple microservices locally.

2. How can we instantiate arbitrary number of services to cover basic and edge cases with tests generically? We want to isolate the components of DMan to reduce the number of moving parts (or variables) in our tests. Therefore, we removed the need for a Spring Cloud instance per microservice inside our testing framework by simulating the auto-configuration process of Spring Boot and by abstracting away dependency injection to specialized test classes. This allows us to construct any MSA layout dynamically without setting up any additional machines.

3. How do we set up the persistence layer that the infrastructure layer(s) of the different microservices interact with such that we can verify that the proper operations are executed? The testing framework starts one instance of the Spring Cloud framework that hosts the persistence of all the microservices instantiated during the tests (the instance of Spring Cloud gets restarted before every test). Repositories can be queried directly to check if the test resulted in the expected operations.

The class diagram of the testing framework is shown in Figure 6.2. Test cases are methods that are annotated with `@Test` and defined on test classes that inherit the base class `TestBase`. The `TestBase` instantiates one instance of `SpringBoot` which scans for the repositories defined inside of the test classes automatically. `MicroserviceGrid` contains all the microservices used for the specific test cases. The `Microservice` class hosts all the components required by DMan for a particular service. When a test case is started, the auto-configuration is simulated by `Microservice`. During auto-configuration, the `AutowireProvider` and `ExternalContractBuilder` interface implementations are changed to `TestAutowireProvider` and `TestExternalContractBuilder` to facilitate autowire functionality for the transaction manager with Spring and communication without feign clients respectively. A `Microservice` is derived from a service definition by `TestBase` using the function `microserviceFrom`. A service definition makes use of nested classes and compactly defines the repositories, data services and models required for the test. An example of a service definition can be found in Figure 6.1. For end-to-end tests microservices use the same class for entities and domain models to reduce the number of classes required to compose a service definition. Additionally, `TestBase` provides utility functions like `dataServiceFor` or `transactionManagerFor` to conveniently allow access to a specific data service or transaction manager respectively.

```
@ImplementExternalService(serviceName = "ServiceB")
private interface ServiceB extends ExternalService {
  @org.springframework.stereotype.Repository
  interface Repository extends JpaRepositoryImplementation<
     Model, Long> { }
  @NoArgsConstructor @RequiredArgsConstructor
  @Getter @Setter @Entity
  class Model extends DomainModelInstanceBase<Long> {
    @Id @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @OneToOne(service = ServiceA.class, modelName = "Model")
    @EnsureConsistency(onInconsistency=NULL_VIOLATING_RECORD)
    private Long AId;
  }
}
```

Figure 6.1: An example of a service definition for testing.

6.1.2 Test Cases

The testsets of DMan cover the core requirements **R1**, **R2**, **R3**. Each testset is now briefly described.

R1 Tests that cover successful and unsuccessful (nested) transactions in the context of an MSA with two and four microservices.

R2 Tests that successfully apply both types of consistency policy (null and delete on violation), in the context of an MSA with two and four microservices. Additionally, tests that fail cover cases that the consistency policy fails to apply due to underlying database constraints in varying MSA configurations.

R3 Tests that cover successful joins and the invalidation of join cache on the deletion of cached models on the dependee service.

6.2 Development Process V2

DMan was developed using the JetBrains IDEA integrated development environment (IDE)[1]. During development a simple MSA of two microservices was used debugged using both the integrated debugger of IDEA and Burp Suite. Burp Suite is a penetration testing

[1] https://www.jetbrains.com/idea/

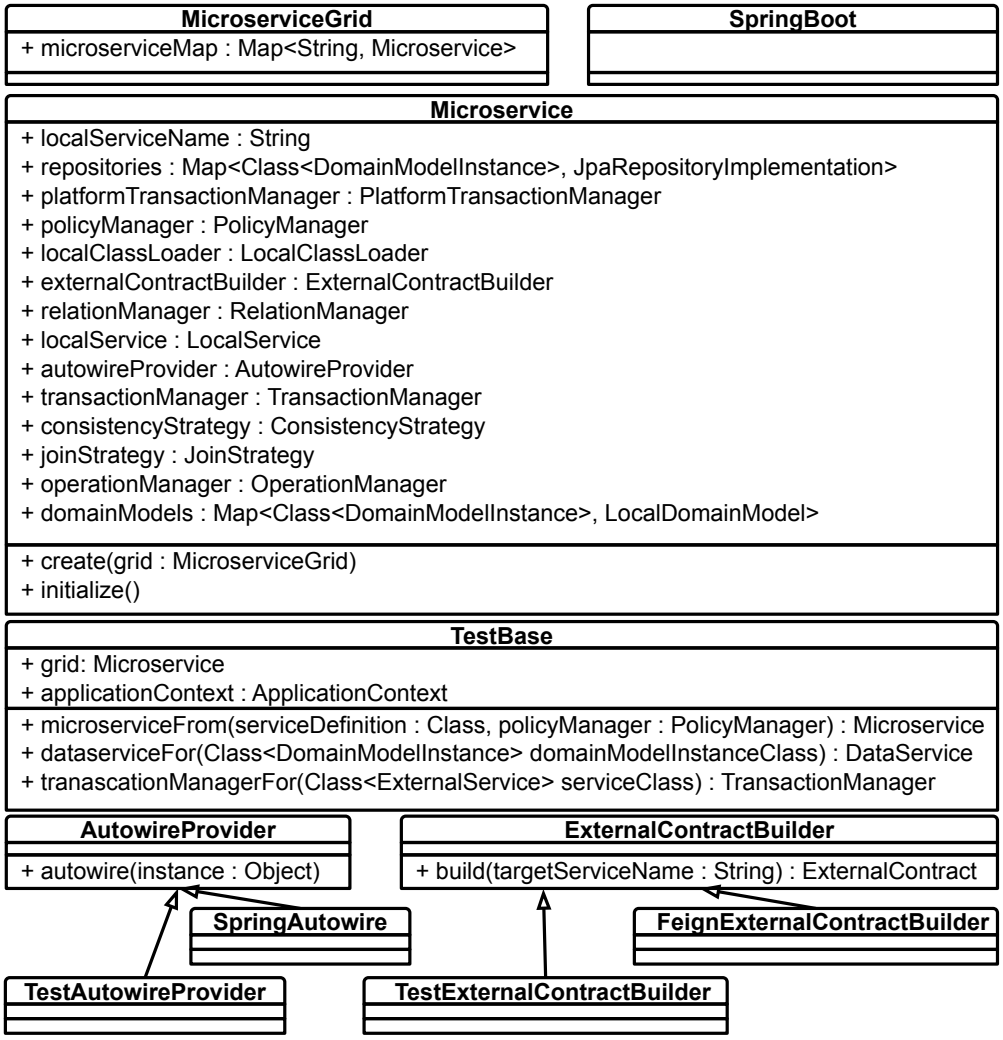

Figure 6.2: Class diagram of the testing framework

application that can intercept HTTP requests. With this tool we could log the requests sent by DMan using the feign clients. Every requirement was tested and debugged in the context of this demo architecture during development. **R5** was tested by manually defining policies in the demo architecture and verifying that the loading of third party code was blocked. **R1** was tested during the development of **R2** and **R3** since they heavily rely on the transaction API. **R2** and **R3** were developed inside of the demo architecture using the previously mentioned tools. **R4** was tested by starting and stopping the second microservice randomly during development.

6.3 Tractable and Non-Repeating Code V3

DMan uses simple annotations to create relational models using preexisting domain model classes (5.6). Based on this relational model, DMan is able to enforce cascading consistency using nested transactions based on different policies using field annotations and maintain read-only copies of model views from external services to make efficient automated joining possible (5.14). The developer does not need to write any repeating code or code related to communication (e.g. HTTP endpoints) and is still able to implement complex distributed data management tasks (5.10) by defining intermediary data services (5.4) and external services (5.2) from simple boilerplate code.

7

Discussion

To reiterate, DMan was introduced with the goal to resolve three data management challenges in the context of an MSA with hot-pluggable microservices (**C1**, **C2**, **C3**, **UC**). From this goal (technical) requirements were derived, that, when satisfied, mark the successful implementation of DMan. Requirements are satisfied when they are both implemented and validated. The design, implementation and validation of each requirement is described in detail by Chapter 4, 5 and 6 respectively. In this chapter we systematically revisit the (technical) requirements and validation methods listed in Chapter 3 with the goal of reflecting on each one such that limitations of DMan can be identified. Afterwards, we cover future work with the purpose of improving DMan, based on the previous reflections.

7.1 Revisiting Requirements

R1 Provide an API for developers that enables distributed transactions using both Saga or 2PC as underlying technique within an MSA (**C1**). — Validated by **V1** and **V2**.

Using the API, developers can dynamically specify and execute distributed transactions automatically. The transaction API lays at the core of DMan since **R2** and **R3** both rely on it. The primary limitation of the API is that error recovery is not fully implemented (e.g. the failure of a machine during the commit operation may result in an inconsistent system state). This was left out of scope intentionally due to time constraints. Additionally, concurrent transactions have not been tested and therefore may introduce previously new bugs in DMan.

R2 Automatically guarantee data consistency between bounded-contexts (**C2**). — Validated by **V1** and **V2**.

Developers can guarantee consistency between bounded-contexts using DMan by annotating domain models with relational metadata. This metadata is used to automatically enforce consistency requirements through intercepting the data operations executed by data services. Initially, 2PC transaction operations were chosen to implement **R2**, since it allows the DBMS to apply underlying database integrity constraints automatically and due to high availability not being a requirement for DMan. However, 2PC heavily trades availability for consistency (41) which, depending on the requirements of the MSA, may not be a viable trade-off. Additionally, the enforcement of the consistency policy may fail due to underlying database constraints on the depender service while validating domain models on read operations. More specifically, this occurs when the dependee deletes a domain model instance that is referenced by an instance on the depender service while it is offline. Consistency enforcement may now fail on the depender service when it validates the instance that contains the invalid reference, due to underlying database integrity constraints (i.e. the entity representing the instance cannot be deleted due to the entity being referenced internally by other tables that are unknown to DMan). This results in the instance only being removed from the data operation by the interceptor but not from the persistence layer due to the transaction being rolled-back. The inconsistency is therefore not visible to the user, but does require manual intervention to be fixed. It is important for developers to consider scenarios in which database constraints may cause consistency enforcement to fail, to prevent these cases from happening. Finally, the cache used by the consistency policy to determine what instance ids are already validated by read operations can grow indefinitely, making it not feasible in the context of large databases.

R3 Manage joins between relations that spawn different bounded-contexts generically and efficiently (**C3**). — Validated by **V1** and **V2**.

DMan provides an API for developers to retrieve and join external domain models. This API leverages interceptable data operations performed by data services and the relational model defined on domain models to automatically retrieve and cache the required external domain models. Similarly to the consistency policy, the cache used by the join policy to store retrieved domain model instance views can grow indefinitely, making it not feasible in the context of large databases.

R4 Services can be added or removed dynamically from the MSA (hot-pluggable) (**UC**). — Validated by **V2**.

The complete relational model is scattered across the MSA since every microservice only contains the part of the model that is relevant to it. Services are hot-pluggable since they can add or remove the relations in their respective relational model from other services dynamically. Additionally, the complete relational model cannot contain circular references by design to, for example, prevent loops when applying the consistency policy. Hot-plugging and removing services dynamically inside more complex MSAs is possible by design, but have not been tested manually or by end-to-end tests due to time constraints.

R5 Services should not trust code from other services unless explicitly accepted (**UC**). — Validated by **V2**.

Services do not trust code from other service since they employ a security policy which requires developers to explicitly configure from what services code should be accepted. When performing the end-to-end tests using our testing framework, services run in the same process and therefore do not use the external class loader, making it not feasible to write automated tests using our testing framework for **R5**. Additionally, the security policy is not secure as is, since there is no authentication system implemented. This means that services can impersonate any other service, defeating the purpose of the security policy.

TR1 The APIs provided by DMan should be tractable and not require repetitive code (**C1**, **C2**, **C3**). — Validated by **V3**.

This requirement is incorporated by design and has been explicitly validated by **V3**. However, the mapping of domain models to entities and vice versa that contain circular references (e.g. two local domain models that both refer to each other), require additional configuration of the mapping library used (i.e. ModelMapper or MapStruct). Additionally, the mapping of null values from domain models to entities by a mapper can overwrite previously set entity fields in the persistence layer, therefore also requiring additional configuration of the mapping library used.

TR2 The services with the MSA should remain loosely-coupled and remain agnostic to the specifics of other services (**UC**).

The design decisions made in Chapter 4 were partially directed by **TR2** such that both the relational and transactional model are agnostic to implementation details of other services (**TR2**). Therefore, services that use DMan remain loosely-coupled.

TR3 The services within the MSA should be isolated from each other (other services cannot be trusted) (**UC**).

The implementation in Chapter 5 took into account **TR3** by physically isolating microservices and limiting the number of endpoints through which they can communicate. **R5** limits the ability of other plugins to run custom transaction code further supporting **TR3**. However, this does not take into account transaction operations that utilize agnostic data services, since they do not require any remote code to execute.

V1 Implementing end-to-end tests that test core use-cases in different MSAs for **R1**, **R2** and **R3** (22).

The end-to-end tests are not executed in a realistic environment, in which microservices are hosted on different machines or in different containers. This results in, for example, **R5** not being end-to-end tested (like discussed previously). Additionally, more end-to-end tests that cover more (edge) use cases of all the requirements of DMan should be implemented since the current number of tests is limited due to time constraints.

V2 Validate **R1**, **R2**, **R3**, **R4**, and **R5** by debugging and prototyping.

During development all requirements were tested within a demo MSA that contained two standalone services and a discovery service. The primary reason to limit to number of services in the demo architecture was to increase the debugability of the system. Nonetheless, DMan should be debugged inside a more complex demo MSA (i.e. an MSA that contains more than two services).

V3 Validate **TR1** by illustrating example use cases of the DMan API and verifying that the code required by the API is both tractable and non-repeating.

TR1 is incorporated by design and has been explicitly validated in Section 6.3. The section illustrates the usage of DMan from the perspective of a developer using previously defined code fragments to illustrate that the required code is both tractable and non-repetitive.

7.2 Future Work

This section describes what future work could be done to further improve DMan. Similarly to Section 7.1, this section is structured by the (technical) requirements and validation

methods defined in Chapter 3 that are relevant for the future work section.

R1 Provide an API for developers that enables distributed transactions using both Saga or 2PC as underlying technique within an MSA (**C1**).

Error recovery. Implement recovery mechanisms for nodes that fail during the execute, commit or rollback transaction phases. The implementation of which could potentially be tested using fault injection testing techniques (24).

Concurrent transactions. Currently, no real testing of concurrent transactions has been done during development or with end-to-end tests. However, this is required to ensure performance in a production environment.

R2 Automatically guarantee data consistency between bounded-contexts (**C2**).

Message delivery guarantees. Implement message delivery guarantees that guarantees a message is delivered and processed once by its recipient by using, for example, a standalone event broker. This will, by extension, guarantee that delete operations get forwarded to depender services on startup, removing the need to validate models on read operations.

Saga variant of consistency policy. Implement the consistency policy using Saga transaction operations to make the MSA eventually consistent and highly available. A Saga variation of the consistency strategy requires DMan to generate idempotent transaction operations that consider both the integrity constraints of the domain models and the entities.

More efficient cache for the consistency policy. Currently, the consistency policy retains all the ids of verified instances in memory. This is not feasible in the context of large databases, therefore an alternative caching strategy could be implemented.

R3 Manage joins between relations that spawn different bounded-contexts generically and efficiently (**C3**).

More efficient cache for the join policy. Similar to the consistency policy, the join policy retains all the retrieved views of external domain models in memory. This is not feasible in the context of large databases, therefore an alternative caching strategy could be implemented.

R4 Services can be added or removed dynamically from the MSA (hot-pluggable) (**UC**).

Support service versioning and load balancing. Service discovery services often can support versioning and load balancing of microservices (20), this is however not yet supported by DMan. Hot-plugging different versions of the same service might be required by an MSA for compatibility reasons.

R5 Services should not trust code from other services unless explicitly accepted (**UC**).

Implement authentication system. For example, implement mutual TLS in combination with a PKI to support microservices making claims about their identify to support better security policies in DMan (12, 45).

TR3 The services within the MSA should be isolated from each other (other services cannot be trusted) (**UC**).

Access Control Level (ACL) policies within data services. ACL policies specify a set of rules which dictate under what conditions a particular operation can be executed on a certain resource. Data services and interceptors can be used to implement such policies to prevent untrusted services from performing unwanted operations.

V1 Implementing end-to-end tests that test core use-cases in different MSAs for **R1**, **R2** and **R3** (22).

End-to-end testing with microservices running on different machines or in different containers. The testing of DMan is currently done by simulating scenarios, and is not yet being done in realistic scenarios.

More tests that cover (edge) use-cases. The current end-to-end test set consists out of a limited number of tests. To ensure the reliability of DMan inside production environment, more (edge) use-cases in different MSA setups should be added.

V2 Validate **R1**, **R2**, **R3**, **R4**, and **R5** by debugging and prototyping.

Debug and prototype DMan in more complex MSAs. DMan should be tested inside a more complex demo MSA (i.e. an MSA that contains more than two services), since it was developed using effectively only two services. Note that the end-to-end tests do consider scenarios with more than two services.

Other interesting future directions of DMan not pertaining to any specific requirement or validation method:

1. **Support polyglot persistence.** DMan currently only supports JPA repositories. However, in theory, a repository can be backed by any type of persistence. Therefore, polygot persistence can be achieved with DMan rather trivially. Nonetheless, this requires work to implement practically.

8

Conclusion

We identified three data management challenges in MSAs through a systammetic literature study (Appendix A). (1) distributed transactions, (2) data relations that spawn different services and (3) joining data from different services efficiently. We drafted requirements for a system named DMan that solves these challenges while taking into account a specific use case. The use case being a platform with hot-pluggable microservices that serve as backend for plugins. We described the designing and implementation process in dept and introduced novel concepts such as the data service layer and the capturing of bounded-context crossing relationships on domain model classes using annotations. In summary, we introduced a peer-to-peer library that implements a distributed transaction system that supports 2PC and Saga for developers to use. Additionally, DMan synchronizes a relational model between services and coordinates data relations and data joining using this model automatically. We validated this system using end-to-end testing and manual testing during development.

References

[1] A. Akbulut and H.G. Perros. **Performance Analysis of Microservice Design Patterns**. *IEEE Internet Computing*, **23**(6):19–27, 2019. cited By 12. 1

[2] Xiaoying Bai, W.T. Tsai, R. Paul, Techeng Shen, and Bing Li. **Distributed end-to-end testing management**. In *Proceedings Fifth IEEE International Enterprise Distributed Object Computing Conference*, pages 140–151, 2001. 41

[3] Doug Barth and Evan Gilman. **Zero Trust Networks: Building Trusted Systems in Untrusted Networks**. 2017. 7

[4] Eric Brewer. **CAP twelve years later: How the "rules" have changed**. *Computer*, **45**(2):23–29, 2012. 7, 8

[5] Hang Chen and Can Cao. **Research and Application of Distributed OSGi for Cloud Computing**. In *2010 International Conference on Computational Intelligence and Software Engineering*, pages 1–5, 2010. 21

[6] Shigeru Chiba and Rei Ishikawa. **Aspect-oriented programming beyond dependency injection**. In *European Conference on Object-Oriented Programming*, pages 121–143. Springer, 2005. 22

[7] Binildas Christudas. **Microservices Security**. In *Practical Microservices Architectural Patterns*, pages 733–777. Springer, 2019. 7

[8] Iuliana Cosmina. **Aspect-Oriented Programming with Spring**. In *Pivotal Certified Professional Core Spring 5 Developer Exam*, pages 277–323. Springer, 2020. 22

[9] Adam L. Davis. *Spring Cloud*, pages 231–246. Apress, Berkeley, CA, 2020. 22, 23, 24, 26

[10] Shahir Daya. *Microservices from theory to practice: Creating applications in IBM bluemix using the microservices approach.* IBM Corporation, International Technical Support Organization, 2015. 5, 6, 7, 8, 21

[11] P. Di Francesco, P. Lago, and I. Malavolta. **Architecting with microservices: A systematic mapping study**. *Journal of Systems and Software*, **150**:77–97, 2019. cited By 68. 1, 6, 21

[12] Wajjakkara Kankanamge Anthony Nuwan Dias and Prabath Siriwardena. *Microservices security in action.* Simon and Schuster, 2020. 51

[13] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. *Microservices: Yesterday, today, and tomorrow.* 2017. cited By 418. 4

[14] Eric Evans and Eric J Evans. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional, 2004. 4, 5

[15] Paul Tepper Fisher and Soloman Duskis. **Managing Transactions**. *Spring Persistence: A Running Start*, pages 137–151, 2009. 30

[16] Martin Fowler. *Patterns of Enterprise Application Architecture: Pattern Enterpr Applica Arch.* Addison-Wesley, 2012. 5, 15, 23

[17] Martin Fowler and James Lewis. **Microservices**. 2014. 1, 5, 12, 15, 21

[18] M. Garriga. **Towards a taxonomy of microservices architectures**. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **10729 LNCS**:203–218, 2018. cited By 25. 3, 4, 5, 7

[19] Richard R Hamming. *Art of doing science and engineering: Learning to learn.* CRC Press, 1997. 1, 10

[20] Stefan Haselböck, Rainer Weinreich, and Georg Buchgeher. **Decision Guidance Models for Microservices: Service Discovery and Fault Tolerance**. In *Proceedings of the Fifth European Conference on the Engineering of Computer-Based Systems*, ECBS '17, New York, NY, USA, 2017. Association for Computing Machinery. 51

[21] David R Heffelfinger. *Java EE 8 Application Development: Develop Enterprise applications using the latest versions of CDI, JAX-RS, JSON-B, JPA, Security, and more*. Packt Publishing Ltd, 2017. 13

[22] Cristian Martínez Hernández, Alexandra Martínez, Christian Quesada-López, and Marcelo Jenkins. **Comparison of end-to-end testing tools for microservices: A case study**. In *International Conference on Information Technology & Systems*, pages 407–416. Springer, 2021. 11, 41, 49, 51

[23] Gregor Hohpe. **Let's Have a Conversation**. *IEEE Internet Computing*, **11**(3):78–81, 2007. 5

[24] Mei-Chen Hsueh, Timothy K Tsai, and Ravishankar K Iyer. **Fault injection techniques and tools**. *Computer*, **30**(4):75–82, 1997. 50

[25] Alexandru Iosup, Laurens Versluis, Animesh Trivedi, Erwin Van Eyk, Lucian Toader, Vincent Van Beek, Giulia Frascaria, Ahmed Musaafir, and Sacheendra Talluri. **The atLarge vision on the design of distributed systems and ecosystems**. In *2019 IEEE 39th International Conference on Distributed Computing Systems (ICDCS)*, pages 1765–1776. IEEE, 2019. 1, 10

[26] Pooyan Jamshidi, Claus Pahl, Nabor C. Mendonça, James Lewis, and Stefan Tilkov. **Microservices: The Journey So Far and Challenges Ahead**. *IEEE Software*, **35**(3):24–35, 2018. 23

[27] D. Jaramillo, D.V. Nguyen, and R. Smart. **Leveraging microservices architecture by using Docker technology**. **2016-July**, 2016. cited By 74. 1, 6

[28] Randy H Katz. **Design transaction management**. In *Information Management for Engineering Design*, pages 56–66. Springer, 1985. 7

[29] Mike Keith, Merrick Schincariol, and Jeremy Keith. *Pro JPA 2: Mastering the JavaTM Persistence API*. Apress, 2011. 13, 14

[30] Seda Kul and Ahmet Sayar. **A Survey of Publish/Subscribe Middleware Systems for Microservice Communication**. In *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT)*, pages 781–785, 2021. 6

[31] R. Laigner, Y. Zhou, M.A.V. Salles, Y. Liu, and M. Kalinowski. **Data management in microservices: State of the practice, challenges, and research directions**. *Proceedings of the VLDB Endowment*, **14**(13):3348, 2021. cited By 1. 5, 7, 8, 18

[32] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li. **Microservices: Architecture, container, and challenges**. pages 629–635, 2020. cited By 1. 1

[33] Raj Malhotra. **Common Use Cases with JPA**. In *Rapid Java Persistence and Microservices*, pages 75–114. Springer, 2019. 13, 30

[34] Raj Malhotra. **Developing microservices with java**. In *Rapid Java Persistence and Microservices*, pages 9–25. Springer, 2019. 23

[35] Andreas Meier and Michael Kaufmann. **Ensuring Data Consistency**. In *SQL & NoSQL Databases*, pages 123–142. Springer, 2019. 7, 8, 18

[36] Sam Newman. *Building microservices.* " O'Reilly Media, Inc.", 2021. 4, 5, 6, 7

[37] Karl Pauls, S McCulloch, RS Hall, and D Savage. **OSGi in action**, 2011. 21

[38] Ken Peffers, Tuure Tuunanen, Marcus A Rothenberger, and Samir Chatterjee. **A design science research methodology for information systems research**. *Journal of management information systems*, **24**(3):45–77, 2007. 1, 10

[39] Dan Pilone and Neil Pitman. *UML 2.0 in a Nutshell.* " O'Reilly Media, Inc.", 2005. 19

[40] Dan RK Ports, Irene Zhang, Samuel Madden, and Barbara Liskov. **Transactional consistency and automatic management in an application data cache**. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10)*, 2010. 8

[41] Chris Richardson. *Microservices patterns: with examples in Java.* Simon and Schuster, 2018. 4, 5, 6, 7, 8, 9, 47

[42] N. Santos, C.E. Salgado, F. Morais, M. Melo, S. Silva, R. Martins, M. Pereira, H. Rodrigues, R.J. Machado, N. Ferreira, and M. Pereira.

A logical architecture design method for microservices architectures. **2**, pages 145–151, 2019. cited By 1. 4

[43] Douglas C Schmidt, Michael Stal, Hans Rohnert, and Frank Buschmann. *Pattern-oriented software architecture, patterns for concurrent and networked objects*. John Wiley & Sons, 2013. 13

[44] Sourabh Sharma. *Mastering Microservices with Java 9: Build domain-driven microservice-based applications with Spring, Spring Cloud, and Angular*. Packt Publishing Ltd, 2017. 4, 5

[45] Ekaterina Shmeleva. *How Microservices are Changing the Security Landscape*. Master's thesis, Aalto University. School of Science, 2020. 51

[46] Ekaterina Shmeleva et al. **How Microservices are Changing the Security Landscape**. 2020. 7

[47] A. Sill. **The Design and Architecture of Microservices**. *IEEE Cloud Computing*, **3**(5):76–80, 2016. cited By 74. 4, 5, 6

[48] J. Soldani, D.A. Tamburri, and W.-J. Van Den Heuvel. **The pains and gains of microservices: A Systematic grey literature review**. *Journal of Systems and Software*, **146**:215–232, 2018. cited By 133. 7

[49] H. Unlu, S. Tenekeci, A. Yildiz, and O. Demirors. **Event Oriented vs Object Oriented Analysis for Microservice Architecture: An Exploratory Case Study**. pages 244–251, 2021. cited By 0. 4, 6

[50] Maarten Van Steen and Andrew S Tanenbaum. *Distributed systems*. Maarten van Steen Leiden, The Netherlands, 2017. 1, 5, 6, 8, 12

[51] Vaughn Vernon. *Implementing domain-driven design*. Addison-Wesley, 2013. 4, 5

[52] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. **Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud**. In *2015 10th Computing Colombian Conference (10CCC)*, pages 583–590, 2015. 1

[53] Craig Walls. *Spring Boot in action*. Simon and Schuster, 2015. 22

[54] Qinglin Wu, Yanzhong Hu, and Yan Wang. **Research on Data Persistence Layer Based on Hibernate Framework**. In *2010 2nd International Workshop on Intelligent Systems and Applications*, pages 1–4, 2010. 14

[55] E.B.H. Yahia, L. Réveillère, Y.-D. Bromberg, R. Chevalier, and A. Cadot. **Medley: An event-driven lightweight platform for service composition**. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, **9671**:3–20, 2016. cited By. 1

[56] He Zhang, Shanshan Li, Zijia Jia, Chenxing Zhong, and Cheng Zhang. **Microservice Architecture in Reality: An Industrial Inquiry**. In *2019 IEEE International Conference on Software Architecture (ICSA)*, pages 51–60, 2019. 21, 24

[57] Yao Zhao, Fei Hu, and Haopeng Chen. **An adaptive tuning strategy on spark based on in-memory computation characteristics**. In *2016 18th International Conference on Advanced Communication Technology (ICACT)*, pages 484–488, 2016. 31

Appendix

A Literature Study

Systematic Literature Review of Microservice Architectures

Thijmen J. Kurk
Vrije Universiteit Amsterdam, The Netherlands
t.j.kurk@student.vu.nl

ABSTRACT

A microservice architecture is an interconnected web of small independent services that each run in its own process while interacting with each other using messages. These architectures are still evolving giving rise to the need for literature reviews. We aim at identifying and classifying definitions and patterns found in microservice architectures described by scientific literature into a concise model. Additionally, we set out to identify challenges found in inter-microservice data management and classify their possible solutions. Towards this goal, we apply well-known systematic literature review methods. Following these methods, we selected 21 relevant studies and iteratively updated our model based on these studies. This work contributes (i) a general model of a modern microservice architectures and (ii) an overview of challenges and solutions within inter-microservice data management.

KEYWORDS

Systematic Literature Review, Microservice, Microservice Architecture, Data Management

1 INTRODUCTION

Nowadays, microservices are the norm when software engineering scalable applications, especially considering that cloud computing and containerization technologies are becoming more prominent [1, 16, 21]. Amazon, Netflix, LinkedIn, Spotify, SoundCloud and other companies [8, 35, 37] are actively adopting and evolving architectures in which microservices are deployed. The first definition of a microservice was introduced by [12] in 2014 as: *'A service that can be automatically and independently be deployed, runs in its own process and communicates using lightweight mechanisms'*. Complementary to this definition, [12] specifies a microservice architecture (MSA) to be a collection of microservices working together towards desired business objectives.

The **goal** of this paper is to analyze published literature on MSAs and classify them. This classification will be done by iteratively creating a model using the definition of, and patterns in MSAs. Additional to this goal we zoom in on inter-microservice data management and classify problems and solutions commonly found therein.

The **audience** of this study are researchers, software engineers and developers who are interested in microservices or are inquisitive about the current state of inter-microservice data management.

The **outline** of this paper is as follows, Section 2 discusses the relevant work we have found and how our research fits into this picture. Subsequently, Section 3 goes into detail on how our systematic literature review is designed. Followed by Section 4 where we define the model and answer our research questions based on the selected literature. Next, in Section 5 we discuss the results. And finally in Section 6 and Section 7 we cover common threats to validity and conclude our study respectively.

To simplify to model used in Section 4, we choose to use the following definition of microservices (Definition 1) and MSAs (Definition 2) both originating from [13], where a process can be either be a process as defined by the operating system (OS) or a collection of threads within an OS process. These definitions generalize better and allow us to define a model that contains less edge cases. For example, using these definitions we can define services used for persistence to be microservices as well, although they do not directly implement logic towards any business objective. Additionally, we use the term service and microservice interchangeably in this paper.

Definition 1 (Microservice, MS). *A microservice is a cohesive independent process interacting via messages.*

Definition 2 (Microservice Architecture, MSA). *A microservice architecture is a distributed application where all its modules are microservices.*

2 RELATED WORK

Numerous secondary studies on MSA have been performed [3, 8, 13, 14, 25, 32, 33]. A majority of these studies aim to provide an overview of what components generally reside in an MSA including but not limited to, how quality attributes [24] can be applied to each of these components [3, 13, 25]. Whereas others concentrate on the patterns in which these components can be utilized most effectively within MSAs [1, 7, 22, 23, 28, 33].

This study distills the work published on MSAs into a model by iterating through studies systematically. In contrast to the work mentioned above, we put the focus on creating a general model of MSAs, instead of enumerating organizational aspects or specific patterns.

3 STUDY DESIGN

This study is designed and carried out by following the guidelines on secondary studies [10, 17, 18]. We provide a replication package, which contains the raw data from each intermediate step executed during the search and selection phase (including any scripts that were used), over at our GitHub[1].

3.1 Research Approach

The goal of this literature study is to provide the reader with a model that contains an overview of components that are generally found inside microservice architectures. Furthermore, we describe the challenges and available solutions that are commonly used for the data management between microservices (i.e., transactions that

[1] https://github.com/ThijmenKurk/literature-study

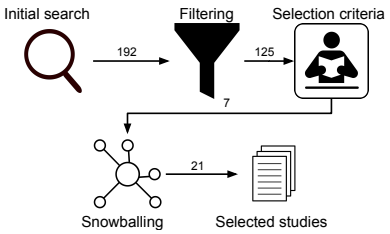

Figure 1: Search and selection process

spawn over multiple microservices). Towards our goal we have drafted the following research questions.

RQ1 How can modern microservice architectures be modelled?
- We drafted this research question to provide a generalizable model for modern microservice architectures derived from scientific literature. This high-level overview will facilitate the next research question by providing context, definitions, and concepts found in microservice architectures.

RQ2 What are the challenges and available solutions used for inter-microservice data management?
- We aim to identify challenges within inter-microservice data management such that we can expose the (potential lack of) available solutions.

3.2 Search and Selection

We now describe the search and selection process of this study. The study design (i.e., the search string and selection criteria) is locked in after the start of the search and selection process to avoid any personal biases. This and other threads to validity are discussed in Section 6.

3.2.1 Initial Search. For this study *Elsevier Scopus*[2] is used as our search engine. This tool has functionality such as the ability to, export search results to different formats (including but limited to CSV and BibTex), apply in-depth filters and search through both citations and references of studies automatically. All of which are used to facilitate the next steps. Our initial selection of 192 studies was a result of the query shown in Figure 3.

3.2.2 Filtering. Using the filter functionality of Scopus we were able to further refine our selection. First, all literature not pertaining to the subject area computer science is removed. Secondly, we filter the literature such that it only contains conference papers, articles and book chapters that are in a finalized state. Thirdly, we exclude any papers that are not written in English. Finally, we omit literature with the subject of microservices in combination with Internet of Things (IoT) and network architectures, since this is out of scope of this review. Refining our selection to a total of 125 studies.

3.2.3 Application of Selection Criteria. The selected studies are manually filtered using the inclusion and exclusion criteria [10, 17, 18] listed below.

I1 Studies focussing on microservices or microservice architectures.
I2 Studies focussing on data management within microservice architectures.
I3 Studies that are peer-reviewed.
I4 Studies that are written in English.
E1 Studies that marginally describe a microservice architecture.
E2 Studies describing the migration process from a monolith to microservice architecture.
E3 Studies that are of bad quality (e.g., many spelling errors, unreadable sentences, etc.)
E4 Studies not available as full-text.

$$(I1 \vee I2) \wedge I3 \wedge I4 \wedge \neg E1 \wedge \neg E2 \wedge \neg E3 \wedge \neg E4 \quad (1)$$

The criteria are applied by following equation 1 during each of the following two steps. First, we read the title and abstract of the study. Last, we read each study full-text. After the manual filtering is done we are left with a selection of 7 studies.

3.2.4 Snowballing. During this phase *forward* and *backward* snowballing [36] is used on the selected studies. On completion, we expanded our selection to 21 studies.

3.2.5 Final Selection. The search and selection process selected 21 studies for this review. An overview of the selected studies can be found in Table 2.

3.3 Data Extraction

During the data extraction phase, every selected study is read full-text and information pertaining to the research questions is stored in a spreadsheet. The goal of this phase it to find patterns within MSAs such that the primary components within these patterns can be described and modelled.

3.4 Data Synthesis

During the data synthesis phase we iteratively go through the extracted data and update our model to fit it. This method is called narrative synthesis and is commonly used when synthesizing literature in the context of systematic literature reviews [26]. The product of this step is a generalized model of MSAs and the challenges and potential solutions within data management, both of which are described in Section 4.

4 RESULTS

In this section we discuss the results of our literature review. The iterative model created from the selected studies is displayed in Figure 2. It defines the main components inside a microservice on an implementation level without detailing any topological specifics. Now the model is described.

4.1 The Model

There are four types of microservices, all of which are interconnected with other microservices through a network. (i) Business microservices implement logic towards a business objective, (ii) utility microservices facilitate business microservices with general functionality (e.g., logging, monitoring, circuit breakers, load balancers, service discovery, etc. [28]), (iii) persistence microservices typically provide database management systems (DBMS) or cache systems to business microservices, and finally (iv) middleware services which facilitate communication between other microservices

[2] https://www.scopus.com/

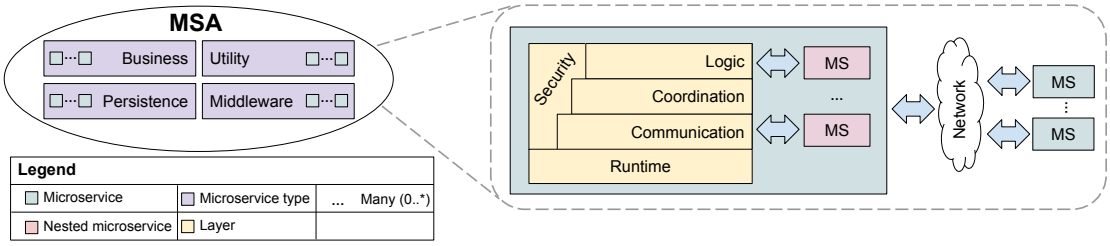

Figure 2: Generalized model of microservice architectures

(e.g., an API gateway, a message broker, a message bus, etc. [13, 28]). An MSA is an interdependent cohesive loosely-coupled composition of these four types of services.

Every microservice can have nested microservices which only the parent microservice is allowed to communicate with. Nested microservices can be colocated on the same node as the parent microservice or hosted on separate nodes [28]. The former is generally the case with the sidecar pattern in which nested utility microservices are deployed to abstract away general functionality from the parent, while the latter is often the case with persistence services [23, 28]. A microservice consists of 3 layers with one overarching layer being security [13, 23, 28, 31] and one foundational layer being runtime. Each layer is now described.

Runtime of a microservice consists of the platform on which it is executed. The industry has evolved such that physical hardware is abstracted away behind many layers of virtualization [28]. There are two levels of virtualization relevant to MSAs, (i) Hardware Abstraction Layer (HAL) virtualization (i.e., virtual machines [28]) and (ii) OS-level virtualization (i.e., containers [16, 28]). Microservices are typically deployed in either, or a combination, of these types of virtualized environment [8].

Communication between microservices consists of three parts [31], (i) the interaction model, (ii) the transportation and finally (iii) the presentation. The interaction model between services can be synchronous or asynchronous. Synchronous communication implies a request/response type pattern, meaning that it is vulnerable for services blocking other services while they are waiting for a response [7, 23, 28]. Asynchronous communication is non-blocking by definition and often comes hand in hand with event-based architectures [34]. However, it requires state management within microservices and typically incurs more communication overhead due to it generally requiring some kind of middleware service, such as a message broker or message bus to facilitate and decouple message delivery [7, 19, 28]. Messages between microservices are transported either via the network or through Inter Process Communication (IPC) depending on performance and scalability requirements. The protocol used for transportation should be selected while considering the desired interaction model, since it can be performed using a variety of protocols [31] (e.g., HTTP, gRPC, XMPP, MQTT, AMQP, etc.). Finally, presentation describes how the data is serialized such that it can be transported between services. Common serializers are JSON, XML and Protobuf [28].

Coordination between services inside an MSA is necessary since it is a distributed system per definition [7, 23, 28]. The coordination within an MSA to perform more complex and elaborate functionalities are either choreography or orchestration based [13, 15, 28]. Orchestration requires composite services that directly coordinates with other services to oversee the process by receiving responses. On the other hand, choreography uses asynchronous events with the publish/subscribe pattern to facilitate collaboration [13, 23, 28].

In essence, the coordination between microservices is about data management [20]. There are various challenges in data management all of which are subject of the next research question.

Logic is the last layer inside of microservices and includes all the programming required to implement the necessary functionality for each particular service. This can be anything due to our wide definition of microservices, ranging from business logic to DBMS logic. One important part of the logic layer is state management. Service can either be stateful or stateless [7]. Statefulness implies session affinity, which in term means less flexibility and scalability [2]. How and if state should be managed within a service can therefore be determined from the scalability requirements of that service.

Security is a layer that overarches all the previously defined layers and is challenging to implement properly. However, it vital within MSAs, due to the large attack surface and complicated connections between services [30]. There are two approaches to securing distributed systems, either a zero-trust-network or a trust-the-network approach [4, 6]. Depending on the security requirements of the MSA, a certain network type should be chosen [4, 6, 13, 30].

4.2 Data Management

The functionality of business microservices is commonly determined by applying domain-driven design (DDD) techniques [11]. In DDD, the problem space of the business is referred to as the domain. This domain can be divided into multiple subdomains each representing a different part of the business. Every subdomain has its own domain model, the scope of which is called a bounded context [9, 28, 29, 34]. From this bounded context, one or more business microservices can be derived [28]. Typically, there is some (minimal) data dependency between the bounded context of these services. Therefore, inter-microservice data management (handled by the coordination layer in the model, Figure 2) is required, given that MSAs are commonly designed with a database per service

pattern [32]. According to the **CAP** theorem, any networked system which shares **P**artitions of data can only pick one of/attain a balance between the following two properties: **A**vailability or **C**onsistency [5]. For example, a highly available MSA cannot be consistent all the time [28]. Therefore, depending on the consistency and availability requirements of the MSA, different solutions to facilitate inter-microservice data management are desirable. We now describe three common data management challenges found within MSAs.

4.2.1 Inter-microservice data relations. Consider two business microservices, one that keeps track of departments and one that keeps track of employees. Every department can have zero-to-many employees. In a relational DBMS, this type of relation is enforced by a foreign key on employee referring to the department of which the employee is a part of. The DBMS enforces policies to keep the database consistent, by for example, cascade deleting employees that are member of a deleted department, or making sure that every employee is member of a valid department. However, this type of enforcement is not trivial if each service has its own (possibly different type of) DBMS [20, 23, 28]. Foreign key emulation is currently resolved using ad-hoc implementations due to the lack of a general solution [20].

4.2.2 Inter-microservice data transactions. A transaction is a unit of work that needs to be completed in its entirety or rolled back. This is challenging to implement in MSAs due to the use of (polyglot) distributed persistence [7, 20, 28]. Consider two business microservices, X_1 that keeps track of inventory and X_2 that is responsible for the placement and tracking of orders. When an order is placed, the stock of the product is updated through the inventory service. A product with only one left in the stock is ordered by two users at the same time. Now X_2 receives two requests to check the stock followed by two requests to update the stock. A naive implementation allows the product to be ordered twice, since X_1 and X_2 do not coordinate the transaction. There are two solutions for this problem, depending on the desired properties of the MSA, displayed in Table 1.

Saga is pattern in which eventual data consistency can be guaranteed by splitting up inter-microservice transactions into a sequence of compensable (potentially retryable) subtransactions each having the scope of only a single microservice [20, 28]. A Saga transaction is successful if each subtransaction has succeeded and can be coordinated using either a choreography or an orchestration based interaction model [28]. If one subtransaction fails, the already succeeded subtransactions are rolled back by executing their respective compensating subtransactions. This makes the MSA eventual consistency and basically available [5].

Two-Phase Commit (2PC) is a pattern in which strong data consistency is guaranteed by orchestrating each operation of the inter-microservice transaction in two phases [20, 28]. The transaction orchestrator starts the first phase by requesting each service to prepare the necessary operations and waiting for each to respond. The second phase executes after each service has successfully prepared and makes the operations permanent. The first phase requires each service to lock resources until the transaction is finalized or aborted since otherwise transactional consistency cannot be guaranteed [20, 27]. This trades availability for strong data consistency [28].

Pattern	Interaction Model	Consistency	Availability
Sagas	Orchestration, Choreography	Eventually consistent	Available
Two-Phase Commit	Orchestration	Consistent	Less available

Table 1: Inter-microservice transaction patterns

4.2.3 Inter-microservice query aggregation and joining. When a microservice requires data from multiple service, it needs to retrieve and aggregate data from each microservice individually and deal with possible inconsistencies by itself [20]. These inconsistencies can form due to, for example, the incorrect implementation of ad-hoc solutions described in Section 4.2.1 or a service failing to return the requested data [28]. Typically, queries that require aggregation or joins are highly optimized inside of DBMSs. Therefore, performing them manually on data from different services implies steep performance penalties. A common solution to this problem is the Command Query Responsibility Segregation (CQRS) pattern. With CRQS, so-called views (read only copies) of tables are prematurely colocated at microservices that require them to speed up aggregation and join operations [28]. Only the microservice owning the table can write to it. Writes are streamed to the views via events using the publish/subscribe pattern, making them eventually consistent [28].

5 DISCUSSION

This literature review aimed to model modern MSAs and the important components within them. This provides researchers with an overview of important aspects and layers to consider while implementing MSAs. We found similar studies within our selection of literature, but these studies considered different aspects of MSAs (e.g., organizational aspects or architectural aspects) instead of implementation aspects.

Additionally, this study derived three challenges and potential solutions for data management problems within MSAs from the selected literature. The lack of solutions for some of these challenges show that MSAs are still evolving and not fully mature yet.

6 THREATS TO VALIDITY

In this section, we discuss the most prominent threats to validity of this study.

Internal validity. The internal validity threat is related to the design and execution of the literature review [17]. This threat is mitigated by defining a detailed research protocol in Section 3 and locking this protocol so that it cannot change after the review is started to prevent personal bias.

External validity. The external validity threat is related to the generalizability of the results [17]. More specifically, for a literature study this would mean that the selected studies not being

representative of the full population (i.e., all the available published literature). To mitigate this form of bias, we used a generic search query (Figure 3) inside of a search engine that indexes scientific literature from multiple sources. Additionally, we utilized both forward and backward snowballing [36] to further expand our selected literature with studies our automatic search might have missed.

Construct validity. The construct validity threat concerns the relation between theory and observation. We mitigated this bias by searching multiple sources using Scopus with only general terms in our search string. Additionally, we rigorously selected relevant studies according to inclusion and exclusion criteria [10, 17, 18].

Conclusion validity. The conclusion validity concerns the degree in which our conclusions are reasonable based on our extracted data. We mitigated this by iteratively defining and updating our model during the data extraction and synthesis phase. Additionally, this threat was mitigated by applying well-known systematic literature review methods [10, 17, 18, 36].

7 CONCLUSION

This study gives an overview of modern microservice architectures in the form of a model and identified three data management challenges and their potential solutions by systematically analyzing the latest studies on microservices using well-known literature review methods [10, 17, 18, 36].

We found that microservices consist of four layers: (i) Runtime, (ii) Communication, (iii) Coordination and (iv) Logic, with one overarching layer being Security. These microservices are connected with each other to form a microservice architecture. There are four types of microservices: (i) Business, (ii) Utility, (iii) Persistence and (iv) Middleware. Additionally, we found three data management challenges: (i) inter-microservice data relations, (ii) inter-microservice data transactions and (iii) inter-microservice query aggregation and joining. We conclude that microservice architectures are evolving but that they are far from mature, given the fact that not all data management challenges have general solutions.

In the future we would like to explore how to solve the challenge described in Section 4.2.1 by proposing a general solution for this problem.

REFERENCES

[1] A. Akbulut and H.G. Perros. 2019. Performance Analysis of Microservice Design Patterns. *IEEE Internet Computing* 23, 6 (2019), 19–27. https://doi.org/10.1109/MIC.2019.2951094 cited By 12.

[2] Yahya Al-Dhuraibi, Fawaz Paraiso, Nabil Djarallah, and Philippe Merle. 2018. Elasticity in Cloud Computing: State of the Art and Research Challenges. *IEEE Transactions on Services Computing* 11, 2 (2018), 430–447. https://doi.org/10.1109/TSC.2017.2711009

[3] N. Alshuqayran, N. Ali, and R. Evans. 2016. A systematic mapping study in microservice architecture. *Proceedings - 2016 IEEE 9th International Conference on Service-Oriented Computing and Applications, SOCA 2016* (2016), 44–51. https://doi.org/10.1109/SOCA.2016.15 cited By 184.

[4] Doug Barth and Evan Gilman. 2017. Zero Trust Networks: Building Trusted Systems in Untrusted Networks. (2017).

[5] Eric Brewer. 2012. CAP twelve years later: How the "rules" have changed. *Computer* 45, 2 (2012), 23–29. https://doi.org/10.1109/MC.2012.37

[6] Binildas Christudas. 2019. Microservices Security. In *Practical Microservices Architectural Patterns*. Springer, 733–777.

[7] Shahir Daya. 2015. *Microservices from theory to practice: Creating applications in IBM bluemix using the microservices approach.* IBM Corporation, International Technical Support Organization.

[8] P. Di Francesco, P. Lago, and I. Malavolta. 2019. Architecting with microservices: A systematic mapping study. *Journal of Systems and Software* 150 (2019), 77–97. https://doi.org/10.1016/j.jss.2019.01.001 cited By 68.

[9] N. Dragoni, S. Giallorenzo, A.L. Lafuente, M. Mazzara, F. Montesi, R. Mustafin, and L. Safina. 2017. *Microservices: Yesterday, today, and tomorrow.* 195–216 pages. https://doi.org/10.1007/978-3-319-67425-4_12 cited By 418.

[10] Tore Dyba, Torgeir Dingsoyr, and Geir K. Hanssen. 2007. Applying Systematic Reviews to Diverse Study Types: An Experience Report. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007).* 225–234. https://doi.org/10.1109/ESEM.2007.59

[11] Eric Evans and Eric J Evans. 2004. *Domain-driven design: tackling complexity in the heart of software.* Addison-Wesley Professional.

[12] Martin Fowler and James Lewis. 2014. Microservices. (2014). http://martinfowler.com/articles/microservices.html

[13] M. Garriga. 2018. Towards a taxonomy of microservices architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 10729 LNCS (2018), 203–218. https://doi.org/10.1007/978-3-319-74781-1_15 cited By 25.

[14] M.S. Hamzehloui, S. Sahibuddin, and K. Salah. 2019. A systematic mapping study on microservices. *Advances in Intelligent Systems and Computing* 843 (2019), 1079–1090. https://doi.org/10.1007/978-3-319-99007-1_100 cited By 3.

[15] Gregor Hohpe. 2007. Let's Have a Conversation. *IEEE Internet Computing* 11, 3 (2007), 78–81. https://doi.org/10.1109/MIC.2007.68

[16] D. Jaramillo, D.V. Nguyen, and R. Smart. 2016. Leveraging microservices architecture by using Docker technology. *Conference Proceedings - IEEE SOUTHEASTCON* 2016-July (2016). https://doi.org/10.1109/SECON.2016.7506647 cited By 74.

[17] Barbara Kitchenham. 2004. Procedures for Performing Systematic Reviews. *Keele, UK, Keele Univ.* 33 (08 2004).

[18] Barbara Kitchenham and Stuart Charters. 2007. Guidelines for performing Systematic Literature Reviews in Software Engineering. 2 (01 2007).

[19] Seda Kul and Ahmet Sayar. 2021. A Survey of Publish/Subscribe Middleware Systems for Microservice Communication. In *2021 5th International Symposium on Multidisciplinary Studies and Innovative Technologies (ISMSIT).* 781–785. https://doi.org/10.1109/ISMSIT52890.2021.9604746

[20] R. Laigner, Y. Zhou, M.A.V. Salles, Y. Liu, and M. Kalinowski. 2021. Data management in microservices: State of the practice, challenges, and research directions. *Proceedings of the VLDB Endowment* 14, 13 (2021), 3348. https://doi.org/10.14778/3484224.3484232 cited By 1.

[21] G. Liu, B. Huang, Z. Liang, M. Qin, H. Zhou, and Z. Li. 2020. Microservices: Architecture, container, and challenges. *Proceedings - Companion of the 2020 IEEE 20th International Conference on Software Quality, Reliability, and Security, QRS-C 2020* (2020), 629–635. https://doi.org/10.1109/QRS-C51114.2020.00107 cited By 1.

[22] A. Messina, R. Rizzo, P. Storniolo, M. Tripiciano, and A. Urso. 2016. The database-is-the-service pattern for microservice architectures. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9832 LNCS (2016), 223–233. https://doi.org/10.1007/978-3-319-43949-5_18 cited By 21.

[23] Sam Newman. 2021. *Building microservices.* " O'Reilly Media, Inc.".

[24] Liam O'Brien, Paulo Merson, and Len Bass. 2007. Quality attributes for service-oriented architectures. In *International Workshop on Systems Development in SOA Environments (SDSOA'07: ICSE Workshops 2007).* IEEE, 3–3.

[25] C. Pahl and P. Jamshidi. 2016. Microservices: A systematic mapping study. *CLOSER 2016 - Proceedings of the 6th International Conference on Cloud Computing and Services Science* 1 (2016), 137–146. https://doi.org/10.5220/0005785501370146 cited By 153.

[26] Jennie Popay, Helen Roberts, Amanda Sowden, Mark Petticrew, Lisa Arai, Mark Rodgers, Nicky Britten, Katrina Roen, Steven Duffy, et al. 2006. Guidance on the conduct of narrative synthesis in systematic reviews. *A product from the ESRC methods programme Version* 1, 1 (2006), b92.

[27] Dan RK Ports, Irene Zhang, Samuel Madden, and Barbara Liskov. 2010. Transactional consistency and automatic management in an application data cache. In *9th USENIX Symposium on Operating Systems Design and Implementation (OSDI 10).*

[28] Chris Richardson. 2018. *Microservices patterns: with examples in Java.* Simon and Schuster.

[29] N. Santos, C.E. Salgado, F. Morais, M. Melo, S. Silva, R. Martins, M. Pereira, H. Rodrigues, R.J. Machado, N. Ferreira, and M. Pereira. 2019. A logical architecture design method for microservices architectures. *ACM International Conference Proceeding Series* 2 (2019), 145–151. https://doi.org/10.1145/3344948.3344991 cited By 1.

[30] Ekaterina Shmeleva et al. 2020. How Microservices are Changing the Security Landscape. (2020).

[31] A. Sill. 2016. The Design and Architecture of Microservices. *IEEE Cloud Computing* 3, 5 (2016), 76–80. https://doi.org/10.1109/MCC.2016.111 cited By 74.

[32] J. Soldani, D.A. Tamburri, and W.-J. Van Den Heuvel. 2018. The pains and gains of microservices: A Systematic grey literature review. *Journal of Systems and Software* 146 (2018), 215–232. https://doi.org/10.1016/j.jss.2018.09.082 cited By 133.

[33] D. Taibi, V. Lenarduzzi, and C. Pahl. 2018. Architectural patterns for microservices: A systematic mapping study. *CLOSER 2018 - Proceedings of the 8th International*

```
TITLE("microservice?") AND (TITLE("architecture") OR
    (TITLE("data") AND (TITLE("coordination") OR
  TITLE("management")))) AND (LIMIT-TO(PUBSTAGE,
     "final")) AND (LIMIT-TO(DOCTYPE, "cp") OR
LIMIT-TO(DOCTYPE, "ar") OR LIMIT-TO(DOCTYPE, "ch"))
       AND (LIMIT-TO(SUBJAREA, "COMP")) AND
        (LIMIT-TO(LANGUAGE, "English")) AND
  (EXCLUDE(EXACTKEYWORD, "Internet Of Things") OR
   EXCLUDE(EXACTKEYWORD, "Network Architecture"))
```

Figure 3: The initial search string and the refined search string after the filtering process

Conference on Cloud Computing and Services Science 2018-January (2018), 221–232. https://doi.org/10.5220/0006798302210232 cited By 87.

[34] H. Unlu, S. Tenekeci, A. Yildiz, and O. Demirors. 2021. Event Oriented vs Object Oriented Analysis for Microservice Architecture: An Exploratory Case Study. *Proceedings - 2021 47th Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2021* (2021), 244–251. https://doi.org/10.1109/SEAA53835.2021.00038 cited By 0.

[35] Mario Villamizar, Oscar Garcés, Harold Castro, Mauricio Verano, Lorena Salamanca, Rubby Casallas, and Santiago Gil. 2015. Evaluating the monolithic and the microservice architecture pattern to deploy web applications in the cloud. In *2015 10th Computing Colombian Conference (10CCC)*. 583–590. https://doi.org/10.1109/ColumbianCC.2015.7333476

[36] Claes Wohlin. 2014. Guidelines for Snowballing in Systematic Literature Studies and a Replication in Software Engineering. In *Proceedings of the 18th International Conference on Evaluation and Assessment in Software Engineering* (London, England, United Kingdom) *(EASE '14)*. Association for Computing Machinery, New York, NY, USA, Article 38, 10 pages. https://doi.org/10.1145/2601248.2601268

[37] E.B.H. Yahia, L. Réveillère, Y.-D. Bromberg, R. Chevalier, and A. Cadot. 2016. Medley: An event-driven lightweight platform for service composition. *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)* 9671 (2016), 3–20. https://doi.org/10.1007/978-3-319-38791-8_1 cited By.

	Document Type	Title	Authors	Year
1	Article	Performance Analysis of Choreography and Orchestration in Microservices Architecture	Kristianto, H., Zahra, A.	2021
2	Conference Paper	Data management in microservices: State of the practice, challenges, and research directions	Laigner, R., Zhou, Y., Salles, M.A.V., Liu, Y., Kalinowski, M.	2021
3	Conference Paper	Microservices: Architecture, container, and challenges	Liu, G., Huang, B., Liang, Z., Qin, M., Zhou, H., Li, Z.	2020
4	Conference Paper	A logical architecture design method for microservices architectures	Santos, N., Salgado, C.E., Morais, F., Melo, M., Silva, S., Martins, R., Pereira, M., Rodrigues,...	2019
5	Conference Paper	A Comparative Review of Microservices and Monolithic Architectures	Al-Debagy, O., Martinek, P.	2018
6	Conference Paper	Towards a taxonomy of microservices architectures	Garriga, M.	2018
7	Conference Paper	Leveraging microservices architecture by using Docker technology	Jaramillo, D., Nguyen, D.V., Smart, R.	2016
8	Conference Paper	A systematic mapping study in microservice architecture	Alshuqayran, N., Ali, N., Evans, R.	2016
9	Conference Paper	Microservices: A systematic mapping study	Pahl, C., Jamshidi, P.	2016
10	Article	The pains and gains of microservices: A Systematic grey literature review	Soldani, J., Tamburri, D.A., Van Den Heuvel, W.-J.	2018
11	Article	Microservices Patterns	Richardson, C.	2018
12	Article	Microservices: Yesterday, today, and tomorrow	Dragoni, N., Giallorenzo, S., Lafuente, A.L., Mazzara, M., Montesi, F., Mustafin, R., Safina, L.	2016
13	Conference Paper	Architectural patterns for microservices: A systematic mapping study	Taibi, D., Lenarduzzi, V., Pahl, C.	2018
14	Article	The Design and Architecture of Microservices	Sill, A.	2016
15	Conference Paper	The database-is-the-service pattern for microservice architectures	Messina, A., Rizzo, R., Storniolo, P., Tripiciano, M., Urso, A.	2016
16	Article	Performance Analysis of Microservice Design Patterns	Akbulut, A., Perros, H.G.	2019
17	Conference Paper	Microservices architecture: Challenges and proposed conceptual design	Munaf, R.M., Ahmed, J., Khakwani, F., Rana, T.	2019
18	Article	Microservices from theory to practices	Daya, S.	2015
19	Conference Paper	Event Oriented vs Object Oriented Analysis for Microservice Architecture: An Exploratory Case Study	Unlu, H., Tenekeci, S., Yildiz, A., Demirors, O.	2021
20	Article	Architecting with microservices: A systematic mapping study	Di Francesco, P., Lago, P., Malavolta, I.	2019
21	Conference Paper	A systematic mapping study on microservices	Hamzehloui, M.S., Sahibuddin, S., Salah, K.	2019

Table 2: Selected studies