Czech Technical University in Prague
Faculty of Electrical Engineering

Department of Computer Science

Microservice pattern Saga as a state machine

Ivan Shalaev

Bc. programme: Software Engineering and Technologies
Branch of study: Architect of Web Applications
Supervisor: Ing. Jiří Šebek

May 25, 2023 in Prague

Declaration

I hereby declare I have written this semester's assignment independently and quoted all the sources of information used in accordance with methodological instructions on ethical principles for writing an academic article.

in Prague ....

Ivan Shalaev

Acknowledgement

I would like to express my sincere gratitude to my teacher, Ing. Jiří Šebek for giving me such a golden opportunity to work on this wonderful project on Microservice pattern Saga as a state machine. Ing. Jiří Šebek is my mentor who has helped me a lot throughout this project by providing important data and information. His valuable words and advice have truly motivated me. The process of preparing this project in collaboration with my teacher is a refreshing experience.

Ivan Shalaev

Result application repository

Ivan Shalaev, Microservice pattern Saga as a state machine, Bachelor Thesis, Czech Technical University in Prague, Faculty of Electrical Engineering, 2023. Available also on *https://gitlab.fel.cvut.cz/shalaiva/saga-orchestrator*

Title:
Microservice pattern Saga as a state machine

Abstract:

This research focuses on investigating the Saga pattern as a state machine in the context of microservices architecture. Implementing the Saga pattern as a state machine offers both theoretical and practical advantages and provides a natural mechanism for representing different states that a Saga can transition between. This approach facilitates a better understanding of Saga behavior, particularly when it involves a complex sequence of local transactions. The state machine framework, which is formal and well-studied, is beneficial for modeling the behavior of distributed systems and aids in the analysis and justification of Saga properties. This work includes the implementation of Orchestration Saga, which is encapsulated in a separate module. This module has an API for run-time Saga creation and execution.

Key words: Microservice Architecture, Saga Pattern, State Machine, Data Consistency, System Behavior Modeling, Distributed Systems.

Název práce:
Mikroservisní vzor Sága jako stavový automat

Abstrakt:

Tento výzkum se zaměřuje na zkoumání vzoru Saga jako stavového stroje v kontextu architektury mikroslužeb. Implementace vzoru Saga jako stavový stroj nabízí teoretické a praktické výhody a poskytuje přirozený mechanismus pro označení různých stavů, mezi kterými může Saga přecházet. Tento přístup usnadňuje lepší pochopení chování Sagi, zejména když zahrnuje komplexní sekvenci lokálních transakcí. Rámec stavového stroje, který je formální a dobře studovaný, je prospěšný pro modelování chování distribuovaných systémů a pomáhá při analýze a odůvodnění vlastností Sagi. Tato práce obsahuje implementaci Orchestrační Ságy, která je zapouzdřena v samostatném modulu. Tento modul má rozhraní pro vytváření a provádění Ságy za běhu.

Klíčová slova: Architektura mikroslužeb, Vzor Saga, Stavový stroj, Konzistence dat, Modelování chování systému, Distribuované systémy.

Contents

List of acronyms

ACID	Atomicity, Consistency, Isolation, Durability
2PC	The two-phase commit
POM	Project Object Model
API	Application Programming Interface
SP	State Propagation

List of Figures

Chapter 1

Introduction

In recent years, there has been a growing trend toward the use of microservice architecture [1] in the design of new software applications. This architectural style is characterized by the decomposition of a monolithic application into a collection of small, autonomous services that can be developed, deployed, and scaled independently. The popularity of this approach is driven by several key benefits, including improved scalability, increased flexibility, enhanced fault isolation, and greater organizational alignment [2].

However, while these new methods can effectively address a wide range of issues and provide specific features, they also introduce unique challenges. In other words, some microservice patterns are derived from another pattern's issues. It could be clearly seen in the following diagram 1.1:

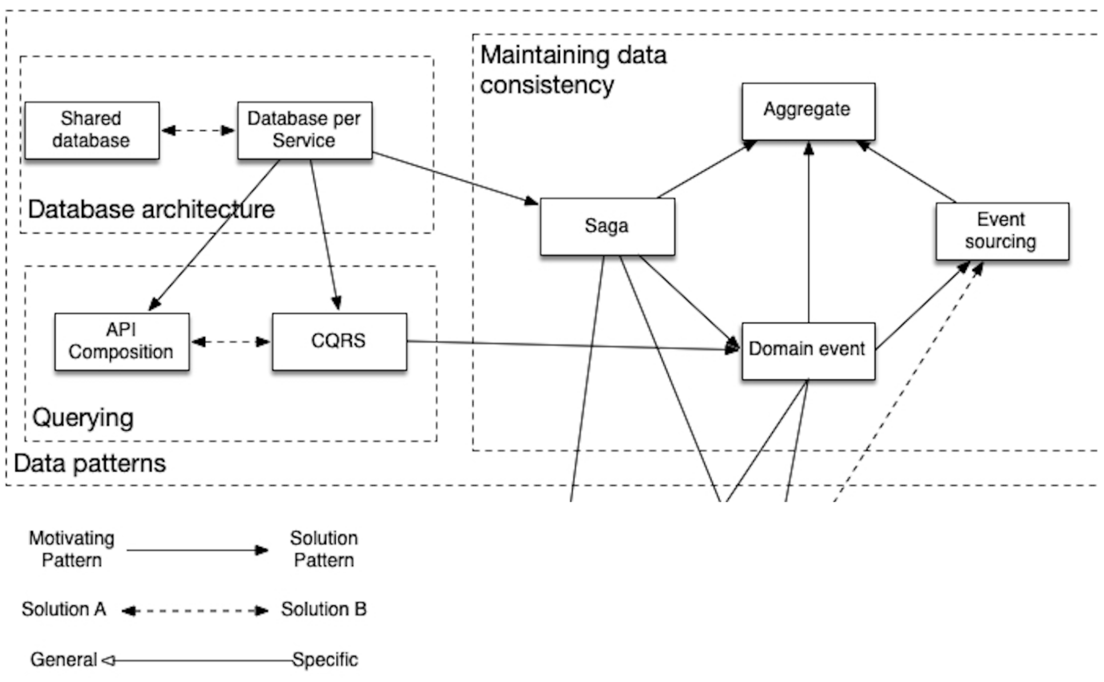

Figure 1.1: Microservice Patterns board [3]

It is crucial to consider the problem that a new design pattern addresses while studying it. By referring to the diagram provided, we can identify the pattern that leads to the problem and approach it from a different angle.

For example, according to the diagram, the Saga pattern is derived from the Database per Service. As the name implies, the Database per Service pattern [4] presumes that every service in topology has its own fully isolated database. This pattern contributes a lot of advantages and it is essential for microservice architecture. By giving each microservice its own dedicated database, it is easier to ensure that each microservice can evolve and scale independently of the others. This independence allows different developer's teams to choose the best technology stack and data models for each microservice and avoid potential conflicts that could arise from sharing a monolithic database [5]. On the other hand, this autonomy can make it difficult to ensure that data remains consistent across services, especially in the presence of failures or network partitions. In other words, after integrating this pattern into our application we will get a system with data inconsistency. The Saga pattern addresses this issue by providing a way to coordinate transactions across multiple services in a consistent and reliable manner [6].

The Saga pattern defines a series of local transactions that are executed by each service, and a set of compensating transactions that can be executed in case of failures. These transactions are executed in a specific order and are connected using asynchronous messaging, which enables the system to follow the "story" of a request's processing. If a failure occurs, the system can revert to the previous consistent state by executing the compensating transactions in the reverse order [7].

Another advantage of the Saga pattern compared to other mechanisms is that it helps to maintain the consistency of the data without locking data across the system [8] (e.g., in contradistinction to the two-phase commit). This eliminates the contention issues caused by locks and enhances performance by decreasing delays.

The purpose of this study is to investigate the Saga pattern, including its features and propose an implementation.

Chapter 2

Analysis

2.1 Saga: Origin

A microservice-based application is essentially a distributed system that is comprised of multiple smaller services that work together to deliver the desired functionality. Despite its benefits, this architecture style faces challenges, particularly in managing transactions that involve multiple services [9].

Adopting a database-per-microservice approach offers several advantages in a microservice architecture [4]. By isolating domain data, each service can utilize its optimal data storage type and schema, independently scale its own database, and remain unaffected by failures in other services. However, maintaining data consistency across multiple service-specific databases presents challenges.

2PC[1] protocol, which is a form of distributed transaction, requires all parties involved in a transaction to either commit or roll back before it can progress [10]. However, certain implementation types, such as NoSQL databases and message brokering, are **not** compatible with this protocol. Moreover, all participating services must be available for the distributed transactions to commit using 2PC, which can potentially reduce overall system availability [11].

Thankfully, there is an alternative to traditional distributed transaction implementation that doesn't greatly affect the system's availability - the Saga pattern.

2.2 Saga: Basics

The Saga architecture pattern provides distributed transaction management in distributed systems using a sequence of local transactions [12]. The Saga pattern was developed to address the need for breaking down long-lived transactions into smaller, more manageable ones. These local transactions are executed within the context of

[1]The two-phase commit

a single service and rely on the assumption that each service utilizes a database that supports ACID[2] transactions to ensure consistency. The key concept introduced by the Saga pattern is the use of compensable local transactions, which refer to transactions that **must be** rolled back if necessary. In general, a Saga can be defined as a sequence of local transactions, all of which possess the capability for compensation. A successful saga flow where no service fails (Figure 2.1):

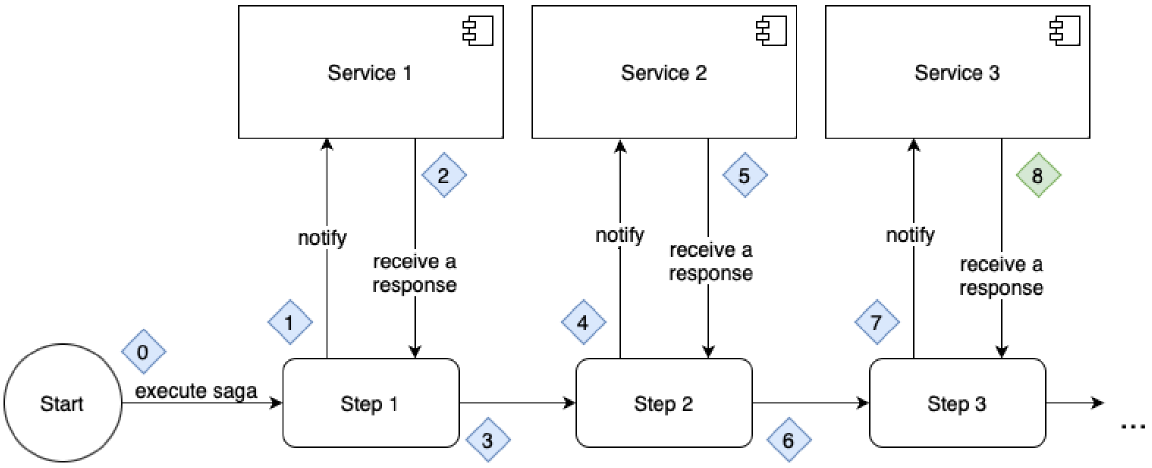

Figure 2.1: Successful Saga flow

In the event of a failure within one of the services, the Saga pattern ensures that all previous local transactions are rolled back and the Saga terminates with an error, see Figure 2.2.

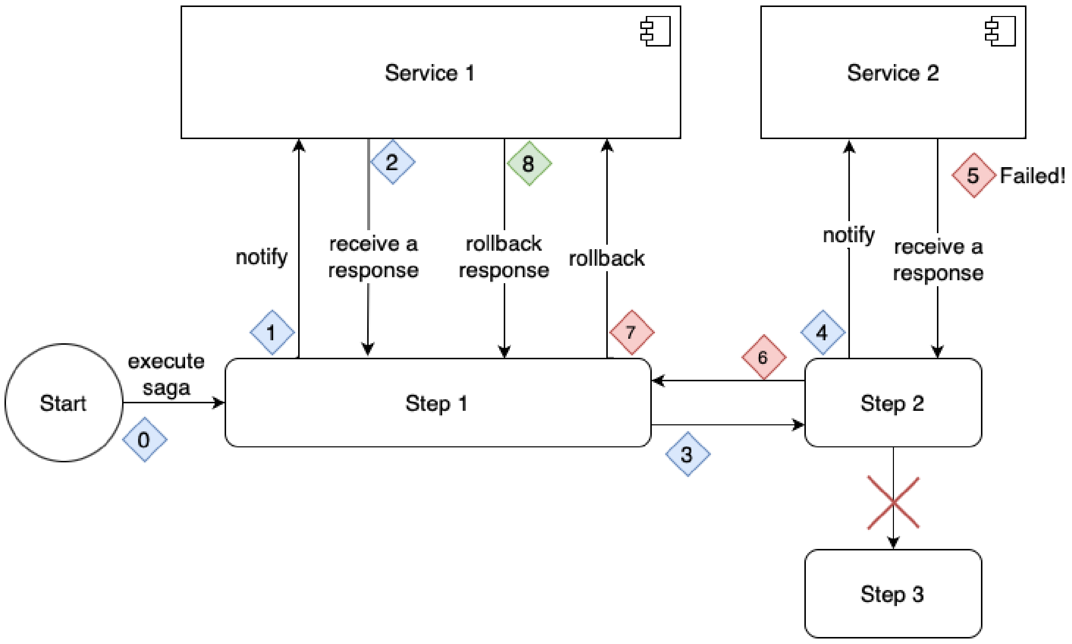

Figure 2.2: Alternative Saga flow

According to Chris Richardson, there are two basic types of saga implementations.[13]

[2]Atomicity, Consistency, Isolation, Durability

1. **Choreography**: the decision making and sequencing among the saga participants. They primarily communicate by exchanging events.

2. **Orchestration**: a saga's coordination logic in a saga orchestrator class. A saga orchestrator sends command messages to saga participants telling them which operations to perform.

2.2.1 Saga Orchestration

Saga orchestration provides a **centralized** mechanism for managing transactions across multiple microservices. The way service are interconnected is shown in Figure 2.3. In this approach, a coordinator service is responsible for executing a series of steps in a defined order and ensuring that the overall system remains in a consistent state. If a microservice fails, the coordinator can take appropriate action, such as rolling back transactions, to ensure that the system remains consistent.

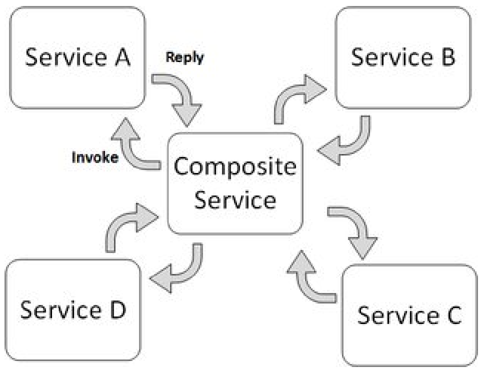

Figure 2.3: Saga Orchestration [14]

2.2.2 Saga Choreography

Choreography, on the other hand, is a **decentralized** approach to coordination, where microservices interact with each other directly without a centralized coordinator. In this approach, microservices communicate with each other using message passing, and the coordination of the overall system is managed through the interactions between microservices which is represented in Figure 2.4. This approach results in a loosely coupled system, but error handling and transaction management can be more complex.

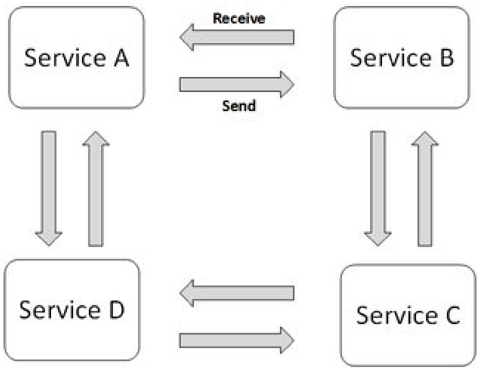

Figure 2.4: Saga Choreography [14]

2.2.3 Saga pattern excellence

In summary, Saga orchestration provides a centralized mechanism for transaction management and error handling, while Choreography results in a loosely coupled system with decentralized control. The choice between these approaches depends on the specific requirements of a system and the trade-offs between centralization and decoupling. Anyways, both two approaches share common features [15]:

1. **Flexibility**: Sagas provide more flexibility in terms of handling failures and ensuring consistency. In a Saga, each microservice can implement its own compensation logic for rolling back changes in case of failures, whereas in 2PC, all microservices must agree to commit or roll back a transaction [16].
2. **Scalability**: 2PC requires that all participants in a transaction be available and respond promptly. This can be a bottleneck in large and distributed systems. Sagas, on the other hand, are more scalable as they do not require all participants to be available and responsive at the same time.
3. **Loose coupling**: 2PC tightly couples the microservices involved in a transaction, while Sagas allow for more loose coupling between microservices as they do not require coordination between participants.

2.2.4 Saga pattern flaws

The Saga pattern is a powerful approach for managing transactions in a distributed system or a microservices architecture. By dividing long-lived transactions into a series of local transactions, it helps to maintain the overall consistency of the system, even when individual services fail. However, like any architectural approach, the Saga pattern is not without its shortcomings. Here, we'll explore some of the notable flaws of the Saga pattern [17].

1. **Failure Handling**: When a failure occurs in a Saga, compensating transactions are triggered to roll back the changes. However, this compensating

transactions themselves could fail, leading to a challenging situation where the system needs to decide how to handle such failures.

2. **Increased Latency**: Since a Saga involves a series of local transactions that need to be coordinated, it could lead to increased latency compared to a monolithic architecture where all data could be updated in a single database transaction.

3. **Increased Complexity in Design**: Implementing Sagas requires a shift in thinking and adds complexity to the design and implementation of the system. It requires developers to identify and implement compensating transactions, manage transaction ordering, and handle failures.

4. **Dependency on Reliable Messaging**: The Saga pattern often relies on reliable messaging to communicate between services, especially in the choreography approach. If the messaging system fails or messages are lost, it could disrupt the correct functioning of the Saga.

In summary, while the Saga pattern offers significant benefits for managing transactions in distributed systems, it's not without its flaws. The choice to use this pattern should be made carefully, considering the specific requirements and constraints of the system, and the trade-offs involved. Implementing Sagas effectively also requires a significant level of expertise and understanding of distributed systems.

Chapter 3

Proposal

Both saga orchestrator and choreography are approaches to managing distributed transactions in a microservices architecture. While choreography is based on event-driven communication between services, the saga orchestrator pattern relies on a centralized component to manage the saga's execution. It is typically a challenging task to choose between these two approaches. Several factors heavily influence the choice between choreography and orchestration in practice.

For this project, we will utilize the orchestration approach to implement saga since it can be more effectively encapsulated within a separate architectural component.

3.1 Saga as a state machine

The implementation of the Saga pattern as a state machine can provide several benefits from both a theoretical and practical perspective [18]

From a theoretical perspective, state machines provide a natural way to represent the different states that a Saga can be in, and the transitions between them. This facilitates a clear and structured understanding of the behavior of a Saga, especially when it involves a complex sequence of local transactions. Additionally, state machines offer a formal and well-studied framework for modeling the behavior of distributed systems, which can be useful when analyzing and reasoning about the properties of a Saga implementation [19].

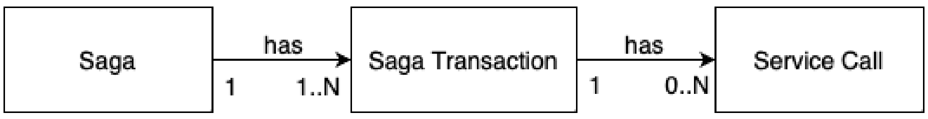

Figure 3.1: Proposed Saga hierarchy model

It is necessary to introduce another crucial abstraction, namely, state propagation, which operates as a means of communication directed in the opposite direction among the three aforementioned abstractions, transmitting information from the service call to the saga. The primary objective of state propagation is to eliminate circular dependencies among these three abstractions, thus adopting a form of the Publish-Subscriber design pattern [20]

3.1.1 State Propagation

In essence, the issue that state propagation resolves arises from the division of Saga into three distinct abstractions. Were we to have a single, overarching state machine, such a problem would not present itself. However, through the introduction of this layer of abstraction, we are able to establish interconnection among all parts of Saga and maintain a requisite level of coupling between them.

The general rule for state propagation is that **it propagates an ended state** of a downstream state machine to an upstream one. The only exclusion will be described later in the Saga Transaction section 3.1.3.

3.1.2 Service Call

Saga operates asynchronous messages by definition. Service call represents a simple request to a particular service. It reaches either succeed or failed states depending on the corresponding response. The state diagram is depicted in Figure 3.2

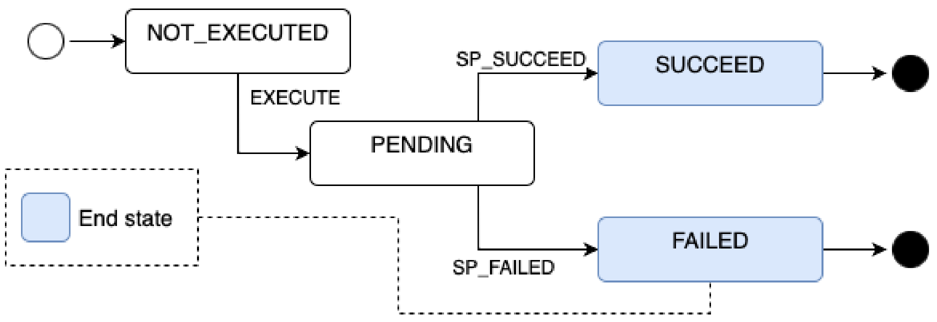

Figure 3.2: Service call state diagram

The prefix SP[1] means *state propagation*. The transition labeled as SP SUCCEED on the diagram signifies that the target service in the service call has successfully processed the request and returned a response that is considered a success. Similarly, the term SP FAILED implies that the response received from the target service was unsuccessful or a failure.

Note that the primary concept of state propagation involves propagating the actual state from a downstream state machine to an upstream one (more in State Propagation section 3.1.1). However, in this specific scenario, it operates in a slightly different manner because the service call functions as a 'boundary' state machine and therefore does not possess any downstream state machines. That's why the state

[1]State Propagation

propagations addressed to the service call is not a state propagation in essence. The decision to utilize state propagation abstraction for service calls is prompted by the desire to standardize the state machine template across all state machines.

3.1.3 Saga Transaction

Saga Transaction represents a sequence of local transactions between Saga and a service which, after execution, turns the service into a consistent state from its Saga point of view. This abstraction serves as a logical encapsulation of the functionality pertaining to requesting a service, enabling the separation of Saga steps. The state diagram of Saga Transaction is represented in Figure 3.3.

The SUCCEED state has been designated as a *semi-end state*. Broadly speaking, in the context of this work, this state is considered as an end state, however, it can be modified in a single scenario: during Saga roll-backing.

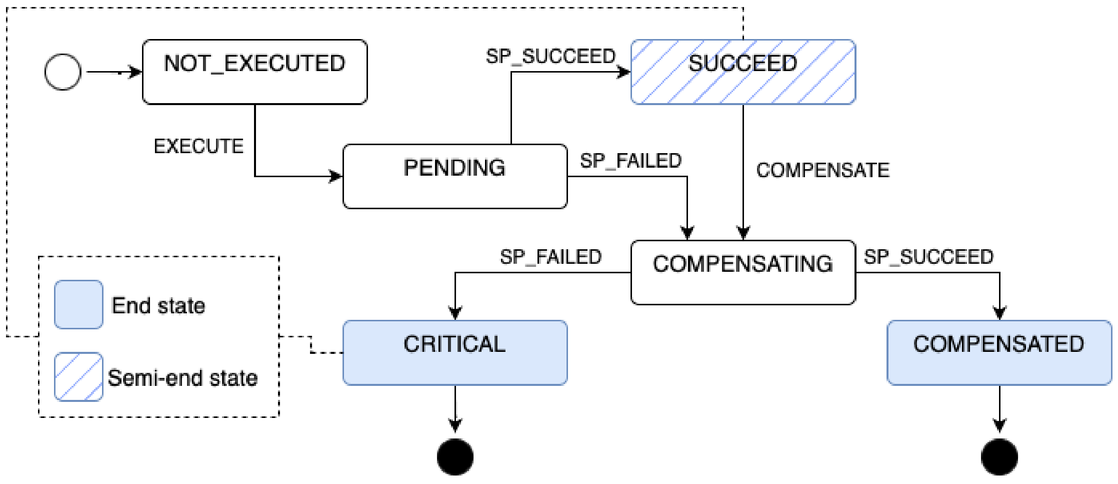

Figure 3.3: Saga transaction state diagram

Service calls under a saga transactions are created and running in the scope of PENDING and COMPENSATION states. That means, that the responsibility of service call creation is on a saga transaction.

3.1.4 Saga

Finally, introducing previous abstractions allows us to develop a comprehensive Saga state diagram design. This abstraction is the most important. All saga transactions under a saga are created during saga creation and running in the scope of PENDING and COMPENSATION states. The Figure 3.4 depicts the Saga state machine diagram.

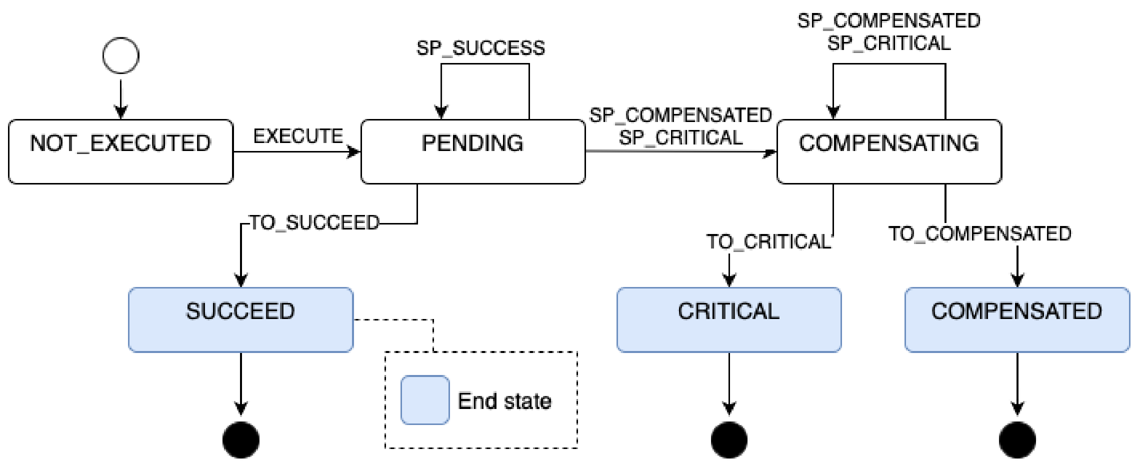

Figure 3.4: Saga state diagram

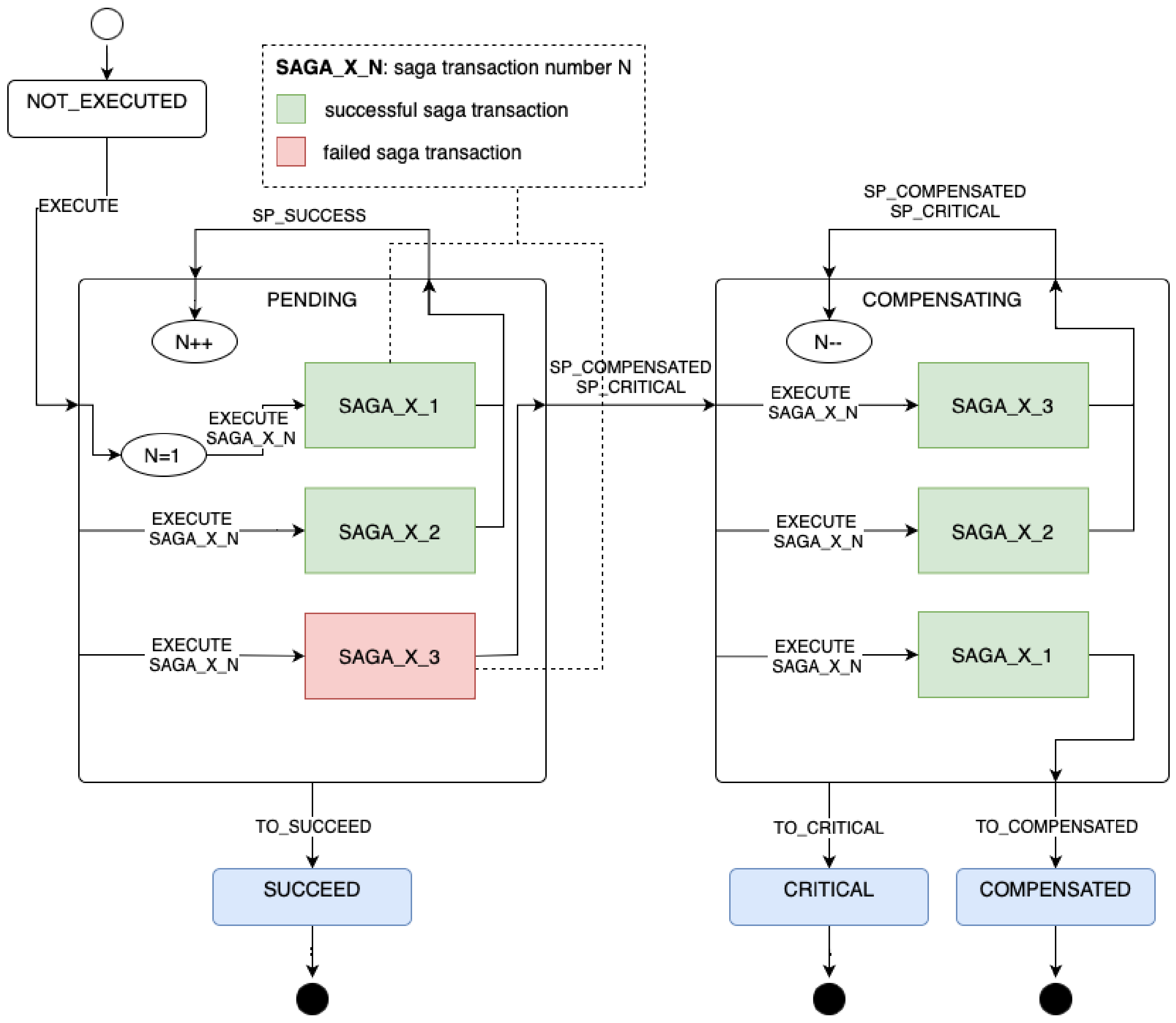

Figure 3.5: How it works together: Saga perspective

3.2 How it works together

To illustrate the functioning of the entire mechanism, let's consider saga state diagram extended by particular saga transactions. In Figure 3.5 is depicted a scenario, when the first two saga transactions passes and the third one fails. Following the compensating stage will successfully roll back all of them.

It should be noted that we have internal transitions, denoted as PENDING-to-PENDING and COMPENSATION-to-COMPENSATING. To distinguish between state changes, a counter N has been introduced. This counter functions as a pointer that identifies which Saga transaction is to be executed following the subsequent entry into either the PENDING or COMPENSATING states.

Let us proceed by presenting a state diagram of Saga transactions, augmented by service calls.

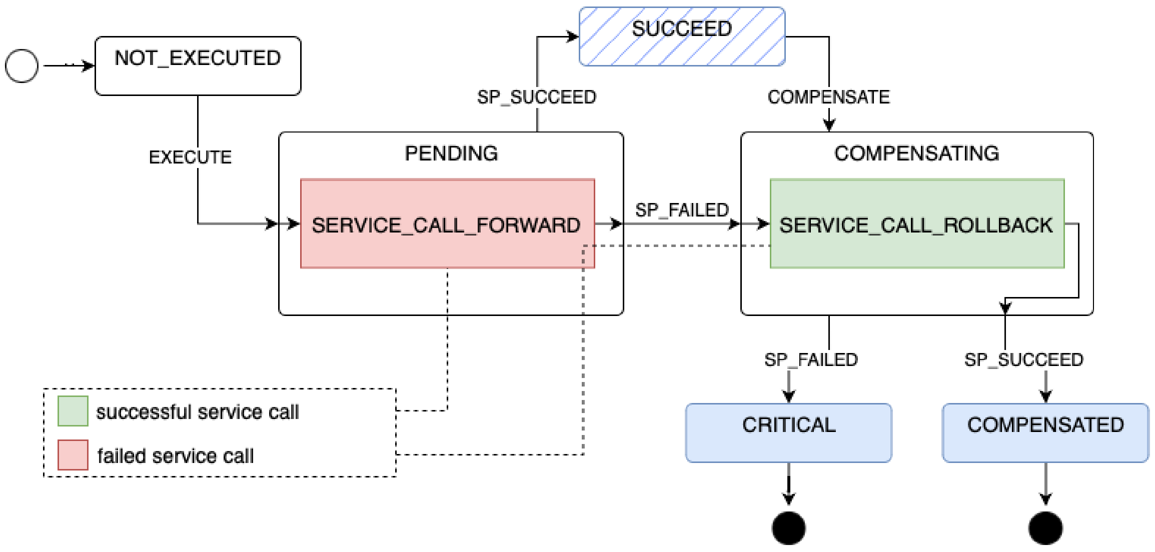

Figure 3.6: How it works together: Saga Transaction perspective

The scenario under consideration in Figure 3.6 proceeds as follows: a saga transaction initiates the first service call, which ultimately fails. Subsequently, it triggers a compensatory service call, which successfully rolls back the previous service call.

In both two examples, we can see that the end state of a downstream state machine serves as a transition event for the upstream one.

3.3 Proof of concept: Design

We shall now examine a simplified, practical instance of Saga implementation. Specifically, we shall assume an application comprising four microservices: A, B, C, and Saga Orchestrator. A corresponding component diagram is presented in Figure 3.7. In an effort to encapsulate inter-service call details, we shall introduce two straightforward abstractions: Response and Request. These abstractions are characterized by a Service Call ID and a content field, which is typed as an Object, meaning that it can assume any form. It is essential to adopt a generalized content type, as it enhances the re-usability of the Saga Orchestrator.

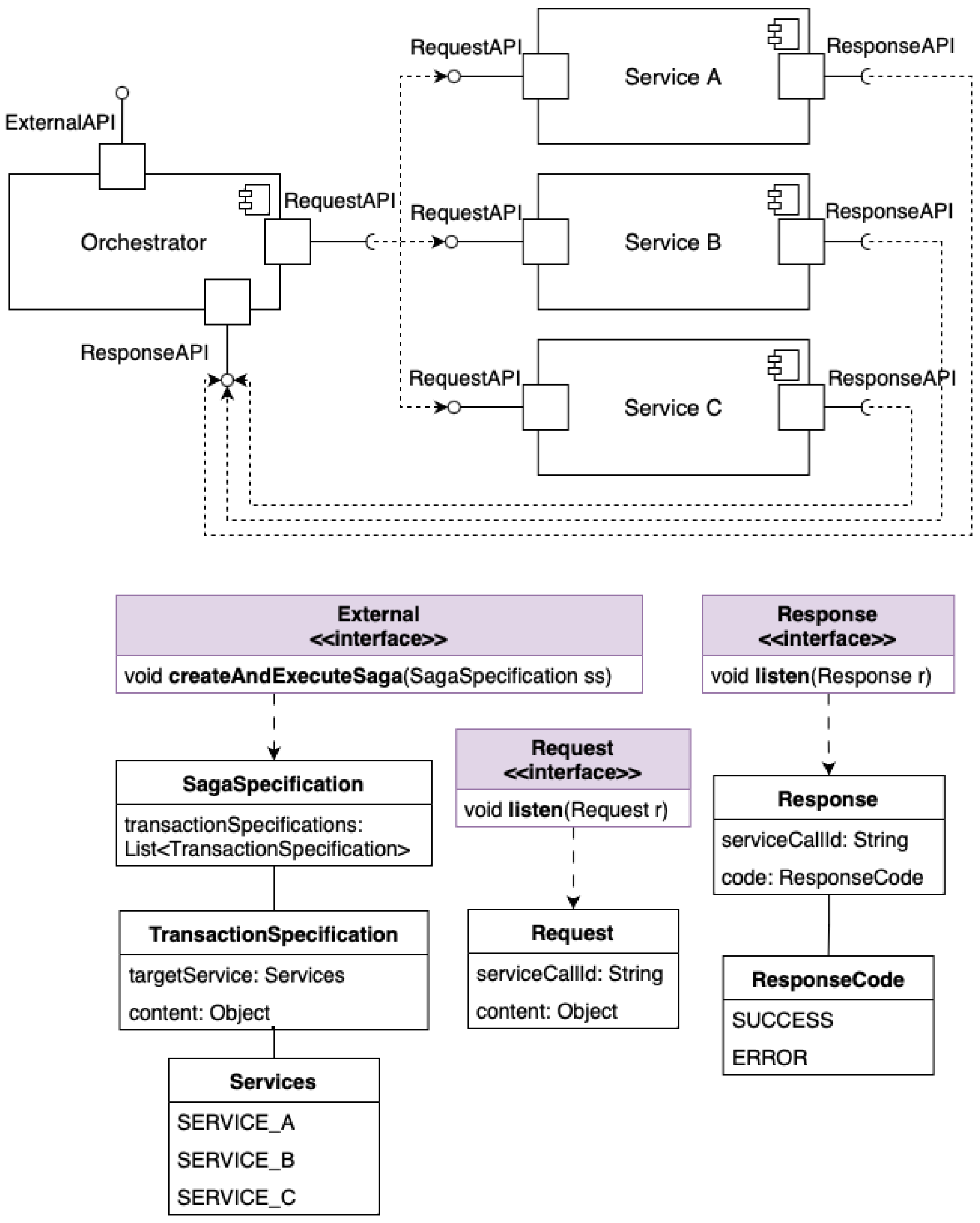

Figure 3.7: Example component diagram

The Saga pattern postulates the asynchronous messaging between services. The subsequent illustration 3.8 demonstrates the architecture of the message broker.

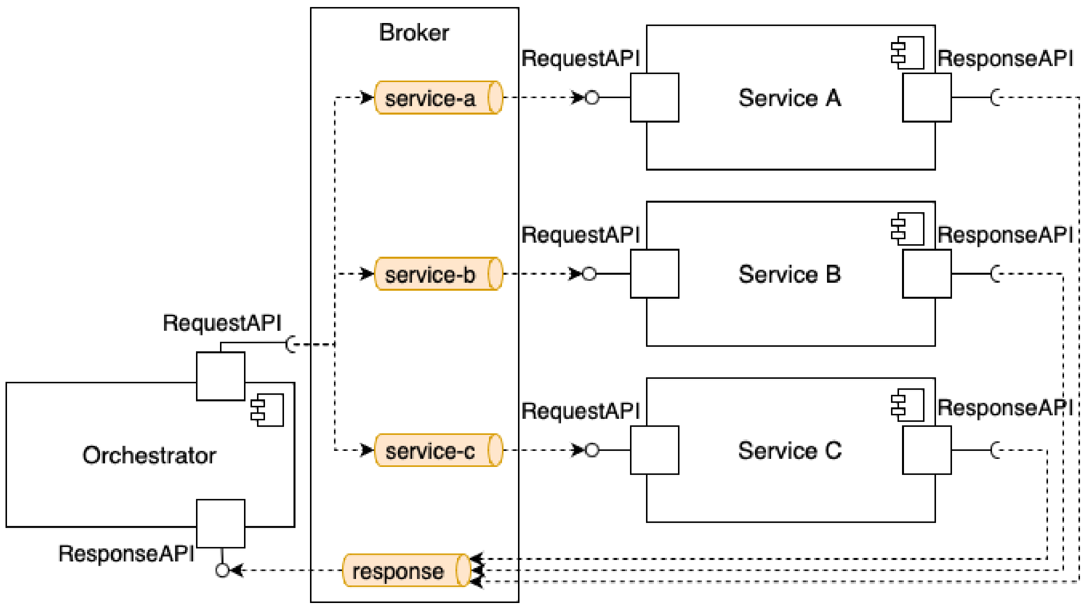

Figure 3.8: Example component broker diagram

As illustrated in the schematic, each participating service possesses a distinct input topic designated by the service's name. In other words, the communication from the Orchestrator to the service is processed through the corresponding service topic. Conversely, interactions from the service to the Orchestrator employ a singular topic, denoted as "Response". The determination of topic allocation is predicated on the type of message and the number of subscribers involved. The guiding principle stipulates that a distinct topic shall be established for each message category and individual subscriber, thereby enabling the delegation of message distribution responsibilities to the message broker.

Chapter 4

Implementation

4.1 Technology stack

To meet the prescribed prerequisites, it is imperative to incorporate external libraries. The forthcoming application will be built on the Spring Boot platform, with a primary emphasis on leveraging the Spring StateMachine framework for its core functionality. In order to proficiently handle asynchronous messages, the utilization of the Spring Kafka extension is essential. Additionally, the inclusion of Spring Data is imperative to facilitate the seamless execution of our Sagas. Lombok is also part of the dependency list because it aids in streamlining Java boilerplate code through the use of annotations. A comprehensive Listing of the project's POM[1] dependencies is provided in Listing 7.1 of Appendix A

4.2 State machines configuration

Utilizing Spring Boot configurations, the Spring StateMachine framework provides an efficient method for setting up state machines. This section delves into the Spring StateMachine configuration of Service Call, Saga Transaction, and Saga state machines, elucidating their respective functionalities and implementation details. Basically, the way of the configuration of each state machine is pretty much the same. By this configuration, we only build a model of the state machine. For this purpose, we have to provide a set of states and mark these states by their role. It could be an initial state or an end state.

4.2.1 Service Call

In order to define a state machine within the Spring StateMachine framework, three essential configurations need to be provided: state configuration, transition config-

[1]Project Object Model

uration, and persistence configuration. The following Listing 4.1 shows the configuration file.

Listing 4.1: Service Call state machine configuration

```java
public void configure( // Persistence configuration
    StateMachineConfigurationConfigurer<ServiceCallStates,
        ServiceCallEvents> config
    ) throws Exception {
    config.withPersistence()
          .runtimePersister( stateMachineRuntimePersister );
}

public void configure( // State configuration
    StateMachineStateConfigurer<ServiceCallStates, ServiceCallEvents>
        states
    ) throws Exception {
    states.withStates()
          .initial( NOT_EXECUTED )
          .state( PENDING )
          .end( SUCCEED )
          .end( FAILED )
          .stateEntry( SUCCEED, serviceCallPropagateSucceed )
          .stateEntry( FAILED, serviceCallPropagateFailed );
}

public void configure( // Transition configuration
    StateMachineTransitionConfigurer<ServiceCallStates,
        ServiceCallEvents> transitions
    ) throws Exception {
    transitions
            .withExternal()
                .source( NOT_EXECUTED )
                .target( PENDING )
                .event( EXECUTE )
                .action( callService ).and()
            .withExternal()
                .source( PENDING )
                .target( FAILED )
                .event( SP_FAILED ).and()
            .withExternal()
                .source( PENDING )
                .target( SUCCEED )
                .event( SP_SUCCEED );
}
```

Note, that Spring StateMachine differentiates 2 types of places from which actions are being invoked. There are so-called *entry actions* and *transition actions*. Following the idea of this work, all end states start with a state propagation. The *callService* action is defined as a transition action, bean definitions of actions are described in detail in subsection 4.2.5. Evidently, it becomes imperative to define certain beans, namely actions and state machine runtime persister (will be defined in section 4.2.4).

4.2.2 Saga Transaction

The uniformity of state machine configuration is maintained throughout the entire application, courtesy of Spring StateMachine. In the next Listing 4.2 we will show all three state machine configuration definitions.

Listing 4.2: Saga Transaction state machine configuration

```
public void configure( // Persistence configuration
StateMachineConfigurationConfigurer<SagaTransactionStates,
    SagaTransactionEvents> config
) throws Exception {
    config.withPersistence()
          .runtimePersister( sagaTransactionStateMachineRuntimePersister );
}

public void configure( // State configuration
StateMachineStateConfigurer<SagaTransactionStates, SagaTransactionEvents>
    states
) throws Exception {
    states.withStates()
          .initial( NOT_EXECUTED )
          .state( PENDING )
          .state( COMPENSATING )
          .state( SUCCEED )
          .end( CRITICAL )
          .end( COMPENSATED )
          .stateEntry( SUCCEED, sagaTransactionPropagateSucceed )
          .stateEntry( COMPENSATED, sagaTransactionPropagateCompensated )
          .stateEntry( CRITICAL, sagaTransactionPropagateCritical );

}

public void configure( // Transition configuration
StateMachineTransitionConfigurer<SagaTransactionStates,
    SagaTransactionEvents> transitions
) throws Exception {
    transitions
            .withExternal()
                .source( NOT_EXECUTED )
                .target( PENDING )
                .event( EXECUTE )
                .action( executeForwardServiceCall ).and()
            .withExternal()
                .source( PENDING )
                .target( SUCCEED )
                .event( SP_SUCCEED ).and()
            .withExternal()
                .source( PENDING )
                .target( COMPENSATING )
                .event( SP_FAILED )
                .action( executeRollbackServiceCall ).and()
            .withExternal()
                .source( COMPENSATING )
                .target( CRITICAL )
                .event( SP_FAILED ).and()
            .withExternal()
                .source( COMPENSATING )
                .target( COMPENSATED )
                .event( SP_SUCCEED ).and()
            .withExternal()
                .source( SUCCEED )
                .target( COMPENSATING )
                .event( COMPENSATE )
                .action( executeRollbackServiceCall );
```

4.2.3 Saga

The configuration of the Saga state machine (depicted in Listing 4.3) is analogous to the previous two, with one minor distinction - the actions for the PENDING and COMPENSATING states are entry actions. In contrast, the Service Call and Saga Transaction only use entry actions for terminal states. For the Saga state machine, we must adopt a different methodology and allocate entry actions to non-terminal states. This is crucial since the executeNextSagaTransaction and compensateNextSagaTransaction actions necessitate the state machine to be in a defined state for execution.

Listing 4.3: Saga state machine configuration

```
public void configure( // Persistence configuration
StateMachineConfigurationConfigurer<SagaStates, SagaEvents> config
) throws Exception {
    config.withPersistence()
          .runtimePersister( sagaStateMachineRuntimePersister );
}

public void configure( // State configuration
StateMachineStateConfigurer<SagaStates, SagaEvents> states
) throws Exception {
    states.withStates()
          .initial( NOT_EXECUTED )
          .state( PENDING )
          .state( COMPENSATING )
          .end( CRITICAL )
          .end( SUCCEED )
          .end( COMPENSATED )
          .stateEntry( PENDING, executeNextSagaTransaction )
          .stateEntry( COMPENSATING, compensateNextSagaTransaction );
}

public void configure( // Transition configuration
StateMachineTransitionConfigurer<SagaStates, SagaEvents> transitions
) throws Exception {
    transitions
            .withExternal()
                .source( NOT_EXECUTED )
                .target( PENDING )
                .event( EXECUTE ).and()
            .withExternal()
                .source( PENDING )
                .target( PENDING )
                .event( SP_SUCCEED ).and()
            .withExternal()
                .source( PENDING )
                .target( SUCCEED )
                .event( TO_SUCCEED ).and()
            .withExternal()
                .source( PENDING )
                .target( COMPENSATING )
                .event( SP_COMPENSATED ).and()
            .withExternal()
                .source( PENDING )
                .target( COMPENSATING )
                .event( SP_CRITICAL ).and()
            .withExternal()
                .source( COMPENSATING )
                .target( COMPENSATING )
                .event( SP_COMPENSATED ).and()
            .withExternal()
                .source( COMPENSATING )
                .target( COMPENSATING )
                .event( SP_CRITICAL ).and()
            .withExternal()
                .source( COMPENSATING )
                .target( CRITICAL )
                .event( TO_CRITICAL ).and()
            .withExternal()
                .source( COMPENSATING )
                .target( COMPENSATED )
                .event( TO_COMPENSATED );
}
```

4.2.4 Persistence

To maintain the persistence characteristics of our system, it is necessary to incorporate a mechanism that enables state machines to persist. Fortunately, the Spring StateMachine framework natively supports this particular feature. An example of such configurations is shown in Listing 4.4

Listing 4.4: State machines persistence configuration

```
@Bean
```

```
public StateMachineRuntimePersister<States, Events, String>
    sagaTransactionStateMachineRuntimePersister(
        JpaStateMachineRepository jpaStateMachineRepository
) {
    return new JpaPersistingStateMachineInterceptor<>(
        jpaStateMachineRepository );
}

@Bean
public StateMachineService<States, Events>
    sagaTransactionStateMachineService(
        StateMachineFactory<States, Events> stateMachineFactory,
        StateMachineRuntimePersister<States, Events, String>
            stateMachineRuntimePersister
) {
    return new DefaultStateMachineService<>( stateMachineFactory,
        stateMachineRuntimePersister );
}
```

Basically, all three different state machines will have such 2 beans each. We will provide only one definition as the only thing changes is the state enumeration and event enumeration in diamond brackets (e.g. Saga will have <SagaStates, SagaEvents>. StateMachineService defined in the Listing above is provided by Spring StateMachine and serves as a sycnhonizing service and cache at the same time.

4.2.5 Actions

As was mentioned previously, there are two different types of action, actions that flow from an upstream to the downstream state machine (e.g. from Saga to Saga Transaction), and actions flows from a downstream state machine to the upstream one. All 3 action configurations for each state machine look pretty much the same at this point. Let's look at Listing 4.5

Listing 4.5: Service Call action configuration

```
@Bean
public Action<ServiceCallStates, ServiceCallEvents> callService() {
    return serviceCallActionService::callService;
}

@Bean
public Action<ServiceCallStates, ServiceCallEvents>
    serviceCallPropagateSucceed() {
    return stateContext -> {
        var stId = getSagaTransactionId( stateContext.getExtendedState() );
        applicationEventPublisher.publishEvent( new
            StatePropagationToSagaTransaction( this.getClass(), SUCCEED,
            stId ) );
    };
}
```

```
@Bean
public Action<ServiceCallStates, ServiceCallEvents>
   serviceCallPropagateFailed() {
   return stateContext -> {
      var stId = getSagaTransactionId( stateContext.getExtendedState() );
      applicationEventPublisher.publishEvent( new
         StatePropagationToSagaTransaction( this.getClass(), FAILED,
         stId ) );
   };
}
```

This is the definition of actions for the Service Call state machine. The actions *serviceCallPropagateSucceed()* and *serviceCallPropagateFailed()* are state propagation actions that direct the flow towards an upstream state machine, in this case, the Saga Transaction. Essentially, all actions aimed at an upstream state machine merely redirect the relevant event to a message queue. This event is then intercepted by the StatePropagationHandler, which will be further elaborated in the subsequent section 4.3. Another type of actions directed to the downstream state machines is simply call the relevant state machine service. These services will be described later in section 4.4.

4.3 StatePropagationHandler

As alluded to earlier, the state propagation mechanism is conceived and structured to mitigate the issue of circular dependencies [21], a common challenge in system design. The root cause of this predicament stems from the methodology we have adopted, particularly, the high degree of inter-connectivity between state machines. This scenario results in an intertwined network of dependencies, which can lead to the manifestation of various system complexities. For further elaboration, we will delve into diagram 4.2. This visual representation elucidates the intricate interdependencies that emerge in the absence of a state propagation layer, thereby highlighting the critical role such a layer plays in preventing the creation of cyclical dependencies.

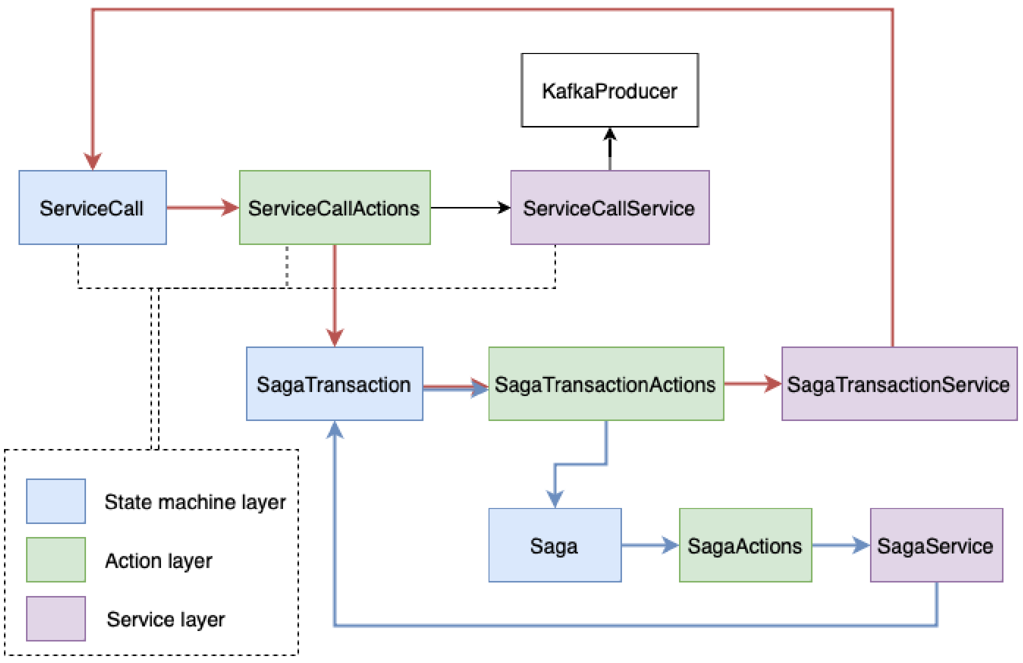

Figure 4.1: Circular dependency diagram

The circles, spawned by this approach are indicated by red and blue colors. To resolve this, we will use native Spring messaging support.

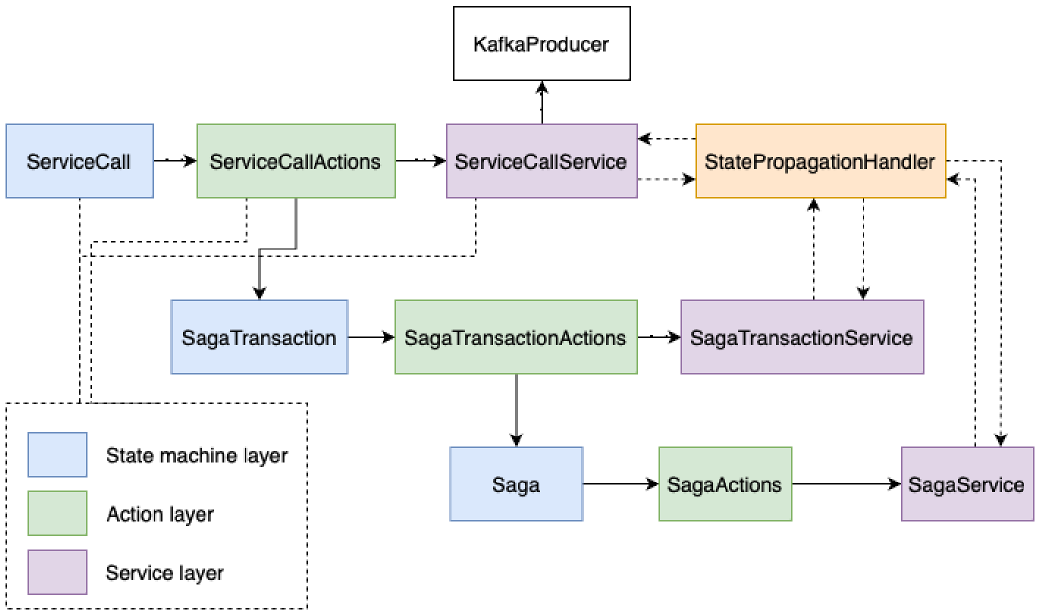

Figure 4.2: Circular dependency diagram: resolved

The subsequent Listing 4.6 provides a representation of the StatePropagationService's structure and composition.

Listing 4.6: State Propagation service

```
private final Map<ServiceCallResponseCode, ServiceCallEvents>
    serviceCallStatePropagationMap = Map.of(
        ServiceCallResponseCode.SUCCESS, ServiceCallEvents.SP_SUCCEED,
        ServiceCallResponseCode.ERROR, ServiceCallEvents.SP_FAILED
);

private final Map<ServiceCallStates, SagaTransactionEvents>
    sagaTransactionStatePropagationMap = Map.of(
        ServiceCallStates.SUCCEED, SagaTransactionEvents.SP_SUCCEED,
        ServiceCallStates.FAILED, SagaTransactionEvents.SP_FAILED
);

private final Map<SagaTransactionStates, SagaEvents>
    sagaStatePropagationMap = Map.of(
        SagaTransactionStates.SUCCEED, SagaEvents.SP_SUCCEED,
        SagaTransactionStates.COMPENSATED, SagaEvents.SP_COMPENSATED,
        SagaTransactionStates.CRITICAL, SagaEvents.SP_CRITICAL
);

@EventListener // from KafkaListener to ServiceCall
public void statePropagationToServiceCall( StatePropagationToServiceCall
    statePropagation ) {
    serviceCallStatemachineService
            .acquireStateMachine(
               statePropagation.getTargetStatemachineId() )
            .sendEvent( just( withPayload(
               serviceCallStatePropagationMap.get(
               statePropagation.getStatePropagationCode() ) ).build() ) )
            .blockLast();
}

@EventListener // from ServiceCall to SagaTransaction
public void statePropagationToSagaTransaction(
    StatePropagationToSagaTransaction statePropagation ) {
    sagaTransactionStatemachineService
            .acquireStateMachine(
               statePropagation.getTargetStatemachineId() )
            .sendEvent( just( withPayload(
               sagaTransactionStatePropagationMap.get(
               statePropagation.getStatePropagationCode() ) ).build() ) )
            .blockLast();
}

@EventListener // from SagaTransaction to Saga
public void statePropagationToSaga( StatePropagationToSaga
    statePropagation ) {
    sagaStatemachineService
            .acquireStateMachine(
               statePropagation.getTargetStatemachineId() )
            .sendEvent( just( withPayload( sagaStatePropagationMap.get(
               statePropagation.getStatePropagation() ) ).build() ) )
            .blockLast();
}
```

The aforementioned Listing predictably features three Spring Messaging event listeners, namely KafkaListener-to-ServiceCall, ServiceCall-to-SagaTransaction, and SagaTransaction-to-Saga. As elucidated earlier, the KafkaListener-to-ServiceCall is not classified as a 'pure' state propagation since KafkaListener does not qualify as a state machine. However, to accommodate a more generalized definition of a state machine, this departure from the norm has been permitted. An additional intriguing feature to note from the aforementioned Listing pertains to event maps. Considering that state machines dispatch their pertinent end states to the StatePropagationService, it is imperative that these end states are accurately mapped to the corresponding upstream transition events. This function is fulfilled by the maps situated at the commencement of the Listing, underscoring their importance in ensuring the correct alignment between end states and transition events.

4.4 State machine services

This level of abstraction functions as the business logic for each specific state machine, with the exception of the state propagation mechanism which is common to all state machines. All these services utilize the StateMachineService bean, an aggregated component provided by the Spring StateMachine framework (more about this service in 4.2.4). In our project, this layer is comprised of two categories of services for each state machine. The first category follows the naming convention of <state-machine-name>Service and is responsible for managing the lifecycle of the state machine. The second category, conforming to the <state-machine-name>ActionService naming convention, is tasked with handling all the residual business logic specific to that state machine.

4.4.1 ServiceCall Services

The following Listing 4.7 shows both types of services for the ServiceCall state machine:

Listing 4.7: Service Call service and Service Call action service

```java
@Service
@RequiredArgsConstructor
public class ServiceCallService {

    private final StateMachineService<ServiceCallStates,
        ServiceCallEvents> serviceCallStatemachineService;

    public StateMachine<ServiceCallStates, ServiceCallEvents>
        buildServiceCall( String sagaTransactionId, Services
        targetService, Object content ) {
        var scSm = serviceCallStatemachineService.acquireStateMachine(
            randomUUID().toString() );
        setSagaTransactionId( scSm.getExtendedState(), sagaTransactionId );
        setTargetService( scSm.getExtendedState(), targetService );
        setContent( scSm.getExtendedState(), content );
        return scSm;
    }
}

@Service
@RequiredArgsConstructor
public class ServiceCallActionService {

    private final ServiceRequestSender serviceRequestSender;

    public void callService( StateContext<ServiceCallStates,
        ServiceCallEvents> stateContext ) {
        var scId = stateContext.getStateMachine().getId();
        var targetService = getTargetService(
            stateContext.getExtendedState() );
        var content = getContent( stateContext.getExtendedState() );
        var request = new ServiceCallRequestDto( scId, content );
        serviceRequestSender.callService( targetService, request );
    }

}
```

It's important to note that each state machine maintains its data storage. This data can be accessed using the following method: sm.getExtendedState().getVariables(), which returns a map of <VariableName, VariableValue>. To make this potentially cumbersome access more user-friendly, static utility access classes have been implemented for each state machine.

Another noteworthy aspect is the ServiceRequestSender dependency. This simply dispatches a message to a selected service via Kafka. In simpler terms, the 'callService' action specified in the ServiceCall state machine configuration offloads the task of action processing to the ServiceCallActionService. This service then employs the ServiceRequestSender to dispatch the given message to the appropriate service.

4.4.2 SagaTransaction Services

The following Listing 4.8 depicts the way the service layer is implemented for SagaTransaction state machine:

Listing 4.8: Saga Transaction service and Saga Transaction action service

```
@Service
@RequiredArgsConstructor
public class SagaTransactionService {

    private final StateMachineService<SagaTransactionStates,
        SagaTransactionEvents> sagaTransactionStatemachineService;
    public StateMachine<SagaTransactionStates, SagaTransactionEvents>
        buildSagaTransaction( TransactionSpecification
        transactionSpecification, String sagaId ) {
        var stSm = sagaTransactionStatemachineService.acquireStateMachine(
            randomUUID().toString() );
        setSagaId( stSm.getExtendedState(), sagaId );
        setTargetService( stSm.getExtendedState(),
            transactionSpecification.getTargetService() );
        setPrimaryContent( stSm.getExtendedState(),
            transactionSpecification.getPrimaryContent() );
        setRollbackContent( stSm.getExtendedState(),
            transactionSpecification.getRollbackContent() );
        return stSm;
    }

    public StateMachine<SagaTransactionStates, SagaTransactionEvents>
        findById( String id ) {
        return sagaTransactionStatemachineService.acquireStateMachine( id
            );
    }

}

@Service
@RequiredArgsConstructor
public class SagaTransactionActionService {

    private final ServiceCallService serviceCallService;

    public void executeForwardServiceCall(
        StateContext<SagaTransactionStates, SagaTransactionEvents>
        stateContext ) {
        var stSmId = stateContext.getStateMachine().getId();
        var targetService = getTargetService(
            stateContext.getExtendedState() );
        var content = getPrimaryContent( stateContext.getExtendedState() );
        serviceCallService
                .buildServiceCall( stSmId, targetService, content )
                .sendEvent( just( withPayload( EXECUTE ).build() ) )
                .blockLast();
    }

    public void executeRollbackServiceCall(
        StateContext<SagaTransactionStates, SagaTransactionEvents>
        stateContext) {
        var stSmId = stateContext.getStateMachine().getId();
        var targetService = getTargetService(
            stateContext.getExtendedState() );
        var content = getRollbackContent( stateContext.getExtendedState()
            );
        serviceCallService
                .buildServiceCall( stSmId, targetService, content )
                .sendEvent( just( withPayload( EXECUTE ).build() ) )
                .blockLast();
    }

}
```

The implementation approach for SagaTransactionService is quite similar to that for ServiceCall. However, SagaTransactionActionService exhibits more complex behavior than ServiceCall. This action service enables us to execute a forward ServiceCall by invoking the relevant function and also executin a rollback ServiceCall.

4.4.3 Saga Services

The subsequent illustration 4.9 demonstrates how the service layer is implemented for the Saga state machine:

Listing 4.9: Saga Service and Saga action service

```
@Service
@RequiredArgsConstructor
public class SagaService {

    private final StateMachineService<SagaStates, SagaEvents>
        sagaStatemachineService;

    private final SagaTransactionService sagaTransactionService;

    public StateMachine<SagaStates, SagaEvents> buildAndStartSaga(
        SagaSpecification sagaSpecification ) {
        var sagaSm = sagaStatemachineService.acquireStateMachine(
            randomUUID().toString() );
        var stIdList = sagaSpecification
                .getTransactionSpecifications()
                .stream()
                .map( ts -> sagaTransactionService.buildSagaTransaction(
                    ts, sagaSm.getId() ) )
                .map( StateMachine::getId )
                .collect( toList() );

        setTransactionIdList( sagaSm.getExtendedState(), stIdList );
        setRunningTransactionOrder( sagaSm.getExtendedState(), -1 );
        setTotalTransactionsCount( sagaSm.getExtendedState(),
            sagaSpecification.getTransactionSpecifications().size() );

        sagaSm.sendEvent( just( withPayload( EXECUTE ).build() )
            ).blockLast();

        return sagaSm;
    }
```

```java
}

@Service
@RequiredArgsConstructor
public class SagaActionService {

    private final SagaTransactionService sagaTransactionService;

    public void executeNextSagaTransaction( StateContext<SagaStates,
        SagaEvents> stateContext ) {
        var stSmNextOrder = getRunningTransactionOrder(
            stateContext.getExtendedState() ) + 1;
        var stSmTotalXAmount = getTotalTransactionsCount(
            stateContext.getExtendedState() );
        if ( stSmTotalXAmount == stSmNextOrder ) {
            stateContext.getStateMachine().sendEvent( just( withPayload(
                TO_SUCCEED ).build() ) ).blockLast();
        } else {
            sagaTransactionService
                    .findById( getTransactionIdList(
                        stateContext.getExtendedState() ).get(
                        stSmNextOrder ) )
                    .sendEvent( just( withPayload( EXECUTE ).build() ) )
                    .blockLast();
            setRunningTransactionOrder( stateContext.getExtendedState(),
                stSmNextOrder );
        }
    }

    public void compensateNextSagaTransaction( StateContext<SagaStates,
        SagaEvents> stateContext ) {
        var stSmNextOrder = getRunningTransactionOrder(
            stateContext.getExtendedState() ) - 1;
        if ( stSmNextOrder == -1 ) {
            stateContext.getStateMachine().sendEvent( just( withPayload(
                TO_COMPENSATED ).build() ) ).blockLast();
        } else {
            sagaTransactionService
                    .findById( getTransactionIdList(
                        stateContext.getExtendedState() ).get(
                        stSmNextOrder ) )
                    .sendEvent( just( withPayload( COMPENSATE ).build() ) )
                    .blockLast();
            setRunningTransactionOrder( stateContext.getExtendedState(),
                stSmNextOrder );
        }
    }

}
```

The SagaService maintains the trend and handles the lifecycle component. A key distinction from other lifecycle services is its ability to create downstream state machines based on input parameters. The SagaActionService enables the execution of the next SagaTransaction or the compensation of a SagaTransaction, which is quite suitable for our needs. For a clearer understanding, please refer back to figure 3.4.

Chapter 5

Testing

The system discussed in this paper is sufficiently complex and requires thorough testing. A number of unit tests were introduced during the development phase, but they are not sufficient for a system of this scale. Furthermore, testing within a microservice architecture demands a unique approach, like service component testing [22].

5.1 Testing approach

The service component testing approach eliminates the need to develop a real microservice for testing another microservice. As a result, we will create stubs for each service (Service A, Service B, Service C). After configuring these stubs to respond as anticipated, we will achieve a state where we can conduct end-to-end testing of our system, since we are decoupled from the other services. This approach is depicted in Figure 5.1.

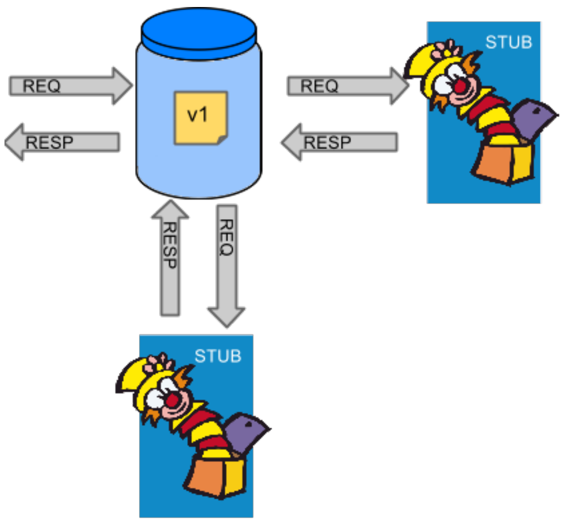

Figure 5.1: Service Component Testing: Stubs [23]

Fortunately, there is a framework, Spring Contract, that enables the creation of stubs, even when using Kafka messaging. Essentially, in Spring Contract we define a contract that afterward will be tested on both API[1] consumer and API producer sides. On the API producer side, we primarily simulate an input call to our microservice and validate the output message. On the API consumer side, our approach to testing involves creating a message listener configured according to the contract. This listener intercepts the message and if it aligns with the contract, it sends a response message defined in the same contract back. The sequence diagram of that process is shown in Figure 5.2

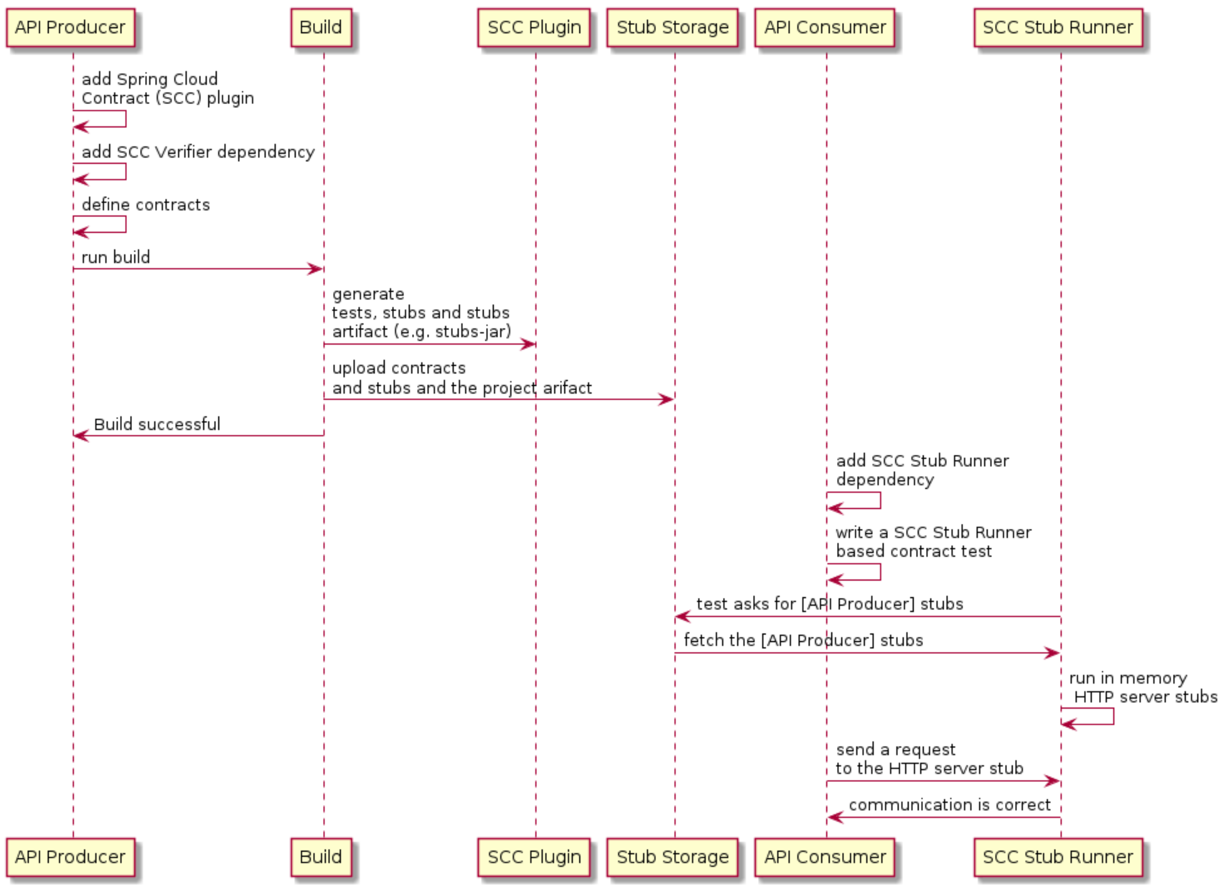
API Producer
Build
SCC Plugin
Stub Storage
API Consumer
SCC Stub Runner
add Spring Cloud Contract (SCC) plugin
add SCC Verifier dependency
define contracts
run build
generate tests, stubs and stubs artifact (e.g. stubs-jar)
upload contracts and stubs and the project arifact
Build successful
add SCC Stub Runner dependency
write a SCC Stub Runner based contract test
test asks for [API Producer] stubs
fetch the [API Producer] stubs
run in memory HTTP server stubs
send a request to the HTTP server stub
communication is correct

Figure 5.2: Spring Contract: Sequence diagram [23]

5.2 Technology stack

Another module, 'Contract', has been introduced specifically for testing. This module will act as a mock for all three services (Service A, Service B, Service C). Additionally, the Contract module has the responsibility of deploying contract stubs. The deployment model is depicted in Figure 5.3.

[1]Application Programming Interface

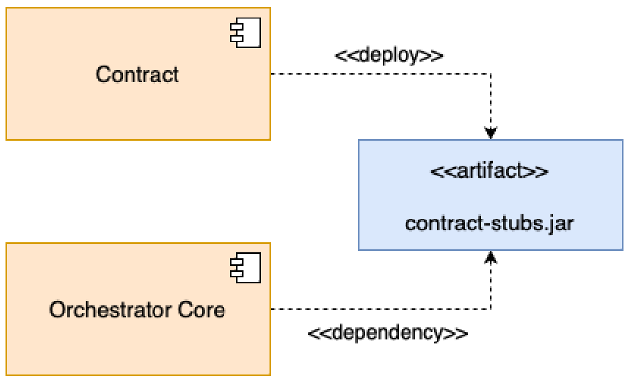

Figure 5.3: Deployment Model: Testing

The Contract module employs two key technologies: Spring Boot and Spring Contract. All other dependencies are derived from these two. The Contract module includes the following dependency in its Project Object Model (POM) is represented in Listing 7.2 of Appendix A

5.3 Spring Cloud Contract modification

Now, let's examine the contracts themselves. The subsequent Listing 5.1 presents simple Kafka message contracts:

Listing 5.1: Plain Spring Cloud contract

```
make {
    input {
        messageFrom('service-a')
        messageBody([
                serviceCallId: 1,
                content: 'AMOGUS'
        ])
        messageHeaders {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallRequestDto')
        }
    }
    outputMessage {
        sentTo('orchestrator')
        body(
                serviceCallId: 1,
                code: 'ERROR'
        )
        headers {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallResponseDto')
        }
    }
},
make {
    input {
        messageFrom('service-a')
        messageBody([
                serviceCallId: 1,
                content: $(consumer(regex('^(?:(?!AMOGUS).)*$')),
                    producer('CONTENT'))
        ])
        messageHeaders {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallRequestDto')
        }
    }
    outputMessage {
        sentTo('orchestrator')
        body(
                serviceCallId: 1,
                code: 'SUCCESS'
        )
        headers {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallResponseDto')
        }
    }
}
```

Each contract initiates with an input block where the topic (messageFrom), message content (messageBody), and message headers are specified. Following the input block is the output message block, where the topic (sentTo), message content (body), and headers are also defined. This contract indicates that for an input that matches the input block, a message matching the output message block will be returned. Take note that these contracts define the messageBody block only for service calls with an ID of 1. Essentially, this implies that any other service call ID will be dismissed, and therefore, no output message will be dispatched.

Nonetheless, our aim is to induce a bit complex behavior. We aspire to broaden the input and output definitions to enhance the functionality of mocks. This can be achieved by introducing a new stipulation to our contract:

$$outputMessage.body.serviceCallId = input.messageBody.serviceCallId$$

Regrettably, Spring Contract doesn't accommodate this feature as its contracts are not designed to tackle such concerns [24]. For this reason, we plan to make minor adjustments to the Spring Contract library. Take a look at the following Listing 5.2 updated contract:

Listing 5.2: Modified Spring Cloud contract

```
make {
    input {
        messageFrom('service-a')
        messageBody([
                serviceCallId: $(consumer(regex('[A-Za-z-_0-9]+')),
                    producer('SERVICE_CALL_ID')),
                content: 'AMOGUS'
        ])
        messageHeaders {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallRequestDto')
        }
    }
    outputMessage {
        sentTo('orchestrator')
        body(
                serviceCallId: $(consumer('$fromInput(serviceCallId)'),
                    producer('SERVICE_CALL_ID')),
                code: 'ERROR'
        )
        headers {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallResponseDto')
        }
    }
},
make {
    input {
        messageFrom('service-a')
        messageBody([
                serviceCallId: $(consumer(regex('[A-Za-z-_0-9]+')),
                    producer('SERVICE_CALL_ID')),
                content: $(consumer(regex('^(?:(?!AMOGUS).)*$')),
                    producer('CONTENT'))
        ])
        messageHeaders {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallRequestDto')
        }
    }
    outputMessage {
        sentTo('orchestrator')
        body(
                serviceCallId: $(consumer('$fromInput(serviceCallId)'),
                    producer('SERVICE_CALL_ID')),
                code: 'SUCCESS'
        )
        headers {
            header('__TypeId__',
                'cz.cvut.fel.orchestrator.api.ServiceCallResponseDto')
        }
    }
}
```

The main difference is that now we introduce consumer/producer values differentiation. The expression '$(consumer(value1), producer(value2))' indicates that *value1* will be utilized for the consumer-side test, while *value2* will be employed for the producer-side test. We won't need a universal matcher on the producer side, as our main focus is on the API consumer. Contracts allows us to match strings using regular expressions [25], the syntax is represented by 'regex()' block. The correlation id logic is implemented through the use of placeholders. Consider the service call id in the output message for the consumer. The '(())' formation, which isn't inherently supported by Spring Contract, is utilized here. This is the segment that we aim to manipulate to attain the correlation id behavior. Since this notation is exclusively employed for the consumer side, let's bring in revised Spring Contract configurations for our orchestrator module (the API consumer) represented on Listing 5.3

Listing 5.3: StubRunnerKafkaTransformer modification

```java
public class CustomStubRunnerKafkaTransformer {
    ...
    public Message<?> transform( Contract groovyDsl, Message<?>
        referenceMessage ) {
        Object outputBody = outputBody(groovyDsl);
        // the point of use insertReferences()
        outputBody = insertReferences( outputBody, referenceMessage );
        Map<String, Object> headers =
            groovyDsl.getOutputMessage().getHeaders().asStubSideMap();
        Message newMessage = MessageBuilder.createMessage(outputBody, new
            MessageHeaders(headers));
        this.selector.updateCache(newMessage, groovyDsl);
        return newMessage;
    }
    ...

    // this function inserts placeholder 'fromInput' by the value
        specified in brackets
    private Object insertReferences( Object outputBody, Message<?>
        referenceMessage ) {
        try {
            var regex = "^\\$fromInput\\((.*)\\)$";
            var objectMapper = new ObjectMapper();
            var outputBodyMapping = objectMapper.readValue( (String)
                outputBody, HashMap.class );
            var referenceMessageBodyMapping = objectMapper.readValue(
                objectMapper.writeValueAsString(
                referenceMessage.getPayload() ), HashMap.class );

            for ( var key : outputBodyMapping.keySet() ) {
                var outputBodyValue = (String) outputBodyMapping.get( key
                    );
                if ( outputBodyValue.matches( regex ) ) {
                    var referenceMessageBodyMappingKey = substringBetween(
                        outputBodyValue, "(", ")" );
                    var referenceMessageBodyValue = (String)
                        referenceMessageBodyMapping.get(
                        referenceMessageBodyMappingKey );
                    outputBodyMapping.put( key, referenceMessageBodyValue );
                }
            }

            return new JSONObject( outputBodyMapping ).toString();
        } catch ( Exception e ) {
            return outputBody;
        }
    }
}
```

Given that we don't have the permission to replace or extend a specific function, we are compelled to rewrite the entire bean. Following this, we will expressly override the bean usage with the '@Import(CustomStubRunnerKafkaConfiguration.class)' annotation on the base test class on the consumer side.

5.3.1 Bean name issue

There is a bug in Spring Contract of 3.1.6 version found during the development. The issue occurs then we try to run the contract test on the consumer side using contracts which accepts more than one input message topics. The following Listing 5.4 shows the source code of a Spring Contract bean:

Listing 5.4: StubRunnerKafkaTransformer bug

```
...
public class StubRunnerKafkaConfiguration {
...
@Bean
@ConditionalOnMissingBean(name = "stubFlowRegistrar")
public FlowRegistrar stubFlowRegistrar(ConfigurableListableBeanFactory
    beanFactory,
        BatchStubRunner batchStubRunner) {
    Map<StubConfiguration, Collection<Contract>> contracts =
        batchStubRunner.getContracts();
    for (Entry<StubConfiguration, Collection<Contract>> entry :
        contracts.entrySet()) {
        StubConfiguration key = entry.getKey();
        Collection<Contract> value = entry.getValue();
        String name = key.getGroupId() + "_" + key.getArtifactId(); // name
        MultiValueMap<String, Contract> map = new LinkedMultiValueMap<>();
        for (Contract dsl : value) {
            if (dsl == null) {
                continue;
            }
            if (dsl.getInput() != null && dsl.getInput().getMessageFrom()
                != null
                    && StringUtils.hasText(
                        dsl
```

```
                         .getInput()
                         .getMessageFrom()
                         .getClientValue()
                         ) ) {
                String from =
                    dsl.getInput().getMessageFrom().getClientValue();
                map.add(from, dsl);
            }
        }
        for (Entry<String, List<Contract>> entries : map.entrySet()) {
            List<Contract> matchingContracts = entries.getValue();
            final String flowName = name + "_" + entries.getKey() + "_" +
                Math.abs(matchingContracts.hashCode()); // flowName
            StubRunnerKafkaRouter router = new
                StubRunnerKafkaRouter(matchingContracts, beanFactory);
            StubRunnerKafkaRouter listener = (StubRunnerKafkaRouter)
                beanFactory.initializeBean(router, flowName);
            ...
            beanFactory.registerSingleton(flowName, listener);
            registerContainers(beanFactory, matchingContracts, flowName,
                listener);
        }

    }
    return new FlowRegistrar();
}

private void registerContainers(ConfigurableListableBeanFactory
    beanFactory, List<Contract> matchingContracts,
         String flowName, StubRunnerKafkaRouter listener) {
      ConsumerFactory consumerFactory =
          beanFactory.getBean(ConsumerFactory.class);
      for (Contract matchingContract : matchingContracts) {
         if (matchingContract.getInput() == null) {
            continue;
         }
         String destination = MapConverter
                                .getStubSideValuesForNonBody(
                                    matchingContract
                                        .getInput()
                                        .getMessageFrom()
                                    )
                           .toString();
         ContainerProperties containerProperties = new
             ContainerProperties(destination);
         KafkaMessageListenerContainer container =
             listenerContainer(consumerFactory, containerProperties,
             listener);
         String containerName = flowName + ".container"; // container
             name: duplicate namings
         Object initializedContainer =
             beanFactory.initializeBean(container, containerName);
```

```
                beanFactory.registerSingleton(containerName,
                    initializedContainer);
                ...
            }
        }
    ...
}
```

This is where Spring Contract creates listeners for input topics in the input section of each contract. If you trace the construction process of the *containerName*, the issue of duplication will become evident. There's a straightforward solution to this issue: we'll adjust the *containerName* by incorporating the hash code of the corresponding contract into its name. The result of such manipulations is represented on Listing 5.5

Listing 5.5: StubRunnerKafkaTransformer bug fix

```
private void registerContainers(ConfigurableListableBeanFactory
    beanFactory, List<Contract> matchingContracts,
String flowName, CustomStubRunnerKafkaRouter listener) {
    ConsumerFactory consumerFactory =
        beanFactory.getBean(ConsumerFactory.class);
    for (Contract matchingContract : matchingContracts) {
        ...
        // contract hash code is added to the containerName
        String containerName = flowName + ".container" +
            matchingContract.hashCode();
        Object initializedContainer =
            beanFactory.initializeBean(container, containerName);
        beanFactory.registerSingleton(containerName, initializedContainer);
    }
}
```

5.4 Saga Flow Test

These preparatory steps enable us to conduct the intended test, which will examine the saga flow from end to end using a mock generated from the contract. Let's begin by showcasing the test configuration employed in this saga flow test:

Listing 5.6: Test configuration

```
@TestConfiguration
@AutoConfigureBefore( StubRunnerKafkaConfiguration.class )
public class TestConfig {

    @Value( "${spring.kafka.bootstrap-servers}" )
    private String bootstrapAddress;

    @Bean
    @Primary
    public ProducerFactory<String, ServiceCallResponseDto>
        testProducerFactory() {
        var configs = new HashMap<String, Object>();
        configs.put( BOOTSTRAP_SERVERS_CONFIG, bootstrapAddress );
        configs.put( KEY_SERIALIZER_CLASS_CONFIG, StringSerializer.class );
        configs.put( VALUE_SERIALIZER_CLASS_CONFIG, StringSerializer.class
            );
        return new DefaultKafkaProducerFactory<>( configs );
    }

    @Bean
    @Primary
    public KafkaTemplate<String, ServiceCallResponseDto>
        testKafkaTemplate() {
        return new KafkaTemplate<>( testProducerFactory() );
    }

}
```

Here in this configuration we are simply define a KafkaTemplate which will be used during stubs auto-generation.

The upcoming Listing 5.7 demonstrates the test itself:

Listing 5.7: Saga flow test

```
@SpringBootTest
// overriding of the default Spring Contract stub configuration bean
@Import( { CustomStubRunnerKafkaConfiguration.class, TestConfig.class } )
// specifying the place stub artifacts are located
@AutoConfigureStubRunner( ids = "cz.cvut.fel:Contract:+:stubs:8090",
    stubsMode = LOCAL )
@DirtiesContext
// runs a Kafka broker in scope of the test
@EmbeddedKafka( partitions = 1, brokerProperties = {
    "listeners=PLAINTEXT://localhost:9092", "port=9092" } )
public class SagaFlowIntegrationTest {

    @Autowired
    SagaService sagaService;

    @Autowired
    SagaTransactionService sagaTransactionService;

    @SpyBean
    ServiceResponseHandler serviceResponseHandler;

    @SpyBean
    KafkaTemplate<String, ServiceCallRequestDto> kafkaTemplate;

    @Captor
```

```
ArgumentCaptor<ServiceCallRequestDto>
    serviceCallRequestDtoArgumentCaptor;

@Captor
ArgumentCaptor<ServiceCallResponseDto>
    serviceCallResponseDtoArgumentCaptor;

@Test
void success() {
// ARRANGE
    var primaryContent1 = "PRIMARY1";
    var rollbackContent1 = "ROLLBACK1";
    var primaryContent2 = "PRIMARY2";
    var rollbackContent2 = "ROLLBACK2";
    var primaryContent3 = "PRIMARY3";
    var rollbackContent3 = "ROLLBACK3";
    var ts1 = new TransactionSpecification( SERVICE_A,
        primaryContent1, rollbackContent1 );
    var ts2 = new TransactionSpecification( SERVICE_B,
        primaryContent2, rollbackContent2 );
    var ts3 = new TransactionSpecification( SERVICE_C,
        primaryContent3, rollbackContent3 );
    var sagaSpecification = new SagaSpecification( List.of( ts1, ts2,
        ts3 ) );

// ACT
    // this function starts builds a saga from saga transaction
    var sagaSm = sagaService.buildAndStartSaga( sagaSpecification );

// ASSERT
    verify( kafkaTemplate, timeout( 5000 ).times( 3 ) )
            .send( argThat( Set.of( SERVICE_A_TOPIC, SERVICE_B_TOPIC,
                SERVICE_C_TOPIC )::contains ),
                serviceCallRequestDtoArgumentCaptor.capture() );
    assertThat( serviceCallRequestDtoArgumentCaptor.getAllValues() )
            .allSatisfy( request -> {
                assertThat( request.getContent() ).isIn(
                    primaryContent1, primaryContent2, primaryContent3 );
            } );
    verify( serviceResponseHandler, timeout( 5000 ).times( 3 )
        ).processResponse(
        serviceCallResponseDtoArgumentCaptor.capture() );
    assertThat( serviceCallResponseDtoArgumentCaptor.getAllValues() )
            .anySatisfy( response -> {
                assertThat( response.getServiceCallId() )
                    .isEqualTo(
                        serviceCallRequestDtoArgumentCaptor
                        .getValue()
                        .getServiceCallId()
                        );
                assertThat( response.getCode() ).isEqualTo( SUCCESS );
            } );
    sleep( 1000 );
    assertThat( sagaSm.getState().getId() ).isEqualTo( SUCCEED );
    assertThat(
           getTransactionIdList( sagaSm.getExtendedState() )
               .stream()
                    .map( sagaTransactionService::findById )
                    .map( StateMachine::getState )
                    .map( State::getId )
                    .collect( toList() )
    ).allSatisfy( SagaTransactionStates.SUCCEED::equals );
}

@Test
void compensated() {
// ARRANGE
    var primaryContent1 = "CONTENT1";
    var rollbackContent1 = "ROLLBACK1";
    var primaryContent2 = "CONTENT2";
    var rollbackContent2 = "ROLLBACK2";
    var primaryContent3 = "AMOGUS";
    var rollbackContent3 = "ROLLBACK3";
    var ts1 = new TransactionSpecification( SERVICE_A,
       primaryContent1, rollbackContent1 );
    var ts2 = new TransactionSpecification( SERVICE_B,
       primaryContent2, rollbackContent2 );
    var ts3 = new TransactionSpecification( SERVICE_C,
       primaryContent3, rollbackContent3 );
    var sagaSpecification = new SagaSpecification( List.of( ts1, ts2,
       ts3 ) );

// ACT
    var sagaSm = sagaService.buildAndStartSaga( sagaSpecification );

// ASSERT
    verify( kafkaTemplate, timeout( 10000 ).times( 6 ) )
           .send( argThat( Set.of( SERVICE_A_TOPIC, SERVICE_B_TOPIC,
               SERVICE_C_TOPIC )::contains ),
               serviceCallRequestDtoArgumentCaptor.capture() );
    assertThat( serviceCallRequestDtoArgumentCaptor.getAllValues() )
           .allSatisfy( request -> {
               assertThat( request.getContent() ).isIn(
                   primaryContent1,rollbackContent1, primaryContent2,
                   rollbackContent2, primaryContent3, rollbackContent3
                   );
           } );
    verify( serviceResponseHandler, timeout( 10000 ).times( 6 )
       ).processResponse(
       serviceCallResponseDtoArgumentCaptor.capture() );
    assertThat( serviceCallResponseDtoArgumentCaptor.getAllValues() )
           .allSatisfy( response -> {
               assertThat( response.getServiceCallId() ).isIn(
                       serviceCallRequestDtoArgumentCaptor
```

```
                                .getAllValues().stream()
                                .map(
                                    ServiceCallRequestDto::getServiceCallId
                                    )
                                .collect( toSet() )
                    );
                    assertThat( response.getCode() ).isIn( SUCCESS, ERROR );
                } );
        sleep( 2000 );
        assertThat( sagaSm.getState().getId() ).isEqualTo( COMPENSATED );
    }

}
```

The provided Java test, SagaFlowIntegrationTest, is designed to conduct end-to-end testing of a Saga pattern, a design pattern used to manage transactions across multiple microservices, using a local Kafka broker.

In this test class, the @SpringBootTest annotation denotes that it's a Spring Boot test, hence it bootstraps the entire application context. Custom configurations are imported using the @Import annotation to override the default Spring Contract stub with a custom one (CustomStubRunnerKafkaConfiguration.class) and to import test configurations (TestConfig.class). The @AutoConfigureStubRunner annotation specifies the location of the stub artifacts.

The test class contains two tests: success() and compensated(). These tests are designed to verify the successful execution of a saga and its proper compensation upon encountering an error, respectively.

In the success() test, a saga is built and started using predefined transaction specifications. The test then verifies that the correct Kafka messages are sent and processed, that the correct response codes are returned, and that the saga and its associated transactions reach the 'SUCCEED' state. The success of this test is guaranteed as its service call specifications do not contain word 'AMOGUS' (see 5.2.

In the compensated() test, a saga is again built and started, but one of the transactions is set to trigger an error. The test verifies that the correct Kafka messages are sent and processed, including the rollback commands for the completed transactions. It then checks that the saga reaches the 'COMPENSATED' state, indicating that the transactions have been correctly rolled back due to the error. Note, that this test fails as the third service call specification contains word 'AMOGUS' in its content, that's why the relevant stub declines this request and return with an error (see 5.2).

Both tests leverage Mockito's @SpyBean to monitor actual beans and @Captor to capture arguments for further assertions. These tests help to ensure that the Saga pattern is working as expected, both in normal operation and when errors occur, thereby improving the reliability and robustness of the system.

As previously stated, the method buildAndStartSaga in SagaService both constructs and initiates the entire saga. The validity of this claim is confirmed in the assertion section.

Chapter 6

Conclusion

Additionally, the Saga pattern provides two different modes of coordination - orchestration and choreography. Orchestration uses a central controller or "orchestrator" to control the interaction between services. On the other hand, Choreography decentralizes the decision-making process, letting each service decide the next service to involve. Each approach has its pros and cons and the choice between them is often dependent on the complexity of the system and the inter-dependencies between the services.

One of the most compelling features of the Saga pattern is its ability to maintain the overall system's consistency, even in the face of individual microservice failures [26]. By breaking down a global transaction into a series of local transactions, each with its own compensating transaction, Sagas ensure that the system can "undo" changes in the event of a failure. This ability to recover from failures is a critical requirement in distributed systems, where individual components may fail or become temporarily unavailable.

However, it is important to mention that using the Saga pattern does require a shift in how we think about consistency. Instead of relying on the database's built-in ACID properties to manage global transactions, the Saga pattern enforces eventual consistency across multiple services. This means that while individual local transactions are immediately consistent, the overall system may not be in a consistent state at all times. Over time, and assuming no failures, the system eventually reaches a consistent state, as each service completes its part of the global transaction.

Overall, the Saga pattern is a robust and resilient strategy for managing transactions in distributed systems, particularly in microservices architectures. It offers a pragmatic trade-off between consistency and availability, contributing to the scalability and resilience of distributed systems.

Chapter 7

Future work

In this project, we have successfully built an extensive Saga pattern utilizing state machines. While this is a commendable achievement, it is important to note that the Saga pattern has its own unique characteristics that need to be addressed.

A significant concern is the issue of isolation deficiency. This problem surfaces as a counteracting measure since the Saga operates as a highly-available system. In simpler terms, a rise in availability often leads to a corresponding drop in isolation, and vice versa. However, in practical, real-world situations, we are likely to need a particular balance between isolation and availability. Achieving this balance is crucial for maintaining system integrity and ensuring smooth operation [27].

Therefore, from my perspective, the logical next step in this project is to design and implement a mechanism that provides us with the ability to fine-tune the level of isolation and availability. This would mean creating a system that can be configured to meet specific needs or situations, making it more adaptable and robust. This would not only enhance the functionality of the Saga pattern but also significantly improve its applicability in diverse scenarios [28].

By focusing on this next step, we can better address the complexities inherent in the Saga pattern and further optimize it for greater efficiency and effectiveness. With the right approach and tools, we can continue to refine our system and push the boundaries of what's possible with state machine-based Saga patterns.

Bibliography

1. WOLFF, Eberhard. *Microservices: Flexible Software Architecture 1st Edition.* 2016.
2. FOWLER, Martin. *Microservices: a definition of this new architectural term.* Available also from: `https://martinfowler.com/articles/microservices.html`.
3. RICHARDSON, Chris. Available also from: `https://www.microservices.io`.
4. RICHARDSON, Chris. *Pattern: Database per service.* Available also from: `https://microservices.io/patterns/data/database-per-service.html`.
5. NADAREISHVILI, Irakli; RONNIE MITRA Matt McLarty, Mike Amundsen. *Microservice Architecture: Aligning Principles, Practices, and Culture.* 2016.
6. BURNS, Brendan. *Designing Distributed Systems: Patterns and Paradigms for Scalable, Reliable Services.* 2018.
7. MAHESHWARI, Deepak. *Distributed Transaction Management in Microservices World.* Available also from: `https://medium.com/nerd-for-tech/distributed-transaction-management-in-microservices-world-22f34718a643`.
8. HELLAND, Pat. Life beyond Distributed Transactions: an Apostate's Opinion. In: *Conference on Innovative Data Systems Research.* 2007.
9. NEWMAN, Sam. *Building Microservices: Designing Fine-Grained Systems.* 2015.
10. FOWLER, Martin. *Two Phase Commit.* Available also from: `https://martinfowler.com/articles/patterns-of-distributed-systems/two-phase-commit.html`.
11. STONEBRAKER, Michael; MADDEN, Samuel; ABADI, Daniel J.; HARIZOPOULOS, Stavros; HACHEM, Nabil; HELLAND, Pat. The End of an Architectural Era (It's Time for a Complete Rewrite). 2007.
12. BAELDUNG. *Saga Pattern in Microservices.* Available also from: `https://www.baeldung.com/cs/saga-pattern-microservices`.
13. RICHARDSON, Chris. *Microservices Patterns: With examples in Java.* 2018. ISSN 9781638356325.
14. BOTTEMA, Benny. *Orchestration vs. Choreography.* Available also from: `https://stackoverflow.com/questions/4127241/orchestration-vs-choreography`.

15. AYDIN, Sahin; ÇEBI, Cem Berke. Comparison of Choreography vs Orchestration Based Saga Patterns in Microservices. In: *2022 International Conference on Electrical, Computer and Energy Technologies (ICECET)*. 2022. Available from DOI: 10.1109/ICECET55527.2022.9872665.

16. BAŞKÖK, Osman. *SAGA Pattern Briefly*. Available also from: https://medium.com/trendyol-tech/saga-pattern-briefly-5b6cf22dfabc.

17. FRIEDRICHSEN, Uwe. *The limits of the Saga pattern*. Available also from: https://www.ufried.com/blog/limits_of_saga_pattern.

18. SINGH, Rohit. *Modelling Saga as a State Machine : An orchestrator driven approach for managing distributed and long-running transactions*. Available also from: https://levelup.gitconnected.com/modelling-saga-as-a-state-machine-cec381acc3ef.

19. ZERO, Project. *The State of State Machines*. Available also from: https://googleprojectzero.blogspot.com/2021/01/the-state-of-state-machines.html.

20. MICROSOFT. *Publisher-Subscriber pattern*. Available also from: https://learn.microsoft.com/en-us/azure/architecture/patterns/publisher-subscriber.

21. UPRA, Sam. *Decoupling Modules using Publisher/Subscriber in Node.js*. Available also from: https://medium.com/@pongpiraupra/decoupling-modules-using-publisher-subscriber-in-node-js-7dd22206ad13.

22. RICHARDSON, Chris. *Pattern: Service Component Test*. Available also from: https://microservices.io/patterns/testing/service-component-test.html.

23. SPRING. *Spring Cloud Contract - Reference Documentation*. Available also from: https://docs.spring.io/spring-cloud-contract/docs/3.1.6/reference/html/using.html#using.

24. GRZEJSZCZAK, Marcin. [N.d.]. Available also from: https://stackoverflow.com/a/46192620.

25. MOZILLA. *Regular expressions*. Available also from: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/Regular_expressions.

26. TAKADA, Mikito. *Distributed Systems For Fun and Profit*. 2023.

27. BANDARAGODA, Tharindu R; TING, Kai Ming; ALBRECHT, David; LIU, Fei Tony; ZHU, Ye; WELLS, Jonathan R. Isolation-based anomaly detection using nearest-neighbor ensembles. *Computational Intelligence*. 2018, vol. 34, no. 4, pp. 968–998.

28. GRAY, Jim. *Transaction Processing: Concepts and Techniques (The Morgan Kaufmann Series in Data Management Systems)*. 1992.

Appendix

A Dependencies

Listing 7.1: Implementation dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.7.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>2.8.11</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
    <version>2.7.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.7.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.statemachine</groupId>
    <artifactId>spring-statemachine-core</artifactId>
    <version>3.2.0<version>
</dependency>
<dependency>
    <groupId>org.springframework.statemachine</groupId>
    <artifactId>spring-statemachine-starter</artifactId>
    <version>3.2.0</version>
</dependency>
<dependency>
    <groupId>org.springframework.statemachine</groupId>
    <artifactId>spring-statemachine-data-jpa</artifactId>
    <version>3.2.0</version>
</dependency>
<dependency>
```

```
    <groupId>org.springframework.statemachine</groupId>
    <artifactId>spring-statemachine-autoconfigure</artifactId>
    <version>3.2.0</version>
</dependency>
<dependency>
    <groupId>org.projectlombok</groupId>
    <artifactId>lombok</artifactId>
    <version>1.18.24</version>
</dependency>
```

Listing 7.2: Testing dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter</artifactId>
    <version>2.7.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <version>2.7.10</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka</artifactId>
    <version>2.8.11</version>
</dependency>
<dependency>
    <groupId>org.springframework.kafka</groupId>
    <artifactId>spring-kafka-test</artifactId>
    <version>2.8.11</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
    <version>2.7.10</version>
</dependency>
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-contract-verifier</artifactId>
    <version>3.1.6</version>
    <scope>test</scope>
</dependency>
<dependency>
    <groupId>org.testcontainers</groupId>
    <artifactId>junit-jupiter</artifactId>
    <version>1.18.0</version>
    <scope>test</scope>
</dependency>
<dependency>
```

```
    <groupId>org.testcontainers</groupId>
    <artifactId>kafka</artifactId>
    <version>1.18.0</version>
    <scope>test</scope>
</dependency>
```