TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Mahdad Khelghatdoust – 182472IVSM

ACTIVEMQ SUPPORT FOR MANAGING CLOUD INSTANCE CONFIGS USING SPRING FRAMEWORK

Master's thesis

Supervisor: Juhan-Peep Ernits
PhD

Tallinn 2020

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Mahdad Khelghatdoust – 182472IVSM

ACTIVEMQ TUGI PILVERAKENDUSTE SÄTETE HALDUSEL SPRING RAAMISTIKUS

magistritöö

Juhendaja: Juhan-Peep Ernits
PhD

Tallinn 2020

Author’s declaration of originality

I hereby certify that I am the sole author of this thesis. All the used materials, references to the literature and the work of others have been referred to. This thesis has not been presented for examination anywhere else.

Author: Mahdad Khelghatdoust

12.05.2020

Abstract

Deploying updated configuration to the applications is one of the most challenging parts of application deployment. Companies have different ways to deploy and apply their configuration changes. Cloud providers have different services for that. For example, Amazon Web Service recently released AppConfig.

Spring Cloud facilitates developers to develop distributed web applications in the popular Spring framework. Spring Cloud Stream Binder, which is a subproject of Spring Cloud, is useful in organizing communication between the components of the distributed application. We can use Spring Cloud Stream Binder to help our application communicate with the message broker and other components. Also, with using Spring Cloud Stream Binder, Spring Cloud Bus, and Spring Cloud Config, we can implement a project to push updated configuration to the client applications. Spring Cloud Stream Binder has several implementations for different messaging queues. Still, at the time of starting the thesis, there was no binder for ActiveMQ, a widely used and efficient messaging queue.

For fixing the problem, we develop a binder for ActiveMQ using Spring Cloud Stream framework. Our approach enables companies using ActiveMQ also to use it for configuration management. To put it in context, we compare it with Amazon AppConfig.

We demonstrate the result in two different sample projects. The first one is a consumer and producer application that communicate with each other through the provided Spring Cloud Stream Binder with ActiveMQ. The second project is using the proposed Spring Cloud Stream Binder with ActiveMQ to push updated configuration to client applications.

Keywords: cloud computing, configuration management, Spring Cloud framework, ActiveMQ

This thesis is written in English and is 48 pages long, including 6 chapters, 19 figures and 3 tables.

Annotatsioon

ActiveMQ tugi pilverakenduste sätete haldusel Spring raamistikus

Konfiguratsioonisätete uuendamine käigusolevates rakendustes on tarkvara halduses üks väiljakutseterohkemaid aspekte. Ettevõtted kasutavad rakenduste ja sätete käikuandmisel erinevaid meetodeid. Pilveteenuste pakkujatel on selleks erinevad teenused, näiteks Amazon Web Services lasi hiljuti välja rakenduse AppConfig.

Spring Cloud lihtsustab hajusveebirakenduste arendust populaarses Spring raamistikus. Spring Cloud Stream Binder on Spring Cloud alamprojekt, mille eesmärgiks on hajusrakenduse erinevate komponentide vahelise suhtluse korraldamine. Kasutada Spring Cloud Stream Binder komponenti, et aidata rakendusel suhelda sõnumihaldussüsteemiga ja teiste komponentidega. Kasutades Spring Cloud Stream Binder, Spring Cloud Bus ja Spring Cloud Config komponente loome lahenduse, mille abil saab levitada värskendatud konfiguratsioonisätteid. Spring Cloud Stream Binder sisaldab tuge mitme erineva sõnumihaldussüsteemi jaoks, kuid käesoleva töö alustamisel ei olnud olemas tuge ActiveMQ jaoks. ActiveMQ on laialdaselt kasutatav ja hea jõudlusega lahendus.

Probleemi lahenduseks arendame Spring Cloud Stream raamistikule ActiveMQ siduskomponendi. Meie lahendus võimaldab ActiveMQ-d kasutavatel ettevõtetel edaspidi ka sätteid ActiveMQ kaudu levitada. Lahenduse kontekstipanekuks võrdleme seda Amazon AppConfiguga.

Tulemust illustreerime kahe erineva näidisrakendusega. Esimene on sõnumite saatja ja vastuvõtja rakendus, kus suhtlus käib üle Spring Cloud Stream Binder komponendi ActiveMQ kaudu. Teises näidisrakenduses kasutatakse Spring Cloud Stream Binder komponenti ActiveMQ-ga uuendatud sätete saatmiseks klientrakendustesse.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 48 leheküljel, 6 peatükki, 19 joonist, 3 tabelit.

List of abbreviations and terms

MQ	*Messaging queue*
AWS	*Amazon web service*
DS	*Design science*
DSRM	*Design science research methodology*
GCP	*Google cloud platform*
POM	*Project object model*
URL	*Uniform resource locator*
JMS	*Java message service*
EC2	*Amazon elastic compute cloud*
S3	*Simple storage service*
IOT	*Internet of things*
JSON	*Javascript object notation*
Rest-API	*Representational state transfer- application programming interface*

Table of contents

1 Introduction 11
1.1 Unit of study 12
1.2 Motivation 12
1.3 Research goal 12
1.3.1 Research questions 13
1.4 Relevant concepts 13
1.5 Research design 14
1.5.1 Research method 14
1.6 Evaluation 15
2 Proposed solution 16
2.1 Spring Cloud Framework 17
2.1.1 Spring Cloud Bus 17
2.1.2 Spring Cloud Config 17
2.1.3 Spring Cloud Stream 18
2.2 Implementation 20
2.2.1 Req 1: The project should follow Spring cloud stream binders packaging structures 20
2.2.2 Req 2: Client Applications should be able to set connection properties of the ActiveMQ using our solution 21
2.2.3 Req 3: Consumers and Producers should be able to define the required properties 21
2.2.4 Req 4: The binder should be able to automatically provision consumer and producer destination by the name that is given in the configuration as a destination name 22
2.2.5 Req 5: Input and output channel should be able to connect to the ActiveMQ broker using our project, and the binder should be able to consume a message and handle it to a consumer 24
2.2.6 Req 6: The project should have a starter config to help clients to use the dependency easily 25

3 Amazon AppConfig 27
3.1 Amazon Web Services 27
3.1.1 Amazon Elastic Compute Cloud 27
3.1.2 Amazon Cloud Watch.... 27
3.1.3 Amazon Lambda Function.... 27
3.1.4 Amazon S3.... 28
3.1.5 Amazon System Manager 28
3.1.6 Amazon Parameter Store 28
3.1.7 Amazon AppConfig.... 28
3.2 The Advantages and Disadvantages of Amazon AppConfig 28
3.2.1 Advantages of Amazon AppConfig 28
3.2.2 Disadvantages of Amazon AppConfig 30
4 Amazon AppConfig Vs. Spring Cloud Config.... 31
4.1 Security.... 31
4.2 Cost.... 31
4.3 Ease of use 32
4.4 Functionality 33
5 Evaluation.... 35
5.1 Use case 1 35
5.2 Use case 2 40
5.3 Use case 3 41
6 Conclusion.... 46
References 47

List of figures

Figure 1: Spring Cloud Stream Application Model 19
Figure 2: Two destination models in ActiveMQ 23
Figure 3: Spring Cloud Stream Binder ActiveMQ Definition 25
Figure 4: Package Structure of Spring Cloud Starter Stream ActiveMQ 25
Figure 5: Pom of Spring Cloud Starter Stream module 26
Figure 6: HelloBinding Interface implementation 36
Figure 7: ProducerController implementation 37
Figure 8: The application.properties for use case 1 37
Figure 9: Dependency list for Producer project 38
Figure 10: HelloBinding implementation for the Consumer project 38
Figure 11: HelloListener Class implementation for the Consumer project 39
Figure 12: The application.properties for the Consumer project 39
Figure 13: Dependency list for the Consumer project 39
Figure 14: Config-Server Main class 42
Figure 15: The Application properties file for Config Server 42
Figure 16: Dependency List for Config Server Project 43
Figure 17: Controller for Config Client project 43
Figure 18: Bootstrap.yml file for Config Client project 44
Figure 19: The dependency list of Config Client project 45

List of tables

Table 1: Design science steps in this study adapted from design science research methodology for information systems [12]. 14
Table 2: Requirement list for Spring cloud stream binder ActiveMQ implementation . 16
Table 3: Comparison between Amazon AppConfig and Spring Cloud Config 33

1 Introduction

Spring framework is one of the most popular and valuable frameworks in the Java language for building web applications. Nowadays, many companies are using the Spring framework and implementing their applications relying on it. In addition to passing requirements, companies need to keep an eye on providing high availability services with downtime close to zero or as little as possible and responding to requests in an appropriate time. For supporting the mentioned problems, companies are using messaging queues to improve availability. There are many different messaging queue applications available, and one of the most widely known, open-source, and Multi-Protocol ones is ActiveMQ [1] [2]. For example, Amazon uses ActiveMQ as the underlying technology for AmazonMQ for providing three nines level of message durability [3].In the modern cloud architecture applications are divided into smaller and independent components that lead to easier development, deployment, and maintenance. Messaging queue provides communication between these distributed applications [1] [4]. The Spring framework also contains some projects to use with a messaging queue; one of these projects is Spring Cloud. Spring Cloud gives some projects that help developers to build some common patterns in cloud distributed systems. One of these patterns is configuration management [5]. But the problem was that at the time of starting the current work, Spring Cloud Stream did not support ActiveMQ.

In this thesis, we are going to clearly explain how we would like to use ActiveMQ and one of the Spring Cloud projects, specifically Spring Cloud Stream in order to make it easier to develop reliable applications. We implement Spring Cloud Stream for ActiveMQ, and we write down the requirements to make it possible to have principled evaluation. Then we implement a client to use Spring Cloud Stream with Spring Cloud Bus and Spring Cloud Config to manage our application configurations and push them directly to the application(s) without any other deployment or downtime in our application.

1.1 Unit of study

This study is about updating configuration properties of client applications using the Spring framework and ActiveMQ. The focus is on the Spring Cloud framework (specifically Spring Cloud Stream Binder) and ActiveMQ. The importance of the problem is confirmed by Amazon recently (November 2019) by announcing AppConfig, which provides a different solution to the configuration management problem [6]. Also Cisco is providing almost the same functionality as Amazon AppConfig [7]. We will compare our proposed solution to Amazon AppConfig. The solution that we propose in the current thesis will help companies who already have ActiveMQ in a project to use it also for configuration synchronization, thus simplifying the setup. Also, having the same technology stack for different tasks in a project helps to handle tasks in a more manageable way than having multiple different technology stacks in the same project.

1.2 Motivation

These days many companies are using ActiveMQ in cloud environments like Amazon or Microsoft Azure[1]. Our solution helps companies that are using ActiveMQ to publish their updated configurations to client applications; also, during this study, we noticed that in November 2019 Amazon released a new feature, AppConfig, that can be used to synchronize configuration parameters with clients. Its behavior is similar to our approach but is relevant in the Amazon environment, so it means that Amazon's solution is not available to be used with other cloud providers.

1.3 Research goal

We would like to study how updating configuration parameters of cloud applications can be achieved using ActiveMQ and Spring Cloud framework in order to keep the development and configuration of the application within one framework. The case study will involve the implementation of the solution for updating configuration parameters in client applications using Spring Cloud frameworks and ActiveMQ. We should mention

[1] A list of companies using ActiveMQ can be acquired from, e.g. (accessed on May 10, 2020) https://discovery.hgdata.com/product/apache-activemq

that our solution will also work with ActiveMQ over cloud service providers like Amazon or Microsoft Azure, etc.

1.3.1 Research questions

In this thesis, we will seek answers for the following main questions:

- How can we use Spring Cloud Stream framework with ActiveMQ?
- How will the proposed solution compare to the recently announced Amazon AppConfig?

For finding an answer to the first question, we will implement the solution using ActiveMQ and Spring Cloud sub-projects. Answering the second question involves running a suite of test applications with our implementation for Spring Cloud Stream and our example application for Amazon AppConfig.

1.4 Relevant concepts

We already know that Apache ActiveMQ is a message broker for communication among different systems using the Java Message Service specification [8] [9]. Also, we know that Spring Cloud Config is one of the Spring Cloud projects, and it is a programming library that gives us server-side and client-side support for externalized configuration in a distributed system. Using Config Server, we have a central place to manage external properties for applications across all environments [10]. In addition to this information, we know that for achieving the goal of the current thesis, we need to use many different Spring Cloud projects that we will explain in the section 2.1.

Although we can use different messaging queues and provide a solution to push real-time configurations to clients, at the time of starting the thesis work, there was no existing solution for pushing updates to client applications using ActiveMQ and Spring Cloud Framework. We should mention at this point that many different companies are using ActiveMQ on different cloud solutions like Amazon Web Service, Microsoft Azure, or Google Cloud Platform, and this solution can help them to solve the configuration management problem more elegantly. Besides, after we had already started the current work, Amazon released a new feature that is called AppConfing. AppConfig makes application configuration faster than with traditional deployment. Amazon AppConfig can handle deployment to many different application types. An application can be a microservice on EC2 instance, or mobile app, or Amazon Lambda or a system you run on behalf of others [6].

1.5 Research design

To answer research questions, we start by reading the existing documents related to different messaging queues; in the second step, we implement the solution for pushing updated configuration properties to Application clients using Spring Cloud Config, Spring Cloud Stream Binder, and Spring Cloud Bus. We read the Spring Framework Github project documentation to understand the current solution based on RabbitMQ, which helps us to implement a solution to support ActiveMQ and also follow the Spring Framework open-source principles. Then we apply an example application using ActiveMQ. Finally, we compare our approach with Amazon AppConfig. The implementation starts from defining essential requirements, and by providing an implementation that supports the use cases.

1.5.1 Research method

In this study, we are going to use the design science research methodology (DSRM) to cover our research plan [11], [12]. Based on the mentioned methodology, we have the following steps:

Table 1: Design science steps in this study adapted from design science research methodology for information systems [12].

Step	Definition
Identify problem and motivation	Define the problem and showing the importance
Define solution	What is the solution? And what would be a better approach?
Design and development	Artifact
Demonstration	Using artifact to solve the problem
Evaluation	Test and evaluate the result based on the requirements

As we can see from Table 1, we have five phases. First, we identify the problem and explain motivation, as we did in the above section. Second, we define our solution as we mentioned we are going to implement a Spring Cloud Stream Binder framework for ActiveMQ. The next phase is designing and developing of the suggested solution. In step five, we demonstrate our solution, and we show a client application that is using our framework to push the updated configuration. Finally, we evaluate our solution by implementing some use cases. In case if there is any failure in the evaluation step, we come back to step three and apply some changes and continue with steps four and five.

1.6 Evaluation

In order to validate and interpret the outcome, we are going to implement some use cases to test our solution and illustrate how it would be used in a real-life scenario. The first use case is one producer project and one consumer project, which communicates with each other as two micro services using our solution and binder application. The second project is a client and server applications. The server application serves the client(s) configurations and also pushes updated configuration to the client(s).

2 Proposed solution

In this section, we are going to explain our solution and illustrate how does it work. For a better and more precise explanation, we decided to start by designing some requirements. Then we introduce some general related topics about Spring framework, and finally, we explain how our implementation answers to the needs.

The requirement list is the following:

Table 2: Requirement list for Spring cloud stream binder ActiveMQ implementation

No	Requirement	Explanation
1	The project should follow Spring cloud stream binders packaging structures	Like other Spring Cloud Stream Binder Implementations
2	Client Applications should be able to set connection properties of the ActiveMQ using our solution	Properties for connection to ActiveMQ
3	Consumers and Producers should be able to define the required properties	Properties for setting up Consumers and producers
4	The binder should be able to automatically provision consumer and producer destination by the name that is given in the configuration as a destination name	It should support the destination for both Consumer and producer
5	Input and output channel should be able to connect to ActiveMQ broker using our project, and the binder should be able to consume a message and handle it to a consumer	It should support two destination models: Topic and Queue
6	The project should have a starter config to help clients to use the dependency easily	The Pom File to handle Spring Cloud Stream Binder Implementation

2.1 Spring Cloud Framework

Spring Cloud is one of the popular projects of the Spring Framework that provides some services to developers to build some common patterns of distributed applications that can be deployed easily and as fast as possible. Some of these common patterns that Spring Cloud delivers are as follows: configuration management, service discovery, one-time token, a control bus, micro proxy, and so on [5]. For supporting each of these patterns, Spring Cloud contains some projects, some of the notable projects under Spring Cloud framework, which are relevant to our topic are the following:

Spring Cloud Stream, Spring Cloud Bus, Spring Cloud Config

For achieving one of the goals of this thesis, which is *pushing updated configuration to client applications using ActiveMQ*, we need to get more familiar with the mentioned Spring Cloud projects. For the said goal, Spring Cloud Config uses Spring Cloud Bus to send a message or push an updated config to clients. Still, for accomplishing this purpose, Spring Cloud Bus itself relies on Spring Cloud Stream, specifically Spring Cloud Stream Binder Implementation. There is a lack of Spring Cloud Stream Binder implementation for ActiveMQ that we are going to implement.

2.1.1 Spring Cloud Bus

Based on the Official Spring Cloud Bus guide [13], this project is being used for connecting nodes of a distributed system with a lightweight message broker. This functionality can be used to manage configuration or broadcast a state change or other management instructions. The Spring Cloud Bus and Spring Cloud Stream Binder Rabbit is using Spring Cloud Stream. We can add our Spring Cloud Stream Binder instead of Spring Cloud Stream Binder Rabbit.

Besides, it is essential to say that the Spring Cloud Bus is an abstraction built on top of Spring Cloud Stream.

2.1.2 Spring Cloud Config

Spring Cloud Config is another project under the Spring Cloud framework. This project provides support for server-side and client-side applications, and the goal of this project is to externalize configuration in a distributed system. There is a direct relation between

Spring Cloud Config Starter and Spring Cloud Config Client and Spring Cloud Starter project.

As we already mention Spring Cloud Config supports client and server sides, we need to use these dependencies so it would be good to mention some of the features of these two projects [10]:

- Spring Cloud Config Server:
  - Providing APIs to support external configuration. It can be name-value pair or content of YAML file
  - Encrypt and decrypt property values
- Spring Cloud Config Client:
  - Bind to the config server and initialize Spring Environment using property resources
  - Encrypt and decrypt property values

2.1.3 Spring Cloud Stream

The last Spring Cloud framework project that is currently relevant is Spring Cloud Stream. Based on Spring Cloud Stream framework official website [14] at the time of writing, the project supports a variety of binder implementations for a different type of queues:

- RabbitMQ
- Apache Kafka
- Kafka Streams
- Amazon Kinesis
- Google PubSub (Partner maintained)
- Solace PubSub+ (Partner maintained)
- Azure Event Hubs (Partner maintained)
- Apache RocketMQ (Partner maintained)

As we can see, there is no support for ActiveMQ, so this is the part that we are contributing and implementing ActiveMQ support for the mentioned project. Also, we should mention at this point that the Spring Cloud Stream project is mainly used for the development of highly scalable event-driven micro services linked to standard messaging systems [5].

Based on Spring Cloud Stream official guide [5], each Spring Cloud Stream Binder project contains the following three core modules:

1. Destination Binder: Components that are in charge of integration with external messaging systems.
2. Destination Bindings: Bridge between the external messaging systems and the application provided by Message Producers and Consumers (created by Destination Binders).
3. Message: The data structure used by producers and consumers to communicate with destination Binders.

In Figure 1, we can see the Spring Cloud Stream Application model:

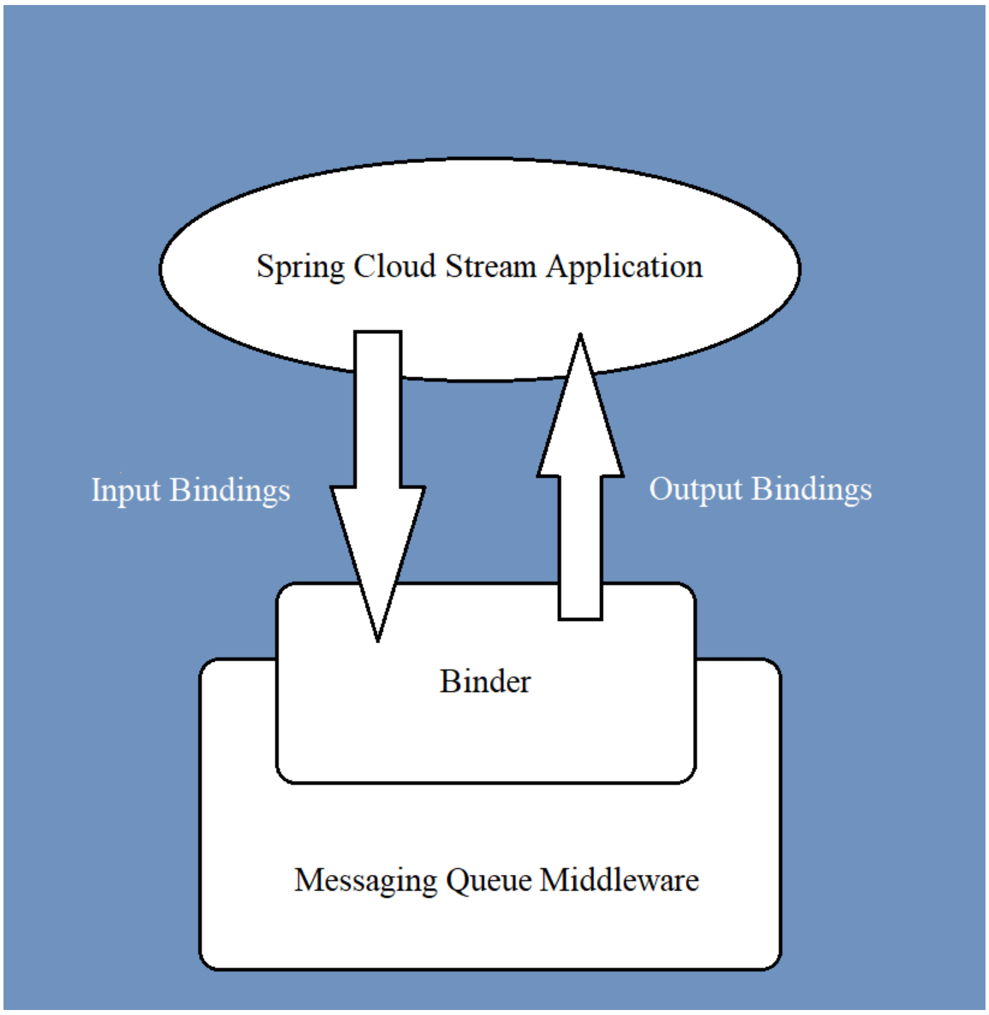


Figure 1: Spring Cloud Stream Application Model

Another fact about using Spring Cloud Stream is that if we want to use a messaging queue, like RabbitMQ, typically, we should write a lot of boilerplate code and properties to connect to RabbitMQ. But by using Spring Cloud Stream, we do not need to write that code because the framework gives us an abstract view, and it handles all the configuration parts automatically.
In the next step, we are going to explain how we would like to implement Spring Cloud Stream Binder for ActiveMQ.

2.2 Implementation

For implementing the Spring Cloud Stream Binder for ActiveMQ in a principled way, we are going to specify how we will satisfy the requirements that we already defined in Table 2.
Also, final implementation can be downloaded from this repository: https://github.com/madkt12/spring-cloud-stream-binder-activemq

2.2.1 Req 1: The project should follow Spring cloud stream binders packaging structures

We should say that Spring Cloud is an open-source project, so we use the Spring Cloud Stream Binder Rabbit Github repository that is based on RabbitMQ as a role model to get familiar with the project and framework structures.
We divide our implementation into the following different modules:

1. Spring Cloud Stream Binder ActiveMQ Core: This project includes core functionalities of our Spring Cloud Stream binder that provides provisioning, properties, and mapping.
2. Spring Cloud Stream Binder ActiveMQ: This project consists of the configuration properties based on client projects, and it has a binder of the ActiveMQ channel.
3. Spring Cloud Starter Stream ActiveMQ: This project is just a dependency on Spring Cloud Stream Binder ActiveMQ to use it in our Spring Cloud Stream client projects. And it is an alternative way to use the project's dependency directly.

We should add this point that all these modules are included in a parent project. Also, the parent project includes all the common dependencies. We will explain and discuss all the mentioned projects in the following parts.

2.2.2 Req 2: Client Applications should be able to set connection properties of the ActiveMQ using our solution

To meet this requirement, we should explain that in the Spring Cloud Stream Binder ActiveMQ Core project, we create a package called properties, and in that class, we declared host, user, and password properties. Also, we defined configuration properties annotation over the class and adding this prefix: spring.cloud.stream.active.binder. The naming convention for this prefix is almost the same as other Spring Cloud Stream Binder projects. For example, for the RabbitMQ, it is like this: spring.cloud.stream.rabbit.binder. So this class helps us to get connection properties from client application and we can use ActiveMQConnectionFactory from ActiveMQ dependency to create a connection whenever it is needed. In addition to this information, for following object-oriented programming rules and design patterns, we defined getters and setters for properties to prevent direct access to the properties.

As we are using Configuration properties annotation here, we need to add Spring Boot dependency, because we need to have this dependency into other projects we decided to add it to the parent project.

2.2.3 Req 3: Consumers and Producers should be able to define the required properties

To satisfy this requirement, we define two classes, one for consumer properties and named it ActiveConsumerProperties and another one for producer: ActiveProducerProperties. Both Java classes are placed in the properties package in the Spring Cloud Stream Binder ActiveMQ Core project. We need to add the following variables:

- Destination: Shows the destination
- Type: It can be Topic or Queue; by default is Queue
- Transactional: True/False; by default is False

We need Spring Cloud Stream to identify our properties, so first, we should add Spring Cloud dependency to the parent project, then we have to implement the BinderSpecificPropertiesProvider interface. This interface helps Spring Cloud Stream to get consumer and producer objects from the binder project, which is Spring Cloud Stream Active Binder here. Also, to complete this part, we have to extend AbstractExtendedBindingProperties and override the getExtendedPropertyBinding method and return our ActiveBindingProperties.class. So, after all these steps, now this configuration helps Spring Cloud Stream to identify consumer, producer, and connection properties.

2.2.4 Req 4: The binder should be able to automatically provision consumer and producer destination by the name that is given in the configuration as a destination name

First, it is good to explain briefly how does ActiveMQ work, and then we elaborate the way that we use it to implement the current part of the project.
Producers create messages and send them to a destination. Consumers receive and make a process over messages [1] [9]. But before that ActiveMQ broker routes messages to one of two following destinations:

1. To a queue: that delivers messages to a single consumer (also known as a point to point model) [9].
2. To a topic: that delivers messages to multiple consumers (also known as a publish/subscribe model) [9].

In the following figure, we show two different destination models in ActiveMQ.

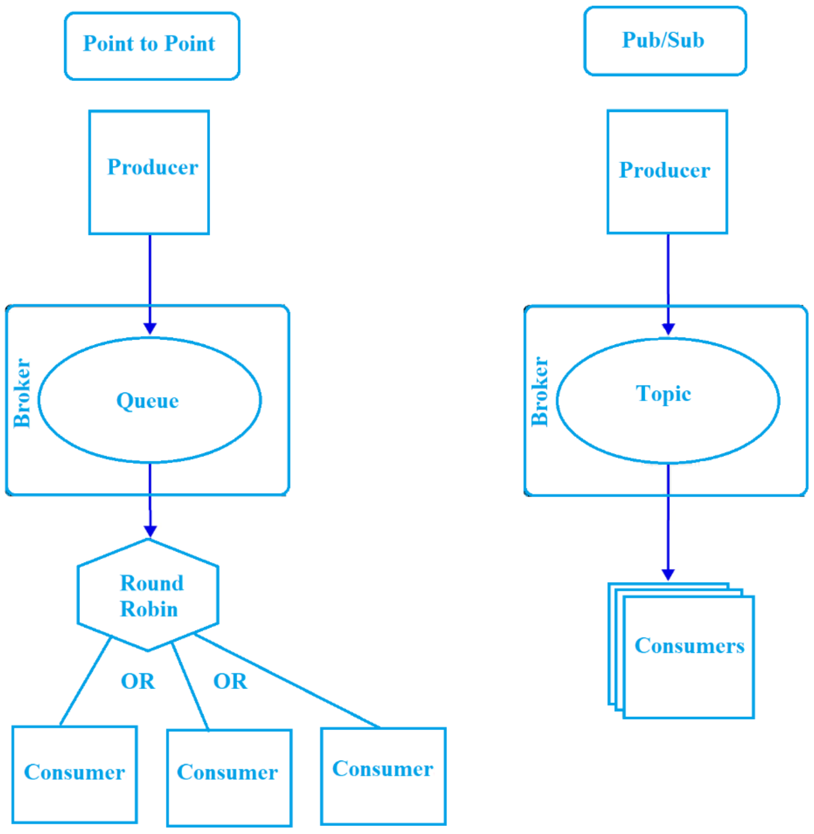


Figure 2: Two destination models in ActiveMQ

As can be seen in Figure 2, the consumers are getting messages from the destination and not directly from producers.

For implementing this part, we need to create the provisioning package in Spring Cloud Stream Active Binder Core and add the ActiveProvisioner class. In this class, we create consumer and producer destinations based on the extended properties that we mentioned in the previous requirement. So, the client properties related to the destination will be used in the class. As we can see from the name of the package and class, this class is in charge of the provisioning of consumer and producer destinations. Again, in this class, we are using ProvisioningProvider interface from Spring Cloud Stream dependency to make our project compatible with it.

The last package of this project is mapping. This package has a class named ActiveMessageProducerMapper. This class extends MessageProducer to create producer

endpoint, as the MessageProducer is an Abstract class, creating an object out of it, is impossible, so we need to implement our own MessageProducer Mapper.
The functionality of this module is that it creates a connection to ActiveMQ and prepares the Consumer endpoint as well as supports the produced message. In more detail, in this section, we check the type of destination if it is a queue or a topic and then take the appropriate action. The above checking is happening on the producer part. As we already mentioned, this module is tightly coupled with the previous module, and it is using all the properties that Spring-Cloud-Stream-Binder-ActiveMQ-Core project contains.

2.2.5 Req 5: Input and output channel should be able to connect to the ActiveMQ broker using our project, and the binder should be able to consume a message and handle it to a consumer

The current requirement depends on all previous requirements.
When sending a message, the message producer has to honor the properties set in the client application. Also, there should be a consumer endpoint that can listen to the proper destination based on the given properties and receive the messages. In this case, we need to create an ActiveMessageChannelBinder class in the Spring-Cloud-Stream-Binder-Active project. This class includes two main methods that are being inherited from AbstractMessageChannelBinder. We need to override the following two methods:

- CreateProducerMessageHandler: It provides the logic to produce the message. Also, it identifies which destination type is required based on the given properties.
- CreateConsumerEndpoint: ActiveMessageProducerMapper is responsible for consuming the message and processed it to the client. Usually, the message producer classes extend MessageProducerSuppourt class from the Spring Integration framework. Still, in our case, as we need to use JmsTemplate instead of MessageTemplate for creating a connection to JMS, and also we need to have a customized message listener, we created our ActiveMessageProducerMapper class. We should mention that this class is in the mapper package in the Spring-Cloud-Stream-Binder-ActiveMQ-Core project.

Our binder relies on the ActiveProvisioning class, which is being initialized in our constructor with ActiveBinderConfigurationProperties.
Finally, Spring Cloud Stream looks at the META-INF package inside the resources folder to find the type of Spring binder of our Spring Cloud Stream binder project. So after

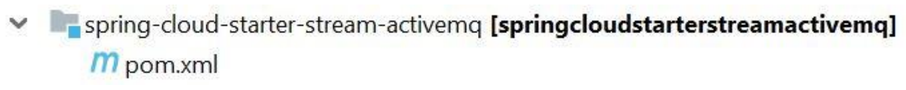
finishing the above classes, we have to write the configuration shown in Figure 3 into Spring.binder file and declare our binder configuration entry.

```
activemq:\
stream.config.ActiveMessageChannelBinderConfiguration
```

Figure 3: Spring Cloud Stream Binder ActiveMQ Definition

2.2.6 Req 6: The project should have a starter config to help clients to use the dependency easily

The configuration goes into a pom file that can be used in the Spring Cloud Stream client projects as a dependency. The structure of this module is simple and straight as shown in Figure 4.



```
spring-cloud-starter-stream-activemq [springcloudstarterstreamactivemq]
    pom.xml
```

Figure 4: Package Structure of Spring Cloud Starter Stream ActiveMQ

The contents of the pom file is given in Figure 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <parent>
        <groupId>org.spring.cloud.stream</groupId>
        <artifactId>spring-cloud-stream-binder-activemq-
parent</artifactId>
        <version>1.0-SNAPSHOT</version>
    </parent>

    <modelVersion>4.0.0</modelVersion>

    <artifactId>spring-cloud-starter-stream-activemq</artifactId>
    <description>Spring Cloud Starter Stream Activemq</description>

    <url>https://projects.spring.io/spring-cloud</url>
    <organization>
        <name>Pivotal Software, Inc.</name>
        <url>https://www.spring.io</url>
    </organization>

    <properties>
        <main.basedir>${basedir}/../..</main.basedir>
    </properties>

    <dependencies>
        <dependency>
            <groupId>org.spring.cloud.stream</groupId>
            <artifactId>spring-cloud-stream-binder-
activemq</artifactId>
        </dependency>
    </dependencies>

</project>
```

Figure 5: Pom of Spring Cloud Starter Stream module

The pom file has some metadata related to the Spring project, and specifically, Spring Cloud projects like URL, organization name for Spring. It also has a dependency on Spring Cloud Stream Binder ActiveMQ, which is one of our modules for using this in the client projects. In addition to the information, the pom file has using the parent element as one out of four modules in the Spring Cloud Stream ActiveMQ Binder project.

3 Amazon AppConfig

3.1 Amazon Web Services

The current chapter is dedicated to explaining how Amazon AppConfig works and what functionality of Amazon Web Services is required for it to function, and then we compare our proposed solution with the new feature of Amazon Web Services. On 25 November 2019, Amazon announced AppConfig, a new service that the customers can use to easily apply their updated configuration properties to the application, which is hosted in Amazon environments [15]. Before we start explaining Amazon AppConfig, we need to understand some concepts of Amazon Web Services.

3.1.1 Amazon Elastic Compute Cloud

Amazon Elastic Compute Cloud or Amazon EC2 supports Amazon Web Services and provide computing capabilities. By using Amazon EC2, we do not need to be worried about our application's hardware, so it means we do not need to spend money on hardware in advance. In addition to this information, Amazon EC2 is scalable; it means we can scale it up or down based on our requirements. Also, it helps customers to deploy and run their applications faster and easier [16].

3.1.2 Amazon Cloud Watch

Amazon Cloud Watch monitors other Amazon web services, resources, and applications that are hosted by Amazon. It collects logs, events, and operational data. Also, it can react appropriately against an alarm and fire some actions at the required time. For example, we can simply monitor different resources usages like CPU usages on our Amazon EC2 instance [17]. Amazon Cloud Watch has an essential role in Amazon web Services.

3.1.3 Amazon Lambda Function

AWS Lambda Function allows us run an application without any managing servers. With the Lambda function, we can run our code for virtually any type of application or backend service - all with zero administration. Just we upload our code, then Lambda takes care of everything required to run and scale the code in case if it is needed [18].

3.1.4 Amazon S3

Amazon S3 is an object storage service providing scalability, data availability, security, and performance. It means users and companies can use it to save and secure any amount of data across a variety of use cases [19].

3.1.5 Amazon System Manager

Amazon System manager is a flexible management service that helps us to manage and administer our workloads. This amazon service is designed to enable easy configuration and management of systems on a large scale, and it makes writing automation artifacts so simple [20]. Amazon System Manager contains many different services; two of these services are essential in our scenario; Amazon Parameter Store and Amazon AppConfig.

3.1.6 Amazon Parameter Store

Amazon Parameter Store gives us a way to handle our configuration data in a secure mode. The service helps us to separate our secrets and configuration data from our code. These parameters also can be plain text or secrets like database passwords. And it can be used in other AWS services. The parameter store is part of the Application management suite of the Amazon System Manager [21].

3.1.7 Amazon AppConfig

Amazon AppConfig is one of the Amazon web service products which is under AWS System Manager in application management suite, which helps customers to manage and deploy application configurations fast. You can use AppConfig with an application hosted on Amazon Elastic Compute Cloud (EC2) instances, Amazon Lambda Functions, Android, IOS (generally all mobile apps), containers and IoT devices, and with several more types of applications [22]. Besides this, Amazon.com, Kinder, and Alexa are using Amazon AppConfig at the time of writing, and it seems that the service is widely used within various products/services offered by Amazon.

3.2 The Advantages and Disadvantages of Amazon AppConfig

3.2.1 Advantages of Amazon AppConfig

Based on the Amazon official documentation [22], AppConfig provides the following benefit:

- Deploy using different source of configuration
    - Amazon AppConfig supports configuration, which is stored in *Amazon Parameter Store*, *System Manager Document*, and recently on 13 March 2020, they announced that AppConfig is supporting *Amazon S3* as a Configuration source as well [23].
- Decreasing errors during configuration changes
    - Amazon AppConfig reduces errors during configuration deployment, using *validation of configuration* changes. It means that before deployment, we can *validate changes by defining a JSON schema or creating some Lambda function*.
- Update application without interruption
    - Amazon AppConfig *deploys configuration changes without any interruption at runtime*.
- Control deployment of changes across the application
    - Amazon AppConfig provides *deployment strategy*, which gives more control over the configuration changes deployment, and by that, we can say how quickly we want our application targets to receive changes.
- Easy rollback during the failure of configuration changes
    - We can easily *enable Amazon CloudWatch to monitor our deployment*, and in case if there is any problem, we can roll back to the preferred configuration version.
- Managing different environments for deploying configuration changes
    - Amazon AppConfig provides us a *configuration profile* that we can set it based on the different environments that we have or based on the sub-component and apply the changes.
- Security management of deploying configuration changes

    - We can *give access to the preferred role* to be able to deploy configuration changes.

3.2.2 Disadvantages of Amazon AppConfig

We can say at the time of writing Amazon AppConfig has the following drawbacks:

- Young product
    - Amazon AppConfig is quite a young product in Amazon Web Services, which is publicly available.
- Is not available in all regions
    - Amazon AppConfig is not available in some regions. E.g. China.
- Using it needs some code changes in the current application
    - We need to configure our application in such a way that our application will be able to poll the updated configuration using the GetConfiguration API periodically.

4 Amazon AppConfig Vs. Spring Cloud Config

In previous chapters, we discussed the benefits and drawbacks of Amazon AppConfig. We also proposed a solution based on the Spring Cloud Stream ActiveMQ Binder, Spring Cloud Config and Spring Cloud Bus that we can use to push updated configuration to the client applications.

In the current chapter, we are going to compare Amazon AppConfig to our solution based on the following criteria: security, cost, flexibility, ease of use and some of the important functionalities. Also we should say that the comparison in this section is tightly coupled with many different points And the result should be interpreted based on the conditions.

4.1 Security

Amazon Web Service helps us to handle role based access and it means we can apply which roles and users are allowed to deploy changes. Still, we should say that in the Spring framework, we can easily enable the Spring Security framework by adding Spring Security dependency to the project and apply role-based access to the Cloud Config part. Also we need to mention that the proposed solution should be used in an internal network so by default it mitigates some risks of having public interface [24].

We know that Amazon AppConfig supports the Amazon Parameter Store as a secure source of configuration. But again, in Spring Cloud Config, we can enable encryption, so based on the public key and private key, we can save encrypted parameters and fetch decrypted parameters. In addition to this information Spring Cloud Config Server supports Vault. Vault is a tool for securely accessing secrets.

As a result, we can say both approaches are following enough security principles. Just by using Amazon AppConfig by default we are using a secure way but for Spring Cloud Config more developments needed.

4.2 Cost

If we already are using Amazon Web Services, then the cost of using Amazon AppConfig is not considerable, and it offers many different ways of billing. But if we are already using an on-premises ActiveMQ instance, then it is more reasonable to use the Spring Cloud Config and our proposed solution.

Generally, the cost of using Amazon AppConfig itself is not expensive. At the time of writing the thesis based on the Amazon system manager guide [25], AppConfig is following the “pay per usage” approach. It means that we pay only for what we are using, and in AppConfig, we pay 0.2 dollars per 1M GetConfiguration API calls and 0.0008 dollars for a configuration change per instance. For example, if we have an application that its configuration changes 5 times per day, and also we have 1000 target that are fetching configuration changes every 5 minutes, after a month the cost of using AppConfig is as follow:

Each month per minute = 30 days* 24 hours* 60 minutes = 43200

Cost of API calls after a month = 1 configuration* 1000 targets* 0.2 calls per minute * 43200 * 0.2 dollars per 1 million calls = 1.728 dollars

Cost of deployments after a month =1 configuration* 1000 targets* 5 changes per day* 30 days* 0.0008 dollars for each deployment= 120

Total cost = 1.728 Cost of API calls after a month + 120 Cost of deployments after a month = 121.728 dollars

But we should add this point that this is just the cost of Amazon AppConfig. If we want to use Amazon Parameter Store or any other service to act as a source of configuration and using CloudWatch to monitor target, then we should pay for those services separately.

As a result, however using Amazon AppConfig with all required features like Amazon CloudWatch and Amazon S3 can invoice a considerable cost but if we want to have same functionality in Spring Cloud Config probably we need to pay more for monitoring tools and services.

4.3 Ease of use

Using AppConfig in an existing application is a little difficult because after setup in Amazon environment, we have to change our application in a way that we call GetConfiguration api periodically to fetch configs and set it to the application. But using

Spring Cloud Config is not that difficult, and we can easily create Spring Cloud Config Server project and add Spring Cloud Config dependency to our project and start using it, in the evaluation part, use case 2 we illustrate how we can do it.

As a result, we can say developing Spring Cloud Config solution can be easier approach specially if we want to apply it into an existing project. Because for applying Amazon AppConfig into an existing project considerable coding needed but for Spring Cloud Config just we need to modify the application properties and add a Spring Cloud Config dependency.

4.4 Functionality

In this section, we compare some of the Amazon AppConfig functionalities that we mentioned in the section 3.2 with Spring Cloud Config solution.

Table 3: Comparison between Amazon AppConfig and Spring Cloud Config

Amazon AppConfig	**Spring Cloud Config**
Deploy using different source of configuration	Spring Cloud Config also supports a lots of different source of configuration. It supports following list: Git, SVN, file System, AWS S3, Vault, Redis and JDBC
Validation of configuration parameters before deploying	Spring Cloud Config Server supports different formats of configuration. It can be Yaml, properties, plain texts and JSON. Validation can be achieved with extra developments. For example there is a separate project in the following git repository:https://github.com/intuit/intuit-spring-cloud-config-validator/blob/master/README.md
Configuration changes without any interruption at runtime	It depends on the server configuration and infrastructure.

Deployment strategy	There is no strategy, and all are depend on the person who is responsible for deployment. However, there is a retry option on the client-side that in case if Config Server application is not available in application start time, it helps to retry and fetch the configuration.
Monitoring and roll back functionality	Monitoring can be set up on the server and client applications. But there is no automatic roll back. Also, there is a health indicator option in config on client side that can be used to monitor client configs.
Managing different environments and configuration profiles	It supports different environment profiles and it can be used as a configuration profiles. But there is no validators for configuration.

As a result, both approaches are providing a lot of functionalities, but we should say that achieving some functionalities in Amazon AppConfig is much easier than Spring Cloud Config.

5 Evaluation

In the current chapter, we are going to evaluate our proposed solution. We decided to implement some test applications and try to use our project as a dependency. We designed some use cases:

- Use case 1: There are two applications (one producer and one consumer), and they are communicating with each other using Spring Cloud Stream framework dependencies and our proposed solution. Both Topic and Queue destination models should be verified in this use case.

- Use case 2: There are three applications (one producer and two consumers), and they are communicating with each other using Spring Cloud Stream framework dependencies and our proposed solution. Both Topic and Queue destination models should be verified in this use case.

- Use case 3: There are two applications (one server and one client), and the server pushes updated configuration to the client. Using the required Spring Cloud Framework and our proposed solution.

5.1 Use case 1

Description:

1. The user calls a rest API in the browser.

2. The producer sends the message based on the configuration to the given destination and channel name.

3. The consumer receives the message based on the configuration from the given destination and channel name.

4. The consumer prints the received message to the console.

5. We repeat the steps for both Topic and Queue destinations.

Implementation:

We create two projects, Cloud-Stream-Producer-ActiveMQ and Cloud-Stream-Consumer-ActiveMQ. The projects can be clone from following GitHub repositories:

- Cloud-Stream-Producer-ActiveMQ: https://github.com/madkt12/cloud-stream-producer-activemq
- Cloud-Stream-Consumer-ActiveMQ: https://github.com/madkt12/cloud-stream-consumer-activemq

In the producer project, we created two packages, one is the binding, and another one is the controller. Also, we create an application.properties in the resources directory of our project. In the binding package, we create an interface called HelloBinding. The interface is responsible for binding message channel names so that Spring Cloud Stream can identify it as a message channel. In Figure 6, you can see our HelloBinding interface implementation.

```
import org.springframework.cloud.stream.annotation.Output;
import org.springframework.messaging.MessageChannel;

public interface HelloBinding {


    @Output("greetingChannel")
    MessageChannel greeting();
}
```

Figure 6: HelloBinding Interface implementation

Now we need to implement our controller class so that the Spring boot framework identifies our REST-API. In Figure 7, you can see our controller class implementation.

```
import org.springframework.messaging.Message;
import org.springframework.messaging.MessageChannel;
import org.springframework.messaging.support.MessageBuilder;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.PathVariable;
import org.springframework.web.bind.annotation.RestController;
import un.ttl.thesis.cloudstreamproduceractivemq.binding.HelloBinding;

@RestController
public class ProducerController {

    private MessageChannel greet;

    public ProducerController(HelloBinding binding) {
        greet = binding.greeting();
    }

    @GetMapping("en/greet/{name}")
    public void publish(@PathVariable String name) {
        String greeting = "Hello, " + name + "!";
        Message<String> msg = MessageBuilder.withPayload(greeting)
                .build();
        this.greet.send(msg);
    }
}
```

Figure 7: ProducerController implementation

As can be seen, we are using the MessageChannel that we defined in the HelloBinding interface. Now we just need to update application.properties file to the following code shown in Figure 8.

```
server.port=8080
spring.cloud.stream.active.binder.host=tcp://localhost:61616
spring.cloud.stream.active.binder.password=admin
spring.cloud.stream.active.binder.user=admin
spring.cloud.stream.active.bindings.greetingChannel.producer.destinati
on=greetings
spring.cloud.stream.active.bindings.greetingChannel.producer.type=queu
e
spring.cloud.stream.default-binder=activemq
```

Figure 8: The application.properties for use case 1

In the application properties file, we set the host, password, and user values of our ActiveMQ broker. We informed Spring Cloud Stream that our default binder implementation is activemq. Then, we inform our binder about bindings channel name and producer destination and type. In this part, we verified Req 2 and 3.

Finally, in the pom file, we have following dependencies as well as the Spring Boot Parent as shown in Figure 9.

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
        <dependency>
      <groupId>org.spring.cloud.stream</groupId>
           <artifactId>spring-cloud-starter-stream-
activemq</artifactId>
      <version>1.0-SNAPSHOT</version>
        </dependency>

   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-stream</artifactId>
      <version>3.0.2.RELEASE</version>
   </dependency>
</dependencies>
```

Figure 9: Dependency list for Producer project

As can be seen, we are just using Spring-Cloud-Starter-Stream-ActiveMQ dependency here from our binder implementation. The scenario covers Req 6.
By running the project and calling the rest-API, we can check the ActiveMQ admin web browser and see the required channel and destination is created there. Also, we can verify that message is in the enqueue and pending list. The scenario covers Req 4 and 5 partially.

Now we need to implement the Cloud-Stream-Consumer-ActiveMQ project. We have two packages, the binding and the listener. The binding package has an interface called HelloBinding as shown in Figure 10.

```
import org.springframework.cloud.stream.annotation.Input;
import org.springframework.messaging.SubscribableChannel;

public interface HelloBinding {

String GREETING = "greetingChannel";

    @Input(GREETING)
    SubscribableChannel greeting();
}
```

Figure 10: HelloBinding implementation for the Consumer project

As the latter is our consumer project we have Input annotation instead of Output, and also we have SubscribableChannel methods return type to connect to the greetingChannel the place where data will be pushed.
We have a listener class named HelloListener in the listener package, as shown in Figure 11.

```
@EnableBinding(HelloBinding.class)
public class HelloListener {

    @StreamListener(target = HelloBinding.GREETING)
    public void processHelloChannelGreeting(String msg) {
        System.out.println(msg);
    }
}
```

Figure 11: HelloListener Class implementation for the Consumer project

We have StreamListener annotation that helps the consumer and Cloud Stream framework to understand that the method is listening to the greetingChannel.

Our application properties file is shown in Figure 12.

```
server.port=9090
spring.cloud.stream.active.binder.host=tcp://localhost:61616
spring.cloud.stream.active.binder.password=admin
spring.cloud.stream.active.binder.user=admin
spring.cloud.stream.active.bindings.greetingChannel.consumer.destinati
on=greetings
spring.cloud.stream.active.bindings.greetingChannel.consumer.type=queu
e
spring.cloud.stream.default-binder=activemq
```

Figure 12: The application.properties for the Consumer project

So in the application properties file, we set the host, password, and user values of our ActiveMQ broker. We informed Spring Cloud Stream that our default binder implementation is activemq. Then, we inform our binder about bindings channel name and consumer destination and type. The scenario validates Req 2 and 3.

Finally, in the pom file, we have following dependencies as well as the Spring Boot Parent as shown in Figure 13.

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>
   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-stream</artifactId>
   </dependency>

   <dependency>
      <groupId>org.spring.cloud.stream</groupId>
      <artifactId>spring-cloud-starter-stream-activemq</artifactId>
      <version>1.0-SNAPSHOT</version>
   </dependency>
</dependencies>
```

Figure 13: Dependency list for the Consumer project

As can be seen, we are just using Spring-Cloud-Starter-Stream-ActiveMQ dependency here from our binder implementation. In this part, we tested Req 6.
By running the project and calling the producer rest-API, we can check the ActiveMQ admin web browser and see the required channel and destination was created there. Also, we can verify that the counter increased in the enqueue and dequeue part. And finally, we can see the output in the consumer console. In this part, we tested Req 4 and 5.
In the last step of the use case, we want to make some changes with application properties to ensure that the topic model works.
So we change the destination types to Topic and restart the application again. And we can see the message in the consumer project console as we expected. In this part, we tested Req 4 and 5 to see that our solution works with the Topic destination.

5.2 Use case 2

Description:

1. The user calls a rest API in the browser.
2. The producer sends the message based on the configuration to the given destination and channel name.
3. All consumers receive the message based on the configuration channel name if the destination type is Topic.
4. All consumers print the received message to the console.
5. We repeat the steps for both Queue destinations, but this time just one consumer receives the message.

Implementation:

We have same project as use case 1. So we have Cloud-Stream-Producer-ActiveMQ, Cloud-Stream-Consumer-ActiveMQ, but we would like to add new instance of the consumer so we would have two consumer projects and named it Cloud-Stream-Consumer-ActiveMQ-1 and Cloud-Stream-Consumer-ActiveMQ-2.

We have exactly the same implementation for both consumers and we just do some changes in application.properties. We change the destination types to the topic and run all projects and call the rest-API to send the message to the destination and check both consumers to verify that they received the message.

As the last test, we would add a new channel to the producer, and by calling two different rest-APIs, we sent two different messages to two different channels that one was a Queue destination, and another one was Topic destination.

At the end of this use case, we tested and verified that all requirements are working well.

5.3 Use case 3

Description:

1. The user pushes the updated configuration properties to a git repository
2. The server fetches the properties
3. The server pushes properties to the client
4. The client received updated configuration

Implementation:

We create two projects, Config-Server and Config-Client. The projects can be clone from following GitHub repositories:

- Config-Server: https://github.com/madkt12/spring-cloud-config-server
- Config-Client: https://github.com/madkt12/spring-cloud-config-client

In this implementation, dependencies are the most crucial part, because we want to make sure that our Spring Cloud Stream Binder ActiveMQ is compatible with Spring Cloud Bus, Spring Cloud Config. So first, we need to say again that, Spring Cloud Config is for pushing a notification to clients relies on Spring Cloud Bus. Spring Cloud Bus for sending notification relies on Spring Cloud Stream, specifically Spring Cloud Stream Binder Implementations.

On the server-side, we need to implement the code shown in Figure 14 in the main class.

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.cloud.config.server.EnableConfigServer;

@SpringBootApplication
@EnableConfigServer
public class TestServerApplication {

    public static void main(String[] args) {
        SpringApplication.run(TestServerApplication.class, args);
    }

}
```

Figure 14: Config-Server Main class

The only change that we have here is EnableConfigServer annotation, and this annotation lets Spring know that this class is our Cloud Config Server. Then we need to implement the application properties, as shown in Figure 15.

```
spring.cloud.config.server.git.uri=https://github.com/madkt12/my-
config.git
management.endpoint.refresh.enabled=true
server.port=8001
management.endpoints.web.exposure.include=*
spring.cloud.stream.default-binder=activemq
spring.cloud.stream.active.binder.host=tcp://localhost:61616
spring.cloud.stream.active.binder.password=admin
spring.cloud.stream.active.binder.user=admin
```

Figure 15: The Application properties file for Config Server

This properties file shows that we have set the git URL that our project fetches configuration. It also can be a simple file on the server or somewhere else. The second line will enable endpoint refresh. Our config server port is 8081. And we also set our stream binder configuration. We should add this point that we need to mention for Spring Cloud Stream that the default Binder is ActiveMQ, so it will not communicate with RabbitMQ instead of ActiveMQ.

The configuration file is on git, and it has some key-pair values. As we can see, there is no channel or destination definition in our application properties file. Spring Cloud Bus will define it automatically.

The Config server project dependencies are shown in Figure 16.

```
<dependencies>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-server</artifactId>
        <version>2.2.2.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-config-monitor</artifactId>
        <version>2.2.2.RELEASE</version>
    </dependency>

    <dependency>
        <groupId>org.spring.cloud.stream</groupId>
        <artifactId>spring-cloud-starter-stream-activemq</artifactId>
        <version>1.0-SNAPSHOT</version>
    </dependency>

    <dependency>
        <groupId>org.springframework.cloud</groupId>
        <artifactId>spring-cloud-bus</artifactId>
        <version>2.2.1.RELEASE</version>
    </dependency>

</dependencies>
```

Figure 16: Dependency List for Config Server Project

After running the project, we see that the configuration is loaded. Then on the client-side, we need to implement the Config Client project. In the Config Client in addition to the main class we have a controller class that simply shows the properties that it fetches from config-server, as shown in Figure 17.

```
@RestController
@RefreshScope
public class ClientController {
  @Value("${db-username}")
  private String dbUsername;

  @Value("${contentful-username}")
  private String contentfulUsername;

  @GetMapping("/db")
  public String showDbUserName() {
    return "The db username is: " + dbUsername;
  }


  @GetMapping("/contentful")
  public String showContentFulUserName() {
    return "The contentful username is: " + contentfulUsername;
  }

}
```

Figure 17: Controller for Config Client project

The class contains an annotation RefreshScope that is being used to load properties value from the Spring Cloud Config Server.
The Application properties file in the project is so simple, and it just includes application port numbers. But there is a bootstrap.yml file that is necessary for any Config Client project that wants to fetch data from the Spring Config Server project. The bootstrap.yml file is shown in Figure 18.

```
spring:
  application:
    name: config-client

  cloud:
    config:
      uri: http://localhost:8001
    stream:
      default-binder: activemq
      active:
        binder:
          host: tcp://localhost:61616
          user: admin
          password: admin

  profiles:
    active: dev
```

Figure 18: Bootstrap.yml file for Config Client project

The name of the application properties is essential, as the Config server typically has to serve up properties for many applications. The cloud.config.url is the Config-server URL. We need to mention for Spring Cloud Stream that the default Binder is ActiveMQ, so it will not communicate with RabbitMQ instead of ActiveMQ. And also we set broker properties.And finally, we can set active profile and fetch properties based on the profile and file name in the git repository.
The Pom file of this project is shown in Figure 19.

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-starter-web</artifactId>
   </dependency>

   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-starter-config</artifactId>
   </dependency>

   <dependency>
      <groupId>org.springframework.cloud</groupId>
      <artifactId>spring-cloud-bus</artifactId>
      <version>2.2.1.RELEASE</version>
   </dependency>

   <dependency>
      <groupId>org.spring.cloud.stream</groupId>
      <artifactId>spring-cloud-stream-binder-activemq</artifactId>
      <version>1.0-SNAPSHOT</version>
   </dependency>
</dependencies>
```

Figure 19: The dependency list of Config Client project

We called the client API, and we received the fetched properties from git. We updated properties and pushed them to the repository by calling the monitor API from Config-Server, the Spring Cloud Config understands which file has been updated and by calling the following API: curl -X POST http://localhost:8001/actuator/bus-refresh Config Server pushes updated configuration to the client. Then Config-Client fetches it. In this communication, Spring Cloud Bus created a Queue automatically and named springCloudBus.

By the above use case, once again, we tested and validated that our solution is compatible with Spring Cloud Bus and Spring Cloud Config. And They are working as they do but with our Spring Cloud Stream ActiveMQ Binder.

6 Conclusion

As illustrated in the evaluation section, the goal of the thesis is reached. The developed framework is a tool that helps developers and operations to push their configuration changes to different application targets easily using ActiveMQ. The framework is also useful in case if the company has many microservices and needs to make a connection between these components. Also, we should add this point that the developed framework can be used along with other Spring Cloud Framework components easily.

We also compared our solution to Amazon AppConfig and discussed the benefits and drawbacks of it in the context of the Spring Cloud framework. The service is quite young, but if a company is already using other Amazon Web Services, Amazon AppConfig can help manage configuration changes.

We compared our solution with Amazon AppConfig regarding security, cost, ease of use, and some of the critical functionality. We explained that Spring Framework could help us to achieve a secure way to handle role-based authentication, which allows us to give push notification access to those users that are eligible to handle it. However, Amazon is making money as a service provider and also, Amazon AppConfig costs for companies, on-premises server, and ActiveMQ instance cost as well. We know that Amazon Web Services provide many different services along with AppConfig, which makes working with it more comfortable. For example, Amazon CloudWatch supports Amazon AppConfig, and we can use this monitoring tool in case of failure in configuration deployment and roll back to the preferred version.

Finally, we should say that the Spring framework is an open-source project. And any open-source project needs an active community to be alive. This framework helps developers who are using ActiveMQ and Spring Cloud framework projects.

References

[1] V. M. Ionescu, “The analysis of the performance of RabbitMQ and ActiveMQ,” *14th RoEduNet International Conference - Networking in Education and Research (RoEduNet NER), Craiova,* pp. 132-137, 2015.

[2] “Apache ActiveMQ,” The Apache Software Foundation, [Online]. Available: https://activemq.apache.org/. [Accessed 12 2019].

[3] “Amazon MQ FAQs,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/amazon-mq/faqs/. [Accessed 02 2020].

[4] “Benefits of Message Queues,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/message-queue/benefits/. [Accessed 01 2020].

[5] “Spring Cloud,” Spring, [Online]. Available: https://spring.io/projects/spring-cloud. [Accessed 01 2020].

[6] “Safe Deployment of Application Configuration Settings With AWS AppConfig,” Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/systems-manager/latest/userguide/appconfig.html. [Accessed 12 2019].

[7] “Method for managing application configuration state with cloud based application management techniques,” [Online]. Available: https://patents.google.com/patent/US20190303212A1/en. [Accessed 04 2020].

[8] A. F. Klein, . M. Ştefănescu, . A. Saied and K. Swakhoven, “An Experimental Comparison of ActiveMQ and OpenMQ,” *Fifth International Conference on Digital Information Processing and Communications (ICDIPC),* pp. 24-30, 2015.

[9] B. Snyder, D. Bosanac and R. Davies, ActiveMQ in Action, Manning Publications, 2011.

[10] “Spring Cloud Config Reference Guide,” [Online]. Available: https://cloud.spring.io/spring-cloud-config/reference/html/. [Accessed 01 2020].

[11] K. Peffers, T. Tuunanen, C. Gengler, M. Rossi, W. Hui, V. Virtanen and J. Bragge, “The design science research process: A model for producing and presenting information systems research,” 2006.

[12] K. Peffers, T. Tuunanen, M. A. Rothenberger and S. Chatterjee, “A design science research methodology for information systems research,” *Journal of Management Information Systems,* vol. 24, no. 3, pp. 45-77, 2007.

[13] “Spring Cloud Bus,” [Online]. Available: https://spring.io/projects/spring-cloud-bus. [Accessed 01 2020].

[14] “Spring Cloud Stream,” [Online]. Available: https://spring.io/projects/spring-cloud-stream. [Accessed 01 2020].

[15] “simplify application configuration with aws appconfig,” Amazon Web Services, 25 November 2019. [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2019/11/simplify-application-configuration-with-aws-appconfig/. [Accessed 12 2019].

[16] “What is Amazon EC2?,” Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/concepts.html. [Accessed 01 2020].

[17] “What Is Amazon CloudWatch?,” Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/WhatIsCloudWatch.html. [Accessed 01 2020].

[18] “What Is AWS Lambda?,” Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/lambda/latest/dg/welcome.html. [Accessed 01 2020].

[19] “Amazon S3,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/s3/. [Accessed 01 2020].

[20] “AWS Systems Manager,” Amazon Web Services, [Online]. Available: https://www.amazonaws.cn/en/systems-manager/. [Accessed 12 2019].

[21] “AWS Systems Manager Parameter Store,” Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/systems-manager/latest/userguide/systems-manager-parameter-store.html. [Accessed 01 2020].

[22] “AWS AppConfig,” Amazon Web Services, [Online]. Available: https://docs.aws.amazon.com/systems-manager/latest/userguide/appconfig.html. [Accessed 12 2019].

[23] “AWS AppConfig announces integration with Amazon S3,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/about-aws/whats-new/2020/03/aws-appconfig-announces-integration-with-amazon-s3/. [Accessed 03 2020].

[24] “Common vulnerabilities and exposures,” [Online]. Available: https://cve.mitre.org/cgi-bin/cvekey.cgi?keyword=spring. [Accessed 04 2020].

[25] “AWS Systems Manager pricing,” Amazon Web Services, [Online]. Available: https://aws.amazon.com/systems-manager/pricing/. [Accessed 02 2020].