Masaryk University
Faculty of Informatics

Use of Transactions within a Reactive Microservices Environment

Master's Thesis

Martin Štefanko

Brno, Fall 2017

Masaryk University
Faculty of Informatics

Use of Transactions within a Reactive Microservices Environment

Master's Thesis

Martin Štefanko

Brno, Fall 2017

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 of the management of transactions across multiple nodes in the 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 transaction models. 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 achieving 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 2PC protocol

2.3 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.

2.3.1 Atomicity

The transaction consists of a sequence of operations performed on different resources or by different participants. Atomicity means that all operation in the transaction are performed as if they were a single operation. When the transaction commits successfully all of its participants are also required to perform a valid commit. Conversely, if the transaction fails and is aborted all performed operations and effects are forced to be undone. This defines a possibility to abort at any point so that all changes done by the transaction will be reverted to the state before the transaction start.

The atomicity is generally achieved by the usage of the consensus multi-phase protocols. Standardized protocol is the two phase commit protocol which is used by the majority of modern transaction systems.

2.3.2 Consistency

The consistency describes that the transaction maintains the consistency of the system and resources that it is 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 semantic knowledge about the resources it manipulates [1]. Therefore achieving this property is the responsibility of the application code.

2.3.3 Isolation

The isolation property takes effect when multiple transactions can be executed concurrently on the same resources. That means 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 which is why the isolation is often also referenced as serializability.

From the perspective of an external user the isolation property means that the transaction appears as it was executed entirely by itself. 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 the inconsistent system.

2.3.3.1 Isolation levels

As transactions were first defined only in the database environment, in practice we distinguish several levels that describe to which extent the isolation succeeds.

2.3.4 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 the catastrophic failure.

2.4 Transaction models

2.5 Distributed transactions

A distributed transaction is the transaction performed in a distributed system. The distributed system consists of a number of independent devices connected through a communication network. Such systems are liable to the frequent failures of individual participants or communication channels between them.

The transaction manager can be implemented as a separate service or being placed with some participant or the client. **TODO**

2.6 Transaction manager

Every transaction is associated with a transaction coordinator or transaction manager which is responsible for the control and supervision of the participants performing individual operations. It is a component liable for coordinating transactions in the sequential or parallel execution across one or more resources. It provides proper and complete execution and it administers the comprehensive result of the transaction. Applications are commonly required only to contact the transaction manager about the start of the transaction.

The main responsibilities of transaction manager are starting and ending (commit or abort) of the transaction, management of the transaction context, supervision of transactions scoped across multiple resources and the recovery from failure.

2.6.1 Local transaction manager

A local transaction manager or a resource manager is responsible for the coordination of transactions concerning only a single resource. Because of its scope it is often build in directly to the resource. The span of the resource is defined by its managing platform.

The resource manager is required to provide a support for the participation in the global transactions that span over several resources. This means that it is effectively capable to handle complete transaction processing to the different transaction manager.

2.7 Failure handling

CA CP theorem - CAP

3 Microservices architecture pattern

This chapter introduces the basic concept of microservices and describes why modern scalable enterprise applications are to be developed implementing this pattern.

3.1 Architecture pattern

Microservices as a subset of a service oriented architecture (SOA) [4] is an architectural pattern which offers an intuitive approach to common problems following the software development. Instead of the SOA which builds the applications around the logical domain, microservices are built around the business model. Each microservice represents the separated part of the system.

3.1.1 Monolithic architecture

When describing microservices, the common way is to start by defining the opposite pattern, the monolithic architecture. When the application is developed in a 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" [5], is tightly coupled with the application, which is using it. Because of the easy development, scalability and deployment of monolithic software this approach is being preferred by the majority of modern enterprise applications. However, when the application needs to be extended or rebuild, it can become difficult to maintain. For instance, even because of the minor change or update in the single component, the scaling and the continuous deployment of the whole application can stagnate.

3.1.2 Microservice architecture

Microservices introduced the application separation into the self-maintained units – services [6]. The service is a single scalable and deployable unit, which is not dependent on any context. This means that the service can be maintained apart from applications which use

it. In addition, every microservice is being developed independently from other services. Each instance is managing its resources and is not able to directly access resources of any other service. This allows each request for data to be processed by the managing service. Service corresponds to component in monolithic architecture.

One of the biggest advantages of microservices is the ability to be deployed to the server, not affecting other applications or services. This allows separated teams to develop and maintain services independently. Applications based on this architecture utilize services by remote procedure calls instead of in-process calls used in monolithic architecture.

3.2 Principles of microservices

This section is inspired by the talk delivered by Sam Newman [7] in 2015 on the Devoxx conference in Belgium. In this presentation he described microservices from the business perspective. By his definition microservices are "Small autonomous services that work together" and they are based on these eight principles.

1. **Modeled around the business domain** – As was stated in the beginning of this chapter, microservices as well as the teams which are maintaining them correspond to the business model. This means that the requirements on their functionality do not change frequently. This architecture scales applications vertically – changes processed in one microservice do not affect other services or the system itself. They allow developers to focus on the particular part of the system rather than some specific technology.

2. **Automation** – The services are managed by teams. The team is responsible for development, administration, deployment and the life cycle of the service infrastructure. When the number of services is small it is possible to maintain them manually but when their number increases, this will became unacceptable. Automation processes like testing or continuous delivery then allow the enterprises to scale more

efficiently and speed up the mechanism of the service coordination.

3. **Hide the implementation details** – Microservices need to use external resources. Often, there is a requirement to share the same resource between two or more services. One possibility to do this is by providing the resource directly. The problem with this approach is that when one service changes the resource other services need to react to the change. The idea of this principle is that each microservice maintains its own resources. It provides an API[1] to access them. This allows the developers to decide what is hidden. Every request for the data must be processed through the public interface.

4. **Decentralization** – Microservice architecture is build around the idea of self-sustaining development. Services are maintained autonomously. When the teams are not dependent on each other, they can work more freely which allows faster improvement. This principle also corresponds to partitioning of responsibilities. It accentuates that relevant business logic should be kept in services themselves and the communication between them must be as simple as possible.

5. **Independent deployments** – This is the most important principle of this architecture. It expands the base provided by the option of independent development. 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 CI[2] build. Co-existing endpoints model describes the situation when customers need to upgrade to the new version of the service. As customers cannot be forced to upgrade at

1. Application Programming Interface
2. Continuous integration

the same time as the release happens, the idea is to make new endpoint which would process updated client requests. Customers use both endpoints depending on the version their applications require. This allows them to easily upgrade. When the 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. For the developers it can be useful to have any feedback from the clients that use their service. The understanding of the API can be supported by a good documentation provided by API frameworks like Swagger [8], or by the service discovery to propagate services and make the discovery of the service providers easier. To combine this information we can use the humane registries [9] which indicate the human interaction.

7. **Failure isolation** – Even if microservices force distributed development it is not a necessity that the failure of one service cannot influence another. This principle describes the distribution of resources to avoid the single point of failure. As there are many vulnerabilities in applications which can break, there is no precise manual on how to attain this principle.

8. **High observability** – Monitoring is an important part of development. As the microservice architecture is distributed it can be complicated to process this information. To make it more accessible aggregation is essential. Storing all logging entries and statistics in one place can highly impact the monitoring process. Another relevant point is to track the service calls as the services typically communicate with other services. By logging this information we can ensure traceability in case of failure.

3.3 Reactive microservices

4 Communication patterns

4.1 Consensus protocols

4.1.1 2PC

4.1.2 3PC raft paxos

4.2 Event based protocols

Eventual consistency

4.2.1 CQRS

5 Saga pattern

A saga, as described in the original publication [10], 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.

5.1 Participants

5.2 Compensations

5.3 Current development support

This section presents the current implementations of the saga pattern available for the enterprise use. The three explored frameworks are Axon [11], Eventuate [12] and Narayana LRA[1] [13].

5.3.1 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 [11]. It is composed on the top of the 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.

5.3.2 Saga development

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

1. Long Running Actions

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 incoming 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.

5.3.3 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.

5.3.4 Narayana LRA

6 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. This application is a backend processing for orders with a simple REST[1] interface.

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.

6.1 Invocation and interaction

An 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 6.1 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 B.

The setup of each example is described in detail in their respective repositories included in the Appendex B. Generally, each service is

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

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

Figure 6.1: Product Information example JSON

a standalone Java application that must run in a separated terminal instance by default located on the local computer address (localhost). Except for the LRA example, all examples are also able to run on the Docker[14] platform using the Docker compose project[15].

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 B. 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.

6.2 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). 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.

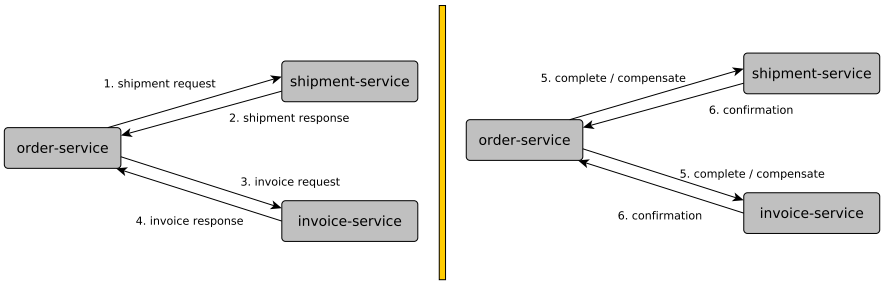

Figure 6.2: The saga model

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 C.

6.3 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.

6.3.1 Platform

Axon service is a Java Spring Boot [16] microservices application. Each service is fully separated and independent Maven project [17]. Every project is standalone runnable application (fat jar[5]) 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 JVM[6] thus one microservice. Except from that, 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[18]. Its main responsibility is the selection of the communication protocol and the choice of the target destination for each incoming command.

Axon provides two options for connecting different services through the distributed command bus – `JGroupsConnector` and `Spring Cloud Connector`. The one used in this quickstart is the Spring Cloud method. The underlying implementation is based on the Netflix Eureka Discovery and Eureka Server combination [19]. Each business service as part of its initialization registers itself with the service `registration-server`

4. for the definition of aggregate please refer to the Appendix A
5. Java Archive
6. Java Virtual Machine

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 AMQP[7] protocol called RabbitMQ message broker [20]. 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. Unfortunaly, by the time of this writing there is no way to distribute commands or events directly by the Axon platform.

6.3.2 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.

7. Advanced Message Queuing Protocol

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.

6.3.3 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.

6.4 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.

6.4.1 Platform

Eventuate service is a microservices application consisting of a set of Spring Boot [16] 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 [21]. 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 `eventuateio/eventuateio-local-console`.

6.4.2 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 [22] 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[8] 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 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.

8. Uniform Resource Locator

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.

6.4.3 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.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[19] "Netflix service registration and discovery."

[20] "Rabbitmq," 2018. [Online]. Available: https://www.rabbitmq.com/

[21] "Apache zookeeper," 2018. [Online]. Available: https://zookeeper.apache.org/

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

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

A The Command Query Responsibility Segregation pattern

CQRS

B The example applications public APIs

B.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}
```

B.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}
```

B.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
```

B.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}
```

C The saga scenarios