TALLINN UNIVERSITY OF TECHNOLOGY
School of Information Technologies

Erik Ehrbach 176066IDDR

Message Traffic Audit Logging between Application and Messaging Server on the Example of AS LHV Pank

Diploma thesis

Supervisors: Toomas Lepikult
PhD

Tiit Hallas
MSc

Tallinn 2021

TALLINNA TEHNIKAÜLIKOOL
Infotehnoloogia teaduskond

Erik Ehrbach 176066IDDR

Rakenduse ja sõnumiserverivahelise sõnumiliikluse revisjonlogimine AS-i LHV Pank näitel

Diplomitöö

Juhendajad: Toomas Lepikult
PhD

Tiit Hallas
Msc

Tallinn 2021

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: Erik Ehrbach

27.04.2021

// Abstract

Audit logging is a critical part of todays financial institutions and their IT-infrastructural components. The outcome of logs is used for risk management in order to understand and trust the systems that engage with data at hand. Logs give an understanding of operations done in a system if the collection of logs is done well.

The aim of this thesis was to propose a log collection solution for any traffic flow that occurs between messaging servers and applications due to the fact that these kinds of data flow operations were not standardized nor were they involved with a uniformed traffic logging operations in the scope of LHV Bank. Although the practical part of this thesis is based on LHV Bank's technical stack, the theoretical solutions are applicable to other environments as well.

The proposed traffic audit logging solutions were implemented in a number of selected applications in LHV Bank's infrastructure and the outcome is now under the maintenance of LHV's IT development teams for future improvements.

This thesis is written in English and is 37 pages long, including 5 chapters, 22 figures and 3 tables.

Annotatsioon

Rakenduse ja sõnumiserverivahelise sõnumiliikluse revisjonlogimine AS-i LHV Pank näitel

Revisjonlogimine on tänapäevaste finantsistutsioonide ning nende IT-infrastruktuuri komponentide lahutamatu osa. Logide eesmärk on manageerida riske ja mõista paremini süsteeme, mis tegutsevad erinevates andmetöötluse protsessides. Kui logimis-operatsioonid on lahendatud jätkusuutlikult ning nende tulem on adekvaatne, pidevas analüüsis ja kasutuses, siis on tagatud väga selge madalatasemeline ülevaade kõikidest süsteemis teostatud toimingutest.

Käesoleva töö eesmärk oli välja töötada ja kasutusele võtta rakenduse ja sõnumiserverivahelise sõnumiliikluse revisjonlogimise normatiiv LHV panga skoobis, kuna just selle kindla andmeliikluse mehhanismid ning tööprotseduurid polnud mainitud institutsioonis varasemalt standardiseeritud ega ühildatud ühtsete logimis-operatsioonidega. Terviklik mehhanism revisjonlogide talletamiseks tagab kõik vajalikud võimalused edasise andmetöötluse ning –liikluse analüüsiks nii infoturbe, äriprotsesside, kui ka süsteemide arenduse vaatevinklist.

Töö käigus väljapakutud revisjonlogimise lahendus sai valitud LHV panga IT-infrastruktuuri kuuluvate ärirakenduste tasemel juurutatud ning logimismehhanismid ning valminud normatiiv on nüüdseks LHV IT-arenduse osakonna kontrolli all võimalike edasiste arenduste ja paranduste näol.

Lõputöö on kirjutatud inglise keeles ning sisaldab teksti 37 leheküljel, 5 peatükki, 22 joonist, 3 tabelit.

List of abbreviations and terms

AJP	Apache JServ Protocol
API	Application Programming Interface
ActiveMq	Open source, multi-protocol, Java-based messaging server maintained by Apache Software Foundation
DLQ	Dead Letter Queue
DML	Data Manipulation Language
HTTP	Hypertext Transfer Protocol
JAR	Package file format. Stands for Java Archive. Also described as a JAR artifact.
JMS	Java Messaging Service
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
MDC	Mapped Diagnostic Context
POJO	Plain Old Java Object
UI	User interface
URL	Uniform Resource Locator

Table of contents

1 Introduction 11
1.1 Background 12
1.2 Motivation and outcome 12
2 Technical prerequisites and requirements analysis 14
2.1 Technical scope 14
2.2 Existing logging practices 15
2.2.1 Incoming HTTP request logging 15
2.2.2 Outgoing HTTP request logging 16
2.2.3 Message broker traffic local file logging 17
2.2.4 Message broker traffic database logging 18
2.3 JMS Message entity and context 18
2.4 Log analysis and monitoring 21
3 Log events 24
3.1 Event context 24
3.2 Incoming message 25
3.3 Outgoing message 27
4 Log collection and implementation 29
4.1 Log row format and configuration 30
4.2 JMS logging library 33
4.2.1 Message payload retrieval 35
4.2.2 MDC context lifecycle operations 36
4.2.3 Automated tests 38
4.3 JMS logging library for Spring framework 38
4.3.1 Logging incoming messages 38
4.3.2 Logging outgoing messages 41
5 Usability 44
5.1 Troubleshooting message broker issues 45
5.2 Analysis and monitoring 46

Conclusion 48
References 49
Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis 52
Appendix 2 – Application HTTP access log file example 53
Appendix 3 – Application HTTP client log file example 54
Appendix 4 – Database-logged message queue traffic examples 55
Appendix 5 – Shared JMS properties enum class 56
Appendix 6 – Automated tests for JMS logging library 58
Appendix 7 – Incoming message listener registry example for logging incoming messages 62
Appendix 8 – JmsMessageSender interface example 65
Appendix 9 – DefaultJmsMessageSender and OutgoingJmsMessage implementation examples 67
Appendix 10 – Log format and library usage guide for messaging server traffic log collection 69

List of figures

Figure 1. Message data on ActiveMQ administrator UI (*User Interface*) dashboard 20
Figure 2. Unconsumed messages in a destination on ActiveMQ administrator UI dashboard 20
Figure 3. Audit trail logging swimlane diagram 22
Figure 4. Application error log entry for specific correlation id 23
Figure 5. Application incoming HTTP request log entry for specific correlation id 23
Figure 6. Incoming message log event flowchart 26
Figure 7. Message broker distribution models 27
Figure 8. Example for incoming message logging pattern in Log4j2 XML configuration 31
Figure 9. Example for outgoing message logging pattern in Log4j2 XML configuration 32
Figure 10. Log4j2 appender and logger configuration example 33
Figure 11. JMS Logger class code example 34
Figure 12. JMS Logger message payload retrieval example 35
Figure 13. JMS Logger MDC population example 36
Figure 14. JMS Logger example to serialize JMS Message properties for MDC 37
Figure 15. JMS Logger MDC clearing example 38
Figure 16. Spring boot's auto configuration example for JMS listeners 39
Figure 17. Outgoing JMS message interface example. 41
Figure 18. Example for outgoing JMS message interface with persistence field support. 42
Figure 19. Example of *DefaultJmsMessageSender* solution of logging outgoing messages. 42
Figure 20. Application error log example on JMS listener thread 45
Figure 21. Lightweight example of incoming JMS log message 46
Figure 22. Lightweight example of retried incoming JMS message in logs 46

List of tables

Table 1. JMS Message interface sub-interfaces 18
Table 2. Log row representation of an outgoing message 44
Table 3. Log row representation of an incoming message 44

1 Introduction

Banking systems today should rely on logging for risk management. Log collection is heavily recommended by the Estonian Financial Supervisory Authority and as per the official recommendations, it is necessary to have a log of actions performed in an information system [1] [2, p. 17]. Thus, if some system triggers a process where client's account balance is debited by some other system, there should be a corresponding audit trail in place which gives relevant answers to how and why the process happened and what were the end results. This kind of log collection should be set in place regardless of process' methodology, choice of technology or synchronicity.

Regarding synchronicity, practicing asynchronous data flow methodologies represents a useful way to keep resources available for other synchronous processes in environments which demand high availability and fast processing time. In LHV Bank, asynchronous data transmission and business use-case process management between two different applications is mostly achieved by using message queues. A queue's mechanism is to store a message sent by the message producer and publish the message for the message's consumer [3]. A key aspect to understand and monitor this kind of data flow, is logging [4, p. 24].

Message queue traffic logging and monitoring can be done in multiple ways. One can log all the messages to a database table, a local log file or configure the logging mechanism to log messages to a centralized log repository. In the example of LHV Bank, some queue message logging mechanisms rely on database tables while some applications propagate log contents to a local log file in their own preferred format.

In order to support future scalability of applications in virtualized environments and retain relevant audit event tracking for security monitoring and auditing purposes, the author believes it would be a provident decision asserting log collection implementations to follow standardized formats and methods. This thesis will focus on an application's message queue traffic log collection methodology to a local log file in the example of

LHV bank while taking into account the technological prerequisites and future scalability plans of the institution.

1.1 Background

LHV was founded in 1999 as an investment union. In 2009 the institution obtained a banking license and business name AS LHV Bank was adopted. There are over 550 people working for the bank and there are more than 247 000 clients that use LHV's banking services [5].

At the time of writing, the author of this thesis has 8 years of experience in software engineering. Having initially started out as an e-commerce product developer in another company, the author has now worked as a software engineer in LHV Bank for 4 years. During that time, the author has gained a lot of interest and practice in domain driven architecture and development with a focus on secure and performance-oriented solutions.

From the perspective of this thesis' subject, the author has had past involvements in the engineering of asynchronous processes for resolving LHV Bank's business needs. The earlier experiences in message queue traffic implementations has given the author a clear sight of the problem at hand and ingrained the theoretical ideas for a solution.

1.2 Motivation and outcome

The motivation for this subject came from a development of a new in-house application which started to exchange data with other systems via message queue and the existing standardized methodologies relied on database tables for audit logging. For that data exchange use-case, a database table audit logging was an overkill and to disregard any potential performance-related bottlenecks while sending and consuming queue messages, a log mechanism to propagate message contents to a log file suited as a better option for audit, monitoring and debugging purposes.

Two of the key reasons omitting database table audit logging are related to usability and performance issues as file-based logging has more pros than cons compared to database logging [4, pp. 30-31]. Application log files are also centralized in LHV, hence the logging standards and methodologies are more acceptable to possible improvements from the log collection point of view.

The problem of having non-standardized and chaotic message queue traffic logging in LHV applications' infrastructure was brought to the attention of senior IT-development staff by the author. As the issue was recognized and acknowledged, it would also be resolved, developed as well as internally communicated by the author under the supervision of relevant stakeholders (development teams, system administrators and security engineers).

The outcome of this thesis is a documented specification of message queue traffic log standards, contents, formats and usage examples in LHV bank's documentation space for development teams, system administrators and other relevant stakeholders. The outcome documentation is also supplemented with relevant libraries to help development teams integrate required mechanisms in an easier manner without any irrelevant overhead. As the final produce will be achieved by using standardized and well-known components given the technical constraints, the outcome and core idea for how to achieve message queue traffic logging is not only applicable to LHV bank and the solution is easily implementable in other companies and environments[1].

---

[1] As the author has no authority nor the required amount of competence to deal with the juridical aspects and regulations for logging, the proposed solutions and a set of logged data may still be a subject to review from a legal point of view. The solutions proposed by the author are put forward from a technical point of view and the author strongly recommends all the endeavors using this thesis' outcome to be examined by a competent person or personnel who specialize in the legal field. The outcome solutions are provided „as is". In no event shall the author be liable for any claim, damages or other liability, whether in an action of contract, tort or otherwise, arising from out of or in connection with the provided outcome or the use or other dealings with it.

2 Technical prerequisites and requirements analysis

In order for any technical improvement to fit well into existing IT-infrastructure, improvements should take into account the present implementations and technical constraints. Backwards compatibility is a must in LHV's IT change management discipline because one of the goals for the institution is to be highly available with minimal downtime. Any service disruption on public channels gets reported to a public status page in real-time [6].

Considering the aforementioned, in author's opinion, one of the requirements for a new standardized message queue logging mechanism should be to support effortless way for development teams migrating to new logging mechanisms. A scalable way to achieve this goal, is to develop a common library for IT-development teams to use. A shared logging library can have its own lifecycle and boundaries to grow in. Backwards compatibility can be achieved and communicated via semantic versioning and relevant documentation [7].

2.1 Technical scope

Vast majority of LHV Bank's IT development teams use JVM-based (*Java Virtual Machine*) programming languages combined with Spring Framework for the development of business applications. It is highly recommended for the development teams to use common and standardized libraries and implement them to business applications as dependencies. The purposes of the shared JAR (*Java Archive)* artifacts can vary from shared business logic to asserting standardized infrastructural methodologies for development teams.

Regarding IT infrastructure, most in-house asynchronous messaging implementations rely on ActiveMQ messaging server (*open source, multi-protocol, Java-based messaging server maintained by Apache Software Foundation*). Communication with ActiveMQ messaging server is achieved by using JMS (*Java Messaging Service*) which provides a common way for JVM-based applications to connect with messaging systems [8].

2.2 Existing logging practices

At the time of writing, LHV Bank's business applications already follow the official recommendation of Estonian Financial Supervision Authority for collecting audit logs about event occurrence. A general Java application in LHV infrastructure has the following local log files (but not limited to):

- Incoming HTTP (*Hypertext Transfer Protocol*) logs (HTTP access logs for requests that the application serves);
- Outgoing HTTP logs (HTTP client logs for requests that the application triggers to other services);
- Java application logs;
- Java standard output logs;

From the perspective of audit trail, these local log files are also supplemented with database level audit logging on every executed DML (*Data Manipulation Language*) command. Every row contains auditable data regarding a row's creation time, modification time as well as user data responsible for the change. Applications tied to message queue traffic implementations are currently either logging messages to local log file in a non-standardized way or to a database via DML commands. These existing solutions are presented in chapters 2.2.3 and 2.2.4 respectively.

Regarding HTTP log collection, the process follows standardized methodology in LHV Bank's business applications for both incoming and outgoing requests. All incoming and outgoing HTTP requests and responses are logged by the application to a local log file in a uniform manner and format.

Due to the fact that an incoming HTTP request and an incoming queue message both serve an entry point to the application and vice-versa, in author's opinion, the current HTTP request logging methodology offers a valuable reference point for any further developments of message queue traffic logging mechanisms.

2.2.1 Incoming HTTP request logging

From an application side, an incoming HTTP request log file contains all the requests done towards the application. The log file contains the following fields:

- Log type, which has values representing a client request row (C) or server response row (S);
- Timestamp;
- Correlation id, which is propagated to any subsequent requests made during the service of the request in question (through HTTP X-Correlation-ID header [9]);
- Request id, which is used to correlate request log rows to response log rows;
- Session id, which represents a hash of client's session id;
- Domain name;
- Client IP address;
- HTTP method;
- HTTP URL (*Uniform Resource Locator*) path;
- HTTP payload, which is a payload of the request or response;
- HTTP response code;
- User id, which represent an audit information of a user id who initiated the request;

A sample for application incoming HTTP request log file can be seen in Appendix 2.

2.2.2 Outgoing HTTP request logging

From an application side, an outgoing HTTP request log file contains all requests done by the application to other services. This log file contains the following fields:

- Log type, which has values representing an application's request row (C) or target server response row (S);
- Timestamp;
- Correlation id, which is propagated to any subsequent requests made during the service of the request in question [9];
- Request id, which is used to correlate outgoing request to a current client's request;
- Session id, which represents a hash of client's session id;
- Log record id, which is used to correlate outgoing requests and responses
- Node name, which is an identifier of the current application node;
- HTTP method;
- HTTP URL;
- HTTP payload, which is a payload of the request or response;

- HTTP response code;
- User id, which represent an audit data of a user id who initiated the request;

A sample for application outgoing HTTP request log file can be seen in Appendix 3.

2.2.3 Message broker traffic local file logging

Applications in LHV IT-infrastructure that do message broker traffic logging to a local log file use varying patterns and formats for log collection. When comparing log rows across multiple applications, they only have common ground with the following log fields:

- Timestamp (occurrence time of the message being sent or received);
- Message destination (a target destination where the message was sent to or received from);
- Message payload (contents of the message);

There are several applications in LHV IT-infrastructure that practice nonstandardized log collection of message broker traffic events to a local file and they are all implementing different log formats. As for log collection requirements, these implementations will not be analyzed any further as these solutions vary to a great extent and their core idea is to provide development or debug related information and they do not log down any audit data except for the previously listed data items.

Compared to incoming and outgoing HTTP logs, the message broker traffic logging implementations tend to omit a lot of context from the log events. Correlation and request data is not logged and some applications have even thought only fit to log outgoing messages.

As it shows, the biggest problems are nonstandardization and insufficient logged data which prevent practical log analysis across multiple applications using correlation, request or user id's. On the other hand, JMS has all the support needed for the passage of request, correlation or audit related context through its message properties and these fields should be used and logged down to simplify log correlation [10] [4, p. 12].

2.2.4 Message broker traffic database logging

Database logging for message broker traffic implementations follows a similar pattern across different applications. There are two tables:

- queue_message_incoming (for incoming messages)
- queue_message_outgoing (for outgoing messages)

Storing incoming messages is done as soon as a message is consumed by the application. If there are any additional business logic processes for message consumption, the storing operation is usually done inside a shared database transaction for these processes.

Storing outgoing messages is done at two stages. First the messages is saved to a table with a flag indicating that it has not been sent yet. After that, the message is sent via JMS. If no errors occured, the database row is updated and the flag indicating send status is adjusted accordingly. Holding a state for the outgoing message status gives applications an advantage to trigger retrial processes for unsent messages.

This solution and the persisted collection of data (see Appendix 4) today satisfies some needs of audit trailing and it is working without issues. Nevertheless, at a larger scale this kind of database logging for an audit trail is not a good solution as there is a database dependency for analysis and in overall, it has too many disadvantages over file logging [4, pp. 30-31].

2.3 JMS Message entity and context

As stated in chapter 2.1, JMS is a widely used API (*Application Programming Interface*) in LHV messaging queue implementations. JMS messaging solutions use a *Message* interface as a root interface to all message types [10]. The *Message* interface and its sub-interfaces declare all the methods every JMS provider needs to implement and they are represented in Table 1.

Table 1. JMS Message interface sub-interfaces

Message sub-interface type	**Content type in Java**	**Description**
BytesMessage	`byte[]`	Content contains a stream of uninterpreted bytes

MapMessage	`java.util.Map.class`	Content contains name-value pairs in an instance of Java Map interface
ObjectMessage	`java.io.Serializable.class`	Content contains a serializable Java object
StreamMessage	`byte[]`	Content is passed as a stream of primitive types in the Java programming language. Filled and read sequentially
TextMessage	`String.class`	Content contains a Java string value

As all types could be implemented for data flow implementations, the logging mechanisms should be able to handle every single one of them unless agreed to retain any usages of some types. In the latter case, the logging mechanism should stop any traffic flow if a message log event is not loggable [4, p. 55].

JMS defines noteable API's for the *Message* interface that may present significant usage for log collection. In author's opinion, the following methods should be considered for logging use [10]:

- getJMSMessageID() which returns a unique ID of the message set by a JMS provider;
- getJMSCorrelationID() which returns a correlating ID of the message set by the producer. The value can be self-defined or used as a response identifier to a corresponding JMSMessageID for a previous message in a Message ID Pattern [11];
- getJMSDestination() which returns a target destination name for the message;
- getJMSReplyTo() which returns a target destination for any expected reply messages and can be omitted if no use-case;
- getPropertyNames() which returns all the property names;
- getStringProperty(String name) which returns a property value as a string for the specified property name;

From an ActiveMQ administrator dashboard, the values that are returned from to aforementioned API operations can be seen on Figure 1.

Headers

Header	Value
Message ID	ID:c758dcfd068c-33147-1617008891061-4:1:1:1:1
Destination	queue://LHV.QUEUE.REQ
Correlation ID	dummyJmsCorrelationId
Group	
Sequence	0
Expiration	0
Persistence	Persistent
Priority	5
Redelivered	false
Reply To	queue://LHV.QUEUE.RES
Timestamp	2021-03-29 12:09:09:057 EEST
Type	

Properties

Property	Value
X-Correlation-ID	5f9a8187-c32a-4b0d-8333-ff042bb57aca
MESSAGE_ID	dummySelfDefinedIdInProperties
USER_ID	53434
someOtherProperty	somePropertyValue

JMSMessageID
JMSDestination
JMSCorrelationID
JMSReplyTo
Property names
Property values

Message Actions

Delete
Copy
Move
-- Please select --

Message Details

{"someJsonKey":"someJsonKeyValue"}

Message contents

Figure 1. Message data on ActiveMQ administrator UI (*User Interface*) dashboard

The highlighted items in the *Headers* section are also presented in a list view where all unconsumed messages can be browsed as Figure 2 shows.

Browse LHV.QUEUE.REQ

Message ID ↑	Correlation ID	Persistence	Priority	Redelivered	Reply To	Timestamp	Type	Operations
ID:c758dcfd068c-33147-1617008891061-4:1:1:1:1	dummyJmsCorrelationId	Persistent	5	false	queue://LHV.QUEUE.RES	2021-03-29 12:09:09:057 EEST		Delete
ID:c758dcfd068c-33147-1617008891061-4:4:1:1:1	dummyJmsCorrelationId	Persistent	5	false	queue://LHV.QUEUE.RES	2021-03-29 14:23:02:875 EEST		Delete
ID:c758dcfd068c-33147-1617008891061-4:5:1:1:1	dummyJmsCorrelationId	Persistent	5	false	queue://LHV.QUEUE.RES	2021-03-29 14:23:04:274 EEST		Delete
ID:c758dcfd068c-33147-1617008891061-4:6:1:1:1	dummyJmsCorrelationId	Persistent	5	false	queue://LHV.QUEUE.RES	2021-03-29 14:23:05:026 EEST		Delete
ID:c758dcfd068c-33147-1617008891061-4:7:1:1:1	dummyJmsCorrelationId	Persistent	5	false	queue://LHV.QUEUE.RES	2021-03-29 14:23:05:469 EEST		Delete
ID:c758dcfd068c-33147-1617008891061-4:8:1:1:1	dummyJmsCorrelationId	Persistent	5	false	queue://LHV.QUEUE.RES	2021-03-29 14:23:05:880 EEST		Delete

Figure 2. Unconsumed messages in a destination on ActiveMQ administrator UI dashboard

Unconsumed messages usually occur on two different scenarions. There are either no consumers that consume messages from a JMS destination or a consumer fails to process the message and does not acknowledge it in a transacted session [12]. ActiveMQ has in-built redelivery support for the latter purpose and a redelivery configuration can be used to retry message consumption for defined times. If a consumer is still unable to process the message during retrials, the message is marked as poisonous and it gets redirected to the DLQ (*Dead Letter Queue*) which is a JMS destination like any other [13].

In LHV, every message in the DLQ is subject for manual actions. A message in DLQ either gets deleted or resent to the original destination. As these activities are done

manually, the information regarding a message's data is communicated internally between a development team and administrator team. A decision whether to delete or resend is based on the nature of the message and the current state of persisted data in a business application.

2.4 Log analysis and monitoring

As message queue traffic serves an entry point and an exit point to and from the application, the collection of logs on that part should provide a low-level understanding of any data flow occurence for audit analysis. As any entry event to an application could have been preceded by multiple other requests through other applications beforehand, an identifier should be propagated to data flow logging to support analysis via correlation [9].

For example, considering we have an HTTP request to application A, which in return communicates with application B and the latter one produces a message queue message which is consumed by application C. In that part, given that HTTP requests are synchronous and message queue traffic is asynchronous, the X-Correlation-ID HTTP header should be logged down to the following log files in the following order:

1. Application A incoming HTTP request log
2. Application A outgoing HTTP request log
3. Application B incoming HTTP request log
4. Application B outgoing message queue log
5. Application B incoming HTTP response log
6. Application A incoming HTTP response log
7. Application C incoming message queue log

Regarding synchronicity, the incoming message queue log entry in application C may occur earlier than listed but it is guaranteed to be at least after the fourth logged entry as message sending gets triggered before application B has finished serving the HTTP request made by application A as Figure 3 shows.

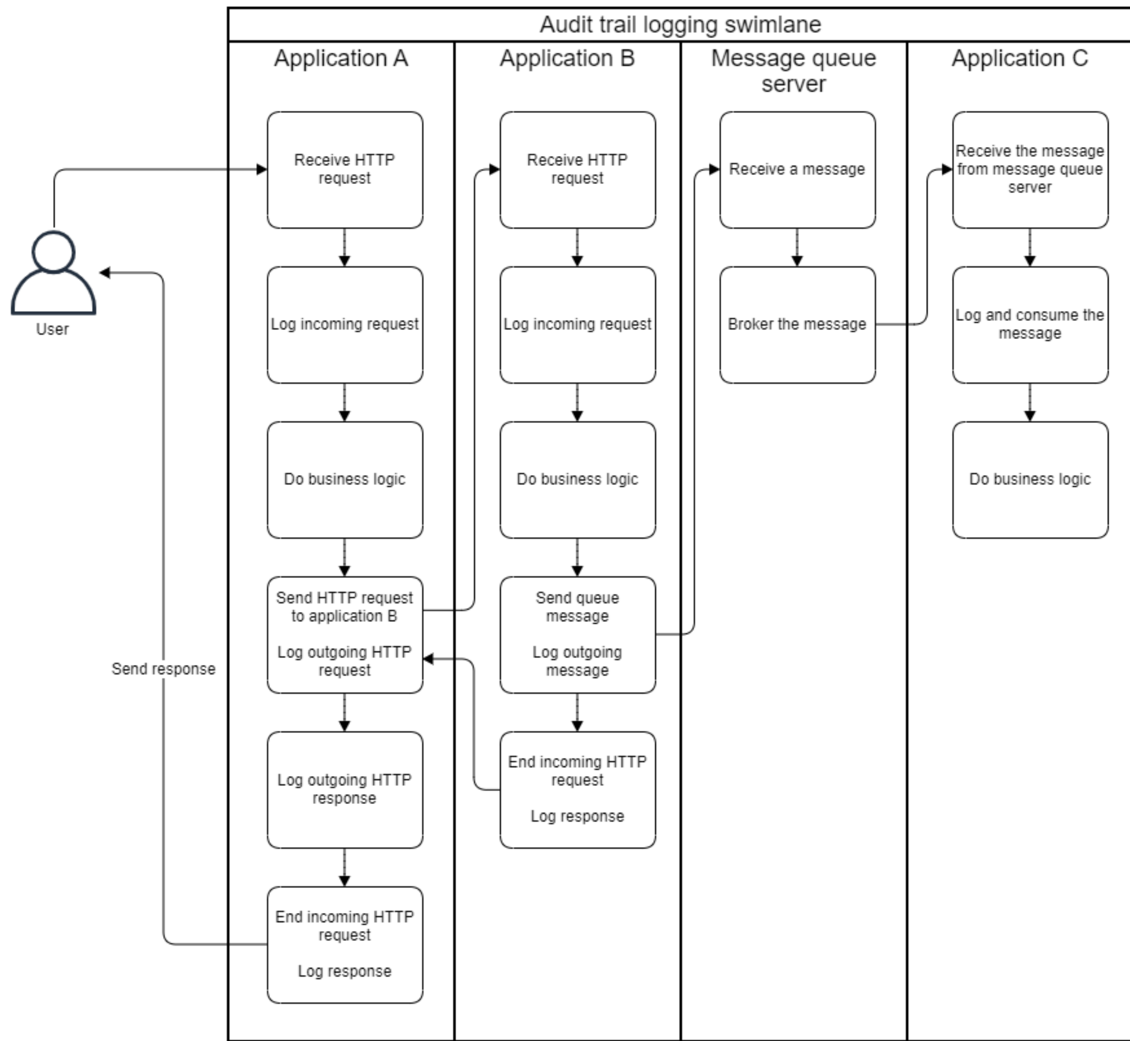

Figure 3. Audit trail logging swimlane diagram

From an application and log monitoring point of view, the X-Correlation-ID header value is a source correlation value for any process in terms of HTTP requests. Considering that business logic processes in LHV IT infrastructure do not always rely on external calls and some processes could be scheduled or running indefinitely inside an application, these operations should fill the correlation context itself in author's opinion. If all the counterparts in a chain of communication have implemented a check for correlation data and fill it, if missing, then all interapplication data flow has correlation info prefilled for all the following operations.

Filling correlation data can also be useful in application monitoring. Considering that there is a monitoring job in place which monitors an application's error logs, logging the

correlation data to every log entry helps troubleshooting as an error could be correlated with a specific HTTP request or an incoming queue message.

For example, on Figure 4 it can be seen that there is a error indicating a bad phone number in a AJP (*Apache JServ Protocol*) thread. As this thread represents a AJP protocol which is used to integrate Tomcat server to Apache installation, we can presume that this error occured on an incoming HTTP request [14].

timestamp	severity	thread	correlation_id	message
2021-04-03 19:46:26.666 +0300	ERROR	ajp-nio-127.0.0.1-2002-exec-10	sGqVacUO	Failed to parse phone number

Figure 4. Application error log entry for specific correlation id

Using the knowledge that the error occured on the service of a HTTP request, we could use the correlation id value to tie the error to a specific request as shown on Figure 5.

timestamp	correlation_id	server	http_method	http_url	request_body
2021-04-03 19:46:26.657 +0300	sGqVacUO	someapplication.lhv.ee	GET	/validate-phone?phoneNumber=ASAUSHFH@231858234	-

Figure 5. Application incoming HTTP request log entry for specific correlation id

Combining the knowledge from Figure 4 and Figure 5, it can be said that the application failed to validate a phone number because the request itself did not include a validatable phone number.

To conclude, logging down an audit trail benefits log analysis across multiple applications as well it saves development teams a lot of time troubleshooting or understanding issues that rise in a production environment. Having an ability to correlate errors, requests and messages supports all the relevant stakeholders to understand and improve their applications.

3 Log events

As stated in chapters 2.2 and 2.3, there are two types of message queue messages for an application. One is an incoming message and the other is an outgoing message. They may be very similar data-wise but from an application side, the incoming and outgoing messages have completely different meanings.

Incoming messages could be viewed as an incoming HTTP requests and since its contents are coming from an external source, in author's opinion, any message consumer must ensure that a message is trustworthy and relevant. Outgoing messages on the other hand are produced by the application itself and a message sent for other applications should be made trustworthy.

Regardless of an incoming message's turstworthyness, a good practice would be to log down any incoming traffic in order to support security monitoring [4, pp. 15, 55]. As for this, in authors opinion, the logging event for an incoming message should occur at the earliest possible point in the flow of consumption.

Outgoing messages on the other hand are the produce of some component in an application as the application is programmed to send out the message at some particular point. From log event context, this particular point in time means that the message has been finalized and is not subject to any data modifications. A lot of context may surround an outgoing queue message. It could be application triggered message by some automatic process or it could be triggered by serving an ongoing HTTP request.

3.1 Event context

As far as context and trustworthyness goes, in author's opinion the logging event should log a relevant context down to support deep-dive log analysis when issues arise, may it be from a security or development point of view. From the author's experience, this practice also retains a good level of trust in the application from a developer's aspect as the knowledge of any traffic flow can be audited and verified at any point in time and a comprehensive data on log row helps to grasp any context around it.

In order to achieve a relevant context to message queue traffic flow, a single message's log row should give answers to the following questions: who, what, where, whence and when [4, p. 25]. In author's opinion, for a message queue traffic logging event, these questions should resemble the following:

- Who – The system or user that triggered the message sending which can and should be bound with a correlation identifier;
- What – The contents of the message;
- Where – The JMS destination name from message producer side and on a consumer side it should also be supplemented with info about which host consumed the message;
- Whence – The system that the message originates from. In terms of consuming a message, a producer application identifier should be described at minimum. In terms of producing a message, the producer should reflect info about which host produced the message;
- When – A timestamp describing the time a message was consumed or produced;

To sum it up, every log event triggered whether by an incoming or an outgoing message should answer these questions. If application A's produced or consumed a message at some particular time, it should be understood who triggered the message, what were the contents, where was it destined to, whence did it occur architecturally (for example a single node in a cluster of application) and when did it occur.

3.2 Incoming message

As stated before, an incoming message is an entry point to the application. Usually, when an application retrieves a message from a message broker, it is destined to execute some logic based on the data received. Considering that, in terms of procedure flow while consuming a message, the author believes that a fundamental aspect is to firstly trigger a logging event for the incoming message.

Having a message logged down before any following procedures start the consumption, provides all the auditability, troubleshooting or analysis options opened. The consumer application could completely fail with consuming the message contents, but despite that, due to having logged down traffic before processing the contents, it would be a

straightforward action to reproduce a possible bug or unhandled data validation case for development teams as a possible error log entry could be correlated with the incoming message traffic log row.

Furthermore, if incoming message consumption events start off with traffic logging, the logging mechanism could have an ability to stop any further message consumption, if it occurs that a message is impossible to log down in a required format due to unknown errors or unhandled programming logic inside the logging operations. On that case, the traffic logger could stop the process and dump all the knowledge it has about the message to the consumer application's log as errors or warnings for further troubleshooting.

To summarize, a flowchart to represent all the aforementioned for incoming message log event is seen on Figure 6.

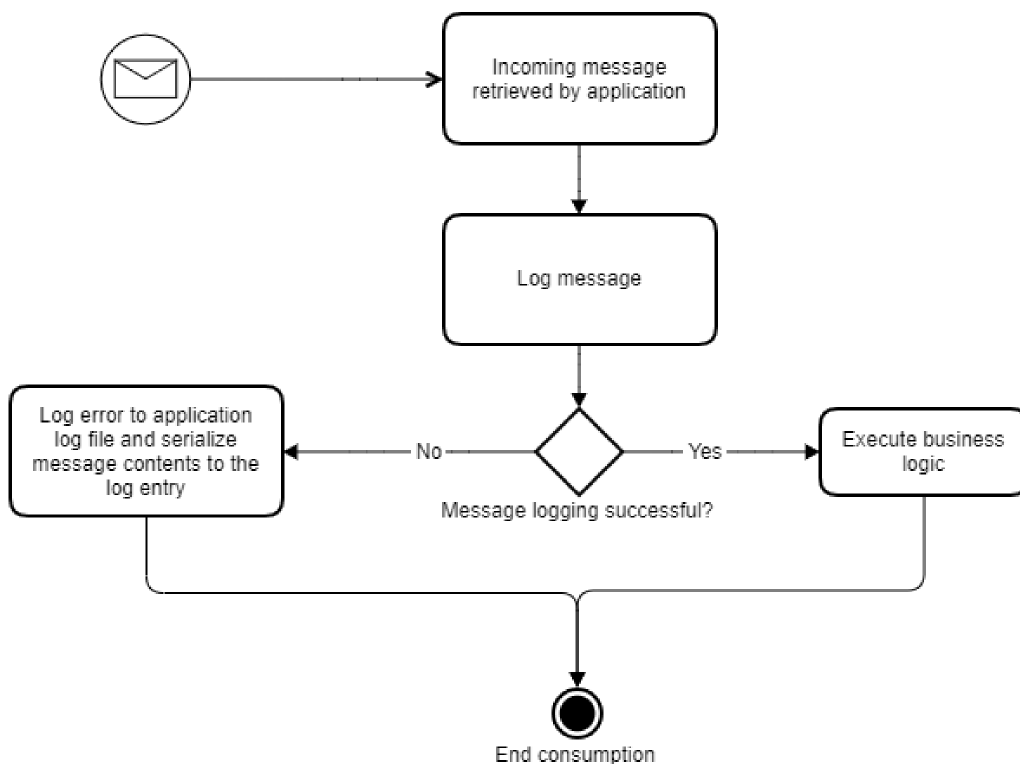

Figure 6. Incoming message log event flowchart

3.3 Outgoing message

Outgoing messages are a produce of a message producer and their sole purpose is to transport data from point to point or to all of the subscribers for a particular message destination [15]. The destinations for these distribution models also have a different meaning as point-to-point is achieved via queues and publish-and-subscribe is achieved via topics as Figure 7 shows [16].

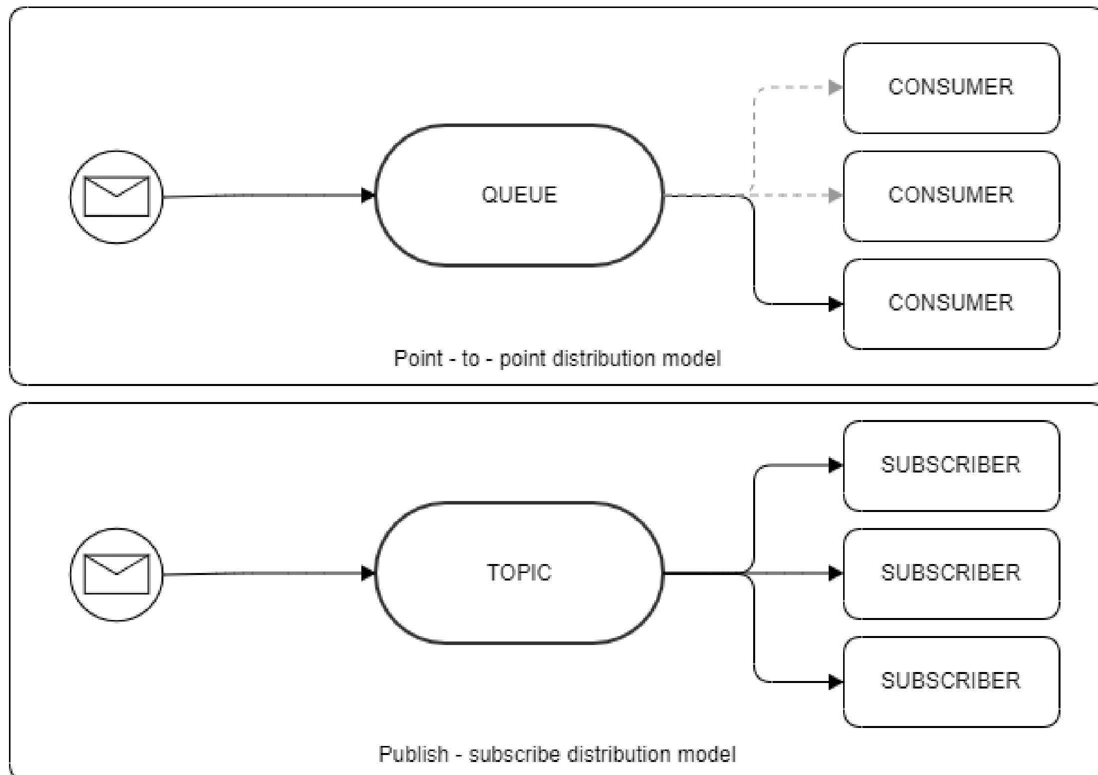

Figure 7. Message broker distribution models

From a log event point of view, the author believes that on a message producer side, the outgoing message log event should describe a distribution type for log analysis. Knowing that a message-producing application has awareness about an outgoing message's destination and the destination is also contracted to JMS *Message* interface API, a logged down distribution type may give additional value and context to the traffic analysis.

For example, knowing that a message travelled to a destination described as *queue://CLIENT.DATA.UPDATE* versus knowing that another message was sent to a destination described as *topic://CLIENT.DATA.SYNC* provides additional context to log

events. The first one is identifiable as point-to-point distribution and the latter one as publish-subscribe distribution.

In LHV, some point-to-point distributions work with request-response arrangement using correlation id pattern [11]. In short, an initial outgoing message is expected to receive an incoming response message to a physical destination in order to fulfill any business requirements. Both messages can be correlated using JMS *Message* interface's correlation id field which is initially generated and filled to the request message on the producer side.

This pattern can also be referred to as the request-reply pattern [17]. In order to not rely solely on physical destinations on this pattern, JMS provides a possibility to use temporary destinations which are created and destroyed programmatically. To support the usage of them, the API has defined a *JMSReplyTo* field for response message producers to refer to when sending a reply. Even if a producer does not create a temporary destination for a reply message and is relying on a physical destination which is predefined, the author believes that development teams should consider the usage of this field in a request-response arrangement based on physical destinations as it could represent additional valuable context to the traffic log event.

Having knowledge about a distribution model itself and also if a outgoing message expects a return could improve log analysis and monitoring. For example, if an application logs down a *JMSDestination* and *JMSReplyTo* for an outgoing message, a monitoring job could use the logged knowledge and correlate outgoing messages with incoming messages based on the destination defined on *JMSReplyTo* field and report or alert any missing responses. Without this kind of field usage, that job monitoring could not operate in a scalable way and it needs to be configured separately for each outgoing message destination and incoming message destination pair.

Ultimately, as outgoing messages are a start point for messaging flow through a message broker, the author believes that there can never be too much context surrounding the outgoing message traffic log event as far as the log collection does not put a message-producing application at risk in terms of availability. As far as author's experience goes, it is more reasonable to log more in the beginning of implementations rather than logging too little.

4 Log collection and implementation

Log collection for JVM-based applications can be achieved by using Java Logging APIs [18]. As the bundled APIs certainly can do the job, the development community tends to lean towards logging frameworks in order to get more flexibility and not to reinvent the wheel [19]. There are several logging frameworks for Java but as LHV IT development teams use Log4j2 logging framework, the log collection for message broker traffic in this thesis will focus on that logging framework.

Log4j2 framework uses MDC (*Mapped Diagnostic Context*) in order to support logging operations and fill log rows with data provided to thread context [20]. In LHV's example, HTTP log collection is heavily based on the MDC. The request or response data is filled at the earliest possible point of interception available during the service of the request or response. There are two intercepting operations – first the request data is filled to thread context and logged down as a request row and after the request is finished and a response retrieved (or produced), the same is done for response data.

As HTTP requests are logged down with interception operations, the author believes a similar logic should apply to any JMS traffic logging. As intercepting and standardized program flow could be shared via libraries as described in chapter 2.1, the interception logic could be implemented in a straightforward manner by all the development teams while eliminating any risk of inconsistent solutions.

In author's experience, the biggest downside of achieving a singular consistent solution with interception logic is inflexibility. For example, native Java APIs for doing HTTP requests do not support any interception logic and these are achieved by using frameworks or libraries [21] [22] [23]. The same applies for JMS and when dealing with Spring framework applications, this interception logic could be achieved by using the Spring JMS support and its capabilities [24]. From a development point of view, the author believes that the traffic logging operations should only depend on the native API and the logging framework in use. If done so, a development team is not dependent on a framework interception logic and could implement the logging calls in their own preffered way if not using a commonly shared application framework.

To tackle that inflexibility issue and still support a common logging mechanism, the author believes that two different implementations are needed in order to support consistency and also flexibility at the same time. The first implementation would be a logging library which includes all the operations needed to execute the audit logging on top of the native JMS API which is described in chapter 2.3 and do operations filling or using the MDC context to answer all the questions described in chapter 3.1. This implementation would do all the logging operations and it would be solely depend on the Java API and a chosen logging framework.

The second implementation would be a framework specific logging library which would depend on the previously described logging library and a commonly used framework. In the example of LHV Bank, the latter would be Spring framework. The goal of this library would be to support all the autoconfiguration, message interception and message sending while executing log collection on traffic flow.

4.1 Log row format and configuration

There are many ways to configure logging formats for Log4j2 framework [25]. In LHV, XML configuration is mostly used for shared loggers and the configuration files are packaged into a JAR artifact for Java runtime to use and refer to. Also, as stated in chapter 2.2.3, the existing messaging server traffic file logging implementations lack context information. Considering all of the criteria, limitations and described existing solutions above, the author believes the log format itself should contain the following fields:

- Host – A host’s identifier, who is producing the log. An IP address or the host’s name
- Log type – for incoming messages a value “C” to represent a client and for outgoing messages a value “S” to represent a server;
- Log datetime in ISO8601 format with timezone [4, p. 58];
- Correlation ID – unique identifier which is usually based on HTTP header X-Correlation-ID for identifying processes throughout multiple applications. If not from the HTTP header, the message producer produced a message while not in request context and generated a new correlation ID.
- Request ID – unique identifier for the current "request", random 8 character string. For consumers, it should be generated automatically when message consumer is

starting to process the message. For producers and if in HTTP request context, it should be the same as the HTTP request id (described in chapters 2.2.1 and 2.2.2);

- User ID – unique identifier for the user who is responsible for producing the message. Usually this should be filled when an application produced a message in client's HTTP request context. If not, it should be filled with an application user, representing the application name:
- JMS Message ID – unique identifier for the message. Generated by the JMS provider when sending the message;
- JMS Correlation ID – a correlating identifier for a message if it's a response to a message. Usually originates from the JMS Message ID of the initial message. Can also be application specific value;
- JMS Destination – a string representation of the JMS destination;
- JMS Reply to destination – a string representation of the JMS reply to destination. Should be filled whenever a producer expects a response for the message;
- JMS Properties - JSON-based (*JavaScript Object Notation*) key-value map of the properties from the message;
- Message body

From a Log4j2 point of view, the configuration for this format could be seen on Figure 8 and Figure 9.

```
<Properties>
    <!--.....-->
    <Property name="jmsListenerLoggingFormat">
     ${hostName}\tC\t%date{ISO8601}%date{Z}
     \t%replace{%X{correlationId}}{^$}{-}
     \t%replace{%X{requestId}{^$}{-}\t%replace{%X{userId}}{^$}{-}
     \t%X{jmsMessageId}\t%replace{%X{jmsCorrelationId}}{^$}{-}
     \t%X{jmsDestination}\t%replace{%X{jmsReplyToDestination}}{^$}{-}
     \t%replace{%X{jmsProperties}}{^$}{-}\t%replace{%message}{^$}{-}%n
    </Property>
    <!--.....-->
</Properties>
```

Figure 8. Example for incoming message logging pattern in Log4j2 XML configuration

```
<Properties>
    <!--.....-->
    <Property name="jmsSenderLoggingFormat">
     ${hostName}\tS\t%date{ISO8601}%date{Z}
     \t%replace{%X{correlationId}}{^$}{-}
     \t%replace{%X{requestId}{^$}{-}\t%replace{%X{userId}}{^$}{-}
     \t%X{jmsMessageId}\t%replace{%X{jmsCorrelationId}}{^$}{-}
     \t%X{jmsDestination}\t%replace{%X{jmsReplyToDestination}}{^$}{-}
     \t%replace{%X{jmsProperties}}{^$}{-}\t%replace{%message}{^$}{-}%n
    </Property>
    <!--.....-->
</Properties>
```

Figure 9. Example for outgoing message logging pattern in Log4j2 XML configuration

As the figures above show, using a tabulation character „*\t*" separation for fields, both patterns are very similar and the only difference is the log type field (*C* vs *S*). All the data that is received from the Log4j2's MDC, is wrapped with a „*X*" conversion character function [26]. Also, the author chose to use Log4j2's replace function to substitute empty values with a dash in order to present a null value as null values should not just be empty [4, p. 56]. The fields not wrapped with a replace function (*jmsMessageId* and *jmsDestination*) can not be empty as per the *Message* interface API combined with LHV's message broker server configuration [10].

When adding the log pattern to a Log4j2 XML configuration, there also needs to be a configuration for specific Logger instances which have a configured appenders for logging [27]. An example of this configuration can be seen on Figure 10.

```
<Appenders>
    <File name="JmsListenerLoggerFile" fileName="jms_in.log">
        <PatternLayout pattern="${jmsListenerLoggingFormat}"/>
    </File>
    <File name="JmsSenderLoggerFile" fileName="jms_out.log">
        <PatternLayout pattern="${jmsSenderLoggingFormat}"/>
    </File>
</Appenders>
<Loggers>
    <Logger name="jmsSenderLogger" level="TRACE" additivity="false">
        <AppenderRef ref="JmsSenderLoggerFile"/>
    </Logger>
    <Logger name="jmsListenerLogger" level="TRACE" additivity="false">
        <AppenderRef ref="JmsListenerLoggerFile"/>
    </Logger>
</Loggers>
```

Figure 10. Log4j2 appender and logger configuration example

For specific use cases like traffic logging in Log4j2 requires a specific Logger configuration. The logger's logging level is also defined as *TRACE* in order to log with the lowest severity and to make sure the traffic log events do not get logged to the wrong file while using this configuration [25].

4.2 JMS logging library

As stated, an initial implementation should represent a Java library which operates on top of the Java JMS API and a chosen logging framework (Log4j2). The logging functionality needs two vital dependents to in order to log a message. First is the configured *Logger* instance itself, which represents a vital part of the Log4j2 framework and provides APIs to trigger a log collection event [28]. Second is the JMS *Message* interface instance which is described in chapter 2.3.

Also, due to use-cases where there is no need to log every message payload down, the author chose to give an opportunity for destination-based configuration for the logging mechanism in order to omit payload logging. This is has proven to be needed when a high volume of traffic is generated and the message contents may not be needed. It is usually relevant in test environments or where an application uses message broker as a way to throttle some processes and produces a message for itself.

All in all, the JMS logging functionality could be achieved by using a simple *JmsLogger* Java class as shown on Figure 11.

```java
import org.apache.logging.log4j.Logger;
import javax.jms.Message;

public class JmsLogger {
    private final Logger logger;
    private final List<String> excludedDestinations;

    public JmsLogger(Logger logger) {
        this(logger, Collections.emptyList());
    }

    public JmsLogger(Logger logger, List<String> excludedDestinations) {
        this.logger = logger;
        this.excludedDestinations = excludedDestinations != null
                ? excludedDestinations
                : Collections.emptyList();
    }

    public void logMessage(Message message) {
        try {
            String destination = message.getJMSDestination().toString();
            String messageContent = "";
            if (excludedDestinations.stream()
                    .noneMatch(destination::contains)) {
                messageContent = JmsMessageUtil.getMessageContent(message);
            }
            JmsLoggerThreadContextUtil
                .populateJmsThreadContextValues(message, destination);
            logger.trace(messageContent);
        } catch (JMSException e) {
            throw new LoggingFailedException("Failed to log JMS message: " +
                message, e);
        }
    }
}
```

Figure 11. JMS Logger class code example

As Figure 11 shows, the *JmsLogger* class depends on two helper classes *JmsMessageUtil* and *JmsLoggerThreadContextUtil* which provide the logger additional help with message payload retrieval and MDC filling for log rows. These helper classes are explained in the following chapters. The class also provides two constructor methods – one for a logger where no destinations are excluded from payload logging and the other as a complementary constructor where a list of destinations can be provided to exclude payload logging for the selected destinations.

Also, it can be seen, that for the Log4j2 logging framework, the final logging operation occurs using the *Logger* interface's *trace* method which correlates to the log severity described in chapter 4.1.

4.2.1 Message payload retrieval

The *JmsMessageUtil* class provides methods for representing a readable payload to the log row. As stated before, the traffic log collection should be able to serialize any sub-interface instance of the JMS *Message* interface or stop and report any errors if the message payload serialization fails. The contents for *JmsMessageUtil* is shown on Figure 12.

```
public final class JmsMessageUtil {
    public static String getMessageContent(Message message)
            throws JMSException {
        var content = "";
        if (message instanceof TextMessage) {
            content = ((TextMessage) message).getText();
        } else if (message instanceof BytesMessage) {
            BytesMessage bytesMessage = (BytesMessage) message;
            bytesMessage.reset();
            byte[] data = new byte[(int) bytesMessage.getBodyLength()];
            bytesMessage.readBytes(data);
            bytesMessage.reset();
            content = new String(data, UTF_8);
        } else {
            throw new JmsMessageContentException("Only TextMessage and " +
                + "BytesMessage are supported for retrieving message " +
                + "content from javax.jms.Message");
        }
        return content;
    }
}
```

Figure 12. JMS Logger message payload retrieval example

The payload retrieval on Figure 12 is supporting two types of messages – *TextMessage* and *BytesMessage*. The *TextMessage*'s API is very straightforward as payload can be retrieved by the *getText()* method [29]. *ByteMessage* is a bit more challenging as it has read-only and write-only modes and for reading, the payload retrieval must start byte retrieval from the beginning of the stream nested inside it and thus, the *reset* method is called before retrieval [30]. Also, after a read has been done, the author believes it should be reset because the message contents could be read again at a later stage of processing.

4.2.2 MDC context lifecycle operations

The author's main idea for the *JmsLoggerThreadContextUtil* in the logger library is to provide support for operating on the MDC provided by Log4j2 framework. As the MDC is based on a running thread, using the *ThreadContext* API, one can add all the necessary data to support log collection [31]. From a interception point of view, the author believes that the same util class which is responsible for filling data, should also have necessary support to clear the context data from the thread whenever an application decides to do so.

In general, the key-value pairs passed to the MDC can be referenced in a log row format as done in chaper 4.1 and ultimately the values from MDC get populated to the log row [20]. Considering the proposed log row format earlier and also that the log collection operation is triggered by a *Logger*'s interface method by passing message content variable into the logging call, in author's opinion the following data should be filled to MDC from the JMS *Message* interface: message id, correlation id, the destination, a reply to destination and all the properties

```
public static final String JMS_MESSAGE_ID = "jmsMessageId";
public static final String JMS_CORRELATION_ID = "jmsCorrelationId";
public static final String JMS_DESTINATION = "jmsDestination";
public static final String JMS_REPLY_DEST = "jmsReplyToDestination";
public static final String JMS_PROPERTIES = "jmsProperties";

public static void populateJmsThreadContextValues(Message message
        ) throws JMSException {
    ThreadContext.put(JMS_MESSAGE_ID, message.getJMSMessageID());
    ThreadContext.put(JMS_CORRELATION_ID,
          message.getJMSCorrelationID());
    ThreadContext.put(JMS_PROPERTIES, getMessageProperties(message));
    ThreadContext.put(JMS_DESTINATION,
          message.getJMSDestination().toString());
    var replyDest = message.getJMSReplyTo();

    if (replyDest == null) ThreadContext.remove(JMS_REPLY_DEST);
    else ThreadContext.put(JMS_REPLY_DEST, replyDest.toString());
}
```

Figure 13. JMS Logger MDC population example

As Figure 13 represents, the operations are done from a static viewpoint and only require a JMS *Message* interface as a parameter. Also, as there could be a scenario where a JMS reply-to destination is not filled to the message, the author chose to remove it from the MDC for these use-cases since Java threads could be reused and old MDC context may not have been cleared up. Lastly, as there could be multiple key-value properties passed with a message as shown in chapter 2.3, the author chose to use another method to retrieve them and it is represented on Figure 14.

```
private static String getMessageProperties(Message message) throws
        JMSException {
    var propertyMap = new HashMap<String, String>();
    var propertyNames = message.getPropertyNames();
    while (propertyNames.hasMoreElements()) {
        String name = propertyNames.nextElement().toString();
        propertyMap.put(name, message.getStringProperty(name));
    }

    try {
        var propertiesJsonMapper = new Log4jJsonObjectMapper();
        return propertiesJsonMapper.writeValueAsString(propertyMap);
    } catch (JsonProcessingException e) {
        throw new JmsMessageContentException("Failed to build " +
                + " JMS Message properties as JSON", e);
        }
    }
}
```

Figure 14. JMS Logger example to serialize JMS Message properties for MDC

As the example (Figure 14) above shows, firstly all the property names from the JMS *Message* instance are retrieved and used to build up a map of stringed key-value pairs. This map is then serialized to a JSON format due to the fact that the properties could contain any possible stringed value in whatever format as the *Message* interface allows it. Also, as traffic logging properties tend to be shared in terms of LHV IT infrastructure, the library also contains an enum class containing properties that could be used by the development teams. The enum class can be seen in Appendix 5.

Lastly, as stated before, the MDC propagation utility class should also contains methods to manage MDC lifecycle in terms of clearing it after a process has finished. For that purpose, an example can be seen on Figure 15.

```
public static void clearJmsThreadContextValues() {
    ThreadContext.remove(JMS_MESSAGE_ID);
    ThreadContext.remove(JMS_CORRELATION_ID);
    ThreadContext.remove(JMS_PROPERTIES);
    ThreadContext.remove(JMS_DESTINATION);
    ThreadContext.remove(JMS_REPLY_DEST);
}
```

Figure 15. JMS Logger MDC clearing example

4.2.3 Automated tests

In order to support any healthy development lifecycle, the author believes that creating automated tests are an appropriate way to prevent future issues and unwanted development-related regression. For the JMS logging library described, the author chose to write integration tests on top of the Spring framework support as the latter is widely used in LHV IT infrastructure.

The tests cover logging for both incoming and outgoing messages and they assert that a preconfigured logger actually does write the JMS traffic event logs to a log file. These tests can be seen in Appendix 6.

4.3 JMS logging library for Spring framework

From author's experience, when dealing with Spring framework's JMS features, the noteable components for interacting with JMS brokers are the *JmsTemplate* bean component and the *JmsListener* annotation. The *JmsTemplate*'s main usage purpose in an application's program flow is to send outgoing messages and *JmsListener* annotation is used to define a method to retrieve messages from specificed destination [24]. For the framework specific logging library, the author will focus on intercepting or wrapping any logging operations with a focus on these components.

4.3.1 Logging incoming messages

In order to simplify any initial JMS traffic logging implementations for development teams, the author believes that incoming message handling should be able to intercept traffic for the existing *JmsListener* annotated methods and work out-of-the-box. This could be achieved by using Spring Boot's auto configuration capabilities to configure all the JMS listener methods to implement a predefined message listener adapter which acts

as an interceptor before a message is passed to the application's own program flow [32] [33].

As this library does not intend to break any existing implementations while attaching it to a Spring project, the author chose to make auto configuration only run if a certain configuration property value is present in the application's configuration files. As the traffic logger also supports disabling payload logging for certain destinations, this option should also be propagated to the auto configuration procedure. A working example of this kind of a configuration can be seen on Figure 16.

```
@Configuration
@ConditionalOnProperty(name = "jms.loggingListenersEnabled", havingValue =
          "true")
@EnableConfigurationProperties(JmsConfigProperties.class)
public class LhvJmsListenerAutoConfiguration implements JmsListenerConfigurer
{
    @Resource
    DefaultMessageHandlerMethodFactory handlerMethodFactory;

    @Resource
    JmsConfigProperties jmsProperties;

    @Bean
    public LoggingJmsListenerEndpointRegistry tracingJmsListenerRegistry() {
        var logger = new JmsLogger(LogManager.getLogger("jmsListenerLogger"),
                jmsProperties.getDestinationsExcludedFromBodyLogging());
        return new LoggingJmsListenerEndpointRegistry(handlerMethodFactory,
                logger);
    }

    @Override
    public void configureJmsListeners(JmsListenerEndpointRegistrar registrar)
    {
        registrar.setEndpointRegistry(tracingJmsListenerRegistry());
    }
}
```

Figure 16. Spring boot's auto configuration example for JMS listeners

As the figure above shows, the configuration is only enabled if a property named *jms.loggingListenersEnabled* has a value of „*true*" set. If applicable, then the configuration class initializes an endpoint registry which takes the *JmsLogger* as one of the arguments.

The whole contents of the *LoggingJmsListenerEndpointRegistry* created for this intercepting process can be seen in Appendix 7 as it contains several Spring Framework related configuration techniques and self-declared classes to achieve the required result.

A noteable feature for the interceptor is how it operates on a message. Similarly to the MDC context filling in the JMS logging library itself described in chapter 4.2, the Spring specific library solution also operates on the Log4j2's MDC in author's proposed solution. This is due to the fact that JMS logging library's mechanism handles incoming and outgoing messages the same way. But as a event context differs for both of them, the interceptor checks message properties and applies relevant complementary data to the MDC itself, considering that a message is guaranteed to be an incoming message at that point.

As per the log row format in chapter 4.1, the native JMS logging library does not populate the following data to the MDC:

- Correlation ID
- Request ID
- User ID

From an incoming message's perspective, as the *Message* intstance properties could contain the correlation and user identifier, these values should be retrieved from there with the help of a shared enum property keys described in Appendix 5. For the correlation identifier – if the value is not defined, the author believes that it should be generated by the logging mechanism itself as described in chapter 2.4. For the request identifier – the value should be generated for every message by the application as it is done the same way for HTTP traffic logging. Lastly, if there exists no user identifier in the message properties, the author believes this value should be left empty as there is no other reasonable way to fill that data.

After populating context values to the MDC and logging the message down, the *Message* interface instance is passed on to the application to consume. After the consumption has been done, the MDC is cleared in order to make sure no context data is left to the MDC for any following operations on that current Java thread.

4.3.2 Logging outgoing messages

While the proposed incoming message interception logic resulted in a solution where no actual development team attention is needed code-wise, the Spring's *JmsTemplate* interception support for outgoing messages is a bit more lacking and the author believes that achieving it may require multiple iterations of deep dive research and development across the Spring Framework's Messaging API and protocols[1] [34] [35].

That being said, the author believes that an alternative approach which wraps logging operations around the *JmsTemplate* is more suitable for making sure that a developer-friendly solution will be achieved. In order to also give flexibility for these logging operations and *JmsTemplate* usages while sending messages, the author believes it is best not to develop a fixed solution with POJOs (*Plain Old Java Object*) but rather benefit from interfaces and also support preconfigured solutions as the latter might be enough for some applications while sending messages to the broker.

Starting off with a POJO interface which ultimately would get converted to a JMS message, a proposed example can be seen on Figure 17.

```
public interface OutgoingJmsMessage {
    Destination getJmsDestination();
    Destination getJmsReplyToDestination();
    String getJmsCorrelationId();
    String getMessageId();
    String getMessageCorrelationId();
    Object getContent();
    String getContentType();
    MessageCreator getJmsMessageCreator();
}
```

Figure 17. Outgoing JMS message interface example.

As it can be seen from the figure above, the *OutgoingJmsMessage* interface contracts all the required fields to send a JMS message in terms of the destination and the message contents. It is also accompanied by message specific optional context fields, which could

---

[1] An existing feature request has been opened on 20th of May 2019 in Spring Framework's Github repository to tackle the *JmsTemplate*'s interception limitation [Online]. Available: https://github.com/spring-projects/spring-framework/issues/22999. [Accessed 20 April 2021]

be filled to achieve more comprehensive understanding of the message on consumer side and also in log analysis. Also, if an application is still requiring database table persistence for tracking unsent messages, the author proposes *OutgoingJmsMessage* interface to be accompanied with another interface which would support persistence related audit fields as Figure 18 shows.

```
public interface PersistentAuditedOutgoingJmsMessage
        <T extends Temporal> extends OutgoingJmsMessage {
    Boolean isSent();
    String getLastRequestId();
    String getCreatedBy();
    T getCreatedDtime();
    String getModifiedBy();
    T getModifiedDtime();
}
```

Figure 18. Example for outgoing JMS message interface with persistence field support.

In order to tie the created POJO interfaces with message sending logic and support applications to use their own preferred way of sending them, a *JmsMessageSender* interface is proposed and it would also be accompanied by a default implementation class *DefaultJmsMessageSender* which does the required minimum for traffic logging and mapping existing audit data from MDC to the *Message*.

```
public Message send(OutgoingJmsMessage outgoingJmsMessage) {
    var messageRef = new AtomicReference<Message>();
    jmsTemplate.send(outgoingJmsMessage.getJmsDestination(), session -> {
        Message message = outgoingJmsMessage
            .getJmsMessageCreator()
            .createMessage(session);
        populateMessageFieldsAndProperties(message, outgoingJmsMessage);
        messageRef.set(message);
        return message;
    });
    var message = messageRef.get();
    jmsLogger.logMessage(message);
    return message;
 }
```

Figure 19. Example of *DefaultJmsMessageSender* solution of logging outgoing messages.

As Figure 19 shows, the *DefaultJmsMessageSender* uses the *JmsTemplate* capabilities to send a message to the designated destination while creating the *Message* interface instance using the previously shown POJO interface's *getJmsMessageCreator()* method. The author believes that outgoing traffic log collection can and should only occur when

a JMS broker has confirmed message delivery and set a *JMSMessageID* to the *Message* object [10]. Due to that, the log collection is to be done outside of the message sending operations. To achieve that, the message is saved to the thread safe *AtomicReference* instance in the sending flow and retrieved later in order to trigger log collection [36].

Also, on Figure 19 a method *populateMessageFieldsAndProperties* is called. As the outgoing message should be trustworthy for any consumer, this method maps any auditable context from the *OutgoingJmsMessage* interface to the *Message* interface sent to the broker. The author's proposed default solution fills the following fields:

- *Message* interface's *JMSCorrelationId* if set;
- *Message* interface's *JMSReplyTo* if set;
- *Message* interface's property "*X-MESSAGE-ID*" if set;
- *Message* interface's property "*X-MESSAGE-CORRELATION-ID*" if set;
- *Message* interface's property "*Content-Type*" if set;
- *Message* interface's property "*X-Correlation-ID*"

The proposed implementations and interfaces for contracting message sending and context filling can be seen in full in Appendices 8 and 9.

5 Usability

For message queue traffic logging to be up to audit logging requirements, the log rows must give answers to the questions of who, what, where, whence and when as stated in chapter 3.1. In order to assert that the log row format answers these questions with the implementation of the logging libraries, the output data for both incoming and outgoing messages is presented in Table 3Table 2 and Table 3.

Table 2. Log row representation of an outgoing message

Field	Value
Host	`PC332145`
Log type	`S`
Timestamp	`2021-04-19T11:31:11,821+0300`
Correlation ID	`fa6465c3-14e6-42fb-9444-e802b20a150c`
Request ID	`tcDNGIE1`
User ID	`9183501`
JMS Message ID	`ID:PC332145-38865-1618821068996-1:1:4:1:1`
JMS Correlation ID	`-`
JMS Destination	`queue://SOME.QUEUE`
JMS Reply To	`-`
JMS Properties	`{"X_SYSTEM_ID":"TransactionSystem", {"X-Correlation-ID":" fa6465c3-14e6-42fb-9444-e802b20a150c"}`
JMS Payload	`{"key":"value", "key2":"value2", …}`

Table 3. Log row representation of an incoming message

Field	Value
Host	`PC332146`
Log type	`C`
Timestamp	`2021-04-19T11:31:12,154+0300`
Correlation ID	`fa6465c3-14e6-42fb-9444-e802b20a150c`
Request ID	`L3nfCNWj`

User ID	9183501
JMS Message ID	ID:PC332145-38865-1618821068996-1:1:4:1:1
JMS Correlation ID	-
JMS Destination	queue://SOME.QUEUE
JMS Reply To	-
JMS Properties	{"X_SYSTEM_ID":"TransactionSystem", {"X-Correlation-ID":" fa6465c3-14e6-42fb-9444-e802b20a150c"}
JMS Payload	{"key":"value", "key2":"value2", …}

As the tables above show, a host named „*PC332145*" has sent a message at „*2021-04-19T11:31:11,821+0300*" to „*queue://SOME.QUEUE*" and a host named „*PC332146*" consumed it at „*2021-04-19T11:31:12,154+0300*". The log entries also both contain the same JMS Message identifier, properties and payload. If dealing with ActiveMQ message broker, the contents could have also be seen in a similar way from the UI as Figure 1 in chapter 2.3 shows.

As the traffic log events certainly do answer all the questions about a message's whereabouts and its contents, these kind of log entries could be used for further log analysis in terms of troubleshooting, audit trailing or monitoring.

5.1 Troubleshooting message broker issues

To understand what went wrong with message consumption, the traffic log events should be correlatable to other system logs. As this was proven to be applicable for HTTP request troubleshooting in chapter 2.4, the author expects the same for JMS traffic troubleshooting.

For example, considering that there is an error in application logs which happened on a thread named „*DefaultMessageListenerContainer-1*" as Figure 20 shows.

host	ts	thread	severity	correlation_id	request_id	message
PC123	2021-04-21 16:55:30.125 +0300	DefaultMessageListenerContainer-1	ERROR	cqft9FVu	sKgs5HrL	Lock request time out period exceeded.

Figure 20. Application error log example on JMS listener thread

Based on the thread name, relevant presumptions could be made that the message should be present in the JMS traffic log. Using the *request_id* value, a corresponding message could be found as Figure 21 shows.

host	tyı	ts	correlation_ic	request_ic	jms_message_id	jms_destination	jms_properties
PC123	C	2021-04-21 16:55:30.08	cqft9FVu	sKgs5HrL	ID:PC456-36720-16	queue://SOME.QUI	{"X-Correlation-ID":"cqft9FVu"}

Figure 21. Lightweight example of incoming JMS log message

Considering the knowledge gathered, it could be said that the consumption of this message from "*PC456*" did not succeed because some locking process took too long and a time out was triggered. As there could be a retry configuration set for message consumption, the *jms_message_id* or *correlation_id* values could be used to find any retry messages that got consumed after the failed message consumption as Figure 22 shows.

host	tyı	ts	correlation_ic	request_ic	jms_message_id	jms_destination	jms_properties
PC123	C	2021-04-21 16:55:40.14	cqft9FVu	Wv0P8zdV	ID:PC456-36720-16	queue://SOME.QUI	{JMSXDeliveryCount":"2","X-C
PC123	C	2021-04-21 16:55:30.08	cqft9FVu	sKgs5HrL	ID:PC456-36720-16	queue://SOME.QUI	{"X-Correlation-ID":"cqft9FVu"}

Figure 22. Lightweight example of retried incoming JMS message in logs

For the retry message log row on the figure above, it can also be seen that there is a new value in the *jms_properties* field named *JMSXDeliveryCount*. This is due to the redelivery operation and the JMS broker has set corresponding property to the message indicating the number of attempts there have been to send this message to a consumer [37].

5.2 Analysis and monitoring

From an application monitoring point of view, the proposed log format and its implementation details provide enough data in author's opinion to develop initial monitoring checks for traffic flow. Some possible scenarios which could be developed are the following (but not limited to):

- Monitoring job to return a count of messages consumed in the last 15 minutes;
- Monitoring job to return a count of messages produced in the last 15 minutes;
- Monitoring job to return a count of messages produced that have not retrieved an expected reply message in the last 15 minutes;

From the perspective of audit analysis, the proposed solution provides a better audit trail as every message is subject for logging and while a database-based audit logging did collect the traffic logs, on the case of a database exception, the log entries got rollbacked without any retention as described in chapter 2.2.4.

Also, as the log entries contain key correlation fields in terms of the correlation, request and user identifier, the traffic log entries could be subject to complex analysis models and security monitoring in author’s opinion.

Conclusion

Logging message traffic between applications and messaging servers could provide a valuable audit trail for any information system. According to the author's research and observation of topics on this matter, no extensive nor comprehensive research or study has been made for this particular type of log collection. This thesis covered the existing logging practices in LHV Bank and formulated a problem regarding the lack of traffic logging for message brokerage on application side. Also, the thesis analysed message contexts and offered solutions to gain the most knowledge from log collection on message production and consumption.

As the proposed solutions in this thesis corresponded most requirement accordance from Tiit Hallas' study "Logging Requirement Analysis and Specification for Development Based on Governmental Institutions of Estonia", the results have proven to be applicable. The outcome libraries have been applied to selected applications in LHV Bank's IT-infrastructure. The log collection methodology and the implementation logic has proven to be a success from the stakeholders' point of view.

The outcome of thesis is two separate Java libraries combined from Appendices 5, 6, 7, 8 and 9 and a document describing log format and library usages (Appendix 10) which is also reflected to LHV Bank's documentation space for the institution's IT-development teams to refer to.

The author of thesis gives out special thanks to Tiit Hallas and Toomas Lepikult for the supervision and contribution to this thesis and also Heiki Hiisjärv for his insights and inputs during the code review procedures.

References

[1] Estonian Financial Supervision Authority, “Requirements for the organisation of the information technology and information security of the subject of financial supervision,” 30 June 2020. [Online]. Available: https://www.fi.ee/en/guides/pangandus-ja-krediit/requirements-organisation-information-technology-and-information-security-subject-financial. [Accessed 13 March 2021].

[2] Estonian Financial Supervision Authority, “Requirements for the organisation of the information technology and information security of the subject of financial supervision,” 2020.

[3] Amazon Web Services, Inc, “Message Queues,” [Online]. Available: https://aws.amazon.com/message-queue/. [Accessed 24 February 2021].

[4] T. Hallas, “Logging Requirement Analysis and Specification for Development Based on Governmental Institutions of Estonia,” Tallinn, 2014.

[5] AS LHV Bank, "Brief history of LHV," [Online]. Available: https://www.lhv.ee/en/about. [Accessed 24 February 2021].

[6] AS LHV Bank, “Status page,” [Online]. Available: https://status.lhv.ee/. [Accessed 13 March 2021].

[7] S. Colebourne, “Best Practices for Designing and Implementing a Library in Java,” Oracle Corporation, [Online]. Available: https://www.oracle.com/corporate/features/library-in-java-best-practices.html. [Accessed 20 3 2021].

[8] Oracle Corporation, “Java Community Process,” 16 March 2015. [Online]. Available: https://jcp.org/en/jsr/detail?id=343. [Accessed 21 March 2021].

[9] Rapid7, “Rapid7 blog,” 23 December 2016. [Online]. Available: https://blog.rapid7.com/2016/12/23/the-value-of-correlation-ids/. [Accessed 21 March 2021].

[10] Oracle Corporation, “Interface Message: Java(TM) EE 7 Specification APIs,” [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/jms/Message.html. [Accessed 24 March 2021].

[11] Oracle Corporation, “Understanding Message ID and Correlation ID Patterns for JMS Request/Response,” [Online]. Available: https://docs.oracle.com/cd/E13171_01/alsb/docs25/interopjms/MsgIDPatternforJMS.html. [Accessed 28 March 2021].

[12] Oracle Corporation, “Controlling Message Acknowledgment,” 2010. [Online]. Available: https://docs.oracle.com/cd/E19798-01/821-1841/bncfw/index.html. [Accessed 29 March 2021].

[13] The Apache Software Foundation, “ActiveMQ Message Redelivery and DLQ Handling,” [Online]. Available: https://activemq.apache.org/message-redelivery-and-dlq-handling.html. [Accessed 29 March 2021].

[14] Apache Software Foundation, “Apache Tomcat 7: The AJP Connector,” [Online]. Available: https://tomcat.apache.org/tomcat-7.0-doc/config/ajp.html. [Accessed 8 April 2021].

[15] IBM Cloud Education, “Message Brokers,” 23 January 2020. [Online]. Available: https://www.ibm.com/cloud/learn/message-brokers. [Accessed 13 April 2021].

[16] The Apache Software Foundation, “How does a Queue compare to a Topic,” [Online]. Available: https://activemq.apache.org/how-does-a-queue-compare-to-a-topic. [Accessed 14 April 2021].

[17] Oracle Corporation, “The Request-Reply Pattern,” 2010. [Online]. Available: https://docs.oracle.com/cd/E19340-01/820-6424/aerby/index.html. [Accessed 14 April 2021].

[18] Oracle Corporation, “Java Logging Overview,” [Online]. Available: https://docs.oracle.com/en/java/javase/15/core/java-logging-overview.html. [Accessed 14 April 2021].

[19] R. Kuć, “Java Logging Best Practices: Sematext,” 3 August 2020. [Online]. Available: https://sematext.com/blog/java-logging-best-practices/. [Accessed 14 April 2021].

[20] The Apache Software Foundation, “Thread Context: Log4j 2 API,” [Online]. Available: https://logging.apache.org/log4j/2.x/manual/thread-context.html. [Accessed 14 April 2021].

[21] Oracle Corporation, “Java 9 HttpClient API reference,” [Online]. Available: https://docs.oracle.com/javase/9/docs/api/jdk/incubator/http/HttpClient.html. [Accessed 14 April 2021].

[22] Oracle Corporation, “Java 8 HttpURLConnection API,” [Online]. Available: https://docs.oracle.com/javase/8/docs/api/java/net/HttpURLConnection.html. [Accessed 14 April 2021].

[23] Tom Akehurst, “Which Java HTTP client should I use in 2020?,” 12 October 2020. [Online]. Available: https://www.mocklab.io/blog/which-java-http-client-should-i-use-in-2020/. [Accessed 14 April 2021].

[24] VMware, Inc., “JMS (Java Message Service): Spring framework documentation,” 13 April 2021. [Online]. Available: https://docs.spring.io/spring-framework/docs/5.3.6/reference/html/integration.html#jms. [Accessed 14 April 2021].

[25] The Apache Software Foundation, “Configuration: Log4j2 Documentation,” 6 March 2021. [Online]. Available: https://logging.apache.org/log4j/2.x/manual/configuration.html. [Accessed 15 April 2021].

[26] The Apache Software Foundation, “Layouts: Log4j2 Documentation,” 6 March 2021. [Online]. Available: https://logging.apache.org/log4j/2.x/manual/layouts.html. [Accessed 20 April 2021].

[27] The Apache Software Foundation, “Appenders: Log4j2 Documentation,” 6 March 2021. [Online]. Available: https://logging.apache.org/log4j/2.x/manual/appenders.html. [Accessed 20 April 2021].

[28] The Apache Software Foundation, “Interface Logger,” [Online]. Available: https://logging.apache.org/log4j/2.x/log4j-api/apidocs/org/apache/logging/log4j/Logger.html. [Accessed 14 April 2021].

[29] Oracle Corporation, “Interface TextMessage: Java(TM) EE 7 Specification APIs,” [Online]. Available: https://javaee.github.io/javaee-spec/javadocs/javax/jms/TextMessage.html. [Accessed 14 April 2021].

[30] Oracle Corporation, “Interface BytesMessage: Java(TM) EE 7 Specification APIs,” [Online]. Available: https://docs.oracle.com/javaee/7/api/javax/jms/BytesMessage.html. [Accessed 14 April 2021].

[31] The Apache Software Foundation, “Class ThreadContext,” [Online]. Available: https://logging.apache.org/log4j/2.x/log4j-api/apidocs/org/apache/logging/log4j/ThreadContext.html. [Accessed 14 April 2021].

[32] VMware, Inc., “Creating Your Own Auto-configuration: Spring Boot,” [Online]. Available: https://docs.spring.io/spring-boot/docs/current/reference/html/spring-boot-features.html#boot-features-developing-auto-configuration. [Accessed 20 April 2021].

[33] VMware, Inc., “Class JmsListenerEndpointRegistry: Spring Framework API,” [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jms/config/JmsListenerEndpointRegistry.html. [Accessed 20 April 2021].

[34] VMware, Inc., “Package org.springframework.messaging: Spring Framework API,” [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/messaging/package-summary.html. [Accessed 20 April 2021].

[35] VMware, Inc., “Class JmsTemplate: Spring Framework API,” [Online]. Available: https://docs.spring.io/spring-framework/docs/current/javadoc-api/org/springframework/jms/core/JmsTemplate.html. [Accessed 20 April 2021].

[36] J. Jenkov, “AtomicReference,” 26 January 2016. [Online]. Available: http://tutorials.jenkov.com/java-util-concurrent/atomicreference.html. [Accessed 20 April 2021].

[37] The Apache Software Foundation, “ActiveMQ Message Properties,” [Online]. Available: https://activemq.apache.org/activemq-message-properties. [Accessed 21 April 2021].

Appendix 1 – Non-exclusive licence for reproduction and publication of a graduation thesis[1]

I Erik Ehrbach

1. Grant Tallinn University of Technology free licence (non-exclusive licence) for my thesis “Message traffic audit logging between application and messaging server on the example of LHV bank”, supervised by Tiit Hallas and Toomas Lepikult
   1.1. to be reproduced for the purposes of preservation and electronic publication of the graduation thesis, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright;
   1.2. to be published via the web of Tallinn University of Technology, incl. to be entered in the digital collection of the library of Tallinn University of Technology until expiry of the term of copyright.
2. I am aware that the author also retains the rights specified in clause 1 of the non-exclusive licence.
3. I confirm that granting the non-exclusive licence does not infringe other persons' intellectual property rights, the rights arising from the Personal Data Protection Act or rights arising from other legislation.

27.04.2021

1 The non-exclusive licence is not valid during the validity of access restriction indicated in the student's application for restriction on access to the graduation thesis that has been signed by the school's dean, except in case of the university's right to reproduce the thesis for preservation purposes only. If a graduation thesis is based on the joint creative activity of two or more persons and the co-author(s) has/have not granted, by the set deadline, the student defending his/her graduation thesis consent to reproduce and publish the graduation thesis in compliance with clauses 1.1 and 1.2 of the non-exclusive licence, the non-exclusive license shall not be valid for the period.

Appendix 2 – Application HTTP access log file example

Log type	Timestamp	Correlation id	Request id	Session id	Domain	Client IP	Method	URL	Payload	Status	User id
C	2021-03-21 10:56:38.246 +0200	e80fdd2a	mpfTgWXY	olALUHjT	sandboxapi.lhv.eu	8.8.8.8	GET	/v1/accounts	NULL	Not applicable for request log row	
S	2021-03-21 10:56:38.514 +0200	e80fdd2a	mpfTgWXY	olALUHjT	Not applicable for response log row				"{…}"	200	53434
C	2021-03-21 10:57:39.246 +0200	564a9b6a	AUYL0AnS	olALUHjT	sandboxapi.lhv.eu	8.8.8.8	POST	/v1/consents	"{…}"	Not applicable for request log row	
S	2021-03-21 10:57:39.450 +0200	564a9b6a	AUYL0AnS	olALUHjT	Not applicable for response log row				"{…}"	201	53434

Appendix 3 – Application HTTP client log file example

Log type	Timestamp	Correlation id	Request id	Session id	Log record id	Node	Method	URL	Payload	Status	User id
C	2021-03-21 10:56:38.246 +0200	e80fdd2a	mpfTgWXY	olALUHjT	FewQ7Wb9	PROD-HOST01	GET	https//api.sid.ee/v1/info	NULL	NULL	53434
S	2021-03-21 10:56:38.471 +0200	e80fdd2a	mpfTgWXY	olALUHjT	FewQ7Wb9	PROD-HOST01	GET	https//api.sid.ee/v1/info	{…}	200	53434
C	2021-03-21 10:56:38.472 +0200	e80fdd2a	mpfTgWXY	olALUHjT	2ArCREgG	PROD-HOST01	POST	https//api.sid.ee/v1/auth	{…}	NULL	53434
S	2021-03-21 10:56:38.471 +0200	e80fdd2a	mpfTgWXY	olALUHjT	2ArCREgG	PROD-HOST01	POST	https//api.sid.ee/v1/auth	{…}	200	53434

Appendix 4 – Database-logged message queue traffic examples

QUEUE_MESSAGE_INCOMING table

QUMI_ID	DESTINATION	CONTENT	CONTENT_TYPE	MESSAGE_ID	CORRELATION_ID	USER_ID	CREATED_DTIME	MODIFIED_DTIME
543	LHV.QUEUE.REQ	0x7B2….7D	application/json	REQasd721	NULL	53434	2021-03-21 10:56:38.246 +0200	2021-03-21 10:56:38.246 +0200
544	LHV.QUEUE.REQ	0x7B2….7D	application/json	REQfsa391	NULL	53434	2021-03-21 10:56:41.105 +0200	2021-03-21 10:56:41.105 +0200

QUEUE_MESSAGE_OUTGOING table

QUMO_ID	QUMI_ID	DESTINATION	IS_SENT	CONTENT	CONTENT_TYPE	MESSAGE_ID	CORRELATION_ID	USER_ID	CREATED_DTIME	MODIFIED_DTIME
421	543	LHV.QUEUE.RESP	1	0x7B2….1E	application/json	RESasd721	REQasd721	53434	2021-03-21 10:56:38.746 +0200	2021-03-21 10:56:38.913 +0200
422	544	LHV.QUEUE.RESP	1	0x7B2….1E	application/json	RESfsa391	REQfsa391	53434	2021-03-21 10:56:41.259 +0200	2021-03-21 10:56:41.514 +0200

Appendix 5 – Shared JMS properties enum class

```
public enum JmsProperty {
    /**
     * Content-Type. Used to describe the content structure of the message.
     * I.e application/json, application/xml
     */
    CONTENT_TYPE("Content-Type"),

    /**
     * X-Correlation-ID. Used to correlate the message throughout any HTTP
     * requests made across different applications
     */
    X_CORRELATION_ID("X-Correlation-ID"),

    /**
     * X-MESSAGE-ID. Used to identify the message in an
     * application/business logic scope. Usually persisted in DB as well.
     */
    X_MESSAGE_ID("X-MESSAGE-ID"),

    /**
     * X-MESSAGE-CORRELATION-ID. Used to identify message(s) in an
     * application/business logic scope in a request-response manner.
     * Usually persisted in DB as well.
     */
    X_MESSAGE_CORRELATION_ID("X-MESSAGE-CORRELATION-ID"),

    /**
     * X-SYSTEM-ID. Used to identify the message origin system. I.e PAYMENTS
     */
    X_SYSTEM_ID("X-SYSTEM-ID"),

    /**
     * X-USER-ID. Used to identify the authenticated user or system.
     */
    X_USER_ID("X-USER-ID"),

    /**
     * X-USER-ID-ROLE. Used to identify the customer/user whose role is
     * acted upon.
     */
    X_USER_ID_ROLE("X-USER-ID-ROLE");
```

```
    @Getter
    private final String propertyName;

    JmsProperty(String propertyName) {
        this.propertyName = propertyName;
    }
}
```

Appendix 6 – Automated tests for JMS logging library

Incoming message log output test

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes = TestApplication.class)
class JmsListenerLoggerTest {
    @Autowired
    private JmsTemplate jmsTemplate;
    private final CountDownLatch lock = new CountDownLatch(1);
    private final AtomicReference<Message> receivedMessage = new
                AtomicReference<>();
    private final JmsLogger jmsListenerLogger = new
                JmsLogger(LogManager.getLogger("jmsListenerLogger"));

    @JmsListener(destination = "SOME.QUEUE")
    public void handle(Message message) throws JMSException {
        ThreadContextUtil.addToContext("correlationId",
                 message.getStringProperty("X-Correlation-ID"));
        ThreadContextUtil.addToContext("requestId",
                 LoggingUtil.getRandomHash());
        ThreadContextUtil.addToContext("userId",
                 message.getStringProperty("X-USER-ID"));
        jmsListenerLogger.logMessage(message);
        receivedMessage.set(message);
        lock.countDown();
    }

    @Test
    void shouldLogJmsBody() throws InterruptedException, IOException,
                  JMSException {
        var destination = new ActiveMQQueue("SOME.QUEUE");
        var replyToDestination = new ActiveMQTopic("SOME.TOPIC");
        this.jmsTemplate.send(destination, (session) -> {
            TextMessage textMessage = session.createTextMessage();
            textMessage.setText("Hello, world - listener test!");
            textMessage.setJMSCorrelationID("ID:someJmsCorrId");
            textMessage.setJMSDestination(destination);
            textMessage.setJMSReplyTo(replyToDestination);
            textMessage.setStringProperty("X-USER-ID", "999");
            textMessage.setStringProperty("X-Correlation-ID",
                "abcdCorrelationId123456");
            textMessage.setStringProperty("test", "value");
            textMessage.setStringProperty("test2", "value2");
            return textMessage;
        });

        lock.await(400, TimeUnit.MILLISECONDS);

        String file = FileUtils.readFileToString(new
                File("jms_in.log"), StandardCharsets.UTF_8);
        String[] split = file.split("\n");
        assertTrue(split.length > 0);
```

```
        String logMessageRow = split[split.length-1];
        String[] logMessageParts = logMessageRow.split("\\|");

        assertEquals(InetAddress.getLocalHost().getHostName(),
            logMessageParts[0],
                "Host name not correct");

        assertEquals("C", logMessageParts[1],
                "Log type not correct");

        DateTimeFormatter formatter =
            DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss,SSSZ");
        assertDoesNotThrow(() ->
            OffsetDateTime.parse(logMessageParts[2], formatter),
                "Log offset datetime not parseable");

        assertEquals("abcdCorrelationId123456", logMessageParts[3],
                "Correlation ID not correct");

        assertTrue(logMessageParts[4].matches("[A-Za-z0-9]{8}"),
                "Request ID not in correct format");

        assertEquals("999", logMessageParts[5],
                "User ID not correct");

        assertEquals(receivedMessage.get().getJMSMessageID(),
                     logMessageParts[6],
                "JMS Message ID not correct");

        assertEquals("ID:someJmsCorrId", logMessageParts[7],
                "JMS Correlation ID not correct");

        assertEquals("queue://SOME.QUEUE", logMessageParts[8],
                "JMS Destination not correct");

        assertEquals("topic://SOME.TOPIC", logMessageParts[9],
                "JMS Reply to Destination not correct");

        assertEquals("{\"X-USER-
            ID\":\"999\",\"test2\":\"value2\",\"test\":\"value\"," +
            "\"X-Correlation-ID\":\"abcdCorrelationId123456\"}",
                logMessageParts[10],
                    "JMS Message properties not correct");

        assertEquals("Hello, world - listener test!",
                logMessageParts[11],
                    "JMS Message content not correct!");
    }
}
```

Outgoing message log output test

```
@ExtendWith(SpringExtension.class)
@SpringBootTest(classes = TestApplication.class)
class JmsSenderLoggerTest {
    @Autowired
    private JmsTemplate jmsTemplate;
    private final JmsLogger jmsSenderLogger = new
```

```
        JmsLogger(LogManager.getLogger("jmsSenderLogger"));


@Test
void shouldLogJmsBody() throws IOException, JMSException {
    ThreadContextUtil.addToContext("requestId",
        LoggingUtil.getRandomHash());
    ThreadContextUtil.addToContext("correlationId",
        "abcdCorrelationId123456");
    ThreadContextUtil.addToContext("userId", "999");

    var destination = new ActiveMQTopic("SOME.TOPIC");
    var replyToDestination = new ActiveMQQueue("SOME.QUEUE");

    AtomicReference<Message> sentMessage = new
        AtomicReference<>();
    this.jmsTemplate.send(destination, (session) -> {
        TextMessage textMessage = session.createTextMessage();
        textMessage.setText("Hello, world - sender test!");
        textMessage.setJMSCorrelationID("ID:someJmsCorrId");
        textMessage.setJMSDestination(destination);
        textMessage.setJMSReplyTo(replyToDestination);
        textMessage.setStringProperty("X-USER-ID",
            ThreadContextUtil.getFromContext("userId"));
        textMessage.setStringProperty("X-Correlation-ID",
            ThreadContextUtil.getFromContext("correlationId"));
        textMessage.setStringProperty("test", "value");
        textMessage.setStringProperty("test2", "value2");
        sentMessage.set(textMessage);
        return textMessage;
    });
    jmsSenderLogger.logMessage(sentMessage.get());

    String file = FileUtils.readFileToString(new
        File("jms_out.log"), StandardCharsets.UTF_8);
    String[] split = file.split("\n");
    assertTrue(split.length > 0);

    String logMessageRow = split[split.length-1];
    String[] logMessageParts = logMessageRow.split("\\|");

    assertEquals(InetAddress.getLocalHost().getHostName(),
        logMessageParts[0],
            "Host name not correct");

    assertEquals("S", logMessageParts[1],
            "Log type not correct");

    DateTimeFormatter formatter =
        DateTimeFormatter.ofPattern("yyyy-MM-dd'T'HH:mm:ss,SSSZ");
    assertDoesNotThrow(() ->
        OffsetDateTime.parse(logMessageParts[2], formatter),
            "Log offset datetime not parseable");

    assertEquals("abcdCorrelationId123456", logMessageParts[3],
            "Correlation ID not correct");

    assertEquals(ThreadContextUtil.getFromContext(„requestId),
        logMessageParts[4],
            "Request ID not correct");
```

```
        assertEquals("999", logMessageParts[5],
                "User ID not correct");

        assertEquals(sentMessage.get().getJMSMessageID(),
            logMessageParts[6],
                "JMS Message ID not correct");

        assertEquals("ID:someJmsCorrId ", logMessageParts[7],
                "JMS Correlation ID not correct");

        assertEquals("topic://SOME.TOPIC", logMessageParts[8],
                "JMS Destination not correct");

        assertEquals("queue://SOME.QUEUE", logMessageParts[9],
                "JMS Reply to Destination not correct");

        assertEquals("{\"X-USER-
            ID\":\"999\",\"test2\":\"value2\",\"test\":\"value\"," +
            "\"X-Correlation-ID\":\"abcdCorrelationId123456\"}",
            logMessageParts[10],
                "JMS Message properties not correct");

        assertEquals("Hello, world - sender test!",
            logMessageParts[11],
                "JMS Message content not correct!");
    }

}
```

Appendix 7 – Incoming message listener registry example for logging incoming messages

```
@Log4j2
public class LoggingJmsListenerEndpointRegistry extends
    JmsListenerEndpointRegistry implements BeanFactoryAware {
    private BeanFactory beanFactory;
    private final MessageHandlerMethodFactory messageHandlerMethodFactory;
    private final JmsLogger jmsLogger;

    public LoggingJmsListenerEndpointRegistry(MessageHandlerMethodFactory
            messageHandlerMethodFactory, JmsLogger jmsLogger) {
        this.messageHandlerMethodFactory = messageHandlerMethodFactory;
        this.jmsLogger = jmsLogger;
    }

    @Override
    public void setBeanFactory(BeanFactory beanFactory) throws BeansException
    {
        this.beanFactory = beanFactory;
    }

    @Override
    public void registerListenerContainer(JmsListenerEndpoint endpoint,
            JmsListenerContainerFactory<?> factory, boolean startImmediately)
    {
        if (endpoint instanceof MethodJmsListenerEndpoint) {
            endpoint = replaceMethodJmsListenerEndpoint(
                  (MethodJmsListenerEndpoint) endpoint);
        }
        super.registerListenerContainer(endpoint, factory, startImmediately);
    }

    private JmsListenerEndpoint replaceMethodJmsListenerEndpoint(
             MethodJmsListenerEndpoint original) {
        MethodJmsListenerEndpoint replacement =
            new LoggingMethodJmsListenerEndpoint();

        replacement.setBean(original.getBean());
        replacement.setMethod(original.getMethod());
        replacement.setMostSpecificMethod(original.getMostSpecificMethod());
        replacement
            .setMessageHandlerMethodFactory(messageHandlerMethodFactory);
        replacement.setBeanFactory(beanFactory);
```

```
        replacement.setId(original.getId());
        replacement.setDestination(original.getDestination());
        replacement.setSelector(original.getSelector());
        replacement.setSubscription(original.getSubscription());
        replacement.setConcurrency(original.getConcurrency());

        return replacement;
    }

    private class LoggingMethodJmsListenerEndpoint extends
             MethodJmsListenerEndpoint {
        @Override
        protected MessagingMessageListenerAdapter
        createMessageListenerInstance() {
            return new LoggingMessageListenerAdapter();
        }
    }

    private class LoggingMessageListenerAdapter extends
             MessagingMessageListenerAdapter {

        @Override
        public void onMessage(final Message message, final Session session)
             throws JMSException {
            populateThreadContextForIncomingMessage(message);
            jmsLogger.logMessage(message);
            super.onMessage(message, session);
            clearThreadContext();
        }

        private void populateThreadContextForIncomingMessage(Message message)
                throws JMSException {
            var httpCorrelationId = message
                .getStringProperty(JmsProperty.X_CORRELATION_ID
                    .getPropertyName());
            if (StringUtils.isBlank(httpCorrelationId)) {
                log.debug("Incoming JMS message does not have a " +
                   + " X-Correlation-ID property set. Generating new UUID.");
                httpCorrelationId = UUID.randomUUID().toString();
            }
            ThreadContext.put("correlationId", httpCorrelationId);
            ThreadContext.put("requestId", LoggingUtil.getRandomHash());

            var userId = message
                .getStringProperty(JmsProperty.X_USER_ID.getPropertyName());
            if (StringUtils.isNotBlank(userId)) {
                ThreadContext.put("userId", userId);
            }
        }

        private void clearThreadContext() {
```

```
            JmsLoggerThreadContextUtil.clearJmsThreadContextValues();
            ThreadContext.remove("userId");
            ThreadContext.remove("requestId");
            ThreadContext.remove("correlationId");
        }
    }
}
```

Appendix 8 – JmsMessageSender interface example

```
public interface JmsMessageSender  {

    Message send(OutgoingJmsMessage message);

    /**
     * Default method to populate the {@link javax.jms.Message} with values
     * from {@link OutgoingJmsMessage}.
     * Also generates or sets the correlation id from ThreadContext to
     * Message property {@link JmsProperty#X_CORRELATION_ID}
     * @param message - message that the fields and properties are populated
     * to.
     * @param outgoingJmsMessage - data object which is holding the values
     * that need to be reflected on the message
     * @throws JMSException - if the JMS Provider fails to set any field or
     * property
     */
    default void populateMessageFieldsAndProperties(Message message,
            OutgoingJmsMessage outgoingJmsMessage) throws JMSException {
        if (StringUtils.hasText(outgoingJmsMessage.getJmsCorrelationId())) {
            message
              .setJMSCorrelationID(outgoingJmsMessage.getJmsCorrelationId());
        }
        if (outgoingJmsMessage.getJmsReplyToDestination() != null) {
            message
              .setJMSReplyTo(outgoingJmsMessage.getJmsReplyToDestination());
        }
        if (StringUtils.hasText(outgoingJmsMessage.getMessageId())) {
            message
              .setStringProperty(JmsProperty.X_MESSAGE_ID.getPropertyName(),
                    outgoingJmsMessage.getMessageId());
        }
        if (StringUtils.hasText(
                 outgoingJmsMessage.getMessageCorrelationId())) {
            message.setStringProperty(
                 JmsProperty.X_MESSAGE_CORRELATION_ID.getPropertyName(),
                    outgoingJmsMessage.getMessageCorrelationId());
        }
        if (outgoingJmsMessage.getContent() != null &&
              StringUtils.hasText(outgoingJmsMessage.getContentType())) {
            message.setStringProperty(
                 JmsProperty.CONTENT_TYPE.getPropertyName(),
                    outgoingJmsMessage.getContentType());
        }
```

```
        var correlationId = ThreadContext.get("correlationId");
        if (!StringUtils.hasText(correlationId)) {
            correlationId = UUID.randomUUID().toString();
            ThreadContext.put("correlationId", correlationId);
        }
        message.setStringProperty(
            JmsProperty.X_CORRELATION_ID.getPropertyName(), correlationId);
    }
}
```

Appendix 9 – DefaultJmsMessageSender and OutgoingJmsMessage implementation examples

```
public class DefaultJmsMessageSender implements JmsMessageSender {
    private final JmsTemplate jmsTemplate;
    private final JmsLogger jmsLogger;

    public DefaultJmsMessageSender(JmsTemplate jmsTemplate,
            JmsLogger jmsLogger) {
        if (jmsTemplate == null) {
            throw new IllegalArgumentException(
                "Argument jmsTemplate cannot be null");
        }
        if (jmsLogger == null) {
            throw new IllegalArgumentException(
                "Argument jmsLogger cannot be null");
        }
        this.jmsTemplate = jmsTemplate;
        this.jmsLogger = jmsLogger;
    }

    /**
     * Default implementation which expects the message creator to just
     * handle the creation of the message body.
     * Fields and properties are populated with overridable {@link
     * #populateMessageFieldsAndProperties(Message, OutgoingJmsMessage)}
     * after message has been created by {@link
     * OutgoingJmsMessage#getJmsMessageCreator()}
     *
     * @param outgoingJmsMessage - object which the message content is
     * created from
     * @return sent message
     */
    @Override
    public Message send(OutgoingJmsMessage outgoingJmsMessage) {
        var messageRef = new AtomicReference<Message>();
        jmsTemplate.send(outgoingJmsMessage.getJmsDestination(), session -> {
            Message message = outgoingJmsMessage
                .getJmsMessageCreator()
                .createMessage(session);
            populateMessageFieldsAndProperties(message, outgoingJmsMessage);
            messageRef.set(message);
            return message;
        });
```

```
        var message = messageRef.get();
        jmsLogger.logMessage(message);
        return message;
    }
}
```

OutgoingJmsMessage implementation example

```
@Data
@Builder
public class OutgoingMqMessage implements OutgoingJmsMessage {
    private Destination jmsDestination;
    private Destination jmsReplyToDestination;
    private String jmsCorrelationId;
    private String messageId;
    private String messageCorrelationId;
    private String content;
    private String contentType;

    @Override
    public MessageCreator getJmsMessageCreator() {
        return (session -> {
            var bytesMessage = session.createBytesMessage();
            bytesMessage
                .writeBytes(content.getBytes(StandardCharsets.UTF_8));
            bytesMessage
                .setStringProperty(JmsProperty.X_USER_ID.getPropertyName(),
                    SecurityContextHolder
                        .getContext()
                        .getAuthentication()
                        .getPrincipal()
                        .toString());
            bytesMessage
                .setStringProperty(JmsProperty.X_SYSTEM_ID.getPropertyName(),
                    "TransactionSystem");
            return bytesMessage;
        });
    }
```

Appendix 10 – Log format and library usage guide for messaging server traffic log collection

JMS logs (ActiveMQ)

Applications should log detailed information about its JMS messages. Messages can be either incoming or outgoing.

JMS incoming/outgoing pattern layouts

Incoming

1. Host – A host's identifier, who is consuming the message.
2. Log type – A value to represent the message type as client (C);
3. Log datetime – ISO8601 format with timezone (*2015-02-05T09:43:37.780 +02:00*);
4. Correlation ID – unique identifcator which is usually from HTTP header (X-Correlation-ID) for identifying processes throughout multiple applications. If not from HTTP header, the producer produced the incoming message while not in request context and generated a new correlation ID. (*9c4d44d7-acf9-4345-90b0-2dc5f2c39996*). Should be retrieved by calling *javax.jms.Message.getString Property("X-Correlation-ID")*
5. Request ID – unique identificator for the current "request", random 8 character string. Generated automatically when message consumer is starting to process the message. (*52g5c9ag*);
6. User ID – authenticated user token represented as string (*123*);
7. JMS Message ID – a unique id for the incoming message. Generated by the JMS provider when sending the message. (*ID:PC123-60303-1599224823894-3:1:54:5:32*)
8. JMS Correlation ID – a correlating id for a message if it's a response to a message. Usually originates from the JMS Message ID of the initial message. Can also be application specific value (ID:PC456-55779-1602707201349-3:1:3156:3:345,

REQ70a289c92a8d4dd5be3b8568d9bc5007, *CLIENTSYSTEM_1613650265472_82754*);

9. JMS Destination – a string representation of the JMS destination from *javax.jms.Message.getJMSDestination()* (*queue://SOME.QUEUE*)
10. JMS Reply to destination – a string representation of the JMS reply to destination from *javax.jms.Message.getJMSReplyTo()*. Should be filled whenever a producer expects a response for the message. (*topic://SOME.TOPIC.RESPONSE*);
11. JMS Properties - JSON-based key-value map of the properties from *javax. jms.Message*;
12. Message body

Outgoing

1. Host – A host's identifier, who is producing the message.
2. Log type – A value to represent the message type as server (S);
3. Log datetime – ISO8601 format with timezone (*2015-02-05T09:43:37.780 +02:00*);
4. Correlation ID – unique identifcator which is usually from HTTP header (X-Correlation-ID) for identifying processes throughout multiple applications. If not in request context, this should be generated. Should be set by calling *javax.jms.Message.setStringProperty("X-Correlation-ID", "9c4d44d7-acf9-4345 -90b0-2dc5f2c39996")* (*9c4d44d7-acf9-4345-90b0-2dc5f2c39996*)
5. Request ID – unique identificator for current request, random 8 character string. If not in request context or it's missing, this may be empty (*52g5c9ag*)
6. User ID – authenticated user token represented as string (*123*);
7. JMS Message ID – a unique id for the outgoing message. Generated by the JMS provider when sending the message. (*ID:PC123-50354-1596054135146-3:1:10:5:152*)
8. JMS Correlation ID – a correlating id for a message if it's a response to a preceding message. Usually originates from the JMS Message ID of the initial message. Can also be application specific value (*ID:PC345-55779-1602707201349-3:1:3156:3:345*, *REQ70a289c92a8d4dd5be3b8568d9bc5007*, *CLIENTSYSTEM _1613650265472_82754*).
9. JMS Destination – a string representation of the JMS destination from *javax.jms.Message.getJMSDestination()* (*queue://SOME.QUEUE*)

10. JMS Reply to destination – a string representation of the JMS reply to destination from *javax.jms.Message.getJMSReplyTo().* Should be filled whenever an outgoing message is expected to retrieve a response for the message. (*topic://SOME.TOPIC.RESPONSE*);
11. JMS Properties - JSON-based key-value map of the properties from *javax. jms.Message*;
12. Message body

Log4j2 config

Pattern properties

```
<Properties>
    <!--.....-->
    <Property name="app">some-system-name</Property>
    <Property name="logDir">${sys:logging.path:-build/logs}</Property>
    <Property name="jmsListenerLoggingFormat">${hostName}\tC
     \t%date{ISO8601}%date{Z}
     \t%replace{%X{correlationId}}{^$}{-}
     \t%replace{%X{requestId}{^$}{-}\t%replace{%X{userId}}{^$}{-}
     \t%X{jmsMessageId}\t%replace{%X{jmsCorrelationId}}{^$}{-}
     \t%X{jmsDestination}\t%replace{%X{jmsReplyToDestination}}{^$}{-}
     \t%replace{%X{jmsProperties}}{^$}{-}\t%replace{%message}{^$}{-}%n
    </Property>
    <Property name="jmsSenderLoggingFormat">${hostName}\tS
     \t%date{ISO8601}%date{Z}
     \t%replace{%X{correlationId}}{^$}{-}
     \t%replace{%X{requestId}{^$}{-}\t%replace{%X{userId}}{^$}{-}
     \t%X{jmsMessageId}\t%replace{%X{jmsCorrelationId}}{^$}{-}
     \t%X{jmsDestination}\t%replace{%X{jmsReplyToDestination}}{^$}{-}
     \t%replace{%X{jmsProperties}}{^$}{-}\t%replace{%message}{^$}{-}%n
    </Property>
    <!--.....-->
</Properties>
```

File appenders

```
<RollingFile name="JmsListenerLoggerRollingFile"
        fileName="${logDir}/${app}-jms-incoming.log"
        filePattern="${logDir}/${app}-jms-incoming.%d{yyyy-MM-dd}.log.gz">
    <PatternLayout pattern="${jmsListenerLoggingFormat}" charset="UTF-8"/>
    <TimeBasedTriggeringPolicy/>
</RollingFile>
<RollingFile name="JmsSenderLoggerRollingFile"
        fileName="${logDir}/${app}-jms-outgoing.log"
        filePattern="${logDir}/${app}-jms-outgoing.%d{yyyy-MM-dd}.log.gz">
    <PatternLayout pattern="${jmsSenderLoggingFormat}" charset="UTF-8"/>
    <TimeBasedTriggeringPolicy/>
</RollingFile>
```

Libraries

Logger library:

1. Add a dependency to your project

```
dependencies {
     implementation 'ee.logging:logger-jms'
 }
```

2. Add Log4j2 patterns and configurations described above

Log incoming message:

If you are using Spring, you should relate to *spring-jms-starter* library for attaching logger automatically to every incoming message listener via autoconfiguration. If not using Spring, you should manually log messages.

```
private final JmsLogger jmsListenerLogger = new
     JmsLogger(LogManager.getLogger("jmsListenerLogger"));

@JmsListener(destination = "SOME.QUEUE")
public void handle(Message message) throws JMSException {
    // fill in thread context
    // ...
    jmsListenerLogger.logMessage(message);
    // ...
    ///do logic
}
```

Log outgoing message:

Since JMS message id gets generated by the JMS provider during send, we need to log the message after that. One example for achieving this would be to use *AtomicReference* which is basically just a volatile reference for retrieving the message object modified by the JMS provider.

```
JmsLogger jmsSenderLogger = new
         JmsLogger(LogManager.getLogger("jmsSenderLogger"));
var messageRef = new AtomicReference<Message>();
jmsTemplate.send(messageObject.getJmsDestination(), session -> {
    Message message = messageObject
        .getJmsMessageCreator()
        .createMessage(session);
    messageRef.set(message);
    return message;
});
var message = messageRef.get();
jmsSenderLogger.logMessage(message);
```

Spring JMS starter library (strongly recommended):

This library provides contracts for sending messages and also contracts for handling persisted messages. It also provides a DefaultJmsMessageSender implementation of sending and logging the message via logger. JmsMessageSender interface has a default method to populate the outgoing message with additional context.

1. Add a dependencies to your project

```
dependencies {
  implementation 'ee.spring:spring-jms-starter'
  implementation 'ee.logging:logger-jms'
  implementation
     'org.springframework.boot:spring-boot-starter-activemq'
}
```

2. Add Log4j2 patterns and configurations described above

3. Enable JMS listener logging which attaches logging capabilities to every *@JmsListener* annotated listener method to log incoming messages. Add config property:

```
jms.loggingListenersEnabled=true
```

4. If you want to omit message body logging for specific destinations you are listening or producing messages to, add the destination to configuration's exclusion list:

```
jms.destinationsExcludedFromBodyLogging=some.queue,some.topic
```

Receiving messages

To receive messages from JMS destination, just create a spring annotated listener method.

```
@JmsListener(destination = "SOME.QUEUE")
public void handle(Message message) {
    //Do logic
}
```

Sending messages

Create your own implementation of *JmsMessageSender* or use *DefaultJmsMessage Sender* and inject a bean instance of it into Spring's context:

```
@Log4j2
@AllArgsConstructor
public class MyJmsMessageSender implements JmsMessageSender {
    private final JmsTemplate jmsTemplate;
    private final JmsLogger jmsLogger;
    @Override
    public Message send(OutgoingJmsMessage outgoingJmsMessage) {
        var messageRef = new AtomicReference<Message>();
        jmsTemplate
            .send(outgoingJmsMessage.getJmsDestination(), session -> {
            Message message = outgoingJmsMessage
                .getJmsMessageCreator()
                .createMessage(session);
            populateMessageFieldsAndProperties(
                  message, outgoingJmsMessage);
            messageRef.set(message);
            return message;
        });
        var message = messageRef.get();
        jmsLogger.logMessage(message);
        return message;
    }
}
```

Create your own implementation of *OutgoingJmsMessage*:

```
@Data
public class MyOutgoingJmsMessage implements OutgoingJmsMessage {
    private Destination jmsDestination;
    private Destination jmsReplyToDestination;
    private String jmsCorrelationId;
    private String messageId;
    private String messageCorrelationId;
    private String content;
    private String contentType;

    @Override
    public MessageCreator getJmsMessageCreator() {
        return session -> {
            var msg = session.createTextMessage();
            msg.setText(this.getContent());
            return msg;
        };
    }
}
```

Call out your *JmsMessageSender* implementation to send the message:

```
@Service
public class MyService {
    private final JmsMessageSender myJmsMessageSender;

    @Autowired
    public MyService (JmsMessageSender myJmsMessageSender) {
        this.myJmsMessageSender = myJmsMessageSender;
    }

    public void myMethodToSendMessage(OutgoingJmsMessage message) {
        myJmsMessageSender.send(message);
    }
}
```