Masaryk University
Faculty of Informatics

Use of Transactions within a Reactive Microservices Environment

Master's Thesis

Martin Štefanko

Brno, Spring 2018

Masaryk University
Faculty of Informatics

Use of Transactions within a Reactive Microservices Environment

Master's Thesis

Martin Štefanko

Brno, Spring 2018

Replace this page with a copy of the official signed thesis assignment and the copy of the Statement of an Author.

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Martin Štefanko

Advisor: Bruno Rossi, PhD

Acknowledgement

thanks

Abstract

abstract

Keywords

transactions, Narayana, JTA, reactive, microservices, asynchronous, saga, compensating transactions

Contents

1 Introduction

2 Transaction concepts

This chapter introduces the basic notions of transactions, their properties and common problems with their management across multiple nodes in distributed systems.

2.1 Transaction

A transaction is an unit of processing that provides all-or-nothing property to the work that is conducted within its scope, also ensuring that shared resources are protected from multiple users [1]. It represents an unified and inseparable sequence of operations that are either all provided or none of them take effect.

From the application point of view there exist several transaction models in which the transactions can be executed. The applicable models in the transaction management are local, programmatic and declarative transactions. All three models will be described in detail in the following section.

The transaction can end in two forms: it can be either *commited* or *aborted*. The commit determines the successful outcome - all operations within the transaction have been performed and their results are permanently stored in a durable storage. The abort means that all performed operations have been undone and the system is in the same state as if the transaction have not been started.

Generally, the achievement of above features may differ. The most common pattern for the transaction processing is a two phase commit protocol with the ACID transactions. Other approaches are based on the relaxation of the one or more of ACID properties to adjust to the real world environments.

2.2 ACID properties

A transaction can be viewed as a group of business logic statements with certain shared properties [2]. Generally considered properties are one or more of atomicity, consistency, isolation and durability. These

four properties are often referenced as ACID properties [3] and they describe the major points important for the transaction concepts.[1]

2.2.1 Atomicity

The transaction consists of a sequence of operations performed on different resources. An atomicity property means that all operations in the transaction are performed as if they were a single unit.

As the word atomicity is an overloaded term in many computational science branches[2], some authors prefer to reference it in the ACID context as the abortability property. The abortability is defined as the ability to abort a transaction on error and have all writes from that transaction discarded [5]. This implies that when the transaction commits successfully, all of its operations are also required to execute a valid commit. Conversely, when the transaction fails and needs to be aborted, all realized operations and effects must be undone.

The atomicity is commonly achieved by the usage of consensus multi-phase protocols. The standardized protocol is the two phase commit protocol (2PC) which is used by the majority of modern transaction systems. The consensus protocols are discussed in detail in the section 2.6.

2.2.2 Consistency

The word consistency refers to restrictions placed on data changes that may happen only in allowed ways. When the data is persisted, it must be valid according to all defined rules which meet the application invariants. The consistency property describes that the transaction maintains the consistency of the system and resources that it is being performed on. When the transaction is started on the consistent system, this system must remain consistent when the transaction ends - it moves from one consistent state to another.

Unlike other transactional properties (A, I, D), consistency cannot be realized by the transaction system as it does not hold any seman-

1. Although, the ACID acronym has been associated with transactions since their beginning, Eric Brewer, the inventor of the CAP theorem, discussed in the later section, stated in his article from 2012 that it is "more mnemonic than precise".[4]
2. typical example is the atomic operation in the thread context

tic knowledge about the resources it manipulates [1]. Therefore, the achievement of this property is the responsibility of the application code.

2.2.3 Isolation

The isolation property takes effect when multiple transactions can be executed concurrently on the same resources. It provides a guarantee that concurrent transactions can not interfere one with another. Therefore each concurrent execution on the shared resource must be equivalent to some serial ordering of contained transactions. This is why the isolation is often also referred to as a serializability.

From the perspective of an external user the isolation property means that the transaction appears as it was executed entirely alone. This means that even if there are multiple transactions in the system executed concurrently, this fact is hidden from the every external view.

As an instinctive extension of the consistency property, the serial execution of the transaction keeps the consistent state. The execution of the transactions in parallel therefore cannot result into inconsistent system.

Isolation levels

In practice we distinguish several levels that describe to which extent the isolation guarantees are provided. Levels are distinguished by simplifications of the locking mechanism in exchange for the faster processing. These levels, in decreasing order, are serializable, repeatable read, snapshot isolation, read committed and read uncommitted isolation. The definitions given in this section are based on the talk given be Martin Kleppmann [6].

Read uncommitted

This is the lowest level which does not place any restrictions on the system. This means that it does not require any form of locking mechanisms to be implemented. It allows dirty reads which is that one transaction may read the not yet committed changes of other transaction.

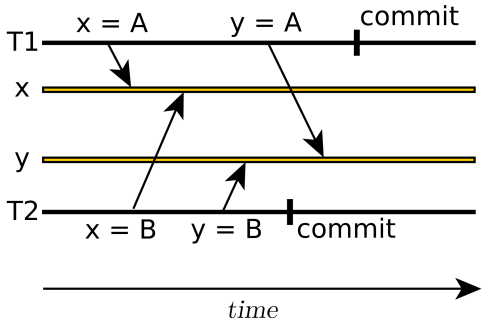

Figure 2.1: Dirty writes

Read committed

Read committed level differs from the previous level in that it prohibits dirty reads and dirty writes. The prevention from dirty reads as discussed in previous section means that the transaction is not allowed to read data that has not been committed by different transaction.

On the other hand, dirty writes mean that the transaction overwrites the uncommitted data. This can be easily described on an example depicted in the figure 2.1. In this execution the resulting data state contains different values for variables (x = B and y = A). However, in any serial execution of transaction the result would be consistent.

Repeatable read and snapshot isolation

Both of these methods represent the same level of isolation preventing the anomaly called read skew which can appear in read committed.

The read skew is a problem of reading the data values in different points of time in which the whole consistent data state may not be ensured. This means that the transaction may read the data change in later point in time which may invalidate already read information from a previous processing.[3]

3. Imagine a bank system with two accounts both with starting balance 500 and a transfer of 100 from account 1 to 2 as a transaction. If an external system reads the balance of the account 2 prior to start of transaction, it will get 500. However,

Both repeatable read and snapshot isolation prevent read skews. From the user point of view they represent the same isolation level, but they differ in implementation. Repeatable read is based on the locking mechanisms which may be complex to maintain in bigger systems. The snapshot isolation is implemented as multi version concurrency control that allows each transaction to read the data as it was in one point of time – the snapshot. The database is required to internally keep track of several states and provide each transaction with the snapshot that is appropriate for its time span.

Serializable

On top of the repeatable read, the serializable isolation prevents the system from one more race condition known as the write skew. The basic principle of a write skew is a transaction that reads data from the storage and then makes a decision based on this information. However, different transactions may write the result into different parts of the database which means they can not conflict. The problem is that by the time the transaction commits, the premise of the decision may no longer be true and the resulting state may break the integrity constraints of the database.

2.2.4 Durability

This property characterizes that all changes done by the transactions must be persistent, i. e. any state changes performed during the transaction must be preserved in case of any subsequent system failure. How the state is preserved usually depends on the particular implementation of the transaction system. Generally, to achieve this property the use of the persistent storage like a disk drive or a cloud is sufficient. Even if this kind of storage is acceptable, it still can not prevent data loss in the case of more critical catastrophic failures.

---

the subsequent read from the account 1 after the transaction has committed would output 400 which would result into inconsistent information.

2.3 Transaction models

As it was already mentioned at the beginning of this chapter, there exist three types of transactions models available for transaction management that define how the transaction is composed – local, programmatic and declarative transactions[4]. This chapter describes each model respectively and examines how it can be applied in applications deployed in the Enterprise Java platform environment.

2.3.1 Local transaction model

The local transaction model derives its name from the fact that transactions are managed by a local resource manager which will be described in the later section. This approach represents the transaction as a connection to the individual resource. Common use-cases for this model include the Java database connectivity (JDBC) or the Java Message Service (JMS) connection providers.

The connection is usually by default configured to commit or rollback the local transaction after each operation, e. g. a database query or sending a message to a queue. In order to control the transaction, it is required to set the boolean flag `setAutoCommit` on the `Connection` interface to false. In this case, the transaction needs to be manually committed or rollbacked with the respective methods of the `Connection` interface.

The major drawback of this model is that local transactions cannot exist concurrently when coordinating multiple resources using an XA[5] global transaction [7] (e. g. we need to update a database and propagate this information to the JMS topic).

2.3.2 Programmatic transaction model

This approach is based on the Java Transaction API (JTA) transaction service implementation. In this model, the developer handles the complete management of the transaction in the source code.

---

4. some sources (e. g. Spring) distinguish only two types of transaction models – the local model and the global model that represents both programmatic and declarative approaches
5. XA stands for *eXtended Architecture*, and it is described in detail in section 2.5

The JTA specification is a set of interfaces that allows developers to manage transactions. Another similar term associated with Enterprise transactions is the Java Transaction Service (JTS). The JTS is a specification for the implementation of the transaction manager which encompasses the JTA specification. It represents a Java mapping of the Common Object Request Broker Architecture Object Transaction Service (CORBA OTS) 1.1 specification. The JTA is required to support both JTS and non-JTS implementations of the transaction management.

Although the JTA specification provides a range of APIs, the only required interface for the utilization of the programmatic transactions is the `javax.transaction.UserTransaction`. The full interface is available at the appendix B. The only concerned methods are `begin()`, `commit()`, `rollback()` and `getStatus()`. The call to the `begin()` will start a new transaction and associates it with the current thread. As the Java platform allows only one transaction to be associated with the thread, a call to the `begin()` method may result into exception in case the transaction has already been started in the current context. The transaction end methods (`commit()` and `rollback()`) perform their respective actions and disassociate the transaction with the thread. The `getStatus()` method returns an integer value representing the status of the current transaction derived from `javax.transaction.Status` class (appendix B).

The most common problem introduced by the programmatic model is that the developer must ensure that the transaction is always terminated in the method that started the transaction [7]. This is often the case when the initiating method ends up with an uncaught exception and for this reason, the transaction needs to be committed or rollbacked before the method returns.

2.3.3 Declarative transaction model

The declarative transaction model is also referred to as the Container-Managed Transactions (CMT). In this model, the supplying, underlying container manages all transactions on the users behalf. This includes starting and the administration of the end phases (either commit or rollback) of transactions. The developer is only required to setup the container with the transaction configuration that declares,

for instance, that the transaction should be rollbacked on any exception.

The main concerned interface used for declarative transactions is the `javax.transaction.TransactionManager`. Although this interface provides same methods that are present in the `UserTransaction` interface used in the programmatic model, users are strongly encouraged to use it only in CMT. In addition to the `UserTransaction`, it introduces two methods that manage the transaction suspensions. The `suspend()` method suspends and disassociates the transaction with the current thread. It returns an object that represents the identification of the running transaction. This transaction object can be afterwards used as an argument to the `resume()` method to associate the transaction again with the current thread and continue the transaction execution.

Another essential method that is often associated with the container transactions is the `setRollbackOnly()`. It declares that the only possible transaction result is to rollback and any consequent actions can not change it.

With the declarative transaction model, the users are required to configure the container with the settings of how individual transactions should be managed. This can be set up through the transaction attribute represented by, for instance, `TransactionAttributeType`, `Transactional.TxType` or `TransactionDefinition` (Spring) classes. The supported values are – Required, Mandatory, RequiresNew, Supports, NotSupported and Never[6]:

- **Required** – If the transaction context is already present on the invocation, it will be used. Otherwise a new transaction is started. This is the most characteristic attribute and it usually configured as a default value.

- **Mandatory** – Similarly to the Required, mandatory transaction attribute represents that the transaction must be present on the execution. However, it requires that the transaction is already started prior to the invocation. Alternatively, it throws

6. Spring adds one more transaction attribute called Nested which represents a single physical transaction with multiple savepoints that it can roll back to [8]

`TransactionRequiredException` if the transaction context can not be found.

- **RequiresNew** – The container will begin a new transaction on every invocation. If there is already a transaction context present, it is suspended for the duration of the processing of the new transaction. This breaks the atomicity property of the former transaction.

- **Supports** – This attribute represents an invocation that is not required to run under the transaction context. It tells the container to use the transaction context, if it exists before the call, or to execute the operation non-transactionally if the transaction is not present.

- **NotSupported** – The method will not be executed within the transaction context. If the transaction exists prior to the invocation, it is suspended and the method is invoked. In other case, the method is immediately started without the initiation of a new transaction.

- **Never** – The container is forbidden to invoke the method if there is a transaction context present. In contrast with the NotSupported attribute which only suspends former transaction, this attribute will throw a runtime exception when the transaction is present before the invocation.

2.4 Distributed transactions

A distributed transaction is any situation where a single event results in the mutation of two separate sources of data which cannot be committed atomically [9]. It represents an extension of the ACID transaction that is executed over a number of independent devices connected through a communication network. The main disadvantage of these transactions is their liability to frequent failures of individual nodes or communication channels that connect them – which is something that the distributed transaction processing (DTP) needs to account for.

Each node is associated with a transaction manager (TM) that manages a local transaction and communicates with other TMs in order to perform a global transaction. Generally, there is one TM selected as a global coordinator that administers TMs participating in the distributed transaction. The coordinator can be allocated with the participating node or can act as a standalone service.

The accomplishment of ACID properties with the frequent partitions failures is very difficult to achieve. Even if the DTP system is able to provide the distributed consensus, it often comes with a performance cost. This lead to the general refusal of the DTP employment in distributed applications in the past.

However, recent network speeds and computational capacities are increasing. This allows consensus protocols (2PC, Paxos) and other DTP solutions (e.g. sagas) to be easily employed in modern, scalable distributed applications.

2.5 Transaction manager

A transaction manager (TM) is a module responsible for the transaction processing, coordination and their sequential or parallel execution across one or more resources. It ensures the proper and valid completion of each transaction. It is also liable for making the final decision whether to commit or rollback the transaction. Clients often communicate with the transaction manager only when they need to start a new transaction.

Main responsibilities of the transaction manager are starting and ending (commit or abort) of transactions, the supervision of transactions scoped across multiple resources and rollback capabilities ensuring the failure recovery. It also manages the transaction context which represents the identification and contains the state of the transaction.

2.5.1 TM types

A local transaction manager or a resource manager is responsible for the coordination of transactions concerning only a single resource. Because of the range of its scope, it is often built in directly into the

resource. The span of the resource is defined by the managing platform, e. g. the JMS context or the TM provided with the database.

The restriction of the local TM scope prohibits its application across various resources. This means that it cannot provide ACID guarantees if the transaction contains, for instance, both the database update and the JMS message send as these resources are handled by different resource managers.

The management of transactions over multiple resources is supported by a global transaction manager. It represents an operation external component that is able to coordinate several resource managers in order to provide ACID transactions spanning two or more different transactional resources.

2.5.2 XA specification

The eXtended Architecture (XA) standard is the X/Open Common Applications Environment (CAE) specification published in 1991 which describes the bidirectional interface between a transaction manager and a resource manager [10]. It maintains two types of components that clients can interact with: the transaction manager (TM) which defines the global TM in a sense described in the previous section and the XA resources that represent local resource managers.

The resource manager implements the XA interface in order to provide a switch that effectively delagates the transaction control to the TM. The XA TM coordination is based on the two phase commit protocol (2PC) which means that the XA interface contains all necessary function calls that needs to available for 2PC – `xa_prepare`, `xa_commit` and `xa_rollback`. The Java mapping of the XA standard is present in the class `javax.transaction.xa.XAResource`.

2.6 Consensus protocols

The consensus problem represents the procedure of achieving the agreement for the shared data value between several components. It has its application in many environments including transactions where the TM needs to conclude whether a transaction can be committed depending on the participants consensus.

The instance of this problem, which solves it for the binary output, is also known as *The Byzantine Generals Problem*. It denotes the agreement of the battle strategy that generals have to conclude. The general in this model may also act as a traitor providing misleading information which represents a failed system component. The original paper from 1982 [11] proved that the consensus can be achieved only if less than one third of the participants is malicious.

A consensus protocol describes a series of steps that solve the consensus problem. These steps can be typically divided into three phases – the selection of candidate values, the exchange of values between participants and the agreement. The final decision of each participant is irreversible. The consensus protocol is correct, if it complies with these properties [12]:

- **Agreement** - all nodes decide on the same value
- **Validity** - the decided value must have been proposed by some participant
- **Termination** - all nodes eventually decide

In the transactions environment, the consensus represents the shared decision whether to commit or rollback the transaction. The following sections describe some of the mostly widely used consensus protocols that may be employed in (potentially distributed) transactional systems.

2.6.1 2PC

The *Two phase commit protocol* is one of the most known employed consensus protocols used not only in the transaction processing. The procedure consists of two phases:

- **The prepare phase** - All participants send their proposals to the coordinator (TM) in which each of them states either that it is able to proceed and commit its work segment or that the transaction needs to be aborted.
- **The commit phase** – After collecting all proposals, the transaction coordinator makes a final decision – if all participants are

able to commit, the transaction can be committed; conversely, if any participant stated that it needs to abort, the transaction is aborted. The final outcome is subsequently forwarded to every participant and the transaction can be finished.

The 2PC protocol is able to handle node failures to some extent through the use of transaction log. However, this do not cover every scenario and certain failures may require manual intervention.

The algorithm expects one node to act as a coordinator. This does not necessarily need to be an elected participant. Any node can act as a coordinator and initiate 2PC prepare phase by asking other participants for their votes. Furthermore, there also exits a decentralized variant but with higher massage complexity.

The main disadvantage is that the 2PC is a blocking protocol. When the participant sends its prepare message, it will block until either a commit or rollback message is received. In case of the coordinator failure, this may oppose significant problems.

2.6.2 3PC

The *Three phase commit protocol* is a consensus protocol introduced in 1982 by Dale Skeen [13]. It extends the 2PC protocol in a non-blocking way – it allows participants to place upper time bounds on the phases completion which assures that resources are not held indefinitely. The three phases are:

- **The prepare phase** – Same as in the 2PC protocol.
- **The pre-commit phase** – If all participants voted in the first phase to commit the transaction, the coordinator sends to every participant a `preCommit` massage. After the participant receives `preCommit`, it will proceed by preparing the commit by locking the required resources assuring that it is able to finish the commit. By this stage the participant can not execute any irreversible actions. If the preparation was successful, it responds to the coordinator with the acknowledgment message.
- **The commit phase** – After the coordinator receives preparation confirmation from all participants (the original paper [13] allows also to specify a majority vote count), it will commence the

commit phase by sending the `commit` or `abort` messages, same as in the 2PC protocol.

The termination is achieved by the timeout boundaries set on every message expedition. If the coordinator timeouts, it always assumes the rollback outcome – after the first phase it cannot proceed as it did not receive votes from every participant and after the second phase if the coordinator fails, the state of the protocol would not be recoverable. However, the participant behaves in the same way after phase one (as it did not receive the outcome from the coordinator, it must assume abort) but if it timeouts after the second phase, it proceeds with the commit. This is allowed as the `preCommit` message is sent by the coordinator only if all participants wanted to commit the transaction in the phase one.

If the coordinator fails, the new coordinator is selected by any election algorithm. This recovery node can determine the outcome of the protocol based on the state of other nodes – if any node received a `preCommit` message, the transaction can be committed (all other participants must have also received the `preCommit` message). If some node did not receive the `preCommit` message, the transaction can be aborted.

By contrast to the 2PC protocol, the 3PC is resilient to more types of failures. Nevertheless, it cannot withstand the network partition. If the partition disassociate nodes that did receive the `preCommit` message from those which did not, the newly selected coordinators on each side of the partition will result into opposite outcomes and thereby an inconsistent system. The resilience capabilities are at the expense of the performance cost which approximately two times higher that in the 2PC protocol [13].

In 1998, Keidar and Dolev introduced an *Enhanced three phase commit* protocol (E3PC) which maintains the consistency in the face of site failures and network partitions [14]. It is using two additional counters that impose linear order on the majorities of the system node while still preserving the same computational complexity as 3PC.

2.6.3 Paxos

The Paxos represents a family of distributed algorithms designed to solve the consensus problem. It has been introduced by Leslie Lamport in 1998 [15].

The idea of the basic variant of the algorithm is reasonably simple. The Paxos distinguishes three types of system nodes: proposers, acceptors and learners. One node can be of more than one type, even act as all of them. Proposers act as client representatives proposing values that the client wants the system to agree on. Acceptors serves as the voting mechanism and all nodes must know the number of acceptors that form a majority. Learners serve as the representatives that can be queried for the decided value. The algorithm runs in two phases:

- **The promise phase** – The proposer first sends its proposed value with a new unique identification number generated from a sequence to all acceptors (or the majority). For each acceptor: if the received id number is higher than the value the acceptor promised to ignore, it will respond with the `promise` message, otherwise it ignores the propose message.

- **The commit phase** – If the proposer collects promises from the majority of acceptors, it sends the `accept-request` message to all acceptors (or the majority) with the same id and the proposed value. If the acceptor did not promise to ignore the received id, it decide on the value and send the `accept` message to all learners. If the id in the message is lower than the promised, the message is ignored. As the `accept-request` message is send to the majority of acceptors, the consensus is reached.

If the acceptor already accepted a value, it appends the accepted id and value to each `promise` message. In this case, the proposer knows that there is some value already decided in the system and it continues the processing with the value from the `promise` message with the highest id. If it wants to update this value, it needs to initiate a new run of the algorithm after the current one has ended.

Many variants of the Paxos algorithm allows it to sufficiently handle almost all types of failures which is why it is employed in the wide range of enterprise systems. In particular, even the basic variant

Protocol	Time (phases)	Message complexity	Client delay
2PC	2	$3(n-1)$	3 RTTs
3PC	3	$5(n-1)$	5 RTTs
Paxos	2	$4(f+1)$ (f = majority)	4 RTTs

Table 2.1: Consensus protocols comparison

solves problems of the 3PC algorithm, namely, network partitions and the restriction to the fail-stop model. Instead, the Paxos protocol is resilient to the fail-recover model which allows individual nodes to recover and continue processing from the point of the failure – which is expected in the modern distributed systems.

The problem that the algorithm can not solve is two proposers that actively compete for the highest proposal number. This happens between phases as the participants `accept-request` is rejected due to the higher `promise` issued to the other proposer. The system is blocked until the conflict can be resolved, e. g., by the exponential back off mechanisms which allows one proposer to wait sufficiently long for the other one to finish. Acceptors are also required to have a persistent storage to avoid providing misleading information in case of the fail-recovery.

2.6.4 Conclusions

This section presented in detail three consensus protocols that can be employed to solve the transaction commit / abort consensus in the distributed systems. There also exists many other algorithms, for instance, the Raft or the Ark which can not be discussed due to the space limitations. The summary of the presented algorithms is available in the table 2.1 (this table represents scenarios without any failures).

The consensus is a very sophisticated and complex problem particularly in distributed environments. As it will be presented in the following chapters, the saga pattern [55] provides a simple and elegant alternative to the distributed consensus for the long lived transaction commit.

3 Microservices architecture pattern

This chapter introduces the concept of microservices and it focuses on why modern, elastic and resilient enterprise systems should be designed an implemented according to this pattern. It also provides an updated microservices status overview from my previous work publication [16].

3.1 Architectural pattern

Microservices are an architectural pattern which offers an intuitive approach to common problems following a software development. They represent a subset of a Service Oriented Architecture (SOA) [17] that advocates creating a system from a collection of small, isolated services, each of which owns their data, and is independently isolated, scalable and resilient to failure [18]. Instead of the SOA, which builds the applications around the system logical domain, microservices are focused around the application business model. Each microservice represents the separated and independent part of the system that interacts with other components only through predefined communication interfaces[1].

3.1.1 Monolithic architecture

The effective way of describing why the microservice architecture is emerging as a practical development style, is to begin with the definition of the opposite pattern – the monolithic architecture. When the application is developed in the monolithic fashion, all of its content is being implemented and deployed as a single archive. Every component, i. e., a unit of software that is independently replaceable and upgradeable [19], is tightly coupled within the application. Because of the easy development of the monolithic software, this approach has been preferred by the majority of edging enterprise applications. However, when the application requires to add a new functionality or to fix a problem, any additional maintenance represents an issue. For

1. throughout the rest of this publication we will be using terms *microservice* and *service* interchangeably

instance, even because of the minor change or update in the single component, the scalability, continuous deployment and the general advancement of the whole application can stagnate.

Monolithic applications present a few advantages – the development model is often easy to adjust to the application requirements[2], the deployment is reduced to single archive (or a small number of archives) and it is easy to horizontally scale by adding more servers behind a shared load balancer. The problems arise when the system becomes large. The monolithic code base is often complex and hard to understand which results into long learning curves [21] and developer concerns. The automatic deployment and the continuous delivery (CD) of the system also decelerate – in order to update one component you have to redeploy the entire application [22]. Although, it is still able to scale horizontally, the replicated server instances take up more resources and overload the container with a slower start up speeds. In general, the monolith also represents a commitment to a particular technology (or even its specific version) which makes the system difficult to maintain and also adapt to new emerging technologies.

3.1.2 Microservice architecture

Microservices introduced the application separation into the self-maintained units – services [23]. The service is a single scalable and deployable unit, which is not dependent on any context. This means that services may be deployed and scaled independently of each other, and may employ different middleware stacks for their implementation [24].

The important attribute of the microservices system is a service isolation. Each microservice is responsible for the management of its own resources and it is prohibited to directly access resources of any other service. This means that each data request must be processed by the operating microservice which is allowed to accordingly control the data access and computation requirements. Services often correspond to components in the monolithic architecture.

Microservices further extends the Law of Demeter which intents to organize and reduce dependencies between classes [25]. As the service

2. the traditional development model represented as the client-server-database or the Model-View-Controller architecture [20]

presumably requires to communicate with other services in order to provide system functionality, this law naturally applies to minimize such coupling among microservices on the distributed component level.

Another standard object-oriented rule that also applies in the microservices environment is the Single Responsibility Principle (SRP) as defined by Robert C. Martin – a class should have only one reason to change [26]. There is a common misconception associated with the microservice architecture – the word *micro* should conform to the service size. Although, this statement is true to some extent (there is no point in creating the microservice of the same size as the monolith), the *micro* should more resemble a scope of the service responsibility. This concept also corresponds to the Unix philosophy: Make each program do one thing well [27].

The separation and loose coupling of microservices provide an ability to deploy each individual service to the production environment autonomously, not affecting other applications or services. This allows isolated teams to develop, maintain and upgrade services independently and to form these teams around the system problem domains.

As microservices represent a stateful entities, to achieve data isolation each service exposes an application programming interface (API) through which it is exclusively able to provide functionality to other services. These APIs are often technology-agnostic to ensure that the technology choices are not constrained [28]. Instead of in-process calls employed in the monolithic architecture, applications based on the microservices style utilize services by remote procedure calls which are often asynchronous. This form of segregation also facilitates the system failure recovery or resilience as each particular microservice breakdown is less prone to influence the rest of the system.

3.2 Principles of microservices

This section describes the microservices architecture from the perspective of the business use cases and the solution architecture. It is based on the work of Sam Newman [28, 29] in which he proposed to build each microservices system on a set of principles. This principles

may differ for various systems (depending of the application and microservices use cases) but in general, they can reduced to these eight principles:

1. **Modeled around the business concepts** – When the microservices applications, together with the teams that are responsible for their maintenance, correspond to the business domain, they are generally more stable – the requirements on their functionality do not change frequently. This allows developers to focus on the particular system segment, rather than on some specific technology stack. Additionally, it also permits services to directly reflect the business requirements.

2. **Adapting a culture of automation** – Because of the service motion, failures or the communication distribution through the network, microservices brings additional complexity to the system. When the number of services increases, the maintenance, administration and deployment can became unmanageable. The automation then presents an essential part of the service life cycle. The practices as the automated service testing, the employment of the continuous delivery or the unification of the deployment strategy over services, allow enterprise systems to scale more efficiently and speed up the mechanism of the service coordination.

3. **Hiding the internal implementation details** – Every microservice generally needs to interact with other services or external systems to provide its functionality to the rest of the system. In order to keep the option of independent development, it is essential that each service hides its implementation details. This can be achieved through the motion of bounded contexts as defined for the Domain-Driven design (DDD) [30]. The bounded context delimits the applicability of a particular model so that team members have a clear and shared understanding of what has to be consistent and how it relates to other contexts. The context is separated by an explicit interface represented as an API which allows teams to specify which utilities of the service can be shared and which must be hidden. Every request for the

service data must be subsequently processed through this public interface.

4. **Decentralizating all things** – Microservice architecture is build around the idea of self-sustaining development which means that services are maintained autonomously. This allows to delegate decision making and authority to the team that is accountable for the service maintenance. The team is then able to take full ownership of the service which with the support of the independent deployment mechanism results into the convenient development, testing and life-cycle management. This principle accentuates that relevant business logic should be kept in services themselves and the communication between them must be as simple as possible. This permits to design systems in a way that adheres to the Conway's law stated in 1967 [31]:

   > Any organization that designs a system (defined broadly) will produce a design whose structure is a copy of the organization's communication structure.

   This principle also affect the system architecture and design. The purpose is to avoid approaches like enterprise service bus (ESB) or other orchestration systems, which can lead to centralization of business logic [28]. In general, architectures build on the choreography patterns rather than orchestration are preferred. The comparison of these two approaches has been investigated in many research works [32, 33, 34].

5. **Independent deployments** – This is the most important principle of the mircroservices architecture. When the service is being deployed, it should be the requirement that it cannot influence the lifespan of any other service. To achieve this, various techniques like consumer-driven contracts or co-existing endpoints can be used. Consumer-driven contracts make services to state their explicit expectations. These requirements are supported by the provided test suite for individual parts of the domain and they are run with each Continuous integration (CI) build. Co-existing endpoints model accommodates consumers to service changes over time. The idea is to make new endpoint which

can process updated client requests while the former endpoint still functions for a limited lifespan. This includes techniques as blue/green releases [35] or canary deployments [36]. Customers can utilize both endpoints depending on the version their applications require which allows them to decide when to upgrade. Once the previous endpoint is no longer in use, it can be safely removed.

6. **Customer first** – Services exist to be called. It is indispensable to make these calls as simple as possible for the customers. The developers can advatage from any feedback from the clients that use their service. To ease the understanding of the service API, it should be supported by a good documentation provided by API frameworks like Swagger [37]. This also includes the service discovery mechanisms to propagate system services and to make the discovery of the service providers more apparent. To combine this information we can use the humane registries [38] which indicate the human interaction.

7. **Failure isolation** – Even if microservices force distributed isolated development, the architecture still needs to protect against the failure propagation between services. This principle supports resources separation to avoid the single point of failure and it is also supported by the service location distribution. As services require to communicate remotely, it is important to account for the network failures. To prevent cascading failures various techniques like timeouts, bulkheads or circuit breakers [39] may be employed. As there are many vulnerabilities in applications which cannot be considered, there is no precise manual on how to attain this principle.

8. **High observability** – Monitoring is an important part of development and production deployment. Because of the microservices system distribution, it is not sufficient to observe actions performed by particular services apart. Instead, the monitoring solution must record the system operations altogether. To make this information more accessible, the aggregation is essential. Storing all log entries and statistics in one place can highly impact the monitoring process. Another relevant issue is to track

the service calls as the services typically communicate with other services. This can be achieved by mechanisms as semantic monitoring and techniques like synthetic transactions or correlation IDs [28]. By logging this kind of information we can ensure traceability in the case of service failure.

3.3 Reactive microservices

Before the definition of what the reactivity means in distributed microservices environments, it is appropriate to start from the basics of what the reactivity signifies in general terms and how these principles may be applied in software architectures. This section introduces the motion behind the reactive design and why it is suitable for the use in the microservices environment.

By the definition in the Oxford dictionary, the word *reactive* symbolizes an exposure of a response to a stimulus or an action in response to a situation rather than creating or controlling it. This definition naturally translates to software systems. However, the interpretation of what the stimuli is in software applications may differ. It might be, for instance, events, messages, requests or failures. The important common property of these impulses is that the development model cannot be implemented in a way to control them. This motions differ from the traditional style of programming models in which the program functioned as a sequence of commands that were always executed in the predefined order and in the maintained, controlled state.

In software systems we distinguish three distinct classes of reactive concerns – reactive systems, reactive programming and reactive streams [40].

3.3.1 Reactive systems

Reactive systems are an architectural style that focuses on the responsiveness. By the definition provided in the Reactive Manifesto [41], reactive systems are also resilient, elastic and message driven which makes them more flexible, loosely-coupled and scalable. Generally, this model provides a straightforward programming interactions and simplified dependency management that is required in modern appli-

cations. The following enumeration explains these essential properties in detail:

- **Responsiveness** is the most important characteristic of reactive systems. It provides a guarantee of a timely response to normal user requests, as well as the rapid failure detection. Reactive systems are expected to establish a sufficient upper bound placed on the system response times to institute an end user assurance in the system usage.

- **Resilience** covers the responsiveness of the system in the case of the system failure. The manifesto states that the system is not resilient, if it becomes unresponsive after any failure. Resilience can be achieved by, e.g., replication, isolation, delegation and loose-coupling. This ensures that the failure in one part of the system cannot effect the system as whole which shadows the component clients from any form of the failure handling.

- **Elasticity** involves the system responsiveness in the case of alternating load. Reactive system is expected to be able to dynamically adjust and scale system resources according to the request traffic. Elasticity also implies that the system must be able to actively replicate and regulate its components and distribute user inputs among them by the scalable, predictive (and possibly reactive) algorithms.

- **Message driven** elaborates on the asynchronous message exchange between the system components that promotes the loose coupling, isolation and location transparency. Explicit message utilization has many advantages, e.g., flow control, load management or monitoring and engagement of the back pressure. The location transparency, based on virtual addresses, decouples individual components. It may also provide a failure management mechanism, in which case the service cannot distinguish between the communication with a single component or a cluster. Additionally, the asynchronicity allows the system to utilize resources in a non blocking way, only when they are required for the request processing.

3.3.2 Reactive programming

Reactive programming is a development model focusing on the observation of data streams, reacting on changes, and propagating them [42]. In the rest of this section, we will be referencing these data streams in the Reactive Extensions methodology as *observables*. An observable is an object that contains dynamically versatile data that represent state which may be of interest to other object. In order to consume the data emitted by an observarble, the interested object must *subscribe* to it.

In practice, reactive programming distinguishes three kinds of observable objects – observable data streams, singles and completables. Observable, as a stream of data, represents an asynchronous reaction. It provides three handlers, namely, for the data result, error handling and the end of the data stream. The single is a special type of observable that depicts the stream of one value. It is associated with an execution of asynchronous operation which provides data and error callback handlers. The completable observable symbolizes the stream without any value. It contrast with the single, it do not return a data value. For this reason, the completable should be configured with the completion and error handlers.

The observable stream can be of two types – a cold or hot observable. The cold observables are lazy loaded. This means that the data stream do not process any tasks until somebody starts observing it. It represents an asynchronous action that is invoked only when there is a consumer interested in the result. When an object subscribes to a cold observable, it receives all data objects contained in the stream which allows them to be shared. Conversely, the hot observable data stream is active before the consumer subscriptions. When the consumer subscribes to the hot observable, it will receive all data values from the stream that are emitted after the subscription. Both cold and hot observables require the user subscription to receive the data values from the streams. If the consumer does not subscribe to an observable stream, the data is lost.

The most important concept of reactive programs is the asynchronicity. On the contrary from the traditional program invocations, this processing model is based upon notifications that are emitted when the data stream produces a value. Each asynchronous operation happens independently of the main program flow which introduces

several new aspects that needs to considered for this kind of programming paradigm. These aspects can be summarized in three simple rules: avoid side effects[3], avoid using too many threads and never block.

One of the most popular implementations of the reactive programming principles in modern systems is the Reactive eXtension (RX). It represents a library for asynchronous and event-based programs by using observable sequences [43]. These extensions combine the observer and iterator patterns with a range of functional idioms to allow developers to easily adapt reactive methods. Reactive extensions provide a broad range of implementations for various programming languages as, for instance, Java (RxJava), C# (Rx.NET) or Kotlin (RxKotlin).

To conclude, it is important to remember that reactive programming does not build a reactive system [44]. It only provides a development model that can be used for asynchronous processing, a task based concurrency model or the non-blocking Input/Ouput (I/O) that may be employed with other reactive principles to create responsive and reliable distributed systems as defined in the Reactive Manifesto.

3.3.3 Reactive streams

Reactive streams represent an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure[4] [45]. It addresses the issue of controlling the load placed on the stream destination in case of the consumption overload. The main focus of reactive streams is placed on the mediating the stream of data among different API components without the requirement to buffer unreasonable amount of data on the receiver side.

This specification aims to provide a minimal set of interfaces and protocols that would describe the operations and entities to achieve asynchronous streams of data with non-blocking back pressure [42]. It mainly serves as an interoperability layer. The current provided Java

3. The side effect is any interaction of the function with the remainder of the program in other way than through its arguments or its return value

4. Back pressure is a form of feedback mechanism that allows consuming object to regulate the load which is beign sent to it. It is typically employed in situations where the publishing object (observable) is able to emitt data items more quickly than the consuming side is able to process them

virtual machine (JVM) implementation includes the Java API, the specification, the technology compatibility kit (TCK) and programming examples.

The API components that are required to be provided by the Reactive Streams implementations are publisher, subscriber, subscription and processor. The publisher provides potentially infinite number of elements in a sequence which are being published according to the subscribers demands. The subscriber subscribes to the data publisher with a call to `Publisher.subscribe(Subscriber)`. The outcome of this operation is signaled to the subscribing consumer by a call to the `Subscriber.onSubscribe(Subscription)` method. The subscriber is able to request data delivery by a call to `Subscription.request(long)` in which it can specify the number of items it is able to consume. This call is then followed by a requested number of `Subscriber.onNext` calls that represent the delivery of the data item to the subscriber. If the requested number of items is `Long.MAX_VALUE`, the request is treated as effectively unbounded. The termination of the data consumption is signaled to the subscriber by the call to one of the `onComplete` or the `onError` methods. The subscriber is also allowed to request the publisher to stop sending data at any time by a call to `Subscription.cancel` method, but there still may be some data received due to the asynchronous nature of the publisher. The processor represents the component which is both the publisher and the subscriber in one instance and it must follow the contract of both interfaces. The full API provided by the Reactive Streams in version 1.0.2 is available in the Appendix C.

All of the API component interfaces discussed above has been already included in the Java development kit (JDK) 9 in the class `java.util.concurrent.Flow`. The reactive streams initiative is supported by the companies like Neflix, Pivotal, Red Hat, Twitter and many others.

3.4 Challenges

As it is common with every architectural style, microservices bring together with the above mentioned benefits also some drawbacks. Although, the development experience showed that microservices are preferred choice over monolithic architecture, they may not be

inevitably suitable for every system. This section describes some of the challenges that may impose problems when building systems based on the microservices pattern.

3.4.1 Distributed systems

The service distribution supports the architecture by the inherent model of the service boundaries. However, the communication over the network brings a few complications that services need to account for.

The first considerable issue is network failures. Because this is something that can not be controlled by the invoking service, it is required that each microservice call is treated with caution (e. g. by setting an explicit timeout). Consequently, every service should be always designed in a resilient way with failure in mind.

Another relevant problem presented by the network overhead is the communication performance. Remote calls, that are required for inter service invocations, represent additional complexity and time consumption that are not present in modular monolithic systems. There are several various techniques that may be employed to improve the general performance like, for instance, decreasing the number of calls or making them asynchronous.

The problems mentioned above have been elaborated in the work of James Gosling who in 1997 extended the draft created by Peter Deutsch which stated wrong assumptions that are commonly being made about distributed systems. These assumptions are known as *The 8 fallacies of distributed computing*:

1. The network is reliable.
2. Latency is zero.
3. Bandwidth is infinite.
4. The network is secure.
5. Topology doesn't change.
6. There is one administrator.

7. Transport cost is zero.

8. The network is homogeneous.

Unfortunately, the original work of authors is no longer available, but Arnon Rotem-Gal-Oz provides in his publication [46] a very detailed descriptions of each individual assumption.

3.4.2 Eventual consistency

The eventual consistency is a model where the system guarantees that if no new updates are made to the object, eventually all accesses (data reads) will return the last updated value [47]. It is a form of weak consistency – the system do not provide any consistency guarantees for a limited time called inconsistency window. The advantage of this model is that the maximal size of the inconsistency window can be computed from system statistics if no failure occurs.

The arguable problem with this model is the delay between the write of the data value and the its actual obtainable update. The reason is once more mainly caused by the service distribution. This may lead to the decrease system usability and customer satisfaction.

Imagine a situation where the customer creates an order in a web interface. After the confirmation, the order request is sent to the service A which starts its processing. Right after the confirmation, the user wants to check if the order was created in the orders section which is provided by a service B. If the message about the order creation has not yet been propagated to the service B (the inconsistent window is still open), it can not respond with the actual updated system state.

If the system allows microservices to make decisions based on inconsistent information, the eventual consistency may lead also to problems in the service coordination. This kind of issues is hard to find as they are often discovered only after the inconsistent window has been closed.

The propagation delay problems are also present in the monolithic systems, but as it was mentioned in the previous chapter, the remote calls are typically remarkably less performant than in process communication. However, in practice applications do not really depend on the strong consistency guarantees to the expected extent (as we

will see in the later chapters, the saga pattern is based on the eventual consistency).

3.4.3 CAP theorem

In 2000, Eric Brewer introduced the idea that there is a fundamental trade-off between consistency, availability and partition tolerance [48] in the distributed system. This proposal is known as *the CAP theroem* and it states that distributed systems can provide at most two of these three properties.

The consistency guarantee ensures that if some value has been written by a specific node, the query placed on any other node must return the same value or the later update. The availability states that if the node has not failed, it must be always able to respond. Finally, the partition tolerance is an ability of the system to continue functioning even if the communication access between two or more nodes has been lost. In other words, this means that services are still operating but they are not mutually reachable.

In practice, the general belief is that for wide-area systems, designers cannot forfeit the partition tolerance and therefore have to make a choice between consistency and availability [4]. The only effective way of how to guarantee that the partition can be avoided is to only a single service.

The CAP theorem has been formally proven in 2002 by Seth Gilbert and Nancy Lynch [49]. The prove is surprisingly simple and can be modeled by just two nodes.

In 2012, Daniel J. Abadi proposed an extension of the CAP theorem called PACELC theorem which is particularly applicable for distributed database systems (DDBSs). The theorem states – if there is a partition (P), how does the system trade off availability and consistency (A and C) or else (E) when the system is running normally in the absence of partitions, how does the system trade off latency (L) and consistency (C) [50]. The ELC part applies only when the system replicates data – this is why this model cannot describe all kinds of distributed systems.

3.4.4 Operations

With the microservices architecture comes an inherent complexity incurred by the service operational management. As services are expected to be dynamically created and destroyed, upgraded, scaled and deployable, it is essential that the system employs techniques which simplify the operations processes. Such methods include automation, continuous delivery (CD) and integration (CI) and possibly external monitoring and orchestration tooling. Many of these approaches are useful even for the monolithic applications, but they become necessary if the system makes use of microservices [51].

In the modern move to the cloud architectures – platform as a Service (PaaS) considerably ease operations tasks. Containerization platforms like OpenShift [52] or Kubernetes [53] simplify the management of networking, automatic scaling, replication, resilience, tracing, monitoring and many other tasks.

All of above mentioned procedures remarkably accelerate the software life cycle process intervals. Therefore, the employment of DevOps (development and operations) principles which promote increased collaboration and shared responsibilities across teams is also essential. DevOps allows not only to deliver products faster while still maintaining reliability assurances, but they also provide capabilities to bring the quality in to the development process itself.

3.4.5 Human factor

Previous sections discussed the complexities of microservices architecture from the technical point of view. This section focuses on teams that are creating individual services and discusses some of reasons which may influence how this mind shift may affect their end performance.

- **Communication** – The team communication is essential. The opinion and problem solution discussion capabilities inside a team directly influence the deliverability of the maintained service. Although, the service boundaries should prevent the requirement for inter service agreements, it is not always possible to avoid it entirely (version management). The commitment to the DevOps culture can also support the communication process.

- **Nonuniform technology** – The possibility to choose the technology stack individually for each service may promote independence and exploring possibilities, but may also impose a maintenance overhead in a long duration.

- **Design shift** – As microservices still represent a relatively young architecture, the design composition change may be hard utilize. Even if there are a few successful employments of microservices (Netflix or Spotify), the inexperienced teams may be hesitant to make a move to the split of the working monolithic applications. Furthermore, the transition itself impose a complex task.

- **Monitoring** – With the high number of services, collecting and processing of the monitoring information may become very complex task. It also affects the problem traceability and debbuging across services.

- **Testing** – Despite the fact that there exists several microservices testing strategies [54] which may help with the test case definitions, the asynchronous communication and the service distribution may still impose significant problems associated with testing development and execution.

4 Saga pattern

A saga, as described in the original publication [55] from 1987, is a long lived transaction that can be written as a sequence of transactions that can be interleaved with other transactions. Each operation that is a part of the saga represents an unit of work that can be undone by the compensation action. The saga guarantees that either all operations complete successfully, or the corresponding compensation actions are run for all executed operations to cancel the partial processing (4.1).

In contrast to the traditional transaction approach, the Saga pattern relaxes some of the ACID properties (namely atomicity and isolation) to achieve availability and scalability with build-in failure management. In practice, applications are mostly not fully restricted to all of the transaction guarantees, so the saga pattern is emerging as a real alternative to traditional ACID transactions.

In this chapter, we will describe the saga processing as it is defined by the initial publication [55], describe how the activities and compensations works in sagas, how the saga can handle the failure recovery and present several modern frameworks that provide the support for the saga implementation.

4.1 Activities

An activity represents a local transaction that is a part of the saga. Each saga can be split into a sequence of activities in which each individually can be implemented with full ACID guarantees. When the activity completes, all results of the performed work are expected to be persisted in the durable storage. This means that the external observer may see the system in intermediate states of the saga execution, as well as that it may also introduce the system into an inconsistent state between the individual activity invocations.

The ability to commit a local transaction breaks the atomicity property as we no longer endure the atomic unit that the enclosing transaction represents. Furthermore, if distinct sagas may be executed in the system concurrently, the isolation (serializability) property is also violated. Although, intermediate saga states may introduce consistency contraventions, sagas guarantee that the state will become consistent

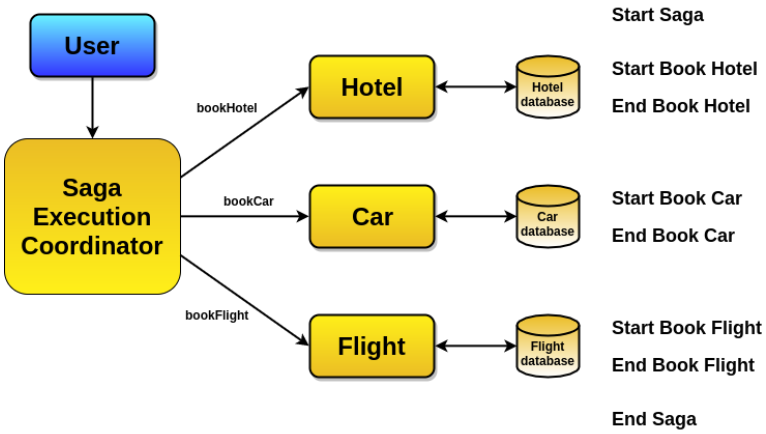

Figure 4.1: Example saga execution

after the saga completes. This principle is called an eventual consistency. It guarantees that even if partial execution may result into the inconsistent state, the system state will get consistent eventually – there exist a point in future in which the system is consistent.

As the definition in the original paper by Garcia-Molina and Salem [55] allows local transactions to interleave, it prohibits any form of dependencies between them. This would imply that the participant cannot depend on results committed by any previous transaction in the saga sequence. However, in modern systems the sequential execution of local transactions is possible to implement in expense of an utilization of a single saga coordinator process throughout the entire sequential execution of this sequence.

4.2 Compensations

Each activity in a saga needs to have an associated compensation transaction. The purpose of the compensating transaction is to semantically undo the work performed by the original transaction. This is not necessarily the contradictory action that puts the system into the same state as it was present before the activity began or generally the saga started.

Imagine that the activity consists of the sending of an email. The saga compensation transaction cannot directly undo the email consignment. Instead it would send another email to the same destination which could explain why the previous action did fail. In this case, we can see that the system is in the state where it has two additional emails being send. However, the comprising system state is expected to be semantically consistent as both transaction and compensation have been defined by the participant. Therefore, the consistency guarantees must be ensured by individual participants at the activity level.

The compensating actions for the individual activities are expected to be idempotent. The main reason for this requirement is the saga failure and recovery management which in detail described in section 4.4.

It is important to keep in mind that even the compensating transactions may fail. There are several options of how the saga management system can handle such situations. The first option is to retry the compensating transaction again, but as the reason of the failure may still be valid, the system can get caught in an infinite retry loop. Another option is to provide a recovery block – a separated block of code which would get executed in place of the primary compensation in case of failure. The last option, which is not elegant but it is practical, is the manual intervention. This is possible to implement due to the saga nature – it does not hold any locks on resources it is being performed on. When the compensation handler is manually repaired, the saga can continue its execution where it has left off.

4.3 Saga execution component and transaction log

The saga execution component (SEC) is a process that is responsible for the saga management. It communicates with the transaction manager which manages activities included in the saga. Both of these components require a transaction log to record their respective interactions[1]. The saga execution component does not require concurrency control because the saga activities can be interleaved.

The entries that may be written to the transaction log are usually associated with the saga or activity lifecycle. The saga log in-

1. it is convenient to share the transaction log between both components

cludes `start-saga` entry followed by one or more `start-activity` / `end-activity` entries and it is finished by the `end-saga`. Optionally, the transaction system may also provide an ability for users to cancel the saga execution with the `abort-saga` command.

Each saga operation is channeled through the saga execution component and it is recorded in to the transaction log before any action may be taken. The transaction log can also contain any parameters associated with the saga execution.

4.4 Recovery modes

The saga paper [55] distinguishes two options to handle the failure that interrupts the saga. These two supported modes are backward and forward recovery.

Backward recovery

A backward recovery mode is the most common way of handling saga failure management as it was described in previous sections. It requires that all activities must define a compensation handler.

When the SEC component receives an `abort-saga` command in the backward recovery mode, it firstly abort the currently executed activity. Then for every previous activity in the reverse order of the original execution, it calls its respective compensation action. After the invocation of the compensation handler corresponding to the first activity is completed, the saga may end and the system is in semantically consistent state as it was before the saga began.

When the saga management system applies the backward recovery mode, the associated transaction log is also used to recover from the crashes of the saga coordinator. After the recovery, once all activities has been completed (committed or aborted), the saga coordinator determines the status of each saga execution by the investigation of transaction log entries.

If the log contains only both `start-activity` and `end-activity` entries for the activities comprised in the saga, than the execution is safe to continue with the next activity which has not been started. Another safe state is when the transaction log contains the `abort-saga`

entry. In this case, it calls all compensation handlers for the referenced saga. This is possible due to the fact that all compensating actions are required to be idempotent.

The only unsafe state, that may be introduced after the saga coordinator recovers, is when the transaction log contains the `start-activity` entry without the corresponding `end-activity`. In this case, the saga coordinator selects the last successfully executed activity (contained in the transaction log) and invokes the compensation handler for this activity and all activities that have been executed before it.

As in the case of repetitive recovery for the same saga, the saga coordinator may call the corresponding compensation handlers repeatedly, the compensation actions are required to be idempotent. The original paper acknowledges that this constraint may be difficult to implement in some applications which is the reason for the introduction of the forward recovery mode discussed in the following section.

Forward recovery

For the use of the forward recovery mode, the transaction management requires that the saga itself is predefined and that the system is able to produce a check point. The check point represents a snapshot of the system state in the particular point in time into which the system can be always restored.

The pure forward recovery mode takes the checkpoint automatically at the beginning of each activity. Furthermore, it also disallows to abort the saga intermediate execution. This effectively eliminates the need to define any compensation actions. If the crash of the SEC occurs, it will abort the last executed activity and restart the saga from the last checkpoint. This approach effectively degrades the saga execution component to a basic persistent transaction executor, therefore loosing most of the saga benefits.

In addition to modes defined above, it is also possible to combine these two approaches into the backward / forward recovery mode. In this mode, the transaction system takes checkpoints in predefined intervals which may be periodical or based on a different criteria

(e. g. the activity complexity). In case of the SEC failure, the system performs the backward recovery to the last defined checkpoint and then continues the saga execution in forward recovery mode.

4.5 Distributed sagas

The notion of sagas can be naturally extended into distributed environments [55]. The saga pattern as an architectural pattern focuses on integrity, reliability, and quality, and it pertains to the communication patterns between services [56]. This allows the saga definition in distributed systems to be redefined as a sequence of requests that are being placed on particular participants invocations. These requests may provide ACID guarantees, but this not restricted and it must ensured by individual participants. Similarly, each participant is also required to expose the idempotent compensating request which can semantically undo the request that is handled by this participant in the saga.

Analogously to the non-distributed systems, the distributed saga management also requires a transaction log which needs to be durable and distributed. The examples of distributed database providers include e.g. Cassandra, RethinkDB or Apache Ignite.

The saga execution coordinator (SEC) is spanned process across the participating services. This process manages and interprets the saga and it persists all processing information into the transaction log. The coordinator does not represent a single point of failure as it is allowed to fail. This is possible because the SEC process does not hold any state data, all of the saga state is held in the distributed log. The general example of the application model employing SEC is available in the figure 4.2.

As all of the above mentioned components are distributed, the saga management system needs to deal with a number of additional problems that are not present in the localized environment. The main problem is that the saga system is required to deal with network and participant failures that may happen between the remote invocations. There are four locations that can encounter the network failure – writes for the beginning and end of the request to the transaction log and the request and response calls to the associated participant. This may

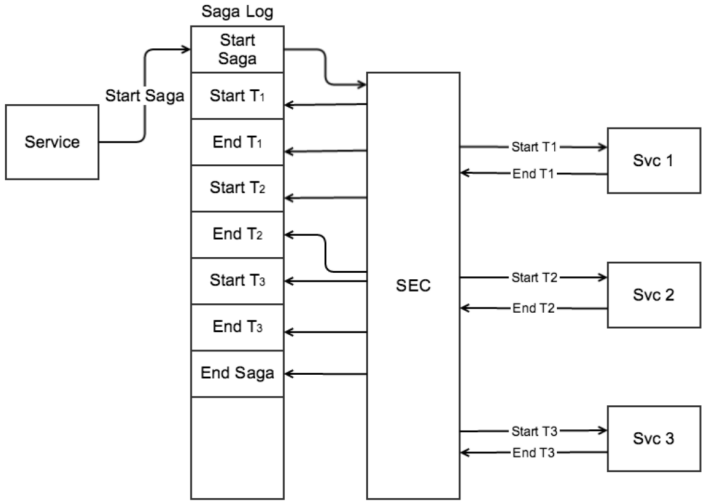

Figure 4.2: Distributed saga example [57]

introduce unnecessary saga aborts[2], but generally the approaches from the non-distributed environment still apply.

4.6 Current development support

This section presents the current implementations of the saga pattern available for the enterprise use. The three explored frameworks are Axon [58], Eventuate [59] and Narayana Long Running Actions (LRA) [60].

4.6.1 Axon framework

Axon is a lightweight Java framework that helps developers build scalable and extensible applications by addressing these concerns directly in the core architecture [58]. It is composed on the top of the

2. if the SEC fails after it has written the `start-request` entry to the transaction log – the recovery cannot determine whether the failure happened before the request, due to the network failure, or the participant response has been lost

Command Query Responsibility Segregation (CQRS) pattern which is described in more detail in the Appendix A.

The Axon framework is based upon the event processing including asynchronous message passing and event sourcing. This allows components to be loosely coupled and therefore easily developed in the the microservices manner.

Saga definition

As the most of the Axon functionality, the easiest way to define sagas in an application is by the use annotations. The annotation `@Saga` marks the class as a saga implementation. In Axon sagas are a special type of event listeners. Each object instance of the saga class is responsible for managing a single business transaction. This includes the management of the saga state information, the execution and handling of the transaction (including start and stop) or the performing of the corresponding compensation actions.

All interaction with the saga class happens only by triggering of events. The ordinary event handlers in saga instances are annotated with the annotation `@SagaEventHandler`.

To start a saga execution the framework needs to receive the event with the special event handler annotated with `@StartSaga`. By default, the new instance will be created only if the corresponding saga can be found.

Ending of the saga can be defined in two means – by the event or by the API call. If the ending event is used, than the conforming event handler needs to be annotated with the `@EndSaga` annotation. Alternatively, the conditional end of the saga can be signaled by the call to the `SagaLifecycle.end()` from some method inside the saga class.

As many instances of the saga class may exist at the same time, there is a need to publish events only to the saga for which they are intended. This is done by a definition of association values. The association value is a simple key-value pair where the key is a property present in the event which forms a connection to the saga instance. The `@SagaEventHandler` annotation contains a custom attribute called the `associationProperty` which denotes the key property in the incom-

ing event. Axon also allows the definition of additional association values by a call to the `SagaLifecycle.associateWith(key, value)` and the `SagaLifecycle.removeAssociationWith(key, value)` inside the saga class.

4.6.2 Eventuate.io

To correctly connect to the Eventuate platform structure, services are required to specify a collection of the application properties. These options define connection and authentication details for Eventuate support services. The Spring Boot application can specify these attributes in an `application.properties` file or as environment variables.

4.6.3 Narayana LRA

The coordinator is responsible for the LRA starting, progress tracking and either completion or compensation.

4.6.4 Eventuate Tram

the platform controls the information transfer. The base Eventuate platform builds the communication exchange on top of the event processing. On the other hand, Tram handles the command responses as messages send trough dedicated channels.

5 Saga implementations comparison example

As a part of the investigation of the each discussed framework from the previous chapter, I created a sample application simulating the saga utilization. The main goal of this quickstart projects is to compare the base attributes of the investigated saga solution frameworks. This includes the comparison of the development model, microservices feasibility, maintainability, scalability, performance and the applicability of the reactive principles within the saga execution. The examples created for this thesis may be considered as artifacts of the one iteration of the design science research [61].

The application represents a backend processing for orders with a simple REST[1] user interface. Users are also allowed to query persisted information through the defined microservices APIs available in appendix E.

All examples are based on the microservices pattern. As every framework is suitable for the use in different environments, each example is achieving the same goal through the different portfolio of technologies. The exact mechanisms used is individual projects will be discussed in more detail in their respective sections.

5.1 Common scenario

A user is able to create the order by a REST call to the dedicated endpoint of the `order-service` microservice. The request must provide a product information JSON[2] containing a product id, a commentary and a price. For the simplicity reasons, the order always consists only of the single product. The figure 5.1 shows the example of the input JSON format for the product data. The complete REST API[3] for each individual example is provided in the Appendix E.

The setup of each example is described in detail in their respective repositories included in the Appendex B. Generally, each service is a standalone Java application that must run in a separated terminal instance by default located on the local computer address (localhost).

1. Representational State Transfer
2. JavaScript object notation
3. Application programming interface

```
{
    "productId": "testProduct",
    "comment": "testComment",
    "price": 100
}
```

Figure 5.1: Product Information example JSON

Except for the LRA example, all examples are also able to run on the Docker[62] platform using the Docker compose project[63].

Every saga invocation is asynchronous - the REST call for the order request directly returns an order identification number in the response. All of the following interactions are documented in individual services by messages that are logged by the underlying platform. The overall saga process can be examined in the `order-service` or in the case of LRA in the `api-gateway` modules.

Users are also allowed to query the persisted saga information (orders, shipments and invoices) by the respective REST endpoints described in the Appendix E. For the CQRS based examples this information is available at the `query-service` microservice, otherwise each service is expected to be responsible for maintaining its individual persistence solution which corresponds with the microservices pattern definition.

5.2 The saga model

The saga pattern used in this application is able to create orders. The order saga consists of three parts – the production of a shipping and an invoice information and if both invocations are successful, the actual order creation. If any part of the processing fails, the whole progress is expected to be undone. For instance, if the shipment is successfully created but the invoice assembly is not able to be confirmed, the persisted shipment information as well as the order must be canceled (also optionally notifying the user that the order cannot be created).

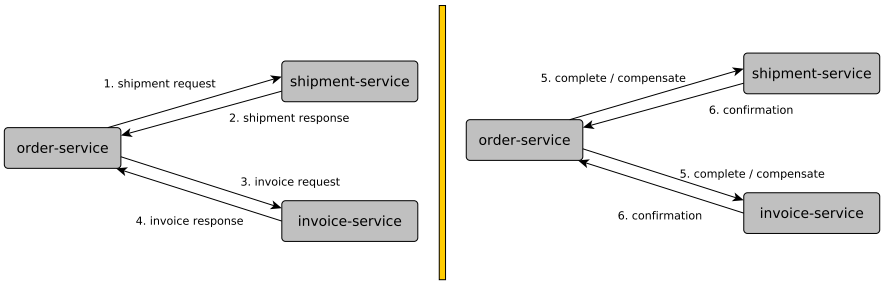

Figure 5.2: The saga model

The graphical representation of the saga progress is available in the figure 6.2.

Every application is able to demonstrate three testing scenarios: the valid pass, the shipment failure and the invoice failure. In the valid scenario after the order is requested, the saga propagation invokes requests for the shipment and the invoice. If the connection between services is stable, both participants successfully return a stub answer and the order is completed.

As most of the applied platforms invoke participants in the synchronous way, we distinguish separated member failures of the shipment or invoice. The shipment failure simulates the termination of saga without the full request coverage. This means that the compensations are distributed to all services including the `invoice-service` which has not received the work request for the order being processed yet. The scenario demonstrates the need of microservices to be able to react to the requests which are not associated with any saga which means actively keeping track of the sagas being currently executed.

The invoice failure scenario on the other hand validates that the saga compensations are executed on all participating services as the shipment is already expected to be completed. Generally, the saga pattern assumes that the compensations of the participants are called in the reverse order of the invocations because of the possible dependencies between them.

To initiate the failures scenarios in examples both `shipment-service` and `invoice-service` are equipped with injected failure conditions.

To invoke the failure the quickstarts expect a product information containing a specialized product identification: `fail-shipment` or `fail-invoice` respectively.

The graphical representation in form of the sequence diagrams for corresponding scenarios is available in the Appendix D.

5.3 Axon service

As it was stated in the previous chapter, the Axon framework is based upon the CQRS principles. Because of this nature, it would be difficult not to follow this pattern. The individual services contain separated aggregates[4] each processing its respective commands and producing various events. Any inter-service interaction is restricted to the use of the command and event buses.

5.3.1 Platform

Axon service is a Java Spring Boot [64] microservices application. Each service is fully separated and independent Maven project [65]. Every project is standalone runnable application (fat java archive (jar)) as the Spring Boot does not use any underlying platform to run microservices.

As a CQRS based quickstart, Axon service uses two different and separated communication channels to exchange information between services: the command bus and the event bus. By default, the Axon framework reduce both channels to one Java Virtual Machine (JVM) thus one microservice. However, developers are also able to specify several specialized ways of the configuration to distribute messages between different services which is used in this Axon quickstart.

The quickstart uses a motion of the distributed command bus which is based upon a different approach then the one used in the traditional single JVM Axon applications. The distributed command bus forms a bridge between separated command bus implementations to transfer commands between different JVMs [66]. Its main responsibility is the selection of the communication protocol and the choice of the target destination for each incoming command.

4. for the definition of aggregate please refer to the Appendix A

Axon provides two options for connecting different services through the distributed command bus – `JGroups Connector` and `Spring Cloud Connector`. The one used in this quickstart is the Spring Cloud method. The underlying implementation employed in this quickstart is based on the Netflix Eureka Discovery and Eureka Server combination [67]. Each business service as a part of its initialization registers itself with the `registration-server` service which function as the Eureka server. Axon is then able to redirect commands to the right service chosen by the value of parameters in the command class annotated by the `@TargetAggregateIdentifier` annotation.

For the distribution of the event bus Axon service uses an external messaging system based on the Spring Advanced Message Queuing Protocol (AMQP) protocol called RabbitMQ message broker [68]. The quickstart uses separated messaging queue for each business service and one separated queue for the `query-service` microservice which is subscribed to all of the processing events. Axon platform provides the direct support for the AMQP so no specific handling of the produced events is required – Axon automatically distributes events to all connected queues.

It is worth mentioning that both the Spring Cloud and the RabbitMQ message broker are required external providers that need to be started before the deployment of the business services. Unfortunately, by the time of this writing there is no way to distribute commands or events directly in the Axon platform.

5.3.2 Project structure

The application is composed of five microservices – the `order-service`, the `shipment-service`, the `invoice-service`, the `query-service` and the `registration-server`. Furthermore it also contains a separated project `service-model` which serves as a support library for the other microservices.

As it was stated in the previous section `registration-server` microservice is a Spring Boot application which function as a Netflix Eureka server. Other business services act as clients for this server, therefore it is required that this service is initialized by the time they try to register.

The `order-service` project is a business microservice responsible for the saga handling. It contains the logic for the order request, the saga initiation and the saga compensation.

Both `shipment-service` and `invoice-service` are business services functioning only as the saga participants. Their only obligation is to provide their respective computations.

The `query-service` is a specific microservice included for the purposes of the CQRS pattern. It collects the information of the prepared orders, shipments and invoices and provides an external APIs for the querying of these resources.

Finally, the `service-model` is a Kotlin and Java Maven application providing the core API for the commands and events used by various business services. This is required as all classes must match in order for particular handlers to be invoked. Furthermore, it also provides common utilities and the logging support. It is mandatory to include this project on the classpath of the every other service.

5.3.3 Problems

Maintenance of the saga structure

The one substantial problem the saga processing in Axon has is the missing structure of the internal life cycle of the saga. Axon only provides ways to indicate the start and the stop of the saga. The actual invocation of the participants, collecting of the responses and handling of the compensations is up to developer as the only way of communication with the saga is through events.

In this application the `OrderManagementSaga` contains two internal classes – `OrderProcessing` and `OrderCompensationProcessing` which are responsible for keeping track of the saga execution and compensation respectively. As production ready sagas can be expected to run in a number of days, this can quickly become the bottleneck of the saga maintenance.

Distribution of events to sagas

Another encountered problem is that by the time of this writing Axon does not provide an easy method for the configuration of the different event bus for saga events than is the one that is used by default. The `@Saga` annotation is preconfigured with the value of the local event bus which is not allowed to be changed. The workaround is to manually register a custom saga manager which is not straightforward from the user perspective when the saga needs to be distributed through different JVMs.

AMQP usage with sagas

When the distributed event bus is processing events from an AMQP queue which the saga class is listening to, the framework does not deliver events correctly to the attached handlers. This may be caused by the incorrect configuration from the previous problem. The workaround used in the quickstart is to artificially wait 1000 milliseconds before delivering the event from the queue to the framework.

CQRS restrictions

As CQRS is a pattern that manages the domain formation of the application, Axon can place a hard requirements for the projects that do not follow the CQRS domain separation. Like it was already presented, sagas in Axon are only a specialized type of the event listener. The only way Axon produces events is trough an interaction with the aggregate instance - events are produced purely as a response to the received command. Therefore the use of Axon sagas in not CQRS environment may be too restrictive for an implementation.

5.3.4 Technology

This example contains a set of Spring Boot applications which can be run directly from the command line. Additionally, it also contains docker and docker-compose configuration to ease the development and deployment operations. In this section we focus on the Spring platform, the Docker description is provided in the section 5.4.4.

Spring Boot

Spring Boot is a framework built on top of the Spring Framework [69]. Its main focus is on the creation of standalone runnable applications that are easily employed in microservices architectures. It favors convention over configuration to allow the usage of Spring features with a little of the Spring configuration [64].

The Spring Boot platform is composed into aggregate modules known as *starters*. The starter is a dependency descriptor which contains dependencies that are required to provide some functionality to the application. In order to use Spring Boot, the `spring-boot-starter` core module must be incorporated. Other useful modules, for instance, `spring-boot-starter-web` or `spring-boot-starter-data-jpa` can be provided to support the microservice adoption. Starters can be packaged with the application using Maven or Gradle build configurations.

Another responsibility of this framework is the packaging of the application. The preferable way for the microservice is to create an executable fat Java archive (jar) which contains all of the application dependencies. The application can then be simply excuted by a `java -jar` command.

A spring Boot microservice must fulfill two requirements – it must follow a Maven layout convention and it must provide an entry point. This can be any class annotated with the `@EnableAutoConfiguration` annotation that starts the Spring Boot context.

Spring provides various ways for exposing available services. Registration servers like Eureka or Consul which are integrated within the Spring cloud [70] can be used as the service discovery mechanisms. For a manual approach, the microservices functionality can be exposed through the RESTful API.

5.4 Eventuate service

Similarly to Axon, Eventuate service is also based on the event sourcing and the CQRS pattern. For this reason, the business execution is managed in the aggregates which correspond to the respective microservices projects. The communication is as a result restricted to the command processing and the event appliance.

This quickstart represents the pure CQRS approach to the saga processing. This means that the whole saga implementation is created by the developer using the platform only for the event and command distribution. For this reason, the Eventuate service is more complex than any other quickstart but for the example purposes, it distinctively demonstrates how sophisticated is the saga administration provided by all remaining platforms.

5.4.1 Platform

Eventuate service is a microservices application consisting of a set of Spring Boot [64] business services, one backing module and a number of support services provided by the Eventuate platform. In this section, I will focus on the Eventuate platform and the services it provides, the business part of the application is described in the following section.

This quickstart is established as a Eventuate Local version of the platform. This means that it uses underlying SQL database for the event persistence and the Kafka streaming platform for the event distribution. Eventuate Local provides five services used by the quickstart that are managed by the platform, namely Apache Zookeeper, Apache Kafka, MySQL database, the change data capture component and the Eventuate console service. The example employs these services as a Docker images included in the provided docker-compose configuration.

Apache Zookeeper service

Apache Zookeeper is an open-source project which enables highly reliable distributed coordination [71]. It maintains a centralized service which supervise various functionalities like handling of the configuration information, synchronization, naming or grouping. The Eventuate Local platform provides its own Docker image tagged as `eventuateio/eventuateio-local-zookeeper`.

Apache Kafka service

The Apache Kafka streaming platform is the service which is responsible for the administration of subscription and publishing mechanisms controlling the event processing for the business microservices. As it is based on the Streams API it allows the platform to react to events in real time. Eventuate manipulates Kafka as the notification service for the event propagation. Eventuate ships its own Kafka version in the `eventuateio/eventuateio-local-kafka` docker image.

MySQL database service

The SQL database used in this application for the event persistence is the MySQL open-source database which is currently the only database supported by the platform. The Eventuate Local maintains two tables – `EVENTS` and `ENTITIES`. This database serves also as a transaction log maintained as a mean for the event sourcing. The containerized version is located under `eventuateio/eventuateio-local-mysql` tag.

CDC service

This service represents the change data capture (CDC) component. The CDC service has two main responsibilities – it follows the transaction log and it publishes each event which is inserted into the `EVENTS` table to the Kafka topic that corresponds to the aggregate for which the event is intended. Eventutate Local supports two options of the execution of the CDC – internally in each business service or as a standalone application. This quickstart applies the Eventuate CDC service `eventuateio/eventuateio-local-cdc-service` as a standalone Docker container.

Eventuate console

The last support service is the `consoleserver`. It provides a simple interface for accessing the information about created aggregate types and the event log. The supplied Docker container image is the `eventuateio/eventuateio-local-console`.

5.4.2 Project structure

This section describes the set of services composing the business side of the application. This set contains four services that cover the saga execution and data processing (`order-service`, `shipment-service`, `invoice-service` and `query-service`), one service (`mongodb`) representing the persistent storage and one additional support module (`service-model`).

All of the business services are a Spring Boot applications based on the Gradle [72] build system. Each microservice is represented as a independent module capable of being separately built and deployed. Even if Spring Boot projects can be executed directly from the command line as ordinary Java applications, this quickstart leverages the Docker approach of the Eventuate Local platform and containerize all of its services.

To connect to the Eventuate platform each service defines a set of environment variables. This information includes the connection and authentication details for the MySQL database, the CDC component and the connection URLs[5] for the Kafka and the Zookeeper services. These variables are specified in the container specification for each individual business service in the `docker-compose.yml` file.

The actual saga execution is managed in the `order-service` microservice. The saga realization implementation is contained in three classes – the `OrderSagaAggregate`, the `SagaEventSubscriber` and the `OrderSagaService`. The first class is an ordinary CQRS aggregate that handles the commands for the saga initialization and the participants outcomes. Conversely, the latter one is the event processor listening for the events produced by the aggregate which is basically a wrapper around the `OrderSagaService` - the class responsible for the remote

5. Uniform Resource Locator

REST calls to the other services and the command dispatching for the `OrderSagaAggregate`. The usage of the separated event listener is required because Eventuate does not allow aggregates to be declared as Spring components. The reason of this defect is described in more detail in the following section.

Except for the normal order API, the `order-service` also provides a management API for the participants to be able to share the information about their processing. This endpoints are hardcoded in the application which may not be acceptable for a production realization.

The `shipment-service` and the `invoice-service` both contain a simple aggregate together with its associated event listener which control the participant interaction with the saga. Each service also accommodate the REST endpoint for the saga request and its possible compensation.

Similarly to Axon service, the `service-model` project acts as a support library for other services. It contains a core API for each business service which needs to be shared and the utilization classes.

The last business microservice is the `query-service`. It performs as a response service providing the information about persisted orders, shipmetns and invoices. It contains an event listener for each designated microservice which in turn preserve the achieved information in the Mongo NoSQL database. This service also provides a simple Swagger interface to ease the user application interaction.

5.4.3 Problems

Complexity

As this project represents a plain CQRS based example it completely demonstrates the background process required for the saga execution. Therefore the complexity of this quickstart may appear more critical than in other projects as the background saga execution often contains many optimizations.

The first complexity problem is that the project contains a great number of the command and event classes. This is required as aggregate classes are only able to consume commands and produce events. For that reason, the communication between components often demands a few additional steps.

The full saga administration is handled by the project from the very beginning. That covers the support of starting, stopping and following the saga execution as well as saga compensations. The restrictions placed by the CQRS pattern furthermore put additional requirements on the saga processing which may not be demanded by other frameworks. Before the Eventuate Tram framework, the Eventuate platform did not provide any saga support.

This quickstart uses the REST architectural style for the remote communication between services. Even if all of microservices are connected to the same MySQL database, they cannot directly propagate commands between each other. This is due to the way Eventuate dispatches commands through the aggregate repository. The aggregate repository represents the database table that is restricted to one aggregate and consequently, it needs to be injected by the platform. For this reason, it needs to declare the target aggregate class and the command type. The sharing of the aggregate class may be very restrictive, especially for microservices applications.

Aggregate instantiation

The Eventuate framework creates the instances of the aggregate classes by a call to the default constructor. This effectively prohibits the use of aggregate instance managed by the underlying server container.

For this reason, each aggregate in this project is separated into two classes – the actual aggregate responsible for the command processing and the event subscriber instance managing the incoming events. The aggregate class is required to extend the `ReflectiveMutableCommandProcessingAggregate` specifying the type of the command interface which allows the classpath instantiation. The event listener is defined by the `@EventSubscriber` annotation which permits it to be constructed as a Spring container component for the dependency injection employment.

This restriction is seconded by the rule stating that each produced event from the aggregate's command processing method must also be applied by the different method of the same aggregate. This limitation exists because of the event sourcing feature providing the ability to replay already executed commands to reconstruct the aggregate's

state in the case of failure. The aggregate then may contain unnecessary empty methods as the saga also requires the propagation of the information to different components (e. g. the REST controller).

Event entity specification

As well as the command type, Eventuate also requires the definition of the event type each aggregate is able to produce. The event class is defined as a value of the `entity` attribute of the `@EventEntity` annotation. This annotation is usually placed on the event interface which implementation represents produced events.

The problem rises when the events needs to be shared between several modules. This is a common requirement as the CQRS pattern requires the query domain to be separated. The event interfaces are therefore included in the common library module as the `service-model` used in this project. The hard coded information of the full name of the aggregate class used in the `@EventEntity` annotation then may become hard to maintain.

Platform structure

The platform structure places the obligation on each developed microservice to conduct with the connecting specification. This means that every service must provide linking information for the Eventuate platform services described in the previous section, namely MySql database, Apache Kafka, Apache Zookeeper and CDC component. This information is manually replicated in each service (restricted to system properties) and therefore predisposed to errors.

In the end it is worth mentioning that as the Eventuate service is the pure CQRS saga example it has a few problems which has been reduced or removed in the later Eventuate Tram implementation that is in detail described in the following section.

5.4.4 Technology

This section provides a detailed view of the Docker platform, for the description of Spring Boot microservices please refer to the section 5.3.4.

Docker

Docker is an open source container platform designed to make it easier to build, secure and manage the widest array of application from development to production both on premises and in the cloud [62]. Docker containers allow applications to run on top of the kernel services provided by the hosting system which considerably effects the application performance. However, it still builds containers on top of the generalized interface which warrants straightforward portability between different machines.

A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it – code, runtime, system tools, libraries or settings [62]. All docker containers that run on the same machine share the kernel services of the host. The images are build on the concept of layers. The layer provides an

abstraction to share common filesystems, configuration and other data that can be reused by several docker containers.

Containers isolate applications from the operating system their running on and also provide the separation from other docker containers running concurrently on the same computer. Instead of virtual machines which provide similar functionality, Docker virtualizes the operating system not the hardware. Docker provides abstraction at the application level.

Docker as a tool is targeted for simple utilization. It provides an unified environment for both developers and administrators supporting the DevOps (development and operations) practices. Particularly, developers profit from portable code that is able to run on any operating platform supporting Docker, while operations gain visibility and management services from comprehensive control panel covering all containerized applications.

5.5 LRA service

The LRA service is the first example in contradiction with previous quickstarts as it does not restrict its services to any particular conventions. Individual services are connected through the exposed REST routes.

5.5.1 Platform

This quickstart is composed as a set of WildFly Swarm microservices applications. Every microservice is designed to be easily deployed to the OpenShift container application platform provided by Red Hat, Inc. Both of these platforms are in detail described in the following sections.

Each service uses the `fabric8-maven-plugin` for the build and the deployment to the OpenShift platform. This plugin provides a straightforward way of declaring the necessary description information the platform requires to orchestrate the service according to the user demands. This project applies the source to image (S2I) toolkit that builds imminent Docker images which can by immediately deployed to the OpenShift.

As it was already presented in the previous chapter Narayana's long running actions are not composed as a platform, rather as a standalone library. This project builds upon the use of the Narayana LRA coordinator which is same as the other services constructed as a WildFly Swarm microservice called `lra-coordinator`.

The traditional requirements placed on the microservices applications are handled by the OpenShift platform. That for instance covers service discovery and location transparency, monitoring, logging, resiliency and health checks (failure discovery).

5.5.2 Project structure

This project consists of five WildFly Swarm microservices and one support module. Each service is designed and configured with the `fabric8-maven-plugin` providing simple deployment to the OpenShift platform. Additionally, services contain a customized Dockerfile specifying environment properties and the target Swarm uberjar file which is used for the source-to-image (S2I) builds in OpenShift.

The support library project is called the `service-model`. This module is rosponsible for the definition of the LRA information, the specification of the exchanged JSON data formats, the characterization of the communication model used in other services and the administration of common utilities.

The `LRADefinition` class denotes the JSON format of the LRA representation. It presents a simplified version of the LRA capabilities for the example purposes. The definition includes only required attributes – the name of the LRA, a list of individual actions that form the LRA and the unspecified object containing the user defined information associated with the LRA. The example LRA definition JSON format is available in the figure 6.3.

The individual actions that compose the LRA are incident to the pattern services in this quickstart use for the communication. The information exchange is based upon the REST architectural style which expects that services adhere to predefined endpoint rules.

The action definition consists of the name, the action type and the service for which the invocation is intended. The `service-model` project contains both `ActionType` and `Service` enumerations that de-

Figure 5.3: LRA definition example JSON

Figure 5.4: Action example JSON

note possible values. The example action JSON is included in the figure 6.4.

The actual deployable microservices project consists of five services – `api-gateway`, `order-service`, `shipment-service`, `invoice-service` and `lra-coordinator`. Every service is configured with the addresses of other services as they reflect the OpenShift/Kubernetes application names. All of exposed APIs are defined in the Appendix E.

The services that provide the LRA execution capabilities are the `order-service`, the `shipment-service` and the `invoice-service`. Every one of these services provides a simple computation that contributes to the LRA realization. Additionally, `order-service` also provides a user invocation endpoint that can initiate the LRA. As all three services eventually subscribe to the LRA when they are called, they all provide REST endpoints annotated by the LRA annotations for the completion and compensation invocations. Each service is configured with an instance of H2 SQL database for the data persistence.

The `lra-coordinator` project is provided by the Narayana framework. Although the LRA specification does not require the application

of the REST architectural style, the `lra-coordinator` operates a set of REST endpoints maintaining the starting and managing of the LRA, gathering the information about active and recovering LRAs, the management of the nested LRAs and the ability of participants to join or leave the LRA. Narayana distributes this project already as a WildFly Swarm binary but this quickstart, for the investigation purposes, builds it still as a part of the S2I deployment.

Even if the `lra-coordinator` presents the REST endpoints for the LRA management, this quickstart invokes the coordinator by the client module provided by Narayana. The `lra-client` dependency provides the `LRAClient` class that is configured with the coordinator location and serves as a proxy separating the user from the actual REST invocations. This class is defined as a CDI bean to enable the dependency injection across the project. `LRAClient` is able to also control the whole LRA lifecycle but the prefered method is to use the annotations present in the `io.narayana.lra.annotation` package.

The last service is the `api-gateway`. This module functions as an interface that makes the LRA execution transparent for the invoking services. The current state of the Narayana handling of LRA calls will be described in the following section.

The `api-gateway` uses the LRA and action definition classes from the `service-module` to handle the LRA processing on the behalf of the initiating user. It exposes a REST interface that consumes the LRA definition JSON.

The actual LRA execution is managed in the `LRAExecutor` class. This class provides one public method `processLRA(LRADefinition lraDefinition, String baseUri) : LRAResult` that is responsible for starting and performing of the LRA, collecting the participants results and closing or compensating of the LRA. The `baseUri` attribute is present because the `api-gateway` service subscribes to the started LRA as the initiator.

As this module was designed for this particular LRA scenario, it is configured to execute the LRA actions (shipment and invoice requests) independently and in parallel. It collects the result of each action and eventually closes or cancels the LRA with methods provided by the `LRAClient`. Certainly, this is an area which could be in a more general execution module further extended with e.g. sequential configuration or LRA nesting.

5.5.3 Problems

The Narayana Long running actions a very nice development model for the microservices applications. As it aims for the compatibility with the MicroProfile specification, it does not restrict services to essentially any particular implementation restrictions. Even if the MicroProfile is restricted on REST invocations, Narayana LRA specification does not require the usage this architectural style for the communication with the coordinator. However, the only implementation that is currently available is based on REST, but in can be efficiently extended to other communication protocols in the future.

The only problem that may occur is that currently, the LRA framework provides only coordination and management capabilities, it does not handle the saga structuring and execution. This may be controversial as any of the previous frameworks did not exposed this functionality neither. The idea was introduced by the Eventuate Tram framework which is described in the following section. This concern may be troublesome for the enterprise ready reactive microservices, as it may place restrictions on their availability under substantial traffic.

In the LRA service is this problem addressed by the `api-gateway` service which functions as a proxy for the `lra-coordinator` that handles the saga execution. This approach does not effect the Narayana LRA management. The only change from the traditional processing is that series of LRA actions are performed by the `api-gateway` instead of the initiating service. This allows the service to be immediately available for the subsequent user requests and to scale the LRA processing independently from the services that utilize the coordination.

5.5.4 Technology

This section describes technologies included in the LRA example – the WildFly Swarm project as underlying microservices provider and the OpenShift and Kubernetes containerization platforms as the quickstart has been designed for the deployment to these environments.

WildFly Swarm

WildFly Swarm is the Red Hat microservices initiative designed to enable deconstructing the WildFly application server and pasting just enough of it back together with the application to create a self-contained executable jar [73].

The traditional Java Enterprise Edition (EE) approach follows the development of the application and its successive deployment to the application server which includes necessary dependencies which the application requires to run. On the other hand, Wildfly Swarm creates a fat Java archive which packages all needed dependencies inside itself. This emulates the packaging of only requisite parts of the application server. The result jar is a standalone runnable Java application which can be executed by the `java -jar` command. It also provides Maven and Gradle plugins to ease the development of Swarm applications.

The default fat jar (also called the uberjar) contains the user application and the needed parts of the WildFly server. Swarm also supports the packaging of the necessary server parts separately from the application. This method is know as the hollow jar and is particularly useful in the containerized environment as the server may be placed in the lower layers that do not require frequent rebuilds.

The individual server parts are being delivered in the packages named fractions. The fraction represents a precise selection of capabilities that can be included in the application. It may denote the exact WildFly subsystem as JAX-RS[6] or CDI[7], or a more sophisticated set of facilities to provide some additional functionality like RHSSO[8].

OpenShift platform

Red Hat OpenShift is an open source container application platform that brings Docker and Kubernetes to the enterprise [52] generally build on top of the Red Hat Enterprise Linux. It provides the deployment, management and monitoring of the containerized software. OpenShift provides automation in the cloud environment that

6. Java API for RESTful Web Services
7. Context and Dependency Injection
8. Red Hat Single Sign-On

enables simple development workflow including easy provisioning, building and deployment of enterprise applications allowing faster delivery to end customers.

The platform provides extensive set of features like self-service maintenance, polyglot (language independent) application support, container-based environment and the automation of application builds, scaling and health management. It can also administer persistence capabilities, the application centric networking and multiple interaction models, e. g. command line tools or the web console.

OpenShift is being developed in several variants. The upstream community project is OpenShift Origin which is a distribution of Kubernetes optimized for continuous application development and multi-tenant deployment [52]. On top of the Kubernetes platform, Origin provides the developer and operations centric tooling, security, logging or pipelining and many other capabilities. Origin is also available as the all in one virtual machine called Minishift which utilizes a local single-node OpenShift cluster.

The second alternative is the OpenShift Online. Currently distributed in version 3, it serves as a Red Hat public cloud application development and hosting service. OpenShift Online provides an integrated environment that allows developers to focus on the application development instead of its management through the set of facilities like source-to-image builds eliminating the Dockerfiles creation, one click deployments through git hooks, automatic scaling according to the traffic and integration with the Eclipse IDE[9].

OpenShift Dedicated offers a managed private cluster which can be hosted on the public cloud like Amazon Web Services or Google Cloud platform. It provides OpenShift services as an isolated platform that aims for the simple and faster transition of user applications to the container based and native cloud environments.

The last OpenShift variant it the OpenShift.io. It provides an open online end-to-end development environment for planning, creating and deploying hybrid cloud services [74]. It supports an integrated approach to DevOps, including tools as one-click container management, machine learning system and integration of many projects like fabric8 or Eclipse Che.

9. Integrated Development Environment

Kubernetes

Kubernetes is an open source project providing automation, scaling and management of containerized applications [53]. It groups the application containers into logical units for easier management and discovery.

The features of Kubernetes include the service discovery, load balancing, automatic container placement or the self-healing for the automated failure recovery and rollbacks. It also manages the storage orchestration, scaling of containers, secrets, container configuration and batch capabilities.

Kubernetes platform is suitable and portable to any cloud environment involving public, private, hybrid clouds and multi-cloud. It allows application containers to be run in the clusters of physical or virtual machines. Kubernetes is not a traditional PaaS (Platform as a Service) solution but it provides platform that many PaaS systems build upon, e. g. OpenShift or Deis.

5.6 Eventuate Sagas

The Eventuate Sagas quickstart is based on the new Eventuate Tram framework which is still, by the time of this writing, under pre-release development. This framework provides several solution to problems with saga management that are present in the Eventuate platform.

5.6.1 Platform

Similarly to the full Eventuate distribution, the Eventuate Tram establishes four services that form the Tram platform: Apache Zookeeper, Apache Kafka, MySQL database and CDC component. The fifth Console service is not present as the platform does not provide this functionality by the time of this writing. As all mentioned services represent the same functionality as they are responsible for in the full Eventuate platform, the individual descriptions of each service is defined in detail in the section 5.4.1.

All services are deployed by the `docker-compose` configuration distributed with the framework. This setup is based on the same

Eventuate docker images for `zookeeper` and `kafka` services, and with `mysql` and `cdcservice` redefined by Tram.

5.6.2 Project structure

As Eventuate Tram does not restrict its services to the CQRS pattern, this project, in contrast to Eventutate service, contains only three business microservices and one support module. Every service is configured in the similar way as for the full platform containing the references and authentication details for the MySQL database, Kafka framework and Apache Zookeeper service. The quickstart is distributed with the predefined `docker-compose` configuration file that enables one command start up of all services.

The last service which is not required for the saga execution and handling is the `mongodb` NoSQL server. This service is present only for the demonstration purposes to allow the data retrieval on individual business services from different storage than the one that is used by the Eventuate platform.

The first microservice `order-service` is responsible for the order requests and saga processing. It also provides the ability to query persisted orders from the remote Mongo database.

The most important element in the `order-service` is the saga definition that is located in the `OrderSaga` class. This definition uses the declarative approach with the fluent API to denote the saga in steps of execution. Every step declares a handler method to be invoked when the step is reached by a reference to private methods in this class.

The step provides an ability to advance the saga execution in three ways – by invoking of the local function, by a call to the remote participant or by the definition of the compensation method for the saga. Furthermore, the participant is able to define individual actions that comprise its engagement in the saga, the compensation handler and several reply handlers that are distinguished by the data object class that is received in the reply. This definition provides a simple in one place saga specification which is suitable for easier maintenance and distribution. This declarative approach provides many advantages that will be detailed in the following chapter.

The last two business services that contribute to the saga execution are `shipment-service` and `invoice-service`. Both of these microser-

vices define several command handler methods associated with the channel that is dedicated to the service.

The channel is a main communication mechanism used in Eventuate Tram. It is denoted by a string name that needs to be specified in the command message as a target destination. The definition of channel associates commands that it is able to receive with command handling methods that are invoked when the corresponding commands are delivered. The command handler returns a `Message` object identifying the outcome of the invocation. The failure outcome of any participant will immediately result in the saga compensation.

Both services are also connected to the Mongo database server in order to provide the browsing of the created shipments or invoices respectively.

In conclusion, Tram remarkably simplified the Eventuate platform for the usage of sagas. Most importantly, it introduced a simple fluent API for the saga definition and the loss of CQRS restrictions. Altogether, Tram platform makes a suitable saga solution for microservices based environment.

5.6.3 Problems

Destination identification

The destination channels in Tram are distinguished by a simple string which may cause problems in the case of name conflicts. Currently, the choice of the handler to be invoked depends on two resources – the name of the channel and the command dispatcher id. When both strings match the same destination even in different services, the platform delivers the commands between handlers in random fashion which may become a complex issue in larger projects.

Command handlers

Handler methods that are referenced in the definition are restricted to the communication model provided by the platform. This allows a single command to be sent to the required destination. Unfortunately, platform does not allow the saga to perform any other operation without the participant invocation which may lead to unnecessary empty commands and channels declarations.

Similarly, the saga may need to interact with the same participant in several different commands. This may cause problems with definitions of compensation and reply handlers as the developer needs to mind the logical grouping of participant invocations.

5.6.4 Technology

Eventuate Tram is based on the same microservices platform structure as standard Eventuate framework and therefore the same technology. For the description of the technology employed in this quickstart please refer to the section 5.4.4.

5.7 Performance test

6 LRA execution extension

7 Conclusion

Bibliography

[1] M. Little, J. Maron, and G. Pavlik, *Java transaction processing*. Prentice Hall, 2004.

[2] M. Musgrove, "Narayana + wildfly," 2015. [Online]. Available: https://developer.jboss.org/servlet/JiveServlet/download/53044-3-129391/jbug-brno-transactions.pdf

[3] T. Haerder and A. Reuter, "Principles of transaction-oriented database recovery," *ACM Computing Surveys*, vol. 15, no. 4, pp. 287–317, 1983.

[4] E. Brewer, "CAP twelve years later: How the "rules" have changed," *Computer*, vol. 45, no. 2, pp. 23–29, feb 2012.

[5] M. Kleppmann, *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, 2017.

[6] M. Kleppmann, "Transactions: myths, surprises and opportunities," 2018. [Online]. Available: https://martin.kleppmann.com/2015/09/26/transactions-at-strange-loop.html

[7] M. Richards, *Java transaction design strategies*, 1st ed. C4Media, 2006.

[8] Spring community, "Spring documentation/Transaction Management," 2018. [Online]. Available: https://docs.spring.io/spring/docs/current/spring-framework-reference/data-access.html#transaction

[9] G. Lea, "Distributed transactions: The icebergs of microservices," 2016. [Online]. Available: http://www.grahamlea.com/2016/08/distributed-transactions-microservices-icebergs/

[10] C. X/Open Company Ltd., *X/Open CAE Specification: Distributed Transaction Processing: CPI-C Specification, Version 2*. Upper Saddle River, NJ, USA: Prentice-Hall, Inc., 1996.

[11] L. Lamport, R. Shostak, and M. Pease, "The byzantine generals problem," *ACM Transactions on Programming Languages and Systems*, vol. 4/3, pp. 382–401, July 1982. [Online]. Available: https://www.microsoft.com/en-us/research/publication/byzantine-generals-problem/

[12] H. Robinson, "Consensus protocols: Two-phase commit," 2008. [Online]. Available: http://the-paper-trail.org/blog/consensus-protocols-two-phase-commit/

[13] D. Skeen, "A quorum-based commit protocol," Ithaca, NY, USA, Tech. Rep., 1982.

[14] I. Keidar and D. Dolev, "Increasing the resilience of distributed and replicated database systems," *J. Comput. Syst. Sci.*, vol. 57, no. 3, pp. 309–324, Dec. 1998. [Online]. Available: http://dx.doi.org/10.1006/jcss.1998.1566

[15] L. Lamport, "The part-time parliament," *ACM Trans. Comput. Syst.*, vol. 16, no. 2, pp. 133–169, May 1998. [Online]. Available: http://doi.acm.org/10.1145/279227.279229

[16] M. Štefanko, "Messaging providers in the silverware microservices platform," Bachelor's thesis, Masaryk University, Faculty of Informatics, Botanická 68a, Brno, Czech republic, 6 2016.

[17] D. Barry, "Service-Oriented Architecture (SOA) Definition," 2016. [Online]. Available: http://www.service-architecture.com/articles/web-services/serviceoriented_architecture_soa_definition.html

[18] J. Bonér, *Reactive microservices architecture*, 1st ed. O'Reilly Media, Inc., 2016.

[19] J. Lewis and M. Fowler, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html

[20] I. H. Sarker and K. Apu, "Mvc architecture driven design and implementation of java framework for developing desktop application," *International Journal of Hybrid Information Technology*, vol. 7, no. 5, pp. 317–322, 2014.

[21] E. Stump P.E., "All about learning curves," *Galorath Incorporated*, 2014.

[22] C. Richardson, "Pattern: Monolithic architecture," 2017. [Online]. Available: http://microservices.io/patterns/monolithic.html

[23] C. Richardson, "Introduction to Microservices | NGINX," 2015. [Online]. Available: https://www.nginx.com/blog/introduction-to-microservices/

[24] W. Hasselbring and G. Steinacker, "Microservice architectures for scalability, agility and reliability in e-commerce," *2017 IEEE International Conference on Software Architecture Workshops (ICSAW)*, 2017.

[25] K. Lieberherr and I. Holland, "Assuring good style for object-oriented programs," *IEEE Software*, vol. 6, no. 5, pp. 38–48, 1989.

[26] R. C. Martin and M. Martin, *Agile principles, patterns, and practices in C#*, 1st ed. Prentice Hall, 2006.

[27] M. D. McIlroy, E. N. Pinson, and B. A. Tague, "Unix time-sharing system: Foreword," *Bell System Technical Journal*, vol. 57, no. 6, pp. 1899–1904, 1978.

[28] S. Newman, *Building Microservices*, 1st ed. O'Reilly Media, Inc., 2015.

[29] S. Newman, "Principles of Microservices," 2016. [Online]. Available: http://samnewman.io/talks/principles-of-microservices/

[30] E. Evans, *Domain-driven design*. Addison-Wesley, 2003.

[31] M. Conway, "Conway's law," 2018. [Online]. Available: http://www.melconway.com/Home/Conways_Law.html

[32] N. Busi, R. Gorrieri, C. Guidi, L. roberto, and G. Zavattaro, "Choreography and orchestration conformance for system design," *Lecture Notes in Computer Science*, p. 63–81, 2006.

[33] R. Dijkman and M. Dumas, "Service-oriented design: A multi-viewpoint approach," *International Journal of Cooperative Information Systems*, vol. 13, no. 04, pp. 337–368, dec 2004. [Online]. Available: https://doi.org/10.1142%2Fs0218843004001012

[34] K. Benghazi, M. Noguera, C. Rodríguez-Domínguez, A. B. Pelegrina, and J. L. Garrido, "Real-time web services orchestration and choreography," in *Proceedings of the 6th International Workshop on Enterprise & Organizational Modeling and Simulation*, ser. EOMAS '10. Aachen, Germany, Germany: CEUR-WS.org, 2010, pp. 142–153. [Online]. Available: http://dl.acm.org/citation.cfm?id=1866939.1866952

[35] V. Farcic, *The DevOps 2.0 Toolkit*. Packt Publishing Ltd., 2016, pp. 222–252.

[36] J. Humble and D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley, 2010, pp. 263–265.

[37] Swagger community, "Swagger framework," 2016. [Online]. Available: http://swagger.io/

[38] M. Fowler, "Humaneregistry," 2008. [Online]. Available: http://martinfowler.com/bliki/HumaneRegistry.html

[39] M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*. Pragmatic Bookshelf, 2018.

[40] C. Escoffier, "The reactive landscape," 2018. [Online]. Available: https://www.slideshare.net/RedHatDevelopers/the-reactive-landscape

[41] J. Bonér, D. Farley, R. Kuhn, and M. Thompson, "Reactive manifesto," 2018. [Online]. Available: https://www.reactivemanifesto.org/

[42] C. Escoffier, *Building Reactive Microservices in Java*, 1st ed. O'Reilly Media, Inc., 2017.

[43] RxJava community, "Rxjava," 2018. [Online]. Available: https://github.com/ReactiveX/RxJava

[44] C. Escoffier, "5 things to know about reactive programming," 2017. [Online]. Available: https://developers.redhat.com/blog/2017/06/30/5-things-to-know-about-reactive-programming/

[45] R. S. community, "Reactive streams," 2018. [Online]. Available: http://www.reactive-streams.org/

[46] A. Rotem-Gal-Oz, "Fallacies of distributed computing explained." [Online]. Available: http://www.rgoarchitects.com/Files/fallacies.pdf

[47] W. Vogels, "Eventually consistent," *Communications of the ACM*, vol. 52, no. 1, p. 40, jan 2009.

[48] S. Gilbert and N. Lynch, "Perspectives on the CAP theorem," *Computer*, vol. 45, no. 2, pp. 30–36, feb 2012.

[49] ——, "Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services," *ACM SIGACT News*, vol. 33, no. 2, p. 51, jun 2002.

[50] D. Abadi, "Consistency tradeoffs in modern distributed database system design: CAP is only part of the story," *Computer*, vol. 45, no. 2, pp. 37–42, feb 2012.

[51] M. Fowler, "Microservice trade-offs," 2015. [Online]. Available: https://martinfowler.com/articles/microservice-trade-offs.html

[52] Red Hat, Inc., "OpenShift container application platform," 2018. [Online]. Available: https://www.openshift.com/

[53] Kubernetes, "Kubernetes," 2018. [Online]. Available: https://kubernetes.io/

[54] T. Clemson, "Testing strategies in a microservice architecture," 2014. [Online]. Available: https://martinfowler.com/articles/microservice-testing/

[55] H. Garcia-Molina and K. Salem, "Sagas," *ACM SIGMOD Record*, vol. 16, no. 3, pp. 249–259, 1987.

[56] U. R. Sharma, *Practical Microservices*, 1st ed. Packt Publishing Ltd., 2017.

[57] C. McCaffrey, "Applying the saga pattern," 2015. [Online]. Available: https://speakerdeck.com/caitiem20/applying-the-saga-pattern

[58] A. framework, "Axon framework," 2017. [Online]. Available: http://www.axonframework.org/

[59] Eventuate.io, "Eventuate.io," 2017. [Online]. Available: http://eventuate.io/

[60] Narayana, "Narayana LRA," 2017. [Online]. Available: https://github.com/eclipse/microprofile-sandbox/tree/master/proposals/0009-LRA

[61] R. J. Wieringa, *Design Science Methodology for Information Systems and Software Engineering*. Springer, 2014.

[62] "Docker," 2018. [Online]. Available: https://www.docker.com/

[63] "Docker compose," 2018. [Online]. Available: https://docs.docker.com/compose/

[64] Spring community, "Spring Boot," 2018. [Online]. Available: http://projects.spring.io/spring-boot/

[65] B. Porter, J. Zyl, and O. Lamy, "Maven," 2018. [Online]. Available: https://maven.apache.org/

[66] Axon community, "Axon framework reference guide," 2018. [Online]. Available: https://docs.axonframework.org/v/3.1/

[67] Spring community, "Netflix service registration and discovery," 2018. [Online]. Available: https://spring.io/guides/gs/service-registration-and-discovery/

[68] Pivotal software, "Rabbitmq," 2018. [Online]. Available: https://www.rabbitmq.com/

[69] D. Woods, "Building Microservices with Spring Boot," 2016. [Online]. Available: http://www.infoq.com/articles/boot-microservices

[70] I. Pivotal Software, "Spring cloud," 2018. [Online]. Available: https://projects.spring.io/spring-cloud/

[71] Apache software foundation, "Apache zookeeper," 2018. [Online]. Available: https://zookeeper.apache.org/

[72] Gradle Inc., "Gradle," 2018. [Online]. Available: https://gradle.org/

[73] A. Gupta, "Wildfly swarm: Building microservices with java ee," 2018. [Online]. Available: http://blog.arungupta.me/wildfly-swarm-microservices-javaee/

[74] Red Hat, Inc., "Openshift.io," 2018. [Online]. Available: https://openshift.io/

A The CQRS pattern

The Command Query Responsibility Segregation pattern describes the separation of the application domain into two distinct parts – the command model which is responsible for the application processing and the query model that is managing the access to the persisted information.

This pattern extends a base given by the Command-query separation (CQS) which was introduced by Bertrand Meyer in his book *Object-Oriented Software Construction*. The main idea is to split the object's methods into two categories – queries which just return a value without changing the state, and commands that change the state of the object and do not return any result.

The commands are typically illustrated as simple objects identified by their respective names which are always expected to be in an imperative tense. They contain all necessary information that is needed to perform the request. Each command is delivered to a specified aggregate's command handler method that matches its identifier.

The query segment is responsible for the presentation of data to the end user. This typically represent methods that return data transfer objects (DTOs) or other data model entities. It can also provide several representations of the presented information or prevent users from multiple round trips by the data accumulation.

An aggregate is a main building block in the CQRS architecture. It represents a data entity that is always kept in a consistent state. The state can be changed by a reaction to the published event. Events are produced (applied) by the aggregate as a reaction to the received command.

To create a more resilient systems, most of the CQRS frameworks also employ an event sourcing mechanism. Every published event is being persisted to the durable storage which allows the aggregate to rebuild its state in the case of failure. This can be done just by replaying (reapplying) of the already published events. It also allows to persist and redeliver events that cannot be transmitted to aggregate during the failed state.

Another advantage of the domain separation is the performance increase. As both sides are allowed to scale up independently, the system can perform more operations concurrently.

The CQRS pattern may be particularly suitable for service (microservices) oriented systems. Because of their distributed nature, it is easy to separate concerns and allow customers to interact with different services depending on their performed activities.

To finalize, although the CQRS pattern has its benefits (ease of complexity and performance support), the practice showed that systems usually need to share the model between command and query sides. If the systems is not build with this patter in mind, the CQRS may place a very hard restrictions that may not be applicable in every application.

B JTA

javax.transaction.UserTransaction

```
void begin();

void commit();

void rollback();

void setRollbackOnly();

int getStatus();

void setTransactionTimeout(int seconds);
```

javax.transaction.Status

```
STATUS_ACTIVE = 0;

STATUS_MARKED_ROLLBACK = 1;

STATUS_PREPARED = 2;

STATUS_COMMITTED = 3;

STATUS_ROLLEDBACK = 4;

STATUS_UNKNOWN = 5;

STATUS_NO_TRANSACTION = 6;

STATUS_PREPARING = 7;

STATUS_COMMITTING = 8;

STATUS_ROLLING_BACK = 9;
```

javax.transaction.TransactionManager

```
void begin();

void commit();

void rollback();

void setRollbackOnly();

int getStatus();

Transaction getTransaction();

void setTransactionTimeout(int seconds);

Transaction suspend();

void resume(Transaction tobj);
```

C Reactive Streams v1.0.2 API

org.reactivestreams.Publisher

```
public interface Publisher<T> {
    public void subscribe(Subscriber<? super T> s);
}
```

org.reactivestreams.Subscriber

```
public interface Subscriber<T> {
    public void onSubscribe(Subscription s);
    public void onNext(T t);
    public void onError(Throwable t);
    public void onComplete();
}
```

org.reactivestreams.Subscription

```
public interface Subscription {
    public void request(long n);
    public void cancel();
}
```

org.reactivestreams.Processor

```
public interface Processor<T, R>
    extends Subscriber<T>, Publisher<R> {
}
```

D The saga scenarios

E The example applications public APIs

E.1 Axon service

Order service

```
POST /api/order
```

Query service

```
GET /api/orders
GET /api/order/{order_id}
GET /api/shipments
GET /api/shipment/{shipment_id}
GET /api/invoices
GET /api/invoice/{invoice_id}
```

E.2 Eventuate service

Order service

```
POST /api/order
POST /management/shipment
POST /management/shipment/fail
POST /management/shipment/compensation
POST /management/invoice
POST /management/invoice/fail
POST /management/invoice/compensation
```

Shipment service

```
POST /api/request
POST /api/compensate
```

Invoice service

```
POST /api/request
POST /api/compensate
```

Query service

```
GET /api/orders
GET /api/order/{order_id}
GET /api/shipments
GET /api/shipment/{shipment_id}
GET /api/invoices
GET /api/invoice/{invoice_id}
```

E.3 LRA service

Order service

```
POST /api/order
GET  /api/health
```

Shipment service

```
POST /api/request
PUT  /api/complete
PUT  /api/compensate
GET  /api/health
```

Invoice service

```
POST /api/request
PUT  /api/complete
PUT  /api/compensate
GET  /api/health
```

LRA coordinator

```
GET  /lra-coordinator
GET  /lra-coordinator/{LraId}
GET  /lra-coordinator/status/{LraId}
POST /lra-coordinator/start
PUT  /lra-coordinator/{LraId}/renew
GET  /lra-coordinator/{NestedLraId}/status
PUT  /lra-coordinator/{NestedLraId}/complete
```

```
PUT  /lra-coordinator/{NestedLraId}/compensate
PUT  /lra-coordinator/{NestedLraId}/forget
PUT  /lra-coordinator/{LraId}/close
PUT  /lra-coordinator/{LraId}/cancel
PUT  /lra-coordinator/{LraId}
PUT  /lra-coordinator/{LraId}/remove
GET  /api/health
GET  /lra-recovery-coordinator/{LRAId}/{RecCoordId}
PUT  /lra-recovery-coordinator/{LRAId}/{RecCoordId}
GET  /lra-recovery-coordinator/recovery
```

API gateway

```
PUT  /api/complete
PUT  /api/compensate
GET  /api/health
POST /api/lra
```

E.4 Eventuate Tram

Order service

```
POST /api/order
GET  /api/orders
GET  /api/order/{orderId}
```

Shipment service

```
GET /api/shipments
GET /api/shipment/{shipmentId}
```

Invoice service

```
GET /api/invoices
GET /api/invoice/{invoiceId}
```