Masaryk University
Faculty of Informatics

Messaging providers in the SilverWare microservices platform

Bachelor's Thesis

Martin Štefanko

Brno, Spring 2016

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: Mgr. Martin Večeřa

Acknowledgement

I would like to express my sincere thanks to my supervisor Mgr. Martin Večeřa for his help, support and patience in my work on this thesis and also everyone who supported me during this time.

Abstract

This thesis concentrates on the integration of two messaging providers into SilverWare microservices – Vert.x and Apache ActiveMQ Artemis. It provides the explanation of the microservice architecture, comparison of three different microservice implementations and the description of integrated components. The practical part summarizes the design and the implementation of both messaging providers which is available as an open-source contribution to the SilverWare project.

Keywords

microservices, SilverWare, messaging, Vert.x, Artemis, ActiveMQ, JMS

Contents

1 Introduction

Microservice architecture pattern [1] is an important part of modern enterprise development. The main advantage of this pattern is the separation of the application into standalone services which represent different business components. For enterprises developed by this model, it is important to provide a form of communication between these services. This thesis aims to implement two components which would provide this functionality in SilverWare microservices.

The SilverWare project [2] is an open-source implementation of the microservice design pattern developed under the Red Hat, Inc. [3]. Even if it is a relatively young project, it is easy to employ because it is built on top of the commonly known technologies like Weld (CDI[1]) or Apache Camel. The more detailed information about the project is provided in the third chapter.

When the application's components are separated from each other, there is a need to support a way of communication between them. This is the main responsibility of messaging providers. The messaging provider implements the functionality that allows an exchange of independent messages between the loosely-coupled heterogeneous software aspects. As these messages are typically asynchronous, they allow the isolation of sender of the message from its receivers providing the mean of indirect communication.

The purpose of this thesis is to integrate two messaging providers into the SilverWare microservices platform.

The first integrated component is the Vert.x platform toolkit for the asynchronous I/O[2] and the event driven programming [4]. Vert.x is lightweight and container-free which is making it a suitable choice for the microservice architecture.

The second component is Apache ActiveMQ Artemis [5] for Java messaging. This non-blocking asynchronous messaging system provides the support for many messaging protocols including the Java Message Service (JMS) as a part of JavaEE[3] specification.

1. Context and Dependency Injection
2. Input / Output
3. Java Enterprise Edition

Artemis, as well as Vert.x, is being developed as an open-source implementation. In SilverWare there is a possibility to inject both of them using CDI microservices. The integration of these providers is published as an open-source contribution to the SilverWare project and it is accompanied with the quickstart examples to present its usage.

This thesis is divided into three logical parts. The first one consists of the introduction and motivation for this thesis included in the first chapter. It also contains a more detailed information about the microservice architecture and its business principles in the second chapter. The third chapter compares the generally known microservice implementations with the SilverWare platform. The more detailed information about the Vert.x and ActiveMQ platforms with the definition of the JMS specification is provided in the fourth chapter.

The second part covers the analysis and the implementation of the messaging providers. The fifth chapter summarizes the design of the implementation promoted by UML class diagrams. In the sixth chapter, I describe the more specific details of the implementation and integration of each component.

The last part concludes the work done in this thesis. The seventh chapter summarizes the unit and integration testing and the results of the performance tests. It also describes the basic information about the quickstart examples. In the final chapter I sum up all of the work results and I suggest the ideas about the possible enhancements in the future.

2 Microservice architectural style

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

2.1 Architecture pattern

Microservices as a subset of service oriented architecture (SOA) [6] is an architectural pattern which offers an intuitive approach to common problems following the software development. Instead of SOA which builds the applications around the logical domain, microservices are built around the business model. Each microservice represents the part of the system.

2.1.1 Monolithic architecture

When describing microservices, the best way is to start by defining its opposite, 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" [1], 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 expanded, it can be difficult to maintain. For instance, even because of the minor change or update in the single component, the scaling and continuous deployment of the whole application can stagnate.

2.1.2 Microservice architecture

Microservices introduced the application separation into the self-maintained units – services [7]. 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, microservices are 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 to 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.

2.2 Principles of microservices

This section is inspired by the talk delivered by Sam Newman [8] 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 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 and 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 service coordination.

3. **Hide the implementation details** – Microservices 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 the 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 the 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 the same time as the release happens, the idea is to make new endpoint which would process the client requests. Customers use both endpoints depending on the version their

1. Application Programming Interface
2. Continuous integration

applications require. This allows them to 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 [9], or by the service discovery to propagate services and make the discovery of service providers easier. To combine this information we can use the humane registries [10] 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 Microservice platforms

The microservice architecture influenced the view of enterprise domains. For that reason, many companies integrated their own implementations to support their business systems. This chapter describes three of them: Spring microservices, WildFly-Swarm and SilverWare itself.

3.1 Spring

Microservices in Spring are utilized out of the Spring Boot. It provides an RMI[1] subsystem for locating services and the common Spring APIs for the client use.

3.1.1 Spring Boot

Spring Boot is a framework built on top of the Spring Framework [11]. It is a typical way of building Spring production ready applications. It favors convention over configuration to allow the usage of Spring features with a little of the Spring configuration [12].

Spring Boot is built into aggregate modules known as starters. The starter contains a well-known stable versions of libraries to provide a required functionality to an application. In order to use Spring Boot, the `spring-boot-starter` module is required. Other useful modules like the `spring-boot-starter-web` to setup Spring MVC[2], REST[3] or the embedded Tomcat server, or the `spring-boot-starter-data-jpa` for the data persistence can be provided to support the microservice adoption. Starters can be packaged with the application using Maven or Gradle dependencies.

Boot is also capable of changing the behavior of the microservice at runtime. This can be achieved through the Spring configuration files or the JVM[4] properties. By default it looks for the files in the directory

1. Remote Method Invocation
2. Model View Controller architecture
3. Representational State Transfer
4. Java Virtual Machine

named `config` relative to the startup path. The path can be changed by setting the `spring.config.location` Java property.

Another responsibility of this framework is the packaging of the application. The preferable way for the microservice is to create an executable fat jar[5] archive which contains all of the application dependencies.

3.1.2 Creating microservice

In order to make a microservice the project must fulfill two requirements – it must follow the Maven style convention for project layout and it must provide an entry point. The entry point can be any class annotated with the `@EnableAutoConfiguration` annotation that starts the Spring Boot application. Additional functionality can be added to the microservice by the use of standard Spring features.

3.1.3 Exposing microservices

As the microservices need to be able to communicate throughout the network, Spring provides various ways for exposing the available services. For this purpose, the registration servers like Eureka or Consul which are integrated into the Spring cloud can be used. For a more direct approach, the microservice functionality can be provided through the RESTful API.

3.2 WildFly-Swarm

This project aims for the decomposition of the WildFly application server. The applications are packaged with the just enough server as they require for their services to run [13]. It also provides Maven and Gradle plugins to ease the production of the WildFly-Swarm projects.

3.2.1 Uberjar

The uberjar is an executable fat jar containing the client application. It accommodates only the needed parts of the application server. Any

5. Java archive

undesirable elements are excluded. It also contains an internal Maven repository of dependencies. The jar can be executed with the `java -jar` command. With this approach, the client can optionally provide a traditional `main(...)` method with the container configuration.

The size of the created uberjar depends on what parts of the server users include. As it can contain multiple WildFly subsystems, it tends to grow on size. However, the size of the uberjar is not typically larger than 50 MB.

Except for creating an executable uberjar, WildFly-Swarm supports the packaging of the application into regular war[6] files. In this case, the war file is expected to be deployed to the server, so the dependencies are not packaged into the archive itself.

3.2.2 Fractions

A fraction is a precise selection of features which should be added to the project. It can be a direct subsystem of WildFly or a more sophisticated set to provide additional functionality. The clients choose which fractions to include in their projects by selecting the dependencies from the `org.wildfly.swarm` package.

3.3 SilverWare

SilverWare project is a minimalistic, easy to use, open-source microservice implementation. It is modularized, highly configurable and container-free [2], so that developers can select components they need for their microservices, possibly reusing existing services. It does not restrict service implementation to any particular language and it is built atop of the generally known frameworks like REST or Apache Camel.

3.3.1 Packaging

Applications in SilverWare are packaged into executable jar archives. They can be started directly form the command line with the `java -jar` command. Application dependencies are not packaged into the

6. Web archive

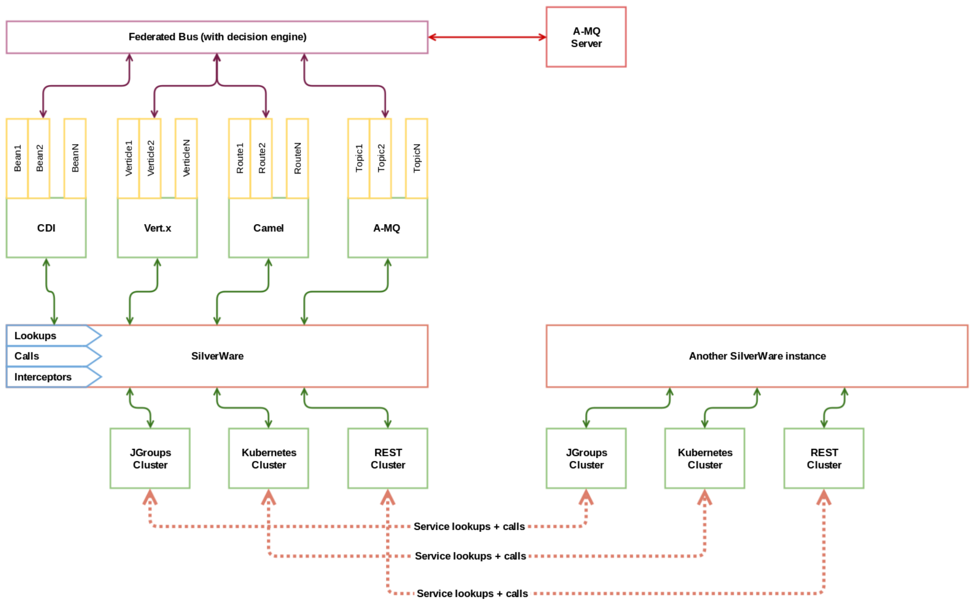


Figure 3.1: SilverWare microservices

jar directly. The jar only contains links to the `{jar.home}/lib` folder. This folder must be shipped with the jar in order to execute it.

3.3.2 Microservice providers

A provider in SilverWare represents a toolkit to provide a desired functionality. This can be the JavaEE specification implementation, the usage of the well-known framework or the management of the full platform. The clients use only the required providers. To add the provider, the dependencies from the package `io.silverware` are being used. The providers are automatically initialized after the platform starts. For this purpose, the `microservices` provider which contains the microservices Boot configuration must be present. Additionally, the `cdi-microservice-provider` is typically included to provide the CDI[7] capabilities.

7. Context and Dependency Injection

3.3.3 Developing microservices

To create the microservice in SilverWare, clients simply annotate the class with the `@Microservice` annotation. This annotation can optionally contain the name of the microservice.

The biggest advantage of SilverWare is the possibility of injecting the desired functionality through the CDI. After adding the required provider dependency, the clients can use the `@Inject` and the `@MicroserviceReference` annotations for injecting the common implementations from the framework APIs depending on the provider configuration.

The clients can also inject other microservices directly into their service. This allows an easy reuse of the already developed microservices.

The providers may take some time to initialize the frameworks beneath them. To allow the microservice to start its execution after the platform is started, SilverWare uses the methods with `@Obeserves MicroservicesStartedEvent` annotated paramater. This event represents that the client can be assured that the required services are present. It can be used for the initial configuration of the microservice.

4 Used technologies

This chapter presents both of the integrated platforms. Additionally, because of the ActiveMQ provider it describes the JMS[1] specification.

4.1 Vert.x

Vert.x is a general purpose JVM[2] application framework. By the definition, "Vert.x is a server framework providing an event-based programming model" [14]. This means that programming is based on asynchronous events. Different components communicate through an *event bus* [15] messaging system by registering event handlers for various actions.

Another important feature of Vert.x is the polyglotism. It enables users to develop applications in many languages not forcing any communication overhead. It is inspired by popular concurrency frameworks like Node.js or Erlang/OTP[3]. Vert.x aims to be non-blocking, asynchronous and high performing.

In Vert.X, programmers write code as a single-threaded application without the need for synchronization. Internally the framework manipulates a thread pool with usually as many threads as is the number of CPUs. Each thread is periodically executing an *event loop* – a check, if any event is executing and it is calling the registered event handlers.

When referring to Vert.x, I mean the Vert.x in version 3.0. Version 3.0 introduced many new features to the platform and it made Vert.x more embeddable.

4.1.1 Verticles

Verticle is the base component of the Vert.x platform. It is the smallest execution unit of code that can be independently deployed. It can be written in any Vert.x supported language. An application can run verticles written in different languages. Verticle is always single-threaded

1. Java Message Service
2. Java Virtual Machine
3. Open Telecom Platform

so it is not vulnerable to synchronization errors and race conditions. Users can scale the applications by deploying more instances of the same verticle but the particular instance is always single-threaded. Different instances of verticles communicate by sending messages through the *event bus*.

Verticles can be run directly from the command line. For this purpose, users need to download and install the Vert.x distribution which is available for download on the Vert.x website [4]. Another option to use Vert.x is as the embedded platform in your own application. For Maven or Gradle projects it can be added as the single dependency. It can be also packaged directly into the jar archives.

4.1.2 Event bus

Event bus is the main communication bridge in Vert.x. It is a transient publish-subscribe messaging system built into the platform. There is always only one instance of the *event bus* for each Vert.x instance. It is responsible for delivering messages between verticles regardless the language they are written in. Although it corresponds to the single instance of Vert.x, it is also capable to provide transmission of messages between verticles deployed in several distinct platforms.

Messages that are being sent through the *event bus* can be of primitive types, Strings, buffers or more complex Objects. Even though any object can be sent, the recommended format of messages is JSON[4] because it is easily understandable by all languages that Vert.x supports. For languages that do not have a first-class support for JSON, Vert.x provides the internal JSON handlers.

4.1.3 Clustered Event Bus

In production it is expected that more than one instance of Vert.x will be running in the JVM at the same time. These distributed instances are able to detect each other and arrange their event buses to form a cluster. They form a peer-to-peer messaging system. There is no main server which would direct the network. This moves Vert.x to the concept of microservices. The platform is built out of loosely coupled applications

4. JavaScript Open Notation

of which each is running different Vert.x instance distributed across the network.

4.1.4 Polyglot platform

As it was mentioned previously, Vert.x supports many programming languages. This includes Java, JavaScript, Groovy, Scala, Ruby, Clojure or Ceylon. As each verticle can be written in different language Vert.x does not force the developer to use any particular language or to use only one of them. This combining allows the selection of the most suitable language for the given task.

It provides an "idiomatic APIs for every language that Vert.x supports" [4]. The main API is always provided in Java. APIs for individual languages are then generated from the Java core API. The core API is fully asynchronous to provide a stable concurrency model.

4.2 ActiveMQ

ActiveMQ is an asynchronous messaging system written in Java. It is composed as a Message-Oriented middleware (MoM) [16]. It is entirely compliant with the JMS specification 1.1. ActiveMQ is an open-source project licensed under the Apache Software License in version 2.0. It aims to provide messaging-oriented integration of applications based on standardized protocols throughout as many languages and platforms as possible [17].

4.2.1 Message-Oriented middleware

The middleware is a layer of software that decouples various components and services that run on different platforms in the network environment. It defines three categories in which it can be implemented:

- **RPC-based** – Remote Procedure Calls allow the applications to connect directly as in the local context.
- **ORB-based** – The Object Request Broker section distributes the application's objects through the network.

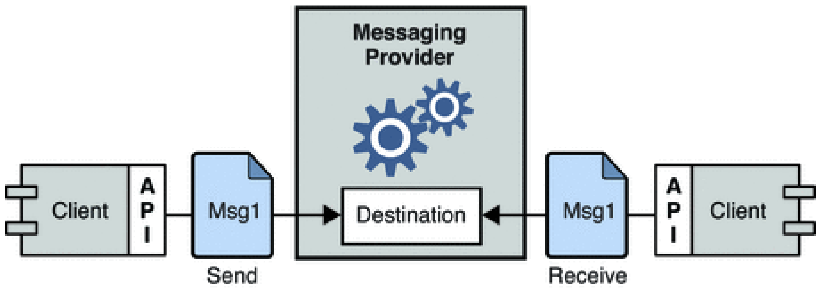


Figure 4.1: Message-oriented middleware architecture [16]

- **MoM-based** – The Message-Oriented middleware concentrates on the applications' communication by sending and receiving messages through the MoM provider.

RPC- and ORB-based middleware is synchronous. That means the service call is blocked until the called component responds. The systems are tightly-coupled, which means that the upgrade of one service can result into change in the another one.

MoM provides benefits of loosely-coupled components by the decoupling of the communication through the messaging provider. The main components of this category are clients, messages and the messaging platform. Clients send and receive messages to and from the destinations. The destination is a mediator used in order to contact other clients.

Messages are being sent in asynchronous manner, i. e. the caller is able to continue processing even after the call invocation. The messages can continue to be sent until the consumer has the available capability to process them. This can be a disadvantage as the component is not in the full control of its resources which can lead to failure. It can be partially prevented by monitoring performance.

Another advantage of MOM is administration and monitoring. As the messages are mediated through the messaging provider, the clients need to interest only in the processing of messages. This allows tasks like reliability, security or performance tuning to be maintained by the platform.

4.2.2 Features

As the ActiveMQ is designed with reliability, high availability and performance in mind, it provides a range of aspects to ease its use [17]:

- **JMS implementation** – It allows a simple change of the JMS provider without the need of rewriting the application.
- **Multiple protocols support** – ActiveMQ provides a wide range of connectivity options. The usage of many different protocols allows more straightforward adaptation of the platform.
- **Integration with application servers** – Except for developing standalone applications it provides the support for server embedding.
- **Client applications** – Even if the ActiveMQ itself always runs in the JVM, the client applications are not restricted to JVM compliant language.
- **Clustering** – It allows many different ActiveMQ brokers[5] to work together. This concept is known as the *network of brokers*.
- **Simple administration** – It is designed with an easy-to-use set of features for broker and client administrating, monitoring and logging purposes.

4.2.3 Apache ActiveMQ Artemis

Artemis is a codename used for the HornetQ code base [18]. This code was donated to the Apache Software Foundation in late 2014. It is possible that one day Artemis will become a successor of ActiveMQ.

ActiveMQ Artemis supports two client messaging APIs, Core client API and Java Message Service (JMS). Because it is implemented as a set of Plain Old Java Objects (POJOs), it can be embedded directly in enterprise applications, deployed as a standalone server or integrated into Java EE application server.

Unlike ActiveMQ, Artemis supports JMS specification in both version 1.1 and 2.0.

5. ActiveMQ instance

4.3 JMS

ActiveMQ, Artemis and many other different MoM providers are providing their own proprietary APIs for enterprise messaging. This led to the requirement of producing one common API to support easier evolution and management for users.

The Java Message Service raised as the standardized way of developing messaging applications. It aims to maintain the portability between different systems and languages. It is written in Java and it serves as an abstraction of individual providers. It provides the common API for sending and receiving messages.

Current version of the JMS is 2.0. Even if it brought the simplified API, the previous version 1.1 is still in frequent use.

4.3.1 Interaction

JMS communication consists of four main elements: JMS clients, messages, destinations and the JMS provider.

JMS clients use the JMS standardized API to access the JMS provider. The two types of clients are producers and consumers. Producers are responsible for creating and sending messages with desired properties such as the message durability. Consumers utilize the messages in the synchronous manner by receiving messages directly, or in the asynchronous manner by providing the message listeners.

The message is the main concept of the JMS. It is a unit of information passed between clients through the provider. It consists of the headers which contain the metadata and the payload – the actual message in the text or binary form.

JMS destination is the place where messages are being sent and from where they can be received. JMS supports two types of destinations: a *queue* and a *topic*. These destinations will be described in the following section.

JMS provider mediates the above mentioned interaction through the specific MoM implementation.

4.3.2 JMS domains

JMS supports two types of messaging domains:

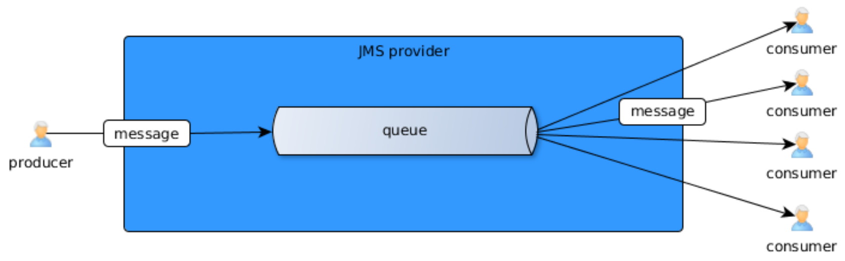


Figure 4.2: Point-to-point domain

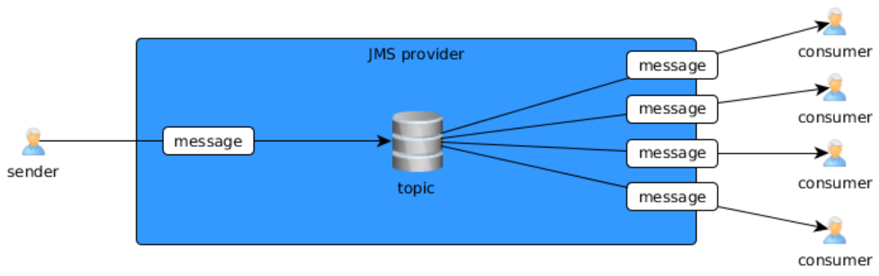


Figure 4.3: Publish-subscribe domain

- **point-to-point** – This domain uses the destination called the *queue*. The *queue* is a FIFO[6] pipe. It delivers messages in the same order as they were sent. In PTP each message is delivered once and only once to a single consumer. When more consumers receive messages from the same *queue*, they are distributed in a round-robin fashion. This destination is represented by the `javax.jms.Queue` class.
- **publish-subscribe** – The destination used in this style is called the *topic*. Producers send messages to the *topic* and consumers register to it in order to receive messages. When the message is sent, it is delivered to all of the subscribers. For the use of the *topic*, the `javax.jms.Topic` class is used.

6. First In First Out

Both domains utilize synchronous and asynchronous processing of messages. In the first case a consumer thread is blocked until a message arrives or the specified timeout passes. In the asynchronous style a consumer provides a message listener. The listener gets notified each time a new message arrives to the destination.

5 Design

The previous chapter introduced the basic concepts of both integrated components. The following chapter explains the design decisions used in this work and it provides a design model of their implementation.

5.1 Providers in SilverWare

The SilverWare consists of several components which represent the microservice architecture. Each component represents a single technology. It can be also referred to as a plugin or a provider because it provides the interface which enables the technology to be used as the microservice.

5.1.1 MicroserviceProvider

Each provider must implement this main interface. It consists of these two methods.

- **initialize(Context)** – This method is called when the platform starts. It serves to perform operations needed for the configuration. The provided `Context` will be described in more detail in the following section. If the initialization was successful, the service is started in a separated thread dedicated to this provider only.
- **run()** – This is the main method containing most of the provider logic. It uses active checking to wait for the thread to be terminated. It implements the provider functionality and it allows the start of the mechanism which the provider represents. Then the requests for the provider are automatically processed by the platform running in the background.

5.1.2 SilverService

This is a generic interface for the `MicroserviceProvider` interface. It provides the possibility of retrieving the current `Context` of the provider. `Context` is carrying all the necessary information needed for the execution. As the information is linked to the specific provider there is an option to originate multiple instances of the same platform in the same JVM.

The `ProvidingSilverService` is one of the interfaces that is extending `SilverService`. This interface is being implemented by the providers which need to offer the ability to provide microservices. It contains two methods:

- **lookupMicroservice(MicroserviceMetaData)** – This method is used to find all available microservices provided by the particular provider. It takes the `MicroserviceMetaData` as an argument which represents the query for microservices. Provider returns the collection of microservices that meet the requirements.
- **lookupLocalMicroservice(MicroserviceMetaData)** – By the default, this method uses the `lookupMicroservice` to locate the local microservices. It can be overridden if the local lookup is different.

The `SilverService` and the `ProvidingSilverService` are the most abstract interfaces in the hierarchy. They are used for the extension by more concrete interfaces of individual providers. These interfaces store the constant context dependent information.

5.2 Vert.x Microservices provider

Vert.x is being integrated into SilverWare as a standalone module which is built on top of the Vert.x platform. This project provides all of the features of Vert.x in version 3.0.

5.2.1 Class verticles

As it was mentioned previously, the platform is assembled as a set of interacting verticles. On the initialization, the provider locates the set of classes which implements the `Verticle` interface.

Apart from Java verticles the provider is able to load groovy verticles as groovy scripts can be compiled into class objects. Groovy verticles are recognized as implementations of the `Verticle` interface or extensions of the `GroovyVerticle` class. As the `GroovyVerticle` cannot be cast to the `Verticle` interface the provider is able to work with both of them respectfully.

Another Vert.x supported language that is capable of compiling into class files is Scala. Unfortunately, the current support of Vert.x platform for Scala is stagnating. It will be no problem to add the support for it to the provider once the platform will function with Scala verticles again.

Each found verticle class is automatically deployed into the platform. The further restrictions of the deployment can be specified by the class annotation which is described in the following chapter. Every deployment is monitored by the logging subsystem.

5.2.2 XML configuration

As Vert.x does not restrict users to only one language, SilverWare provides a way to deploy verticles from languages other than Java through an XML[1] file. It contains the names of the verticles to deploy. This name can optionally consist of the prefix defining the verticle factory to be used to load the verticle.

If no prefix is provided the name is considered as an absolute or a relative path to a verticle. This allows the users even to deploy verticles that are not located on a classpath. Verticle factory is derived from the suffix of the file.

If neither prefix nor suffix succeeded, the name is considered a fully qualified name of the Java class. This is an optional way to deploy more instances of the same verticle.

5.2.3 Verticle deployment

The verticle deployment process is configurable. For this purpose, Vert.x provides the `DeploymentOptions` class. These options can be optionally specified with the verticle deployment.

To ease the deployment of verticles the provider administers an annotation. This annotation can be used with Java and Groovy verticles that are compiled to the class files. It is placed directly on the verticle class definition and it allows to configure all possible options. Individ-

1. Extensible Markup Language

ual properties that can be set will be discussed in the Implementation chapter.

The verticles specified in the XML configuration file can provide the deployment information as the attributes on specific elements. An example of this configuration is provided in the appendix B.

As the deployment of the verticle can fail the provider contains an asynchronous handler which monitors the process.

5.3 ActiveMQ Microservices provider

ActiveMQ module in SilverWare is providing the messaging services maintaining the JMS specification in versions 1.1 and 2.0. It is based on the Apache ActiveMQ Artemis project [5].

5.3.1 Connecting to the server

The provider administers the client side of the messaging platform. This means it does not provide the ability to start and manage the A-MQ[2] server. The client connects to the remote server through the connector URI[3].

Although it is established on the Artemis project, it also supports other A-MQ servers from various providers. It also presents the option to connect to different URIs and different servers from the same microservice.

5.3.2 Configuration

The client can connect to the already started A-MQ server through the JNDI configuration file or directly through the provider API.

The JNDI is automatically used when the `jndi.properties` file is found on a classpath. It must contain one required property named `java.naming.factory.initial`. It defines the A-MQ initial factory used to create connections to the server. The factory settings must be compatible with the used A-MQ server.

The other option is to specify connection properties directly in the microservice implementation. This functionality is provided through

2. ActiveMQ
3. Uniform Resource Identifier

the annotation that can be set on the injection points. This annotation will be described in the Implementation chapter.

5.3.3 JMS support

As this design aims to be implementation independent it provides the JMS specification APIs. The client can inject the main connection classes: the `Connection` for the JMS 1.1 and the `JMSContext` for the JMS in version 2.0.

The `ConnectionFactory` used for creating the above mentioned classes is created by the provider. The client has no direct access to this instance.

Each injection of connection classes as well as the connection factory are maintained by the provider. This means the client do not need to close any of these resources in order for the application to work properly. Every other resource created by the user from these injections, for example the `Session` or the `JMSProducer`, needs to be closed by the user accurately.

The CDI injections can be adjusted for different use cases by the annotation attributes. The annotation allows users to specify the different URI, the initial context factory used to create connections to the server or the security identity for the user of this connection point. In case of JMS 2.0, users are also able to specify the session mode as the `Session` and the `Connection` classes were merged into the `JMSContext`.

Except for the above mentioned settings, it also allows users to choose the connection point to be shared or to create a new instance of it. This reuse of resources can ease the work with them across multiple microservices. It does not have a major impact on the performance as the sophisticated operations are performed by the provider.

6 Implementation

The previous chapter defined the integration design. In this chapter I describe implementation details of each provider respectfully.

6.1 Vert.x

The `VertxMicroserviceProvider` is the main class of the vertx module. It is built on top of the Vert.x platform and is responsible for its management. It consists of two primary aspects to initialize and run Vert.x.

In the initialization phrase it processes the context and looks up the available verticles. In the second stage it retrieves the `Vertx` instance and deploys the loaded verticles. Each verticle is deployed automatically if the user does not specify it differently. After that it actively waits for the interrupt, after which it closes the Vert.x platform and exits the provider.

6.1.1 Locating verticles

The provider searches for verticle instances to be deployed in two ways. As Silverware is a Java project the provider is able to detect the Java verticle instances directly. It looks up all classes on classpath that implements the `io.vertx.core.Verticle` interface as each Java verticle must implement it.

Except for this interface, Groovy verticles are also recognized as subclasses of the `io.vertx.lang.groovy.GroovyVerticle` class. As the `GroovyVerticle` cannot be casted to the `Verticle` interface, I provided a workaround. These classes are being prefixed by the provider with the prefix extension `groovy:`, which forces the platform to load them with the Groovy verticle factory. The example of configuring Maven to compile Groovy classes will be discussed in testing chapter.

Second way to specify verticles' names is an XML configuration file. In this file names are passed as strings so it is possible to deploy verticles from every language that Vert.x supports.

As the SilverWare project consists of several modules and by default each verticle is automatically deployed, it is necessary to provide a way to exclude some of the verticles from the deployment. For this purpose, the list of forbidden verticles is provided in the `VertxUtils` class.

6.1.2 Deployment annotation

The purpose of this annotation is to allow the user to define deployment properties of each verticle. It is used for the verticle deployments which are managed by the provider. This annotation allows users to specify options that can be set on the `DeploymentOptions` class. These options are:

- **type** – This determines which verticle type this instance is. For this option I have created a simple enumeration `VerticleType` which contains three values:
  - **STANDARD** – The default verticle type. The code of standard verticles is executed in the same event-loop thread. This allows users to write the application as single-threaded as long as no new threads are created inside the verticle. This is the most common verticle type.
  - **WORKER** – The worker verticles are similar to the standard verticles but they are executed in the thread from the Vert.x worker thread pool. They are used to run blocking code to avoid suppressing any event loop threads from execution.
  - **MULTI_THREADED_WORKER** – The worker thread that can be executed concurrently by different threads.
- **instances** – This number specifies the number of instances of this verticle that should be deployed to the platform. Users are also allowed to deploy more instances from another verticle or microservice.
- **isolationGroup** – This option defines the isolation group name used to separate classes specified by the `isolatedClasses` property. By default, Vert.x use the same classloader to load all verticles on the classpath. A new classloader is created by Vert.x to load the isolated classes. For instance, this can be used to deploy two verticles with the same name to the Vert.x platform.

- **isolatedClasses** – A comma separated array of class names or package wildcards to be loaded by the different classloader.

- **extraClasspath** – This property allows users to define classpath entries to be added to the default classloader. As for the isolated classes this can be a fully specified class names or wildcards.

- **ha** – The high availability. When this option is enabled it sets the Vert.x platform to redeploy any verticle deployed to the instance of Vert.x when this instance is destroyed unexpectedly.

- **config** – This string specifies the path to the JSON file containing configuration information of individual verticles. This configuration is accessible in verticle through the `Context` object or directly through the `config()` method.

6.1.3 Vert.x configuration XML

This file is always unique to simplify the configuration. The provider looks up for the file named `vertx-config.xml` on the classpath. This configuration is optional. The structure of XML with examples of verticle specifications is described in appendix B.

To validate the structure of the XML configuration file, I have created an `vertx-config-schema.xsd`. The validation against this schema is run as a part of the loading process. Users can also use it to check the configuration file before the platform starts.

6.1.4 Vert.x utilities

To keep the provider as simple as possible I created a utility class `VertxUtils`. This class main responsibility is to store the list of forbidden verticles. This list is loaded as a part of provider initialization from the XML file `forbidden-verticles.xml` to allow simple configuration.

Another function of this class is to load the user's XML configuration. It validates it against the schema and provides the set of verticle names with options to the provider.

It is also responsible for extracting the deployment options. It supplies two static methods to get the information from the `@Deployment` annotation or from the `<verticle>` tag.

`VertxConstants` is another utility class used for storing constants the Vertx provider can use.

6.2 Apache ActimeMQ

The module `activemq-microservice-provider` is established on the `ActiveMQMicroserviceProvider` class. This class maintains the creation and the initialization of connections to the server and it provides the CDI injection capabilities for the client-to-server communication.

6.2.1 JNDI initialization

The provider initializes by default from the JNDI configuration file `jndi.properties` if it is present on the classpath. The file must contain a required property `java.naming.factory.initial` which defines the initial context factory used to connect to A-MQ server. This factory is specific for different servers. When no factory property or no JNDI file is provided, the default `ActiveMQInitialContextFactory` class from the Artemis project is used for this purpose. The initial context factory can be later changed by the `@JMS` annotation.

JNDI configuration can contain other additional properties that can be set on the A-MQ server. These options differ depending on the used server specified with the initial context factory.

6.2.2 JMS annotation

This annotation is optionally placed on any of the connection injection point, i. e. the `Connection` or `JMSContext` field injection. It contains the following configurable attributes:

- **serverUri**: This option specifies a different URI to be used for this connector. This URI must be configured on the desired server or no connection can be created. If it is not provided by the user, the provider will use the default URI obtained from the JNDI configuration.

- **userName**: The user name which specifies the user identity of this connection. This is used along with security enabled settings on the A-MQ server.
- **password**: The user password used in conjunction with user name in the security identification.
- **connectionType**: The value defines what king of injection should be used for this connection. The ActiveMQ provider supports two kinds of injection defined by the `ConnectionType` enum:
  - **SHARED**: It represents the connection shared between all microservices. The connection use lazy initialization so it is not created unless it is required. The shared connections with provided security credentials differ from the non-secured ones.
  - **INJECTION**: This option creates a new connection for this individual CDI injection.
- **sessionMode**: This option is valid only for the JMS 2.0 and it indicates which session mode should be used on the `JMSContext` class. For the traditional `Connection` class this is not required as it is specified on the `Session` creation.
- **initialContextFactory**: Used in collaboration with the serverUri attribute to specify the initial context factory for the connection to the different server than was specified in the JNDI configuration. As in the JNDI file this must correspond to the used server on which the URI is defined.

6.2.3 ConnectionProvider

The `ConnectionProvider` class encapsulates the connection providing for the selected URI. Internally, it maintains the connection factory used to create the `Connection` and `JMSContext` instances for individual versions of the JMS respectfully. This includes the creation of secured and non-secured connections and the sharing of shared instances.

JMS 1.1 contains only one non-authenticated shared instance for one URI. Apart from that, it contains one shared connection for each of the security credentials requested with the shared connection type. For this purpose an internal private class `JMSCredentials` is provided.

The JMS 2.0 is more complicated because the provider needs to track the session mode of the `JMSContext`. It creates one shared context for every session mode and a default one which may vary depending on the execution environment. Even if the context with the default mode is already present in shared instances, if no mode is specified the new context will be created as there is no possibility to estimate its mode.

The security in JMS 2.0 is also maintained by the `JMSCredentials` class. This class additionally contains a session mode attribute which is used to distinguish authenticated instances. This means that shared instances are created for the security credentials and the session mode.

7 Test framework

Each provider is supplied by the basic test suite. These tests ensure the promised functionality requirements are met.

7.1 Vertx test suite

Vertx tests are based on the platform usage to display the features of the provider. It tests the automatic injections of Java and Groovy verticles as well as the loading of verticles through the XML configuration file. It presents the way of implementing verticles in different programmatic languages like JavaScript to establish a HTTP[1] server with which microservices from the test suite communicate. It also tests the utilities provided by the `@Deployment` annotation in the code and the attributes in the XML.

7.1.1 Groovy deployment test

As a part of `VerticleDeploymentTest` in the Vert.x test suite the groovy verticle must be compiled to a class object in order to be deployed. For this purpose, the `gmaven-plugin` is used. This plugin serves just for the testing function. If the user wants to take advantage of the automatic deployment of groovy verticles, it is his responsibility to compile them. The example of plugin configuration can be seen in the provider `pom.xml` file or in the quickstart discussed in the following section.

7.2 ActiveMQ test suite

To demonstrate the provider functionality, the tests start an embedded instances of Artemis and ActiveMQ servers. They display the CDI injection capabilities, the usage of the automatic JNDI configuration and the `@JMS` annotation in both supported versions of the JMS.

The Artemis server setup is obtained from the broker configuration XML files located in the `brokers` folder. To test the security settings

1. Hypertext Transfer Protocol

the `ActiveMQSecurityTest` uses security enabled configuration which contains an example of security restrictions.

7.3 Quickstart examples

The quickstart examples for each provider can be found in a Silverware-Demos repository on GitHub [19].

7.3.1 Vertx helloworld

To present the Vert.x microservices provider functionality four examples are provided. Each of these modules is prefixed with the `vertx` prefix.

The most basic quickstart is `vertx-helloworld`. It shows the automatic deployment of the simple Java Verticle. The microservice shows how to embed the Vert.x provider through SilverWare and how to access the injected instance of a `Vertx` class to set up the consumer on it. The verticle then in periodic intervals sends messages to the *event bus* which are consequently processed in the microservice.

The second example, `vertx-xml-helloworld`, presents the XML configuration file deployment of the JavaScript verticle. The configuration file `vertx-config.xml` is located in the `resources` folder. The verticle sends messages to the event bus as in the previous example.

The `vertx-groovy-quickstart` shows the deployment of Groovy verticles. It presents the way of configuring the `gmaven-plugin` for the compilation of Groovy classes in the Java project. The compiled verticle is automatically deployed to the platform and communicates with the microservice through the *event bus*.

The last quickstart `vertx-deployment-options` shows the way of specifying deployment options. It contains two verticles: Java verticle containing the `@Deployment` annotation and the JavaScript verticle with attributes set on the `verticle` element. Both verticles use the JSON configuration file to display their functionality.

7.3.2 ActiveMQ quickstarts

For the ActiveMQ provider there are three quickstart projects implemented. They are prefixed with the `activemq` prefix. Each example presents the configuration and the way of sending and receiving messages in both supported versions of the JMS specification.

The `activemq-jndi` quickstart displays the basic structure of activeMQ microservice through the JNDI configuration file. It uses the default initial context factory `ActiveMQInitialContextFactory` from the Artemis project to connect to the embedded instance of the Artemis broker. Two microservices send and receive two messages in both supported JMS versions.

The second quickstart `activemq-jms-annotation` shows the configuration with the `@JMS` annotation. This annotation specifies the server URI and initial context factory so no JNDI configuration file is present. It also presents the way of configuring a message listener for continuous receiving of messages from both JMS 1.1 and 2.0.

The last example `activemq-secured` introduces the security enabled ActiveMQ microservice. The security configuration is specified in the `broker.xml` file. The embedded instance of broker is set up with a predefined user which is used for the authentication. The microservice determines the user identity on each CDI injection by the `@JMS` annotation.

7.4 Performance testing

To present the applicability of the provided solution I implemented and executed a performance test for each provider respectfully. The performance was measured by the PerfCake testing framework [20].

7.4.1 Vert.x performance test

This test is based on the throughput of HTTP servers. The first server is established by the CDI REST provider. The class `RestfulMicroservice` annotated with the `@Gateway` annotation serves for this purpose. The second server is started from the `HttpVerticle` class.

Both classes use the `EchoService` microservice as a resource to assure the same conditions. It is injected directly into the REST microservice as it is annotated with the `@Microservice` annotation. As the verticle instance is created by the Vert.x platform, making it a microservice would result into the creating of two instances of the same class. For this reason, the direct injection into the verticle is impossible.

A workaround for this problem suppresses the automatic deployment of the verticle by the `@Deployment` annotation. It is deployed manually from a observer method of the `RestfulMicroservice` class. This observer makes a call to the `EchoService` which in its `@PostConstruct` method saves the injected instance into a static field. This instance is than retrieved directly by the verticle.

The PerfCake testing scenario is built on the throughput of each HTTP server in 30 seconds with the use of 100 threads. It is measured in iterations per second. Iteration means the successful request call to the server.

7.4.2 ActiveMQ performance test

This test maintains a RESTful interface for obtaining messages from the testing *queue*. It provides a ActiveMQ microservice and a POJO class that retrieves messages with the standard JMS. Both of these options are available under different resource paths.

The test fill in the testing *queue* in the observer method of the `RestfulMicroservice` class with 60 000 messages. PerfCake scenario individually retrieves 30 000 messages from each resource path. The results are compared by the time they took to fetch all messages and on the throughput with 100 threads.

7.4.3 Test results

The results of both tests proved that the provided solution is suitable from the performance perspective. Appendix C contains the graphs of average performance for each test respectfully. The full results and the whole implementation of both performance tests are available online as open-source projects on my github pages [21].

8 Conclusion

The goal of this thesis was to implement two messaging platforms as microservice messaging providers into the SilverWare microservices platform. The two integrated components were Vert.x and Apache ActiveMQ Artemis.

The Vert.x was realized in the `vertx-microservice-provider` module. This module maintains the shared `Vertx` instance to manage the verticle deployments. This platform instance is available directly through the CDI injection.

The Vert.x provider supports the automatic loading and deployment of the Java and Groovy class verticles. It also allows users to specify the deployments by the XML configuration file. It maintains the `@Deployment` annotation which can affect the verticle deployment. For the XML file, these options can be specified as attributes on the `verticle` element.

The `activemq-microservice-provider` module represents a client-to-server JMS communication. This provider is based on the Apache ActiveMQ Artemis project. It simplifies the connection of the microservice to the server through the injection of the connecting classes. In the future, the broker capabilities can be provided in a separate module.

This provider offers support for the JMS specification in versions 1.1 and 2.0. It is able to connect to different servers from the same microservice and to configure security identity for individual users.

This thesis described the microservice architecture along with the reasons why it should be used in the enterprise development. Additionally, the Spring and WildFly implementations of microservices were compared with the SilverWare project.

The design and the implementation of both messaging providers were supported by the UML class diagrams. The implementation was also promoted by the unit and integration test framework. The executed performance tests have shown that the solution is usable in the production environment. Furthermore, several quickstart examples were created to present the providers' usage.

The whole implementation of this thesis was provided as an open-source contribution to the SilverWare project.

Bibliography

[1] J. Lewis and M. Fowler, "Microservices," 2014. [Online]. Available: http://martinfowler.com/articles/microservices.html [Accessed: 2015-12-10]

[2] Silverware, "SilverWare: Microservices," 2015. [Online]. Available: http://www.silverware.io/ [Accessed: 2015-12-10]

[3] Red Hat, Inc., "The world's open source leader | red hat," 2015. [Online]. Available: http://www.redhat.com/en [Accessed: 2015-12-10]

[4] Vert.X, "Vert.x," 2015. [Online]. Available: http://vertx.io/ [Accessed: 2015-12-10]

[5] Apache Software Foundation, "ActiveMQ Artemis," 2015. [Online]. Available: https://activemq.apache.org/artemis/ [Accessed: 2015-12-10]

[6] D. Barry, "Service-Oriented Architecture (SOA) Definition," 2016. [Online]. Available: http://www.service-architecture.com/articles/web-services/serviceoriented_architecture_soa_definition.html [Accessed: 2015-11-24]

[7] C. Richardson, "Introduction to Microservices | NGINX," 2015. [Online]. Available: https://www.nginx.com/blog/introduction-to-microservices/ [Accessed: 2015-11-24]

[8] S. Newman, "Principles of Microservices," 2016. [Online]. Available: http://samnewman.io/talks/principles-of-microservices/ [Accessed: 2016-4-6]

[9] Swagger community, "Swagger framework," 2016. [Online]. Available: http://swagger.io/ [Accessed: 2016-3-18]

[10] M. Fowler, "Humaneregistry," 2008. [Online]. Available: http://martinfowler.com/bliki/HumaneRegistry.html [Accessed: 2016-3-18]

[11] D. Woods, “Building Microservices with Spring Boot,” 2016. [Online]. Available: http://www.infoq.com/articles/boot-microservices [Accessed: 2016-4-15]

[12] Spring community, “Spring Boot,” 2016. [Online]. Available: http://projects.spring.io/spring-boot/ [Accessed: 2016-5-5]

[13] WildFly-Swarm community, “WildFly Swarm User’s Guide,” 2016. [Online]. Available: https://wildfly-swarm.gitbooks.io/wildfly-swarm-users-guide/content/index.html [Accessed: 2016-4-17]

[14] W. Seongmin, “Inside Vert.x. Comparison with Node.js. | CUBRID Blog,” 2016. [Online]. Available: http://www.cubrid.org/blog/dev-platform/inside-vertx-comparison-with-nodejs/ [Accessed: 2015-11-23]

[15] Vert.X, “Vert.x Core Manual - Event Bus,” 2016. [Online]. Available: http://vertx.io/docs/vertx-core/java/#event_bus [Accessed: 2015-11-24]

[16] Oracle documentation, “Message-Oriented Middleware (MOM) (Sun Java System Message Queue 4.3 Technical Overview),” 2016. [Online]. Available: https://docs.oracle.com/cd/E19316-01/820-6424/aeraq/index.html [Accessed: 2016-2-26]

[17] B. Snyder, D. Bosnanac, and R. Davies, *ActiveMQ in action*. Manning, 2011.

[18] JBoss community, “HornetQ,” 2016. [Online]. Available: http://hornetq.jboss.org/ [Accessed: 2016-4-5]

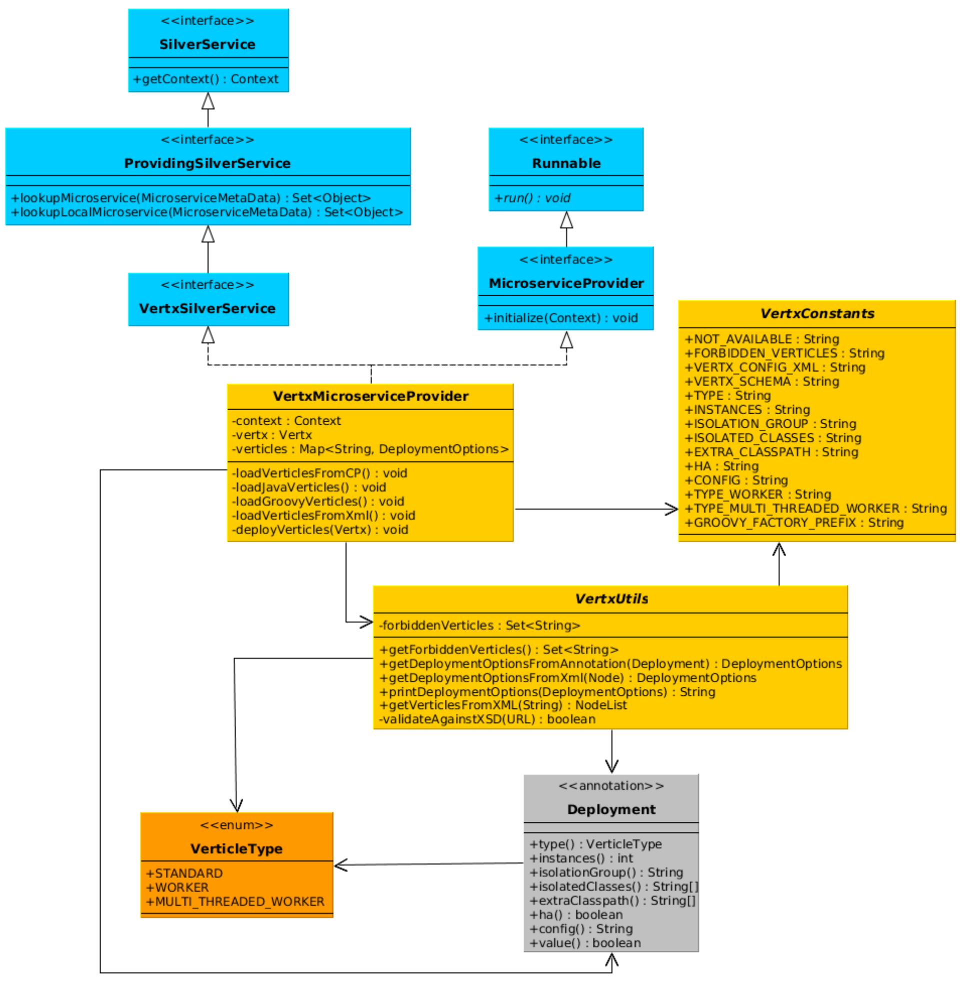
[19] SilverWare-Demos, “Silverware-demos,” 2016. [Online]. Available: https://github.com/px3/SilverWare-Demos [Accessed: 2016-3-31]

[20] PerfCake community, “Perfcake,” 2016. [Online]. Available: https://www.perfcake.org/ [Accessed: 2016-5-7]

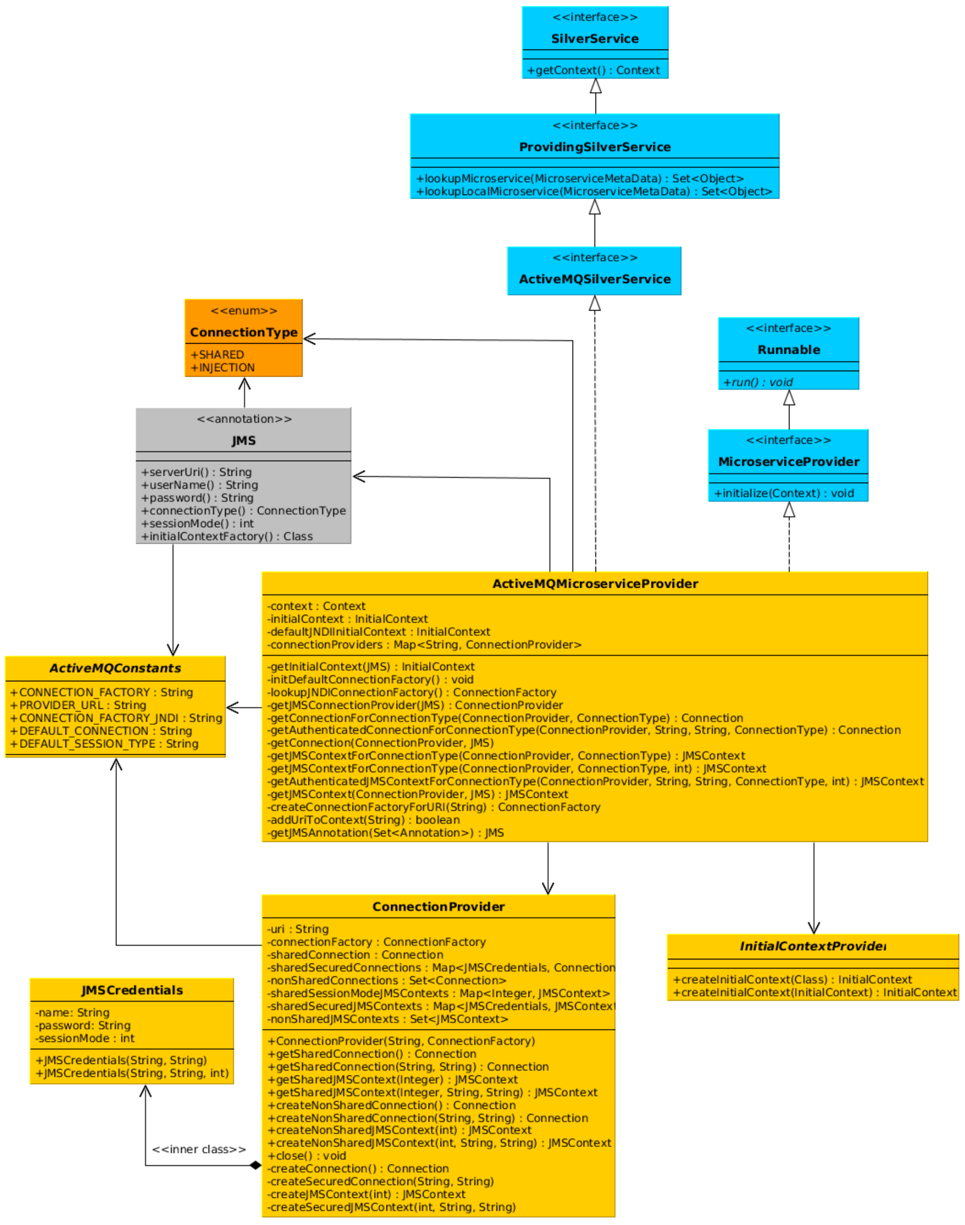
[21] xstefank, “xstefank repositories,” 2016. [Online]. Available: https://github.com/xstefank [Accessed: 2016-5-7]

A UML class diagrams

Vert.x Microservice Provider


<<interface>>
SilverService
+getContext() : Context
<<interface>>
ProvidingSilverService
+lookupMicroservice(MicroserviceMetaData) : Set<Object>
+lookupLocalMicroservice(MicroserviceMetaData) : Set<Object>
<<interface>>
Runnable
+run() : void
<<interface>>
VertxSilverService
<<interface>>
MicroserviceProvider
+initialize(Context) : void
VertxConstants
+NOT_AVAILABLE : String
+FORBIDDEN_VERTICLES : String
+VERTX_CONFIG_XML : String
+VERTX_SCHEMA : String
+TYPE : String
+INSTANCES : String
+ISOLATION_GROUP : String
+ISOLATED_CLASSES : String
+EXTRA_CLASSPATH : String
+HA : String
+CONFIG : String
+TYPE_WORKER : String
+TYPE_MULTI_THREADED_WORKER : String
+GROOVY_FACTORY_PREFIX : String
VertxMicroserviceProvider
-context : Context
-vertx : Vertx
-verticles : Map<String, DeploymentOptions>
-loadVerticlesFromCP() : void
-loadJavaVerticles() : void
-loadGroovyVerticles() : void
-loadVerticlesFromXml() : void
-deployVerticles(Vertx) : void
VertxUtils
-forbiddenVerticles : Set<String>
+getForbiddenVerticles() : Set<String>
+getDeploymentOptionsFromAnnotation(Deployment) : DeploymentOptions
+getDeploymentOptionsFromXml(Node) : DeploymentOptions
+printDeploymentOptions(DeploymentOptions) : String
+getVerticlesFromXML(String) : NodeList
-validateAgainstXSD(URL) : boolean
<<annotation>>
Deployment
+type() : VerticleType
+instances() : int
+isolationGroup() : String
+isolatedClasses() : String[]
+extraClasspath() : String[]
+ha() : boolean
+config() : String
+value() : boolean
<<enum>>
VerticleType
+STANDARD
+WORKER
+MULTI_THREADED_WORKER

ActiveMQ Microservice Provider



<<interface>>
SilverService
+getContext() : Context
<<interface>>
ProvidingSilverService
+lookupMicroservice(MicroserviceMetaData) : Set<Object>
+lookupLocalMicroservice(MicroserviceMetaData) : Set<Object>
<<interface>>
ActiveMQSilverService
<<enum>>
ConnectionType
+SHARED
+INJECTION
<<interface>>
Runnable
+run() : void
<<annotation>>
JMS
+serverUri() : String
+userName() : String
+password() : String
+connectionType() : ConnectionType
+sessionMode() : int
+initialContextFactory() : Class
<<interface>>
MicroserviceProvider
+initialize(Context) : void
ActiveMQMicroserviceProvider
-context : Context
-initialContext : InitialContext
-defaultJNDIInitialContext : InitialContext
-connectionProviders : Map<String, ConnectionProvider>
-getInitialContext(JMS) : InitialContext
-initDefaultConnectionFactory() : void
-lookupJNDIConnectionFactory() : ConnectionFactory
-getJMSConnectionProvider(JMS) : ConnectionProvider
-getConnectionForConnectionType(ConnectionProvider, ConnectionType) : Connection
-getAuthenticatedConnectionForConnectionType(ConnectionProvider, String, String, ConnectionType) : Connection
-getConnection(ConnectionProvider, JMS)
-getJMSContextForConnectionType(ConnectionProvider, ConnectionType) : JMSContext
-getJMSContextForConnectionType(ConnectionProvider, ConnectionType, int) : JMSContext
-getAuthenticatedJMSContextForConnectionType(ConnectionProvider, String, String, ConnectionType, int) : JMSContext
-getJMSContext(ConnectionProvider, JMS) : JMSContext
-createConnectionFactoryForURI(String) : ConnectionFactory
-addUriToContext(String) : boolean
-getJMSAnnotation(Set<Annotation>) : JMS
ActiveMQConstants
+CONNECTION_FACTORY : String
+PROVIDER_URL : String
+CONNECTION_FACTORY_JNDI : String
+DEFAULT_CONNECTION : String
+DEFAULT_SESSION_TYPE : String
ConnectionProvider
-uri : String
-connectionFactory : ConnectionFactory
-sharedConnection : Connection
-sharedSecuredConnections : Map<JMSCredentials, Connection
-nonSharedConnections : Set<Connection>
-sharedSessionModeJMSContexts : Map<Integer, JMSContext>
-sharedSecuredJMSContexts : Map<JMSCredentials, JMSContext
-nonSharedJMSContexts : Set<JMSContext>
+ConnectionProvider(String, ConnectionFactory)
+getSharedConnection() : Connection
+getSharedConnection(String, String) : Connection
+getSharedJMSContext(Integer) : JMSContext
+getSharedJMSContext(Integer, String, String) : JMSContext
+createNonSharedConnection() : Connection
+createNonSharedConnection(String, String) : Connection
+createNonSharedJMSContext(int) : JMSContext
+createNonSharedJMSContext(int, String, String) : JMSContext
+close() : void
-createConnection() : Connection
-createSecuredConnection(String, String)
-createJMSContext(int) : JMSContext
-createSecuredJMSContext(int, String, String)
InitialContextProvider
+createInitialContext(Class) : InitialContext
+createInitialContext(InitialContext) : InitialContext
JMSCredentials
-name: String
-password: String
-sessionMode : int
+JMSCredentials(String, String)
+JMSCredentials(String, String, int)
<<inner class>>

B Vert.x XML configuration

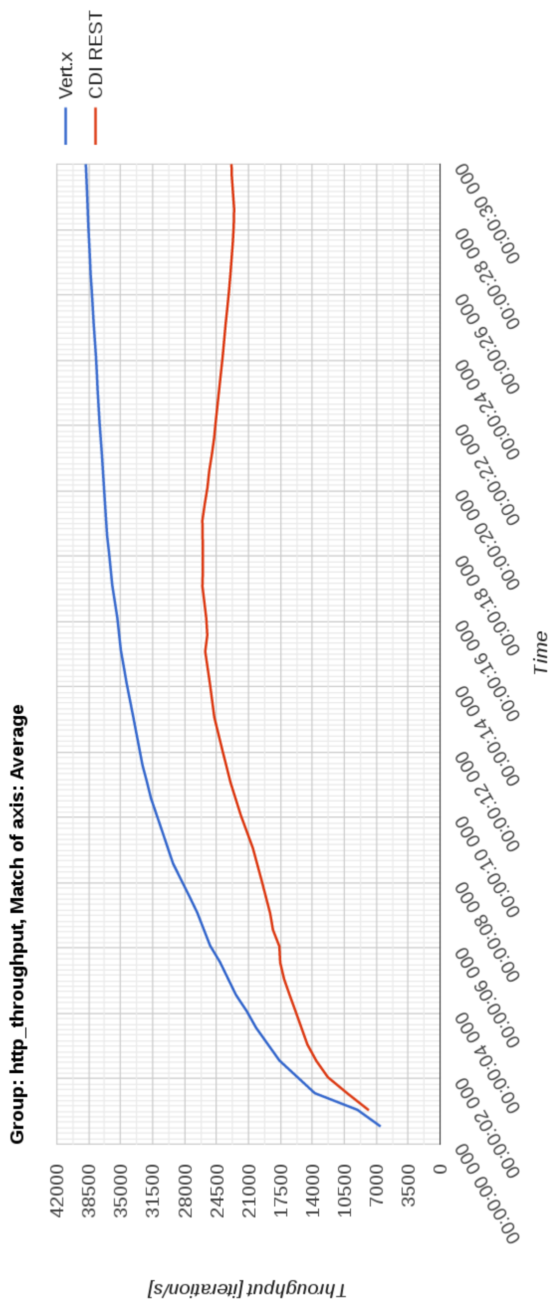
```
<vertx>
   <verticles>
      <!--verticle factory prefix-->
      <verticle>
         groovy:your.package.GroovyCompiledVerticle
      </verticle>
      <!--relative path to the verticle file-->
      <verticle>
         path/to/javascriptVerticle.js
      </verticle>
      <!--Java class name-->
      <verticle>
         your.package.YourVerticle
      </verticle>
      <!--usage of attributes to set the deployment-->
      <verticle type="worker" instances="10">
         path/to/verticle
      </verticle>
      ...
   </verticles>
</vertx>
```

Verticle tag attributes:

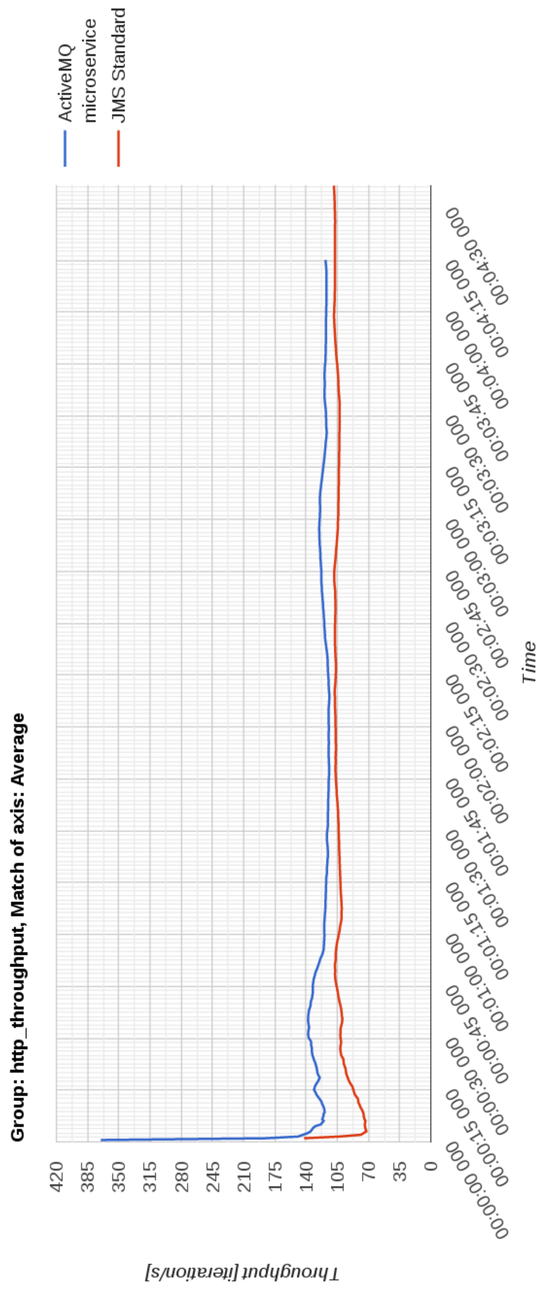
- `type` – {"standard", "worker", "multi-threaded-worker"}
- `instances` – Integer
- `isolationGroup` – String
- `isolatedClasses` – String list
- `extraClasspath` – String list
- `ha` – Boolean
- `config` – String

C Performance tests results

Vert.x performance test results



ActiveMQ performance test results


Group: http_throughput, Match of axis: Average
ActiveMQ microservice
JMS Standard
Throughput [iteration/s]
Time
00:00:00 000
00:00:15 000
00:00:30 000
00:00:45 000
00:01:00 000
00:01:15 000
00:01:30 000
00:01:45 000
00:02:00 000
00:02:15 000
00:02:30 000
00:02:45 000
00:03:00 000
00:03:15 000
00:03:30 000
00:03:45 000
00:04:00 000
00:04:15 000
00:04:30 000