Självständigt arbete på grundnivå

Independent degree project - first cycle

Datateknik
Computer Engineering

JMS communication
Communicating with a legacy JMS broker using JavaScript

Patrick Bylund
Abstract

The objective of this thesis is to research how to use JavaScript to communicate with a legacy JMS broker, specifically OpenMQ 4.1 and implementing a proof-of-concept of this. This proof-of-concept should be able to send and receive JMS messages. The necessary preparations for designing this application was to research possible ways of communicating with different JMS brokers using other languages than Java. This research showed that the only viable method in order to communicate with this version of OpenMQ is to create a Java bridge between JavaScript and the JMS broker. The results shows that it’s possible to use JavaScript to communicate with OpenMQ 4.1 by using a Java Bridge together with NodeJS. This result is performed by making an integration test, by first sending a message with a certain ID to the API and then receiving a response at the intended response destination with the same ID. The front-end of the proof-of-concept provides a web application UI for performing this integration test. Future improvements of this implementation can be done by extending the Java Bridge and API in such a way that it can be used against many different JMS brokers. The configuration data used can also be stored in more reliable ways, if multiple users are using the API and web application at the same time, for example by using a database instead of JSON files.

Keywords: JMS, OpenMQ, legacy broker, asynchronous messages.
Foreword

I want to thank Dewire for providing me with the opportunity of doing this thesis, specifically my supervisor Jakob Pedersen who have helped me and given me feedback during my work. I would also like to thank my supervisor at the university Martin Kjellqvist who took his time to help me get things started.
# Table of Contents

Abstract.................................................................................................................................. iii
Foreword.................................................................................................................................... iv
Terminology................................................................................................................................ vii

1 Introduction.......................................................................................................................... 1
  1.1 Background and problem motivation............................................................................. 1
  1.2 Overall aim...................................................................................................................... 1
  1.3 Scope.............................................................................................................................. 2
  1.4 Concrete and verifiable goals......................................................................................... 2
  1.5 Outline........................................................................................................................... 2

2 Theory................................................................................................................................... 4
  2.1 Messaging....................................................................................................................... 4
  2.2 Java Message Service.................................................................................................... 4
  2.2.1 Point-to-point (PTP)................................................................................................. 5
  2.2.2 Publish-and-subscribe.............................................................................................. 6
  2.2.3 JMS Message........................................................................................................... 6
  2.3 OpenMQ........................................................................................................................ 6
  2.4 STOMP.......................................................................................................................... 7
  2.5 UMS.............................................................................................................................. 7
  2.6 NodeJS.......................................................................................................................... 7
  2.7 NPM.............................................................................................................................. 8
  2.7.1 Net......................................................................................................................... 8
  2.7.2 Express.................................................................................................................... 8
  2.7.3 Child process.......................................................................................................... 8
  2.7.4 Body-parser............................................................................................................ 8
  2.8 JSON............................................................................................................................. 8
  2.9 REST............................................................................................................................. 8
  2.10 Angular......................................................................................................................... 9
  2.11 Typescript.................................................................................................................... 10
  2.12 Dependency Injection................................................................................................. 10
  2.13 Promise......................................................................................................................... 10

3 Methodology......................................................................................................................... 12
  3.1 Research......................................................................................................................... 12
  3.1.1 Multi-threading and Asynchronous programming.................................................. 12
  3.2 Tools............................................................................................................................. 12
  3.2.1 Programming languages......................................................................................... 13
  3.2.2 Editors and libraries............................................................................................... 13
  3.2.3 Illustration tools....................................................................................................... 13
  3.3 Integration testing........................................................................................................... 13
  3.3.1 Sending messages.................................................................................................... 14
  3.3.2 Receiving messages............................................................................................... 14
3.4  Front-end..............................................................................................................14

4  Implementation.....................................................................................................16
  4.1  Architecture........................................................................................................16
  4.2  Design considerations.........................................................................................17
  4.3  Application Programming Interface.................................................................18
      4.3.1  Communication..........................................................................................19
      4.3.2  Sending messages.......................................................................................20
      4.3.3  Receiving messages...................................................................................21
      4.3.4  Configuration.............................................................................................22
  4.4  Java Bridge..........................................................................................................23
      4.4.1  Communication with Java Message Service..............................................23
      4.4.2  Communication with the API.......................................................................24
  4.5  Web Application..................................................................................................25
      4.5.1  Components.................................................................................................25
      4.5.2  Data handling..............................................................................................26

5  Results.....................................................................................................................27
  5.1  Front-end..............................................................................................................27
      5.1.1  Status bar.....................................................................................................27
      5.1.2  Configurable Brokers..................................................................................27
      5.1.3  Configurable Destinations..........................................................................28
      5.1.4  Saved Messages........................................................................................29
  5.2  Integration test.....................................................................................................30

6  Conclusions..............................................................................................................32
  6.1  Goals and result..................................................................................................32
  6.2  Future improvements.........................................................................................32
  6.3  Ethical aspects....................................................................................................33

References.................................................................................................................34
# Terminology

## Acronyms

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMS</td>
<td>Java Message Service</td>
</tr>
<tr>
<td>MOM</td>
<td>Message Oriented Middleware</td>
</tr>
<tr>
<td>JNDI</td>
<td>Java Naming and Directory Interface</td>
</tr>
<tr>
<td>REST</td>
<td>Representational State Transfer</td>
</tr>
<tr>
<td>IDE</td>
<td>Integrated Development Environment</td>
</tr>
<tr>
<td>HTTP</td>
<td>HyperText Transfer Protocol</td>
</tr>
<tr>
<td>API</td>
<td>Application Programming Interface</td>
</tr>
<tr>
<td>UI</td>
<td>User interface</td>
</tr>
<tr>
<td>TCP</td>
<td>Transmission Control Protocol</td>
</tr>
<tr>
<td>URI</td>
<td>Uniform Resource Identifier</td>
</tr>
<tr>
<td>JSON</td>
<td>JavaScript Object Notation</td>
</tr>
</tbody>
</table>
1 Introduction

1.1 Background and problem motivation

Software development is an ever changing industry. Tools, frameworks and even standards used in a project one year can be outdated the next year. The new and fancy project becomes a legacy system in no-time. It’s often too expensive to rebuild such a system with state of the art tools, especially large ones. The ability to integrate and communicate with these systems is strongly dependent on the APIs they provide, possibly forcing old technologies to be used even in new projects – and this is a problem.

In Java, communication between systems can be done with an API called Java Message Service (JMS). It decouples and decreases the dependencies between the systems. However, JMS is just a specification with guidelines which different vendors must implement. In turn, the vendors are allowed to use any protocol as long as the JMS API contract is fulfilled. It happens that vendors stop developing their implementation of the API resulting in the problem described above.

Within the organization Dewire, a JMS provider called OpenMQ is in use as a broker. This OpenMQ broker is used mainly by system testers who today manually send test data using a command line interface. A solution to make this process easier is sought for. This study will investigate if there's a way of implementing a JMS client that can communicate with this broker using an interpreted language like JavaScript and provide a proof-of-concept of this.

1.2 Overall aim

The overall aim is to identify and implement a technical solution for communicating with an OpenMQ broker using JavaScript. This will be performed by finding a way of communicating with an OpenMQ broker provided by Dewire, by using a Java client as a bridge to the broker that uses the native OpenMQ implementation of JMS. This Java client will be designed with the possibility of adding additional providers in the future. The JavaScript implementation will use NodeJS as a back-end, making it possible for users to configure brokers, configure queues and topics and send and receive messages. Receiving messages can be made by making a subscription of a certain queue or topic and specifying a certain queue where the response will be placed. The NodeJS back-end will provide an abstraction in the form of Representational State Transfer (REST). An API with support for Web Sockets or long polling for integration testing will be implemented. The aim is also to identify how Java and JavaScript compare when it comes to asynchronous calls in the sense of suitability.
1.3 Scope
There are many different types of implementations of the JMS API and many different JMS providers. There are many different versions of these providers as well with different features. The proof-of-concept application will be created to work against the JMS provider OpenMQ, specifically version 4.1.

1.4 Concrete and verifiable goals
The following goals should be fulfilled:

• Implement a Java client communicating with OpenMQ, acting as a bridge between the NodeJS back-end and the broker.

• Make Java client extendable with new JMS providers such that new providers can be added if necessary.

• Create front-end with the possibility of configuring brokers, configuring queues and topics and be able to send/receive messages. Also make it possible to specify a certain queue to place responses and make subscriptions on a specified queue or topic.

• An API that implements long polling or web sockets in order to perform integration tests.

• Find a way to provide error handling between the NodeJS back-end and Java client.

1.5 Outline
In chapter 2, a description of the theory behind all the different tools, libraries and technologies used in this thesis is provided. A description of the different approaches that is possible to use to create this proof-of-concept is provided in chapter 3, also a description of how the sending and receiving process works. Chapter 4 describes how the different parts of the proof-of-concept is implemented. Chapter 5 presents the result of the front-end and shows how to perform an integration test. Conclusions with personal reflections about the result, are described in chapter 6. It also describes future work and ethical aspects.
2 Theory

The theory section describes the used technologies and concepts used, in order to give a better understanding of the report.

2.1 Messaging

Messaging makes it possible for different software components and applications to communicate without the necessity of knowing anything about the system it communicates with. In this sense, messaging is loosely-coupled. The sender can send a message to the destination and the receiver can retrieve this message from the same destination when it's ready. This is in contrast to tightly coupled technologies, where the sender and receiver needs to know internal details about the system it communicates with. In messaging, the sender and receiver only needs to know which message format is used and what destination to send the messages to. [1]

2.2 Java Message Service

Java Message Service is an API that define interfaces in order to allow loosely coupled, reliable and asynchronous messaging. The API is reliable in the sense that it can give guarantees that a message is delivered only once and doesn't miss any messages. Clients that receive messages don't need to request them, the provider implementing JMS can deliver them when they are ready which makes JMS asynchronous. [1]

The specification of JMS provides two different domains for communication, which is two common approaches to messaging. Point-to-point and publish/subscribe. Both types of communication can be used in the same application. [2]

For more details about the different communication domains, see 2.2.1 and 2.2.2.

Different JMS providers may have differences in their messaging implementations. In order for JMS clients to be portable, they are expected to communicate through JMS interfaces which are portable. JMS administered objects are used to perform this, which is created by administrating the provider. There are two types of these JMS administered objects. ConnectionFactory, the object the client uses to create a connection with the provider and Destination, an object used to specify the destination of the messages. These objects are placed in a JNDI (Java Naming and Directory Interface) namespace. [3] Figure 1 illustrates this concept. Figure 2 illustrates the relationship between the different parts of JMS.
2.2.1 **Point-to-point (PTP)**

In PTP, each message corresponds to a specific queue. These messages exist until the client extracts them or they expire. The messages only have one consumer and the client can consume the message when it wants to, even if the client is currently down when the sender sends the message. In PTP the receiver also sends an acknowledgment that the message was successfully processed. [5] See figure 3 for an illustration of PTP.
2.2.2 Publish-and-subscribe

In publish/subscribe messaging, the messages can have multiple consumers. Instead of using queues, the clients address their messages to a topic which many other consumers can subscribe to. [5] See figure 4 for an illustration of publish-and-subscribe.

2.2.3 JMS Message

Messages in JMS consists of three parts: header, properties and body. The only required part of the message is the header. The header contains values that are used to identify and route the messages. The properties of a message makes it possible to add optional fields to the header or application specific properties or properties specific to a certain provider of JMS. [6]

2.3 OpenMQ

OpenMQ is an implementation of the JMS API that provides many enterprise features as a Message Oriented Middleware (MOM). It provides an asynchronous and loosely coupled integration between system components. The main building block of OpenMQ is message brokers that manages the messages between producers and consumers. [7] See figure 5 for a diagram showing the major components of OpenMQ.
2.4 **STOMP**

Simple Text Oriented Messaging Protocol (STOMP) makes it possible for any message broker that supports STOMP, to communicate without being dependent on language, platform or broker implementation. It's easy to implement a client using this protocol since its design is similar to HTTP, which makes it useful for creating client applications in other languages. [8]

2.5 **UMS**

Universal Message Service (UMS) enables OpenMQ to be accessed using HTTP. This way it's possible to communicate with OpenMQ using the same reliability of JMS messaging, by using any language. It makes use of the UMS API which transforms the communication into JMS messages. This makes it possible to communicate with OpenMQ using any client written in any language. [9]

2.6 **NodeJS**

NodeJS is a JavaScript run time designed to build scalable and asynchronous network applications. It's based on the V8 JavaScript engine used by Chrome. The model NodeJS uses is asynchronous I/O that doesn't block. [10]. NodeJS is used for both enterprise and high-performance applications. JavaScript together with NodeJS is seen as a universal tool used for both front end and back-end. [11]
2.7 **NPM**

NPM is a package system in NodeJS that contains reusable code that developers can share with other developers. One of the big advantages of using packages is the ease of using already tested and well-defined modules for solving tasks, instead of reinventing the wheel. It's easy to update and maintain the dependencies, by using a special JavaScript Object Notation (JSON) file called `package.json`. [12]

2.7.1 **Net**

A built-in module in NodeJS that provides the developer with asynchronous functions for creating servers and clients. This module can for example be used to create Transmission Control Protocol (TCP) socket connections. [13]

2.7.2 **Express**

Express is a NodeJS module that provides a set of functionality for web applications, making it easy to quickly create APIs. [14] It can for example create a so called app object which has methods for routing HTTP requests, making it easy to handle HTTP requests such as GET, PUT, POST. [15]

2.7.3 **Child process**

A built-in module in NodeJS providing the possibility of spawning child processes which can be communicated with using regular process communication. The child process is started asynchronously, so the process can run without blocking the event loop in NodeJS. [16]

2.7.4 **Body-parser**

NodeJS Module that helps extracting the body of HTTP requests. This makes it easier to handle POST requests. [17]

2.8 **JSON**

JSON stands for JavaScript Object Notation and is a text format for structuring data. The design of JSON is intended to be simple and minimal for both reading and parsing and works well against JavaScript. [18] In JSON objects can be represented as key/value pairs using curly brackets, lists can be represented as arrays using brackets, regular values can contain the regular types such as string, number, object, array, true, false and null. [19]

2.9 **REST**

REST[20] which is known as Representation State Transfer, is a set of architectural principles often said to be fulfilled when the following four basic principles of design are used:
• HTTP methods - HTTP methods such as GET, PUT, POST are used and should adhere to the RFC 2616 where GET is intended to fetch resources, while for example POST is used to create resources on the server.

• Uniform Resource Identifier (URI) - Resources on a RESTful API are described using URIs. These URIs should be intuitive and easy to understand. A way to achieve this is to see URIs as a hierarchy, like a directory structure.

• Statelessness - Each request from the client to the server must contain all the necessary data to understand the request. Nothing on the server should be saved and used in order to understand the request.

• Data format – Using common MIME-types for minimizing coupling between clients and the RESTful service. This lets users decide if they want to use JSON, XML, etc...

2.10 Angular

Angular[21] is a framework of several libraries, used mainly for building client applications in HTML and languages like JavaScript or TypeScript. The idea is to create templates in HTML with a special markup that Angular uses and having components manage these templates. The logic is abstracted away into something called services. Modules in Angular, is used to abstract components and services into one part.

An Angular application consists of the following main building blocks:

• Modules

• Components

• Templates

• Metadata

• Data binding

• Directives

• Services

• Dependency injection

See figure 6 for an illustration of these building blocks.
2.11 Typescript

Typescript[22] is designed for maintaining larger JavaScript programs. JavaScript is a subset of Typescript, which means that regular JavaScript and ES2015 syntax works in a TypeScript program. In addition to this, Typescript also provides things like type annotations, compile time checking and more ways of expressing object oriented concepts which makes the code more readable.

2.12 Dependency Injection

Dependency injection is a pattern of design, where the idea is to make the code of a certain module in the application, independent of the code it uses to communicate with other modules. A common reason for using dependency injection is that it makes testing easier, since the internal parts of a certain module of a program, doesn't need to know the details of another module it depends on. [23]

2.13 Promise

A Promise[24] is a JavaScript object that represents a result that may or may not contain the result of an asynchronous operation in the future. The intended way of interacting with promises is by using its then method, which will provide a callback that receives the eventual value of the promise, or an error message if it fails. A promise can be in the following states:

- Pending – not fulfilled yet
- Fulfilled – operation successful
- Rejected – operation failed
3 Methodology

A description of the tools, design choices and methods used in order to achieve the intended goals from 1.4 will be presented here.

3.1 Research

Before designing and implementing the solution, researching possible methods to achieve the desired result was performed. Researching how to create a client to communicate with OpenMQ using another programming language than Java was necessary, since JMS is an API for Java only, while the proof-of-concept is intended to be written mainly in JavaScript. This research helped clarifying that the two only viable options was to either reverse engineer the proprietary protocol used by OpenMQ and remake this in JavaScript, or use a Java client as a bridge between the JavaScript client application and the JMS broker.

Since JMS doesn't provide a wire protocol and the communication protocol OpenMQ uses is proprietary, the decision was to make a Java client that acts as a bridge between the NodeJS API server and the JMS broker. [25] Using a protocol like STOMP or UMS which later versions of OpenMQ has support for would be ideal, but this was not possible to do with the broker used in this case since its version is 4.1 and UMS and STOMP was introduced in version 4.3. [26]

3.1.1 Multi-threading and Asynchronous programming

On a multi-threaded multicore system, where multiple processes are running, true parallelism can occur since the processes can simultaneously run on each different core at the same time. This in contrast to for example NodeJS which runs in only a single process and thus can't run in parallel. NodeJS can still provide I/O in parallel by using asynchronous I/O, by returning the control to the caller as soon as the data is ready to be used. This is done by using callbacks in JavaScript. When using asynchronous I/O, the application won't get blocked when waiting for I/O operations. [27]

The advantage of using event-driven programming like in NodeJS, is that it's scalable and efficient in terms of control when developing, since events can easily be handled while multi-threading can be more complex. Problems that are hard to find, like deadlocks and shared resources is also a common problem in multi-threading. [27]

3.2 Tools

The choice of tools for achieving the result is partly based on my own experience and what Dewire prefers to use.
3.2.1 Programming languages
When creating the intermediary bridge between JMS and the API, Java will be used and is responsible for communication between JMS and the NodeJS API by creating threads listening for TCP socket connections.

JavaScript will be used as the language for implementing the RESTful API in NodeJS. While Typescript will be used for Angular 2 for the front-end.

3.2.2 Editors and libraries
When developing the Java bridge of the proof-of-concept, NetBeans will be used as an Integrated Development Environment (IDE). With the help of NetBeans, the application is also bundled into a JAR file.

When developing the NodeJS API, the following modules have been used:

- Express
- Net
- Child_process
- Body-parser

Express is used for easily implementing a RESTful API by providing routes using HTTP. Net is used for establishing TCP sockets against the Java bridge. Child_process is used to spawn processes, which is used to spawn a JAR file. Body-parser is used to be able to use HTTP POST inside Express.

Angular is the framework Dewire preferred to use when creating the front-end. In this case, Angular 2 will be used together with Typescript. The reason for using Typescript is because Angular-CLI uses Typescript by default and is recommended generally by the community since most sources online about Angular 2 uses Typescript.

3.2.3 Illustration tools
The diagram software yEd is used for making the illustrations.

3.3 Integration testing
Integration tests are performed in order to test that a certain message sent to a certain destination on the broker, is able to be received at a specified destination. In order to provide this possibility, the ability to send and receive messages to the API, which in turns redirects these messages to the Java bridge and JMS broker is provided.
3.3.1 Sending messages

Sending a message can be done both to a message queue and a topic in JMS. Therefore the capability of specifying if the message is supposed to be sent to a queue or topic is necessary.

Without some kind of message ID, the message consumer may consume messages unrelated to the messages already sent, since JMS doesn't care about the order of the messages. This is not a problem when using topics which contains messages that can be consumed by many different clients. Messages in queues however can only be received once by one user and disappear after being consumed. A way to come around this problem is by attaching a message ID to each message and storing these messages in the API. The user of the API is responsible for generating the message ID, since message IDs main purpose is to track the messages for testing. All of the necessary data when sending a message to the API is sent as attributes using JSON.

3.3.2 Receiving messages

In order to test if each sent message to the API actually arrives at the intended destination, it must be possible to receive messages from a queue or a topic. In this case, it should be able to check if a message with a specified ID is available, if not, it will close the connection. If the message is available, it will return the message and then close the connection. This way it's possible to make an automatic check that regularly checks the API for a certain message ID and when a response is available it will return that response and the user will know that the message sent is received successfully.

The API will provide a local data structure containing the received messages already that's been sent to the broker. Therefore the receive endpoint on the API will simply fetch messages from this local data structure instead of fetching from the broker.

3.4 Front-end

The front-end will be a web page created using Angular 2 and Bootstrap which provides a user friendly interface for using the API and minimizes the amount of work a system tester needs to do. The following functions will be present in the front-end:

- Configuration of brokers
- Configuration of queues/topics
- Configuration of messages in XML-format.
- Automatically generate random input for certain fields
JMS Communication – Communicating with a legacy JMS broker using JavaScript
Patrick Bylund

- Save/send messages
- Receive messages

In order to make the build process easier, Angular-CLI is used to setup the directory structure, install all necessary packages and provide start modules and components. This way, most necessary tools are added for a typical Angular project and it's possible to start developing directly against the Angular application. The only necessary addition was Bootstrap, in which version 3 was used. The reason for using version 3 is because, since version 4 of Bootstrap, glyph icons were removed [28]. The standard version of Bootstrap NPM uses is currently an alpha version of Bootstrap, so manually installing Bootstrap 3 by using "npm install bootstrap@3" was necessary.

Developing the front-end module by module is the intended method of achieving a front-end. This way, each different part of the front-end, for example broker list, queue/topic list and message configuration part, can be focused on without worrying about the details of other parts of the application. When all these different modules are finished, they can finally be put together into the finished product. It's easier to test this way and a lot easier to focus on each relevant part separately.
4 Implementation

This section will describe how each of the different building blocks of the proof-of-concept is implemented. The proof-of-concept is built around three main building blocks:

- A RESTful API that contains the necessary endpoints for communication with the Java bridge
- A Java bridge which is a Java application acting as a mediator between JMS and the RESTful API.
- A front-end which will be the actual user interface for the system testers using the system

The API is the users way of communicating with the Java bridge, which in turn communicates with the JMS broker. The front-end is created to make sending messages easier by configuring commonly used data. Figure 7 illustrates this. More details about each building block in chapter 4.3, 4.4 and 4.5.

Figure 7: Overview of how the main building blocks communicate

4.1 Architecture

In order to get an understanding and overview of how the architecture of the proof-of-concept looks like, figure 8 shows the underlying concept of how each part is intended to interact.
4.2 Design considerations

When receiving messages from JMS using the receive method defined in the MessageConsumer interface, the call synchronously blocks execution until a message is consumed. The intended functionality of this proof-of-concept however is to provide a receive method that checks if a message is available and then closes the connection regardless of the result. A timeout or an alternative method like an instant receive function is necessary to achieve this. The problem with timeout on receive, is that a too small timeout value will always return null. Determining a timeout value according to the load of the queue or topic and broker itself would be required in a case like that.

Another problem is that the JMS receive function will consume all messages without considering who the message belongs to, if the message belongs to a queue. The proposed solution to these problems is to make the NodeJS API asynchronously poll the intended response queue/topic until it gets the right message, after sending a message. This asynchronous polling function will be invoked every time the send endpoint is used. This received message will then be stored locally inside the API if it succeeds in fetching the message. When the user then wants to receive a message from the API, the API will check this locally stored data structure instead. This will make it possible to receive messages instantly without blocking and it will also make it possible to pick out certain message IDs. Figure 9 shows an overview of this process.
This process is mainly used for integration testing, because the message is expected to be first sent to a queue/topic on the JMS broker, where the receiver when receiving the message sends a response to another queue/topic (the response queue/topic). This response message from the receiver of the sent message is what the API listens for after sending the message.

### 4.3 Application Programming Interface

The API is implemented as a RESTful API in NodeJS, with the help of the NPM library Express. This application will act as the API between the client/front-end and the Java bridge/JMS. The client should be able to send messages contained in JSON represented data to this API and also receive messages. Since the API is based on REST, the users communicate with the API by using HTTP and URIs. The API is implemented in such a way that one endpoint is used for sending messages to a broker and one endpoint is for receiving messages from a broker. These endpoints are described in more detail in chapter 4.3.2 and 4.3.3.

The communication between the front-end web application and the API is also done by using endpoints, but these endpoints are for fetching configuration data and doesn't communicate with any broker. These endpoints are described more in detail chapter 4.3.4.

Figure 10 shows an overview of all the endpoints on the API. These endpoints are abstracted away into their own modules on the server, in order to give each of the different modules their own purpose. All these files containing their respective routes, are placed in the “/routes” folder in the API root directory.
4.3.1 Communication

In order to use the JAR file bundled from the Java bridge, the NodeJS module named 'child_process' is used which in this implementation automatically spawns the Java bridge when the API is started. What this module does is the same as starting a Java application from the command line, by specifying the "-jar" parameter and JAR file to start. Since the API spawns this JAR file automatically when started, no interaction with Java is needed.

The communication between the Java bridge and the API, is done using TCP sockets asynchronously, using the NPM library Net. These TCP sockets can be easily handled by using events, these common events can be bound to a callback. The events used in this implementation are:

- 'connect' – invoked when TCP socket is established
- 'data' – invoked whenever data is received from TCP socket
- 'close' – invoked when TCP socket is closed
- 'error' – invoked when an error occurs with socket

Whenever the API sends data to the Java bridge, in order for the Java bridge to understand what action to make, an attribute called “action” is added. This attribute is in the Java bridge handled with a conditional statement in order to determine the relevant action. The format of the complete JSON sent to the Java bridge is:
JMS Communication – Communicating with a legacy JMS broker using JavaScript
Patrick Bylund

```json
{
  action : "SEND",
  data : dataObjToSend
}
```

Where “SEND” means sending a message, which defines what action to take, using the data specified. The data contains a JSON object containing all the data to use.

### 4.3.2 Sending messages

Sending messages is done using the HTTP verb POST on an endpoint called “/send”. What this endpoint expects is a JSON object. See figure 11 for overview of this sending format. A specification of this sending format is:

- BrokerUri – URI to broker
- targetQueue – Name of the queue to send to
- targetTopic – Name of the topic to sent to
- responseQueue – Queue to send response to
- responseTopic – Topic to send response to
- id – Unique ID generated by client in order to track message
- data – The message to send

Either targetQueue or targetTopic must be set in order for the Java bridge to be able to decide if it should send the message to a queue or topic. Depending on if targetQueue is set as an attribute in the received JSON or if targetTopic is set as an attribute, this is easily checked. ResponseQueue and responseTopic are both optional, but if none are set, a temporary queue will be used with a random name.

If a mandatory attribute is not set when sending the JSON, the API endpoint will send an error object containing an HTTP status code 400 and the following attributes:

```json
{error : 1, msg : errorMessage}
```

If sending the data was successful, a HTTP status code of 200 is sent and an asynchronous polling function using Net is invoked before closing the socket.
connection. This polling is invoked in order for the sent message to be able to start being received by the API for temporary storage, so the user later can receive something from the API. This asynchronous polling function polls the response queue or response topic for the specified message ID sent, until it receives the data. When the data is received, it will store this data into a map locally and is stored there until someone receives the data. In that sense, this map acts as an intermediary data structure until each message in this map is consumed.

![Figure 11: Format of the JSON data sent from API to Java Bridge](image)

### 4.3.3 Receiving messages

Receiving messages can be performed in two ways. Either by calling the receive endpoint by specifying the ID as a parameter:

```
/receive/:id
```

Or by specifying all attributes like query variables:

```
/receive/?host={host}&port={port}&queue={queue}&topic={topic}&id={id}
```

The first one is used when making integration tests. What this does is check if the local map data structure on the API contains a message with the specified ID, if it contains the specified ID it will return that message. This endpoint doesn't connect to any JMS broker, it's only used to collect a message that's already gone through the whole integration process from sender to receiver to response destination.

The second type of the receive endpoint using query variables, polls a specified queue or topic on a specified broker URI. This can be used to fetch messages without caring about message ID.
When the API wants to fetch a message from the Java Bridge, the same type of format for the JSON is used as when sending messages like figure 11. The difference is that instead of “SEND” used as action, it uses “FETCH” instead. Also, responseQueue and responseTopic are unused attributes, since it only wants to receive messages. Instead, targetQueue or targetTopic is used as the queue or topic to receive messages from.

4.3.4 Configuration

The configuration route files are made for storing, deleting and managing data about brokers, queues/topics and messages. These endpoints on the API are used by the front-end in order to make the testing process easier when making integration tests. There are 3 different files for this purpose, which makes it possible to configure the following things:

- Brokers
- Destinations (queue/topic)
- Messages

By managing these, the tester can save data about brokers, destinations and messages commonly sent and reuse them. This configuration data is stored as a list of JSON objects in JSON files (one for each type of configuration) in the API server directory. Figure 12 illustrates how the different formats of each object looks like.

<table>
<thead>
<tr>
<th>Broker</th>
<th>Destination</th>
<th>Messages</th>
</tr>
</thead>
<tbody>
<tr>
<td>URI : string</td>
<td>name : string</td>
<td>name : string</td>
</tr>
<tr>
<td>description : string</td>
<td>type : string</td>
<td>data : string</td>
</tr>
</tbody>
</table>

Figure 12: Format of the different configuration data objects

4.4 Java Bridge

The Java bridge is a small Java program that acts as an intermediary between the JMS broker and the API. This program will use the native JMS code to communicate with the broker, which is not else possible in for example JavaScript. Communication between this bridge and the API will be performed using TCP sockets. For every socket connection, a separate thread is created on the Java bridge which takes care of the communication between JMS and the API. These threads are handled by a thread pool in order to provide a limit to the amount of threads running at the same time, so the memory consumption can be managed. In order to provide a smaller footprint, the communication
with JMS occurs without deploying an application server. The Java bridge is compiled into a JAR file which the API uses independently.

The Java Bridge is split into different classes for each purpose, the communication class with JMS, the server socket class for communicating with the API and the client socket class that manages a connection to the API as a thread. Details about these different functions are described in chapter 4.4.1 and 4.4.2. Figure 13 shows how the different parts of the Java Bridge interacts.

![Diagram of Java Bridge building blocks and its communication](image)

**Figure 13: Java Bridge building blocks and its communication**

### 4.4.1 Communication with Java Message Service

The main responsibility of the Java Bridge is to provide communication with the JMS broker, in this case OpenMQ. This implementation provides the minimum functionality for making it possible to send and receive to a topic and queue on that particular broker.

Here is a list of the methods implemented in the Java Bridge for communicating with the JMS broker:

- `sendQueueMessage(String queueName, String responseQueue, String messageToSend, String URI)`
- `receiveQueueMessage(String queueName, String URI)`
- `sendTopicMessage(String topicName, String messageToSend, String ip)`
- `receiveTopicMessage(String topicName)`

These methods are invoked by a client thread, depending on the action received by the API.
4.4.2 Communication with the API

The communicating with the API is provided at separate threads. When the Java Bridge is started, a separate thread for the server socket is started, which constantly polls for client connections. Whenever a client connects, it is separated into a thread for handling that particular client socket connection. This client thread is stored into a thread pool in the server class, for limiting the amount of threads started at the same time. The Java class Executors is used for this thread pool, by invoking `newFixedThreadPool` to create the thread pool using a fixed size.

The client thread is where all parsing of received data occurs from the API and where the Java Bridge decides what action to take depending on the received data. The data expected from the client is a JSON object, which is parsed by using the Java class `JSONObject`.

The first attribute the client expects from the received JSON object is the 'action' attribute. This attribute determines what action to make against the JMS broker. In this implementation, the following actions are used:

- “SEND”, send data to a queue or topic
- “FETCH”, fetch data from a queue or topic

These values are checked with a simple conditional statement. Depending on if the action is to send data or receive data, another attribute is used to determine what type of destination it should perform this action against. The 'data' attribute which is another JSON object, contains the attributes necessary to determine if the Java Bridge should send/fetch the message against a queue or a topic. If the 'data' object has the attribute `targetQueue` set, it's supposed to use a queue and if the attribute `targetTopic` is set, it's supposed to use a topic.

When it has finally determined what action to make and against what type of destination, it will invoke the correct method from the JMS communication class. When this method is done, the client socket will send a response with the data to the API.

4.5 Web Application

The web application is implemented using Angular 2 and Bootstrap. Each different requirement on the front-end application is separated into an Angular module together with the necessary components and services, so that each different module of the front-end works separately. A description of the different modules, components and services are provided here. Figure 14 illustrates the different building blocks of the web application.
4.5.1 Components

Both the broker list, destination list and message list uses one component for showing the whole list of data. In order to provide the opportunity to show/hide the component, a member variable for the component class named 'show' is added. This variable is used together with *ngIf on a <div> tag, in order for the user to be able to toggle the visibility on the list in question. Above this div tag, a regular HTML link is added which points to a function toggleVisible() which simply toggles this variable, making the ngIf condition change.

The list itself is iterated in the HTML template by using *ngFor from Angular. This is used on a list-group-item from Bootstrap. This way it creates a list item for each item the component currently holds.

In order for the user to know what item in the list is selected, a member variable keeps track of this. A [ngStyle] condition is used on each list item in order for the component to show different background styles depending on the member variable keeping track of selected state.

The MessageComponent also contains a component called MessageViewer-Component. The purpose of this component is to show a message preview of the currently selected message.
4.5.2 Data handling

In order for the components to get data from the API, a service class is dependency injected into each component. Each service class takes care of communicating with the API, in order to fetch the configuration data. The service class BrokerService fetches broker configuration data, DestinationService fetches queue/topic configuration data and MessageService fetches saved messages. In addition to this, the service class IntegrationTestService is used to communicate with the /send endpoint on the API, in order to send a message to the JMS broker.

Each service class used for configuration contains the following methods for handling data against the API:

- Get list of data
- Add new item to list
- Delete item from list

These methods use the built-in Http library from Angular in order to communicate with the API. Each method in the service calling the API returns a promise. This promise is then handled in the component, whenever the service is used.
5 Results

The results will present the web application of the implemented proof-of-concept application and an integration test. The front-end in 5.1 will show how administrating is performed and how the user interface is designed. In 5.2, a presentation of an integration test is performed, showing that it's possible to send a certain message and receive it using the same system.

5.1 Front-end

The front-end is split into different components with different purposes for the sake of making it more modularized, which makes it easier to focus on each task separately. The different components together fulfills all of the intended goals from the introduction, which is to make a front-end with the possibility of configuring brokers, queues/topics and messages. Details about these components are described in chapter 5.1.2, 5.1.3 and 5.1.4. The front-end also fulfills the goal of making it possible to specify a response queue or response topic, and send messages.

All different components have the possibility of being hidden on the view, using a button. See figure 15 for an example of how this button looks like.

Figure 15: Button to show and hide the broker configuration component

5.1.1 Status bar

The status bar is located at the top of the web application at all times. This component contains the title of the web application and the current status of what is selected from the configurable brokers, destinations and messages. It will show “Not defined!” for all parts that aren't picked yet in red text, while it shows a green text with the name of the configured broker, destination or message if a selection has been made. Figure 16 shows how this status bar looks like when a message is selected, but not a broker and a destination.

Figure 16: Status bar showing which broker, destinations and message that is currently selected

5.1.2 Configurable Brokers

The configurable brokers are shown in a list format, where the URI of the broker is shown as a link in blue color, while a description is shown together with
the URI. Clicking one of the brokers will select this particular broker. Adding new brokers is done by providing an URI and a description in a form. See figure 17 to see the design of the broker list.

![Broker Configuration](image)

**Figure 17**: List of configurable brokers

Each broker item in the list also have a button to the right which can be clicked to remove the broker from the list.

### 5.1.3 Configurable Destinations

The list of configurable destinations are shown using the name of the queue/topic. It's possible to specify using a select box when adding a new destination, if the destination is supposed to be a queue or a topic. See figure 18.

![Destination Configuration](image)

**Figure 18**: List of configurable destinations

What type of destination it is, is shown to the right, together with a delete button as well. This makes it possible to see what kind of destination it is, which is not possible by checking only the name. See figure 19.
Figure 19: Type of the destinations and delete button is shown to the right of each item

5.1.4 Saved Messages

The list of saved messages are saved using a name of the message and the message data. See figure 20.

Figure 20: List of saved messages, with the first message selected

Whenever a message is selected, a preview with the message is shown. If the message is in XML, it is formatted in a way such that it's more readable. See figure 21.
5.2 Integration test

The necessary steps are presented here, in order to make an integration test on the implemented proof-of-concept.

The first necessary step, is to start the server, which automatically spawns the Java Bridge as a child process. The server is by default accessed at localhost:3000 using HTTP. When the server is started, the first step is to send a message in JSON format, consisting of the following attributes:

- brokerUri
- targetQueue or targetTopic
- responseQueue or responseTopic
- id
- data

This is done by using HTTP POST to the server endpoint localhost:3000/send. This will also (inside the server) automatically start receiving the message and poll for it until it's available to consume, which means that after sending the
message, it's possible to regularly check the server endpoint `localhost:3000/receive`, in order to consume the message. This endpoint expects a message ID, in order to consume the correct message.

The front-end is designed to make the integration testing process automatic, by first allowing the user to specify what configurations in the UI to use as described in 5.1. These configurations are:

- Broker
- Destination
- Message

After selecting a broker, destination and message to send, choosing the section “Integration test” and inputting a “response destination” is the next step. The response destination can either be a queue or a topic and is not a mandatory attribute, because leaving this out will use a temporary response queue. When clicking the button “Send message”, the message is sent. See figure 22.

![Integration Test](https://example.com/figure22.png)

**Figure 22: How it looks like after sending a message using the UI**

After sending the message, it will start the process of making regular calls to the API, checking for the message if it's available in the API for consumption. JavaScript will, by regularly checking in a certain interval, check the API until the sent message ID is available to consume. The web application will fetch and show this message automatically in the User Interface (UI) when it's available on the API and fetched and ready to use in the web application. See figure 23.
Getting a successful result using this process and making sure an integration test is successful, is thus proven possible by checking the attached message ID on the received message and making sure that the ID corresponds to the message ID generated when sending the message. This is related to the intended goals from the introduction, which was to be able to communicate with OpenMQ using JavaScript which is possible since the web application and API is able to communicate with the broker successfully. Also the intended goal that it should be possible to make integration tests is fulfilled.
6 Conclusions

The conclusion will be a reflection of the goals and improvements that can be made in the form of future work. It will also give a brief view of the ethical aspects.

6.1 Goals and result

The main goal of this proof-of-concept was to provide a way of communicating with a legacy broker using JavaScript instead of Java. This goal is fulfilled since it's possible to perform a successful integration test against a broker. An integration test performs both a send operation and receive operation against the broker. In addition, a successful integration test also shows that the spawned Java Bridge which acts as an intermediary between JMS and the NodeJS server works as intended according to one of the goals from the introduction.

Another purpose of the integration tests is that the system testers at Dewire should be able to easily configure and use brokers, destinations and messages and send these and automatically get a response. The possibility of doing this from a web user interface as described in the goals fulfills this purpose.

The error handling between the NodeJS server and front-end is provided as HTTP status codes and an error message. These errors can easily be handled by the client whenever a request is done to the server and the incorrect status code is received. So the implementation as it is leaves most of the error handling to the front-end and if they are not handled they will be silently ignored. This is also how it is supposed to be, since the NodeJS server acts as a RESTful API returning JSON objects using HTTP.

6.2 Future improvements

Extending to use many different brokers was not provided in the implementation. It would be possible to achieve this by adding the necessary JAR files for each broker implementation that needs to be implemented. The only problem would be how the bridge would know what broker to use when communicating with it.

When making the configurable parts of the front-end, I decided to use JSON text files instead of a database when storing the brokers, destinations and messages. The reason for this choice was because the data in question was only configuration data and using a database would add additional complexity on the whole implementation, for example requiring to listen on a certain port and created locally on the machine with its own error checking etc... One potential problem of using JSON files, is the case when the API is running as a server machine accessed by many users at the same time. In case many users save data
at the same time, two or more users may write to the intended file at the same time, which would reject some of the changes.

6.3 Ethical aspects

The proof-of-concept sends all data in plain text without any form of encryption or security, so in case a user sends sensitive data or expects a sensitive response, this would not be safe if it runs on an external server. Another aspect of this, is that the server doesn't provide any form of authentication, which means that if the server runs publicly, anyone can access it. In case the configurable test data contains saved messages, destinations or brokers that are considered confidential, this might be a relevant ethical consideration. In this thesis however, this remains unknown and it's not easy to give any concrete ethical aspects on this particular problem.
References


2.4 Two Messaging Styles

2.3 Administration

2.5 JMS Interfaces


2.5 JMS Interfaces


JMS Communication – Communicating with a legacy JMS broker using JavaScript
Patrick Bylund 2017-06-08

Retrieved 2017-04-01

[12] NPM “What is npm?”
https://docs.npmjs.com/getting-started/what-is-npm
Retrieved 2017-05-01

https://nodejs.org/api/net.html
Retrieved 2017-05-01

https://expressjs.com/
Retrieved 2017-05-01

[15] Express 4.x API “Application”
https://expressjs.com/en/4x/api.html
Retrieved 2017-05-01

[16] Node.js v7.10.0 Documentation “Child Process”
https://nodejs.org/api/child_process.html
Retrieved 2017-05-09

[17] NPM “body-parser”
https://www.npmjs.com/package/body-parser
Retrieved 2017-05-09

Retrieved 2017-05-03

[19] JSON.org “Introducing JSON”
http://www.json.org/
Retrieved 2017-05-03

[20] IBM “RESTful Web services: the basics”
Retrieved 2017-05-01

https://angular.io/docs/ts/latest/guide/architecture.html
Retrieved 2017-05-09

[22] TypeScript Language Specification Version 1.8
https://github.com/Microsoft/TypeScript/blob/master/doc/spec.md
Retrieved 2017-06-05
JMS Communication – Communicating with a legacy JMS broker using JavaScript
Patrick Bylund

[23] “Inversion of Control Containers and the Dependency Injection pattern”
https://martinfowler.com/articles/injection.html
Retrieved 2017-05-19

[24] Promises/A+ Specification
https://promisesaplus.com/
Retrieved 2017-05-19

“What JMS does not include”

[26] https://docs.oracle.com/cd/E26576_01/doc.312/e24948/release-notes.htm#GMRLN00002
“What New Features in Message Queue 4.3”
Retrieved 2017-05-04

[27] Node.js: Using JavaScript to Build High-Performance Network Programs

[28] Bootstrap Migration “Components”
https://v4-alpha.getbootstrap.com/migration/
Retrieved 2017-05-08