Payment Services

Payment Services Event Bus
Software Architecture Document

Version <0.3>

Revision History

Date	Version	Description	Author
04-22/2015	0.1	Initial Release	Travis White
19th January 2016	0.2	Updated after PoC exercise using RabbitMQ in AWS	George Harley
3rd February 2016	0.3	Incorporating initial feedback. Added event headers from EA's event specification document.	George Harley

Table of Contents

Software Architecture Document

1. Introduction

1.1 Purpose

This document provides a comprehensive architectural overview of the system, using a number of different architectural views to depict different aspects of the system. It is intended to capture and convey the significant architectural decisions which have been made on the system.

The intent of the Payment Services Event Bus Software Architecture Document is to provide the guide for developing, implementing, and evaluating the correctness of the overall system design, as well as the individual components relative to their placement within the system.

1.2 Scope

The Payment Services Event Bus Software Architecture Document provides a guiding standard for development and implementation of the individual components and supporting infrastructure. It is the measure of technical correctness of the final system and system components.

1.3 Definitions, Acronyms, and Abbreviations

Term	Meaning
PSEB	Payment Services Event Bus
WMQ	WebSphere Message Queue
JMS	Java Message Service
AMQP	Advanced Message Queueing Protocol
AMQPS	Advanced Message Queueing Protocol Secure
NAS	Network Attached Storage
HA	High Availability
PCF	Pivotal Cloud Foundry

1.4 References

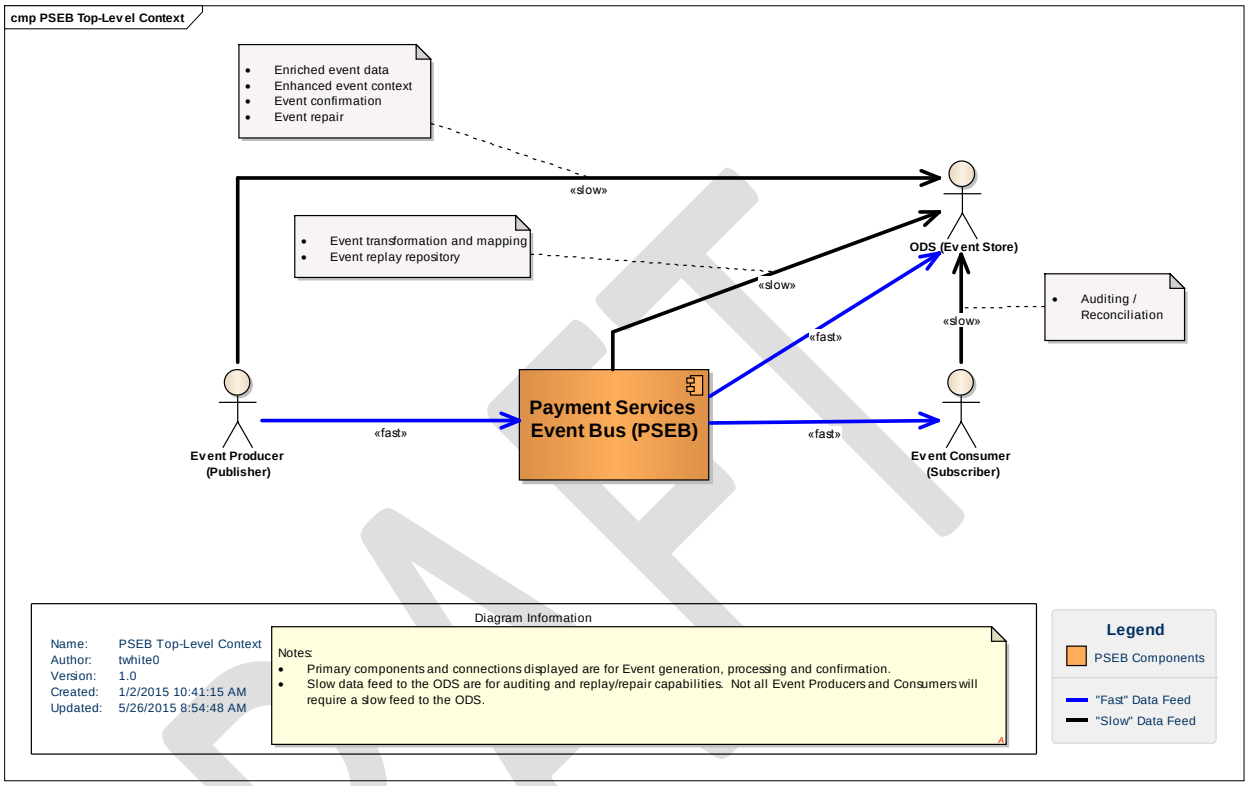
RabbitMQ	http://www.rabbitmq.com
AMQP version 0.9.1	https://www.rabbitmq.com/resources/specs/amqp0-9-1.pdf
MapDB	http://www.mapdb.org
SQLite	https://www.sqlite.org
Spring Integration	http://projects.spring.io/spring-integration
Dozer	http://dozer.sourceforge.net

1.5 Overview

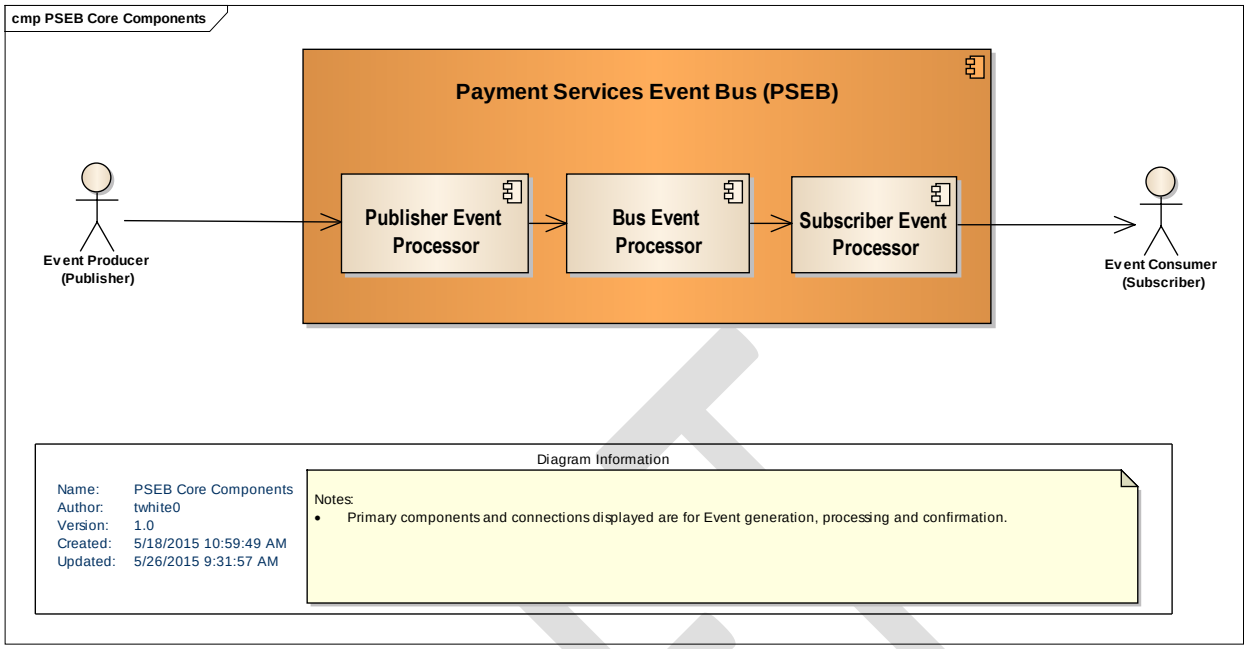
*[This subsection describes what the rest of the **Software Architecture Document** contains and explains how the **Software Architecture Document** is organized.]*

2. Architectural Representation

*[This section describes what software architecture is for the current system, and how it is represented. Of the **Use-Case**, **Logical**, **Process**, **Deployment**, and **Implementation Views**, it enumerates the views that are necessary, and for each view, explains what types of model elements it contains.]*



Basic EA diagram outlining Publishers, Event Store, Subscribers, and ODS surrounding the PSEB.


cmp PSEB Core Components
Payment Services Event Bus (PSEB)
Publisher Event Processor
Bus Event Processor
Subscriber Event Processor
Event Producer (Publisher)
Event Consumer (Subscriber)
Diagram Information
Name: PSEB Core Components
Author: twhite0
Version: 1.0
Created: 5/18/2015 10:59:49 AM
Updated: 5/26/2015 9:31:57 AM
Notes:
Primary components and connections displayed are for Event generation, processing and confirmation.

3. Assumptions

RabbitMQ will be the messaging middleware for the event bus. RabbitMQ broker instances will be deployed and run as clusters on virtualized servers that utilize Discover's nascent Infrastructure as a Service (IaaS). These virtualized servers will be hosted in Discover data centres.

Event processing logic will be implemented with Spring Integration. The processing will be built as Spring Boot applications and packaged for deployment in standalone executable Jar files. These Spring Boot applications will be capable of running on Discover IaaS instances or else in Pivotal Cloud Foundry spaces. Environment-specific configuration will be encapsulated in external properties files rather than in application code.

Event processing nodes will only contain integration logic not client application logic.

Event message routing that does not require access to the payload will make use of RabbitMQ message routing functionality. Event message routing that depends on payload will be considered to be integration logic and so be carried out using Spring Integration.

Events will be published and consumed as JSON documents.

The event bus will convey event messages to subscribers in their published format. Where a subscriber application has a requirement to receive an event message in a different format the message will be routed to an event processing node that will carry out the necessary transformation and subsequent routing of the transformed message to the specific subscriber(s).

Publishers and subscribers will be presented with an AMQP/AMQPS interface.

The Data Services ODS will be the event store for the bus.

4. Architectural Quality Requirements

[This section describes the software requirements and objectives that have some significant impact on the architecture; for example, safety, security, modifiability, privacy, use of an off-the-shelf product, portability, distribution, and reuse. It also captures the special constraints that may apply: design and implementation strategy, development tools, team structure, schedule, legacy code, and so on.]

4.1 Availability

The event bus must provide a very high level of availability to clients; ideally meeting the "five nines" target where overall system down-time is less than five minutes per year. In the context of the event bus availability should be judged using the following criteria:

- It should be possible for new connections to be made to the bus for the purposes of publishing or consuming event messages.
- All messages published to the event bus receive a confirmation/acknowledgement if requested in the channel set up.
- All messages consumed by subscribers to the event bus should be well formed and valid against any message structure agreed during the subscriber's on-boarding process.
- No event messages will be lost. All queued messages will also be persisted to disk so that they are available after a broker instance restart. If a message cannot be successfully processed then it will be routed to a destination agreed during the client's on-boarding process. Typically this will be one or more dead letter queues where error logging and fault analysis can be carried out.
- All bus configuration settings will be durable and capable of surviving restarts. This applies to broker instances (exchange definitions, queue definitions, etc.) and event processor JVMs.
- Event processing logic will not be deployed and run as single instance applications (and therefore be potential single points of failure) but instead will be deployed as at least two instances, possibly more depending on the rate at which event messages flow through a particular integration path.

4.2 Scalability

The event bus must be capable of meeting increases in event message throughput. Part of satisfying this requirement will be met by incorporating sufficient head room in the event bus processing capacity to cope with occasional spikes in event publishing rates. In addition, it must be straightforward to add new software and hardware resources to the event bus that will provide it with additional queueing and processing capacity when load trends indicate it is appropriate to do so. Similarly, it must be straightforward to remove software and hardware resources as necessary.

The event bus should remain available during any operations to alter its queueing and processing capacity.

4.3 Reusability

The event bus shall support all DFS payment brands including Diners Club, Discover, and the PULSE networks.

4.3.1 Hardware Reuse

The event bus will not require specialised hardware resources and will instead run on hardware that is common across the CPP. Instances of the RabbitMQ broker software will execute on standard IaaS servers. Instances of event processing software will run in standalone Java VMs capable of being deployed and run in both the IaaS and PCF environments.

4.3.2 Software Reuse

The software elements of the event bus can be broken down into the following categories:

- Messaging oriented middleware
- Event processing logic
- Client access libraries

4.3.2.1 Message oriented Middleware

The messaging oriented middleware underpinning the event bus will be RabbitMQ, a mature open source message broker commercially supported by Pivotal which is in wide use throughout the software industry. It is an implementation of advanced message queueing protocol (AMQP) which is an open standard governed under the auspices of OASIS. Other implementations of AMQP are available from vendors including, but not limited to, Microsoft (Windows Azure Service Bus), IBM (MQ Light), and Apache (Qpid and ActiveMQ). RabbitMQ is available on most of the major operating systems – AIX is not supported – and has client bindings available across a very large number of popular programming languages including Java, C, C++, C#, Python, Ruby and many more. RabbitMQ is written in the open source Erlang language and runs as an application on the Erlang runtime. The Erlang runtime is a stable, mature platform designed for building large-scale, distributed, fault-tolerant systems. Originally only used in the telecommunications industry, it is now found in many software domains.

4.3.2.2 Event Processing Logic

Processing of event messages that requires access to and manipulation of the message payload (e.g. message transformation or enrichment) will be carried out using Java technologies already prevalent in Discover production systems such as Spring Integration and Spring Boot. No specialist programming language skills or build systems will be necessary when working on these components.

4.3.2.3 Client Access Libraries

Client access to the event bus covers the areas of publishing messages, subscribing to receive specific types of messages and the process of actually receiving them. The software required to accomplish these functionalities will be written in Java and made available in binary form as libraries that can be added to the dependencies of a client application using standard build and dependency management frameworks.

4.4 Security

The event bus shall be PCI-DSS compliant.

The event bus shall validate the identity of client applications (publishers and consumers) and enforce fine grained authorisation to bus resources. In this context the term “resources” refers to RabbitMQ virtual hosts (used in order to provide multi-tenancy), exchanges, queues, and the management console.

Event bus clients will have the minimal set of permissions necessary to publish and consume messages from RabbitMQ. Event bus clients will not be authorised to create or destroy any RabbitMQ resources.

Where necessary to do so, RabbitMQ virtual hosts will be used to maintain isolation between different PSEB clients and their respective messages. A theoretical example of this would be the creation of a RabbitMQ virtual host whose users, exchanges, and queues were used for conveying messages that needed to be kept separate from other message types flowing on the event bus. Resources in a dedicated virtual host would not be accessible to resources defined in other virtual hosts.

The event bus shall integrate with remote security services developed for CPP that can provide centralised authentication, authorisation, and access control data.

The event bus shall log all client access and operations for the purposes of non-repudiation and auditing.

It is expected that sensitive information will be tokenised before being included in any event messages published to the bus. Where this condition is met the need to enable secure encrypted connections to the event bus is essentially removed. Should circumstances occur where it is not possible to tokenise or otherwise disguise sensitive data then client connections to the event bus will be made using AMQPS – the AMQ protocol over an SSL transport layer. Ultimately, the event bus must be capable of supporting AMQPS so that the choice between flowing messages in the clear or in encrypted form can be made on a client by client basis.

All communications between distinct RabbitMQ *clusters* will be made using AMQPS. An example of cluster to cluster communication would be the routing of messages from one data centre to another.

Persistent messages temporarily stored in the file system will be made secure from unauthorised access.

5. Size and Performance

[A description of the major dimensioning characteristics of the software that impact the architecture, as well as the target performance constraints.]

5.1 Software Dimensioning Characteristics

Each of the following sections deals with a software aspect of the event bus software that can impact on its runtime performance. In this context the term performance covers the rate of message throughput, the latency of event messages, and the availability of the system.

5.1.1 Message Delivery Guarantees

Message publishing over a transactional channel is synchronous and can result in the publishing application spending a great deal of time blocked while it awaits an acknowledgement signal from the RabbitMQ message broker. For this reason the event bus will eschew transactional publishing for the more lightweight "Publisher Confirms" alternative where message acknowledgement signals are returned asynchronously. Using this approach adds responsibility to the client inasmuch as it will need to maintain state about all published messages for which an asynchronous acknowledgement has not been received. This functionality will be part of the event bus client publishing library.

5.1.2 Acknowledgement of Consumed Messages

Similar to publishing, message consumption with transactions has an adverse impact on the rate at which messages can be consumed from a queue. To improve on that, client consumers will use basic acknowledgements with a pre-fetch count greater than one. In that scenario consumers will effectively batch up their acknowledgements to RabbitMQ so that the amount of network communication is reduced. The optimal value of the pre-fetch count will depend on a number of factors such as the network speed and the size of the event messages and so will be configurable through use of an external property.

5.1.3 High Availability Queue Mirroring

For any highly available queue in a RabbitMQ cluster the more duplicates/mirrors that exist in the cluster the slower the rate at which messages can be published and consumed. This is a consequence of the amount of intra-cluster communication required to keep all of the mirrors in sync. Ideally, each queue in the event bus will be duplicated to only one other instance of the RabbitMQ cluster which provides fault tolerance with minimal impact on messaging throughput.

5.1.4 Increasing Concurrency in Event Processing Nodes

The AMQP Inbound Channel Adapter that will be a key component of the Spring Integration flow developed for each event processing node can be configured to consume messages with varying degrees of concurrency. Configuring the adapter for a higher number of concurrent consumers has the potential to increase message throughput in the bus.

5.1.5 Automatic Message Expirations

In the event that subscriber applications become unavailable, either through an outage or a network partition, messages will be left in RabbitMQ queues and not be consumed. Specifying an automatic expiry time for each message in a queue will help mitigate against queues reaching their maximum depth should subscribers be unavailable for long periods of time. In the interests of not losing any messages all bus queues should be created with a dead letter exchange (DLX) specified so that expired messages are not discarded but rather are automatically routed to the DLX and ultimately to some dead letter queue for error handling.

5.2 Target Performance Constraints

The event bus will be able to process approximately 2000 event messages per second at steady state, rising to 4000 event messages per second at peak times.

6. Use-Case View

[This section lists use cases or scenarios from the use-case model if they represent some significant, central functionality of the final system, or if they have a large architectural coverage—they exercise many architectural elements or if they stress or illustrate a specific, delicate point of the architecture.]

The list of architecturally significant use cases is as follows:

- Publishing of an event
- Receiving of a subscribed event
- Transformation of event en route to subscriber
- Content filtering of event en route to subscriber
- Authentication of event publisher/subscriber
- Routing of events between data centres
- Adding a new subscriber
- Removing a subscriber

Each of these use cases are described in more detail below.

6.1 Publishing of an Event

6.1.1 Level

This is a system level use case

6.1.2 Success Scenario

1. Publisher application invokes publish method exposed by Event Bus Client Library
2. Event Bus Client Library authenticates to RabbitMQ broker
3. Event Bus Client Library creates message payload and headers
4. Event Bus Client Library adds constructed message to collection backed by persistence solution provided by publisher application (e.g. MapDB or SQLite for file based storage)
5. Event Bus Client Library invokes RabbitMQ method to send message contents to the target exchange
6. RabbitMQ sends an acknowledgement back to Event Bus Client Library confirming that the published message has been safely persisted and routed by the receiving exchange
7. Event Bus Client Library removes the acknowledged message from the persisted collection of messages awaiting an acknowledgement

6.1.3 Extensions

2a: Event Bus Client Library is already using an authenticated connection to RabbitMQ

- Existing connection can be reused without need to authenticate
- Returns to Success Scenario at step 3

6a: Event Bus Client Library does not receive an acknowledgement for the published message indicating that the message may not have reached the target RabbitMQ broker

- Message object eventually expires from persisted collection
- Message expiry handler logs failure and raises client exception

6b: Event Bus Client Library receives a negative acknowledgement from RabbitMQ which will typically indicate that the target exchange does not exist

- Message is removed from the persisted collection
- Message expiry handler logs failure and raises client exception

6c: Event Bus Client Library receives a basic return from RabbitMQ indicating that the message was received by the broker. However in this scenario the message cannot be routed by the exchange (typically because nothing has bound to the exchange to receive this kind of message).

- If the exchange has an "alternate exchange" defined for it – as all RabbitMQ exchanges used in the event bus should – then the message will automatically be routed to the alternate exchange
- The alternate exchange will route the message to the event bus dead letter exchange (DLX) for error logging and queueing by RabbitMQ

6.2 Receiving of a subscribed event

6.2.1 Level

This is a system level use case

6.2.2 Success Scenario

1. Subscriber application supplies message consumer function to Event Bus Client Library. This will typically be an instance of a Java 8 functional interface such as **java.util.function.Consumer<String>** capable of accepting a JSON string. Setting up the Event Bus Client Library with this operation will be done via Spring configuration
2. Event Bus Client Library authenticates to RabbitMQ broker
3. Event Bus Client Library registers message listener method for one or more queues of interest
4. Event Bus Client Library message listener method is invoked with a message object consumed from RabbitMQ queue of interest
5. Event Bus Client Library extracts the JSON string payload from the received message
6. Event Bus Client Library invokes the Subscriber application's consumer function

6.2.3 Extensions

2a: Event Bus Client Library is already using an authenticated connection to RabbitMQ

- Existing connection can be reused without need to authenticate
- Returns to Success Scenario at step 3

6a: Subscriber application's consumer function throws an exception during processing of message payload and the message "redelivered" flag is false

- Event Bus Client Library message listener method catches exception thrown by consumer
- Event Bus Client Library checks message's "redelivered" flag
- Event Bus Client, on finding the message "redelivered" flag to be *false*, re-throws the original exception. This triggers a basic rejection of the message.
- RabbitMQ receives the message rejection and places the message back on the queue

6b: Subscriber application's consumer function throws an exception during processing of message payload and the message "redelivered" flag is true

- Event Bus Client Library message listener method catches exception thrown by consumer
- Event Bus Client Library checks message's "redelivered" flag
- Event Bus Client, on finding the message "redelivered" flag to be *true*, throws a new **org.springframework.amqp.AmqpRejectAndDontRequeueException** (wrapping the original processing exception). This triggers a basic rejection of the message with the "requeue" flag set to false
- RabbitMQ receives the message rejection and, because "requeue" is false, routes the message to the queue's dead letter exchange (DLX) for error logging and queueing by RabbitMQ

6.3 Transformation of Event en route to Subscriber

6.3.1 Level

This is an internal use case that can be included in use cases "Publishing of an Event" and "Receiving of Subscribed Event".

6.3.2 Success Scenario

1. Event Processor Application (typically implemented as a standalone Spring Boot application developed with Spring Integration) consumes message from pre-configured RabbitMQ queue
2. Event Processor Application validates payload of received message against JSON schema
3. Event Processor Application parses message payload into a valid Java object graph using a Spring Integration primitive
4. Event Processor Application service activator uses contents of message object graph to build a separate Java object graph whose types are based on the structure of the new message. A Java bean mapping framework such as Dozer or equivalent could be employed for this step
5. Event Processor Application serialises new Java object graph into a JSON string using a Spring Integration primitive
6. Event Processor Application builds a new message from the new JSON string making use of headers from the original message
7. Event Processor Application publishes new message to a pre-configured RabbitMQ queue

6.3.3 Extensions

2a: Received message payload is invalid

- Event Processor Application catches exception thrown by message validator

- Event Processor Application throws a new **org.springframework.amqp.AmqpRejectAndDontRequeueException** (wrapping the original validation exception). This triggers a basic rejection of the message with the "requeue" flag set to false
- RabbitMQ receives the message rejection and, because "requeue" is false, routes the message to the queue's dead letter exchange (DLX) for error logging and queueing by RabbitMQ

4a: Error building new Java object graph for resulting message

- Event Processor Application catches exception thrown by object mapper
- Event Processor Application throws a new **org.springframework.amqp.AmqpRejectAndDontRequeueException** (wrapping the original validation exception). This triggers a basic rejection of the message with the "requeue" flag set to false
- RabbitMQ receives the message rejection and, because "requeue" is false, routes the message to the queue's dead letter exchange (DLX) for error logging and queueing by RabbitMQ

6.4 Content filtering of event en route to subscriber

6.4.1 Level

This is an internal use case that can be included in use cases "Publishing of an Event" and "Receiving of Subscribed Event".

6.4.2 Success Scenario

1. Event Processor Application (typically implemented as a standalone Spring Boot application developed with Spring Integration) consumes message from pre-configured RabbitMQ queue
2. Event Processor Application validates payload of received message against JSON schema
3. Event Processor Application parses message payload into a valid Java object graph using a Spring Integration primitive
4. Event Processor Application's Spring Integration filter executes logical test against one or more field values from the message Java object graph with the outcome of the test being that the message should not proceed any further in the Spring Integration flow. The Java object graph is discarded.

6.4.3 Extensions

4a: Spring Integration filter test is passed signifying message can continue in Spring Integration flow

- Java object graph is passed from filter to next Spring Integration node
- Event Processor Application Spring Integration primitive serialises Java object graph into JSON form
- Event Processor Application builds a new RabbitMQ message from the new JSON string making use of headers from the original message
- Event Processor Application publishes new message to a pre-configured RabbitMQ queue

6.5 Authentication of event publisher/subscriber

6.5.1 Level

This is an internal use case that can be included in use cases "Publishing of an Event" and "Receiving of Subscribed Event".

6.5.2 Success Scenario

1. Event Bus Client Library attempts to connect to RabbitMQ broker instance over AMQP/AMQPS with credentials in the form of a username and password which have been read from an external properties file located in a secure location on disk
2. RabbitMQ broker instance issues a remote call to the Security Authentication Service API with the supplied username and password
3. Security Authentication Service API validates user credentials
4. RabbitMQ completes AMQP/AMQPS connection with Event Bus Client Library

6.5.3 Extensions

3a: Security Authentication Service API response reveals user credentials to be invalid

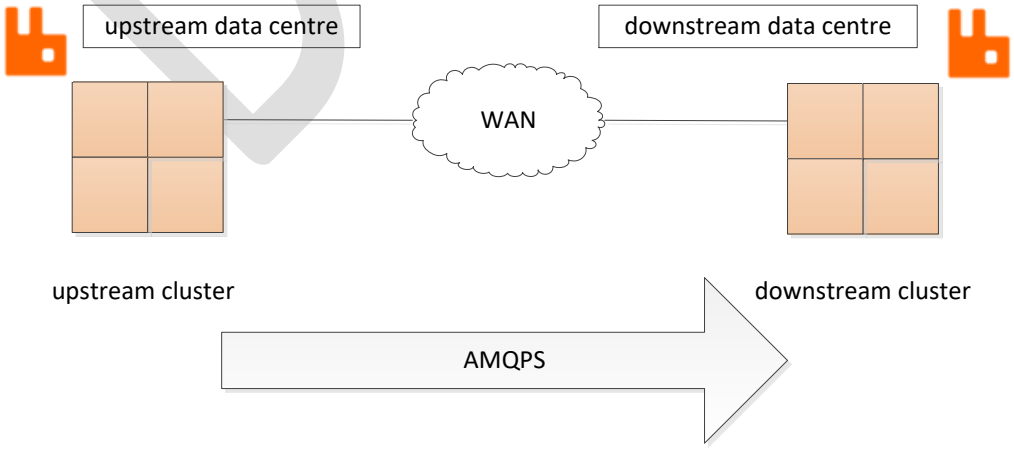
- RabbitMQ aborts connection attempt
- Event Bus Client Library receives error response from RabbitMQ

6.6 Routing of events between data centres

6.6.1 Level

This is an internal use case that will be transparent to client applications using the event bus.

6.6.2 Success Scenario



1. Administrator creates a RabbitMQ "federation upstream" definition in the downstream cluster referencing any node in the upstream cluster
2. Administrator creates a RabbitMQ "federation policy" definition in the downstream cluster referencing the newly created "federation upstream"
3. Subscriber application instance in downstream data centre is started
4. Event Bus Client Library used by subscriber application in downstream data centre authenticates to downstream RabbitMQ cluster and awaits messages
5. Subscriber application instance in upstream data centre is stopped

6.7 Adding a new subscriber

6.7.1 Level

This is a system level use case.

6.7.2 Success Scenario

1. As part of the Subscriber on-boarding process the structure of received event messages is agreed.
2. As part of the Subscriber on-boarding process any limitations on the time that received messages are allowed to remain in the consumer queue before being automatically expired are agreed.
3. As part of the Subscriber on-boarding process the error processing of undeliverable event messages is agreed. This will typically involve identifying one or more dead letter queues (DLQ) that undeliverable messages will automatically be sent to. In addition, any scheduled (e.g. batch) processing of dead lettered messages will be agreed upon.
4. Subscriber application is developed using Event Bus Client Library message consumption functionality.
5. A new dedicated RabbitMQ queue is created for the new subscriber's client library to consume messages from. The new queue must be configured with a dead letter exchange (DLX) that un-consumable messages are automatically routed to. If a message expiry time has also been agreed upon then this will be configured for the new queue. The new queue will not be bound to any exchanges at this stage.
6. Subscriber application is started so that basic connectivity with its associated RabbitMQ queue (which is still not bound to the existing RabbitMQ routing topology for the event type of interest) may be verified.
7. One or more test messages will be sent to the new subscriber queue using either the RabbitMQ management console or a call to its underlying REST API. The subscriber application's event client library should enable the subscriber application to successfully consume these test messages from its queue.
8. The new subscriber queue will be bound to an exchange in the routing topology for the event type of interest. Event messages received at the bound exchange may now start to flow to the queue.

6.7.3 Extensions

5a: Subject to the existing routing topology for the desired event type a new RabbitMQ exchange may also need to be created.

- New exchange is created.
- The new exchange must be configured with an alternate exchange (AX) that un-routable messages will automatically be sent to. This may be the same as the DLX associated with the subscriber application's queue.
- The new exchange may be bound to an upstream exchange.

5b: The new subscriber application needs to receive the event message in a structure that requires transformation of the event message.

- A new event processor application will be developed using Spring Integration. The application will be built as a standalone Spring Boot Jar that reads from a queue and publishes to an exchange which the subscriber application's queue will be bound to.
- A new dedicated RabbitMQ queue will be created for the new event processor to consume messages from. The new queue must be configured with a dead letter exchange (DLX) that un-consumable messages are automatically routed to. If a message expiry time has also been agreed upon then this will be configured for the new queue. The new queue will not be bound to any exchanges at this stage.
- The new event processor application is started to verify basic connectivity with its inbound queue. One or more test messages may be sent to the queue.

8a: The new subscriber application needs to receive the event message in a structure that requires transformation of the event message.

- Once the subscriber application's queue has been bound to its upstream exchange the event processor's queue can also be bound to its upstream exchange.

6.8 Removing a subscriber

6.8.1 Level

This is a system level use case.

6.8.2 Success Scenario

1. Subscriber application's queue is unbound from its upstream exchange. The queue (and therefore the subscriber application) will immediately cease to receive any new event messages.
2. Once any remaining messages have been consumed from the subscriber application's (unbound) queue the queue may be deleted. This may be done using either the RabbitMQ console or its REST API.

6.8.3 Extensions

1a: The subscriber application required custom event transformation carried out on received messages.

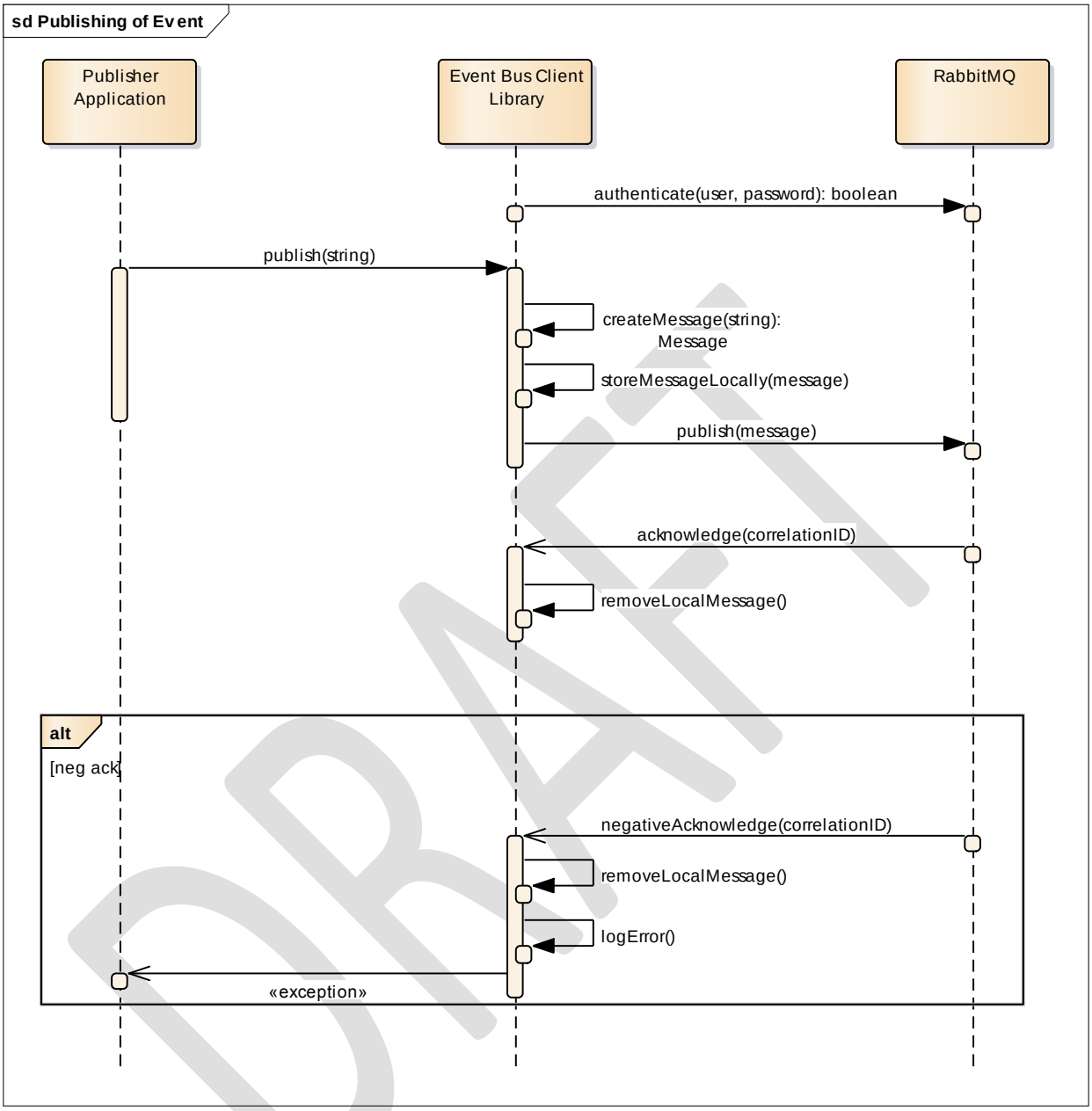
- The transformation event processor's inbound queue is unbound from its upstream exchange. This unbinding step should be carried out first, before the subscriber application's queue is unbound.
- Once any remaining messages have been consumed from the event processor's inbound queue the event processor application should be gracefully stopped
- The event processor's inbound queue may now be deleted.

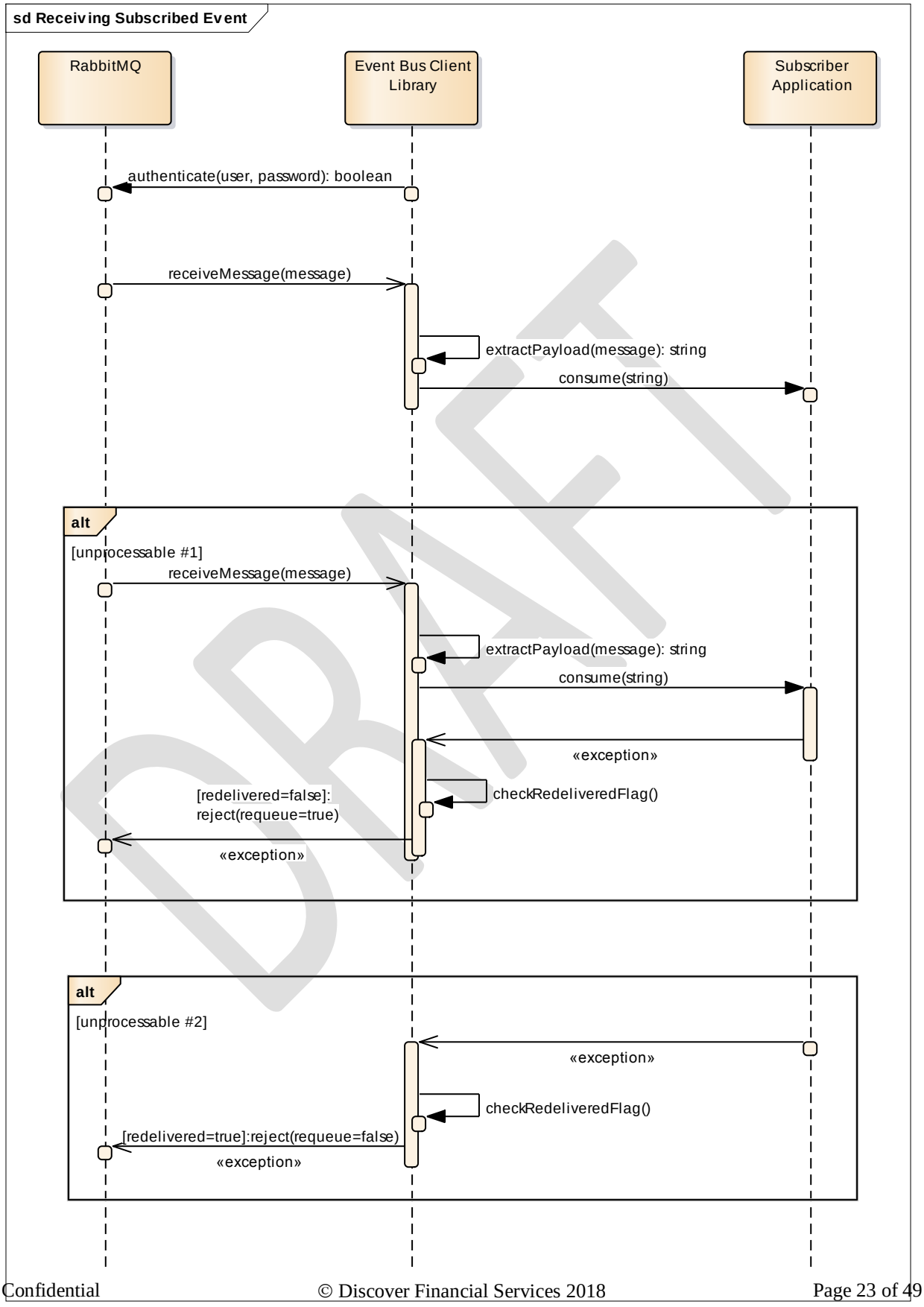
2a: The subscriber application was the sole user of a RabbitMQ exchange which is now deemed no longer necessary.

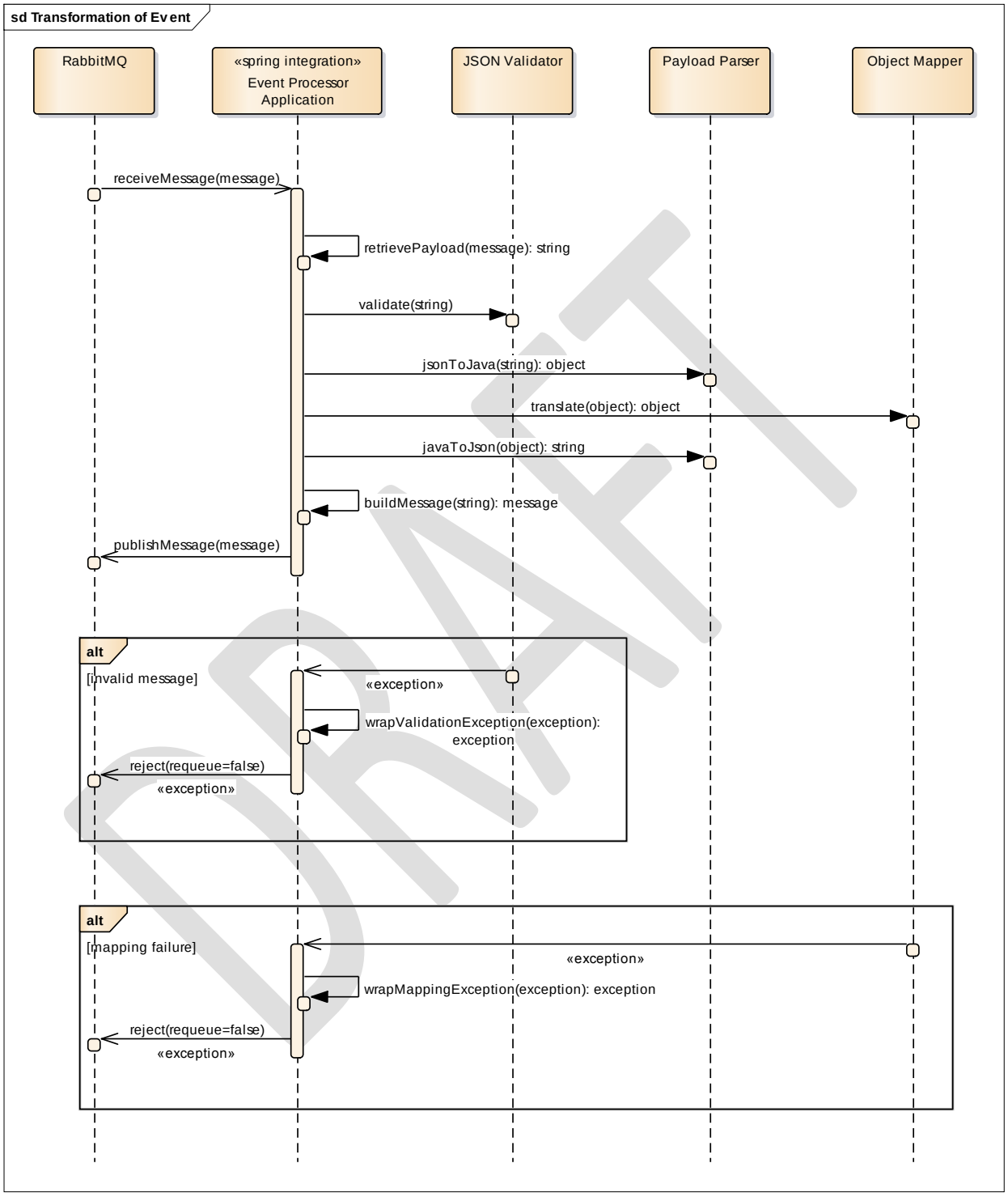
- The exchange may be deleted using the RabbitMQ management console or its REST API.

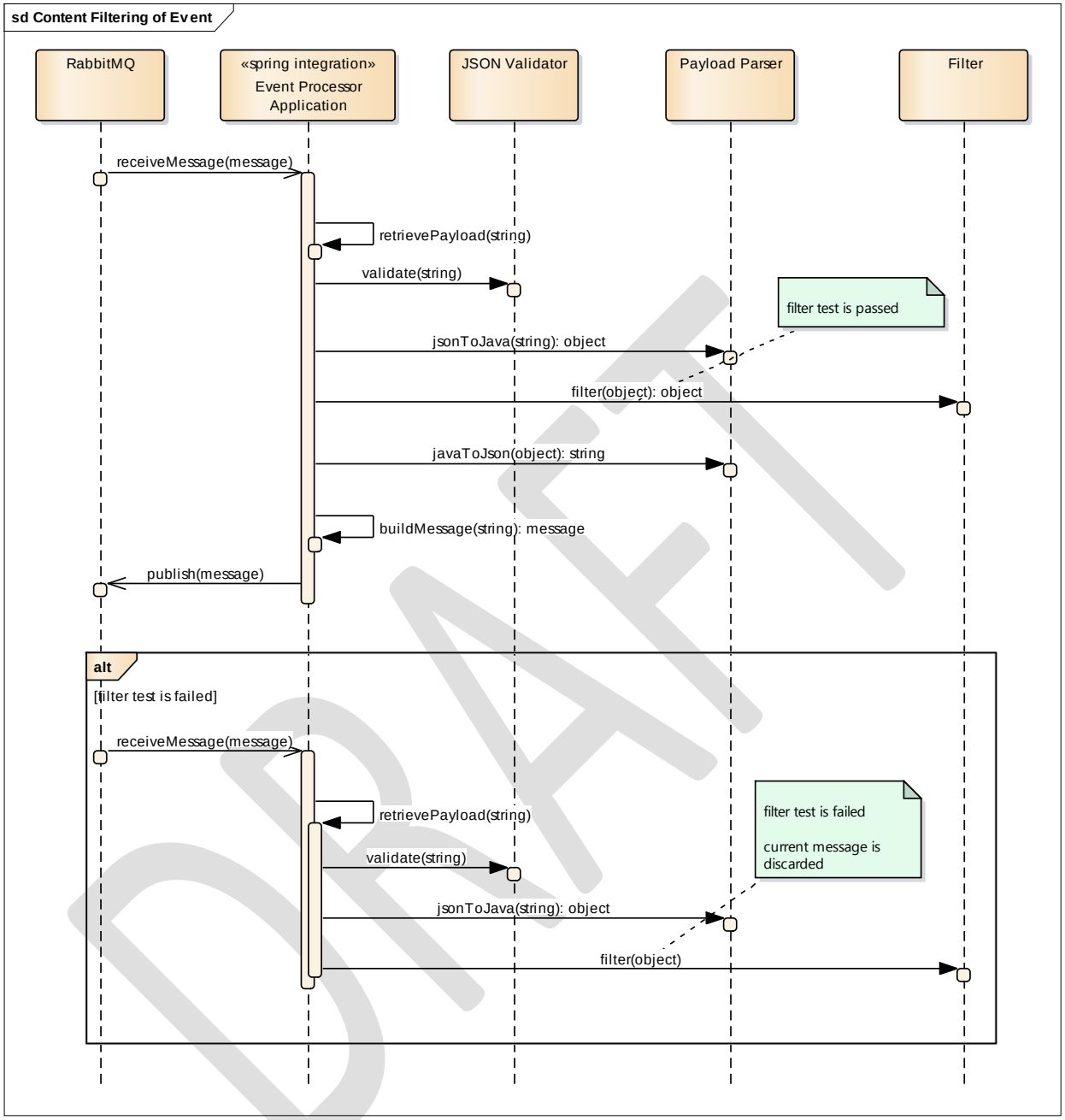
6.9 Use-Case Realizations

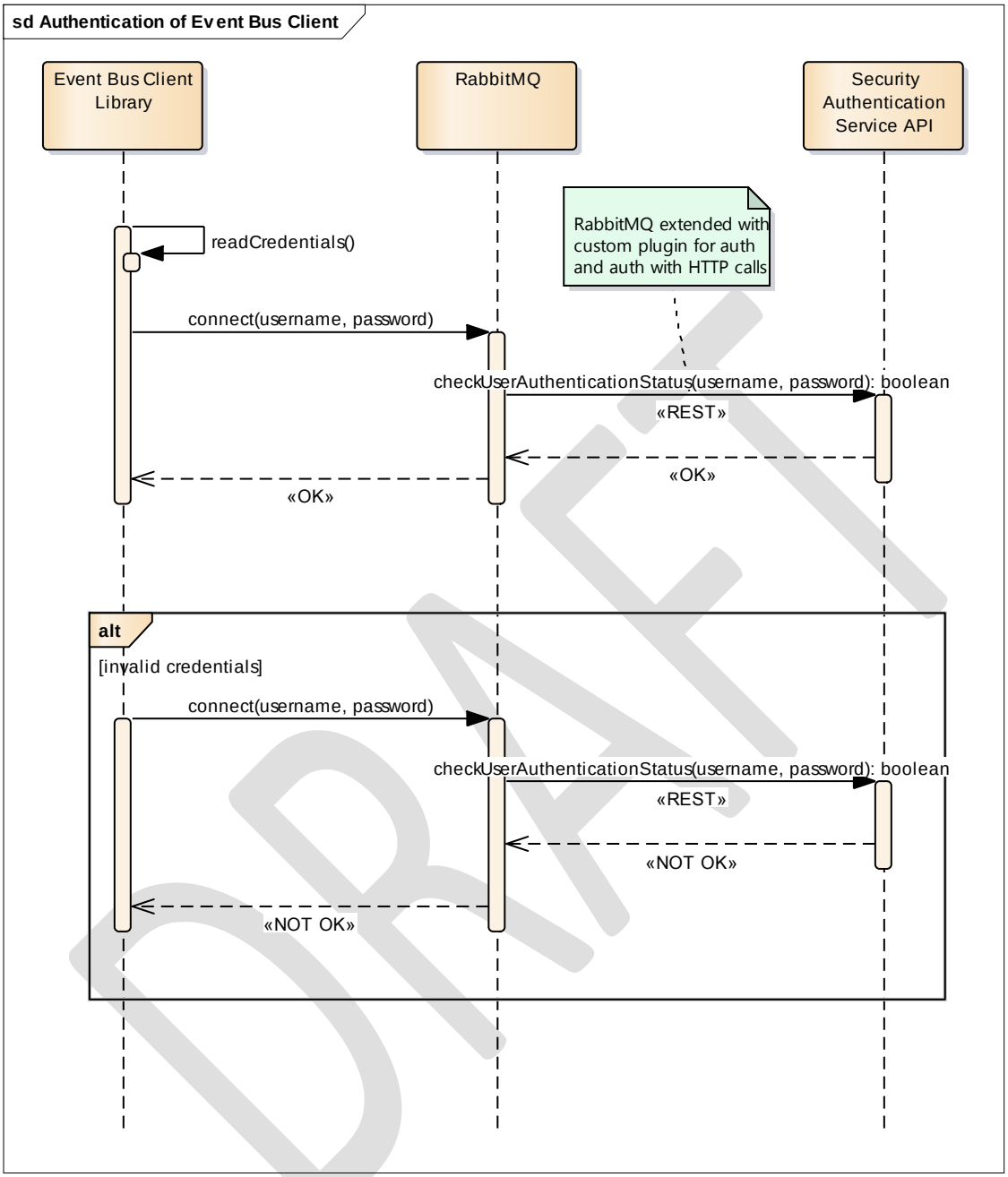
[This section illustrates how the software actually works by giving a few selected use-case (or scenario) realizations, and explains how the various design model elements contribute to their functionality.]

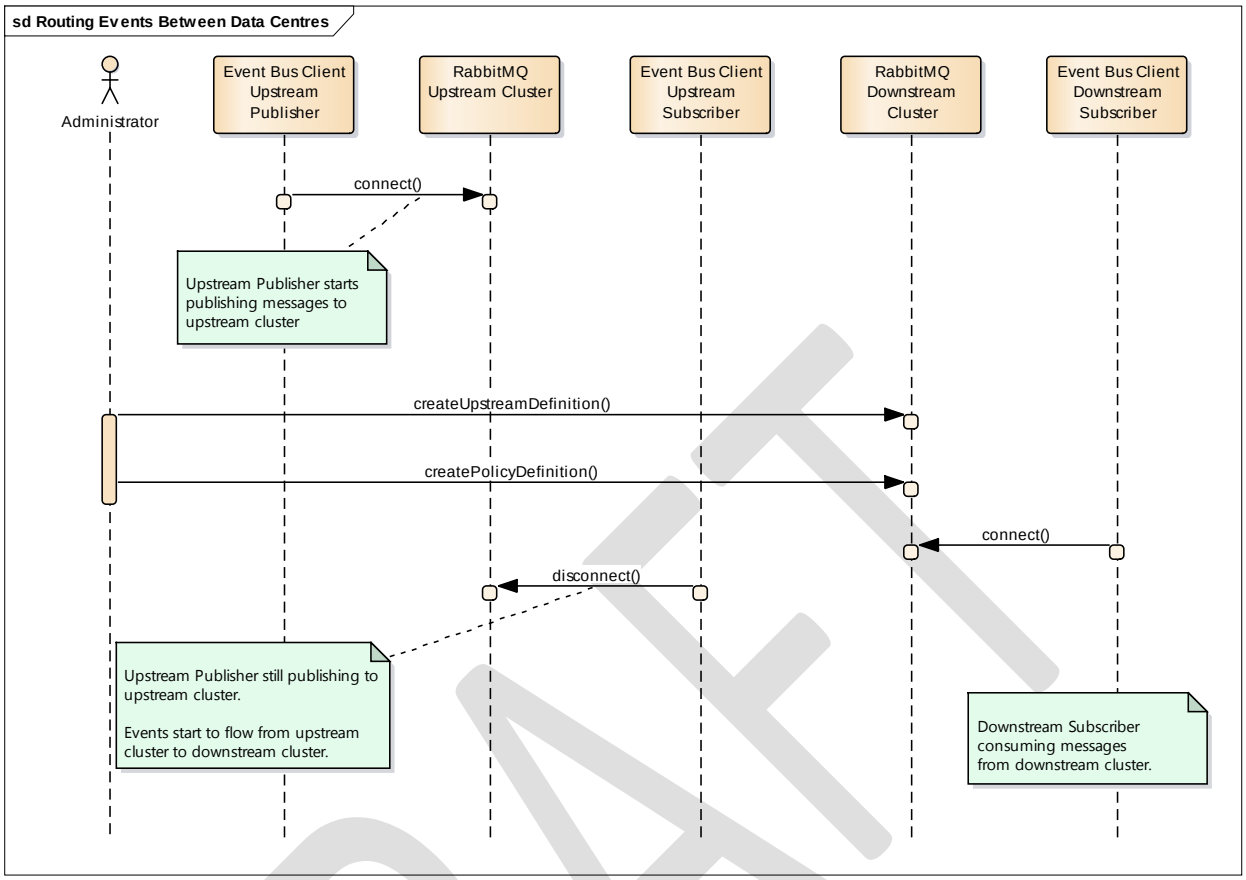

sd Publishing of Event
Publisher Application
Event Bus Client Library
RabbitMQ
authenticate(user, password): boolean
publish(string)
createMessage(string): Message
storeMessageLocally(message)
publish(message)
acknowledge(correlationID)
removeLocalMessage()
alt
[neg ack]
negativeAcknowledge(correlationID)
removeLocalMessage()
logError()
«exception»


sd Receiving Subscribed Event
RabbitMQ
Event Bus Client Library
Subscriber Application
authenticate(user, password): boolean
receiveMessage(message)
extractPayload(message): string
consume(string)
alt
[unprocessable #1]
receiveMessage(message)
extractPayload(message): string
consume(string)
«exception»
checkRedeliveredFlag()
[redelivered=false]:
reject(requeue=true)
«exception»
alt
[unprocessable #2]
«exception»
checkRedeliveredFlag()
[redelivered=true]:reject(requeue=false)
«exception»


sd Transformation of Event
RabbitMQ
«spring integration»
Event Processor
Application
JSON Validator
Payload Parser
Object Mapper
receiveMessage(message)
retrievePayload(message): string
validate(string)
jsonToJava(string): object
translate(object): object
javaToJson(object): string
buildMessage(string): message
publishMessage(message)
alt
[invalid message]
«exception»
wrapValidationException(exception):
exception
reject(requeue=false)
«exception»
alt
[mapping failure]
«exception»
wrapMappingException(exception): exception
reject(requeue=false)
«exception»


sd Content Filtering of Event
RabbitMQ
«spring integration»
Event Processor
Application
JSON Validator
Payload Parser
Filter
receiveMessage(message)
retrievePayload(string)
validate(string)
filter test is passed
jsonToJava(string): object
filter(object): object
javaToJson(object): string
buildMessage(string): message
publish(message)
alt
[filter test is failed]
receiveMessage(message)
retrievePayload(string)
filter test is failed
current message is
discarded
validate(string)
jsonToJava(string): object
filter(object)


sd Authentication of Event Bus Client
Event Bus Client Library
RabbitMQ
Security Authentication Service API
RabbitMQ extended with custom plugin for auth and auth with HTTP calls
readCredentials()
connect(username, password)
checkUserAuthenticationStatus(username, password): boolean
«REST»
«OK»
«OK»
alt
[invalid credentials]
connect(username, password)
checkUserAuthenticationStatus(username, password): boolean
«REST»
«NOT OK»
«NOT OK»

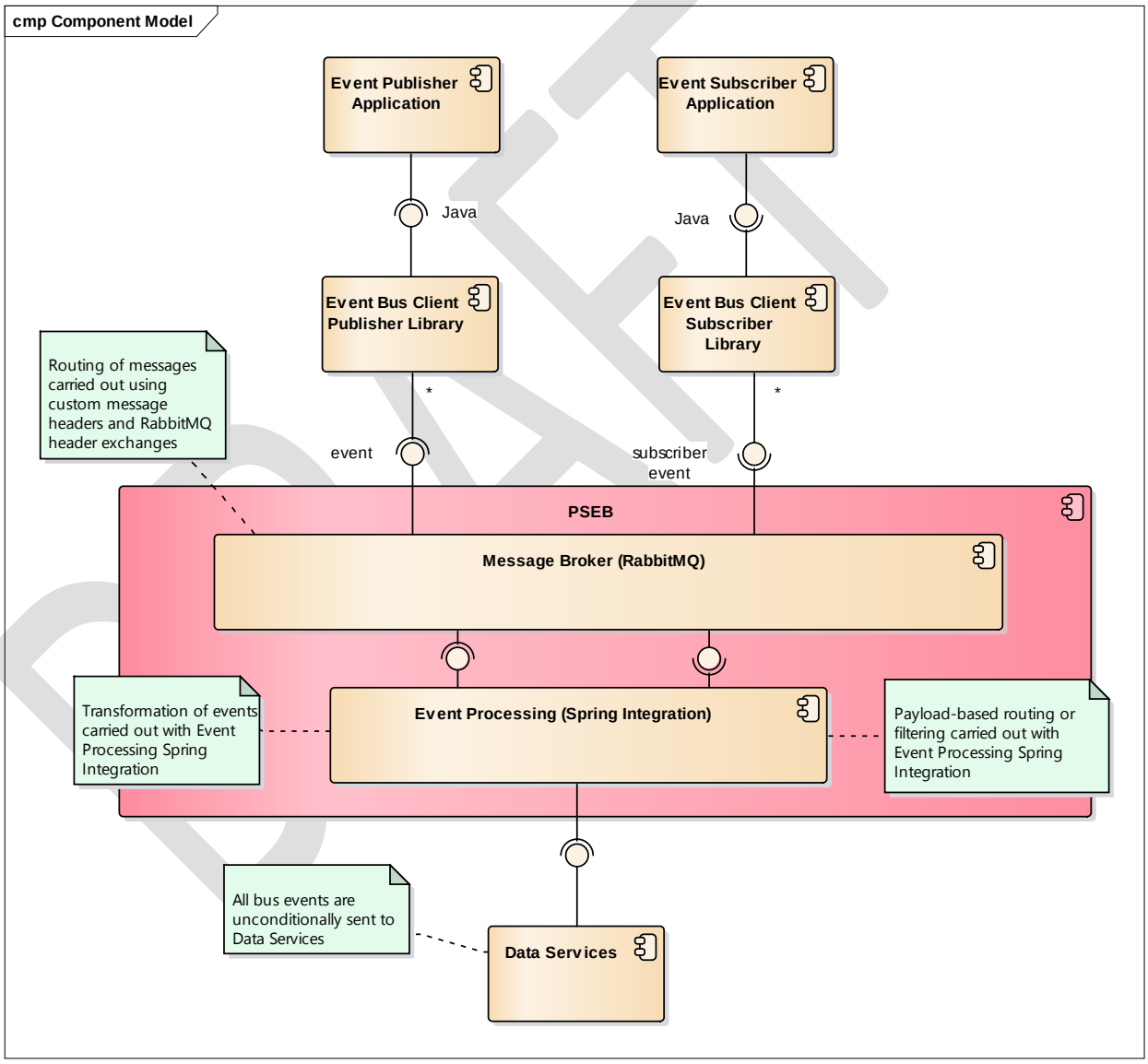

sd Routing Events Between Data Centres
Administrator
Event Bus Client Upstream Publisher
RabbitMQ Upstream Cluster
Event Bus Client Upstream Subscriber
RabbitMQ Downstream Cluster
Event Bus Client Downstream Subscriber
connect()
Upstream Publisher starts publishing messages to upstream cluster
createUpstreamDefinition()
createPolicyDefinition()
connect()
disconnect()
Upstream Publisher still publishing to upstream cluster.
Events start to flow from upstream cluster to downstream cluster.
Downstream Subscriber consuming messages from downstream cluster.

7. Logical View

[This section describes the architecturally significant parts of the design model, such as its decomposition into subsystems and packages. And for each significant package, its decomposition into classes and class utilities. You should introduce architecturally significant classes and describe their responsibilities, as well as a few very important relationships, operations, and attributes.]

7.1 Overview

[This subsection describes the overall decomposition of the design model in terms of its package hierarchy and layers.]



The Payment Services Event Bus will support the publishing and consumption of events from any client application capable of interfacing securely with RabbitMQ over AMQP. Routing of events between publishers and subscribers will, as far as possible, make use of the message routing patterns supported by the basic RabbitMQ exchange types. Conditional routing requiring access to event payload details will be carried out by dedicated event processing logic realised as single-purpose Spring Integration applications. Event message transformations – for instance, where a subscriber expects to receive an event message in a structure different to that originally published into the bus – will also be carried out by event processing Spring Integration applications. In order to abstract the low level details of interfacing with RabbitMQ away from publishers and subscribers a client library will be developed to provide applications with a simplified programming interface for authenticating to and using the PSEB.

7.2 Architecturally Significant Components

[For each significant package, include a subsection with its name, its brief description, and a diagram with all significant classes and packages contained within the package.

For each significant class in the package, include its name, brief description, and, optionally, a description of some of its major responsibilities, operations, and attributes.]

7.2.1 Message Broker

The message broker component of the PSEB is responsible for event message queueing and the majority of routing between clients. The open source RabbitMQ broker, which implements AMQP version 0.9.1, will fulfil this role.

7.2.1.1 Message Broker Routing

In order to understand the routing decisions described further on in this document a quick introduction to some key concepts is required. In AMQP compliant brokers the queueing and routing tasks are split between two distinct constituent parts:

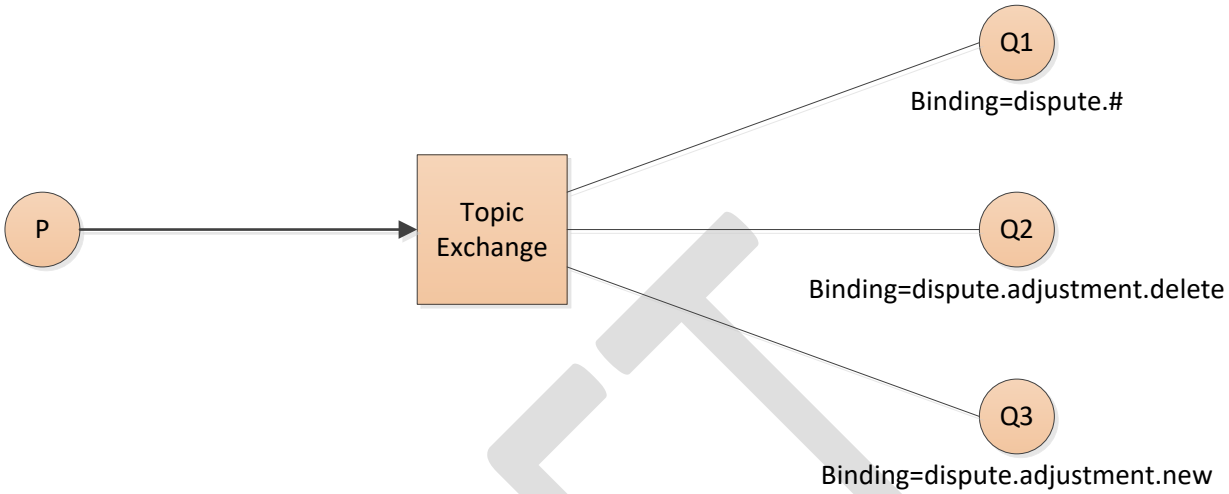
- The exchange, which accepts messages from publishers and routes them to queues
- The queue, which stores messages and forwards them to consumer applications

The bridge between these two parts is a relationship known as a “binding” that informs exchanges where to send received messages. Bindings can relate to the properties of a specific message (typically one or more custom header values set by the publisher) or to a “routing key” property which is specified as part of the publishing operation. An exchange will examine the properties of each message it receives and look for matches against its current set of bindings. When a match is found the message will be sent to the queue or exchange associated with the matching binding. Some key points to note:

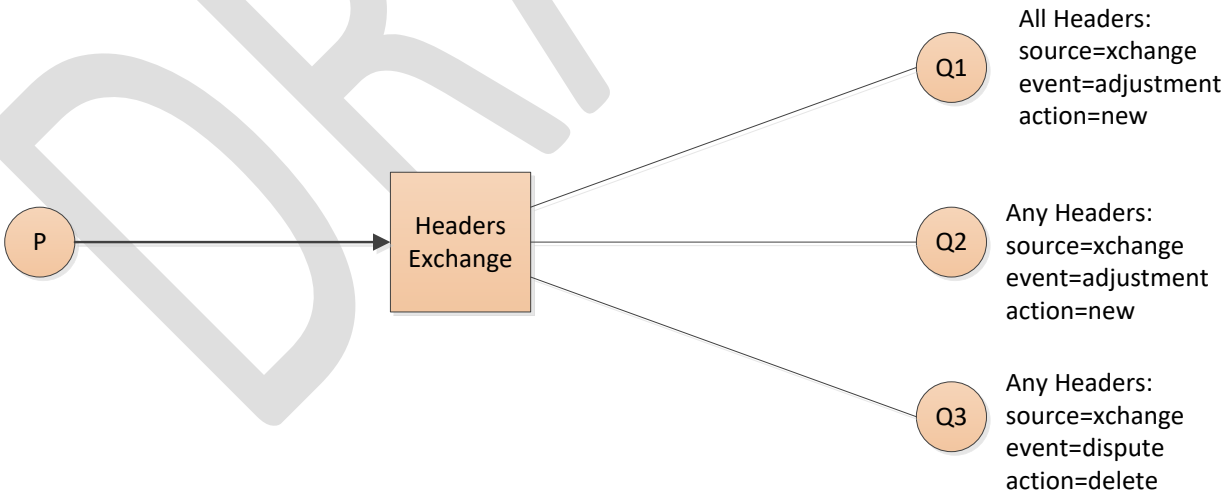
- Exchanges can bind to other exchanges which results in exchange-to-exchange routing
- Depending on the exchange type, bindings can contain wild card elements
- It is possible for an exchange to have multiple queues or exchanges bound to it
- It is possible for a queue or exchange to bind to multiple exchanges

The AMQP model opens up the possibility for varied message routing inside the message broker. Point-to-point and publisher-subscriber patterns of messaging are possible but they are by no means the only options available. All AMQP-based routing is declarative and specified using RabbitMQ administration tools such as the management GUI. There is no code involved.

For PSEB purposes the two most interesting types of AMQP exchange are the **topic exchange** and the **headers exchange** since both are capable of supporting either point-to-point or publish-subscribe semantics depending on how subscriber bindings are declared.



The **topic exchange** tries to match the routing key of each incoming message against the binding key of each subscribed queue. In the fictitious example above, if publisher P sends a message with routing key value "dispute.adjustment.new" to the topic exchange then the message will be routed to Q1 (its binding key is a wildcard pattern that matches the routing key) and to Q3 (its binding key is an exact match with the published message's routing key) but not to Q2 (no match between routing key and binding).



The **headers exchange** looks to match the headers property contained by the published message with the headers that subscribers have expressed an interested in as part of their binding. A subscriber can bind for messages that contain ALL of the headers of interest or else ANY of the headers of interest.

In the fictitious example above, if publisher P sends a message with headers {source=xchange, event=adjustment, action=new} to the headers exchange then a copy of the message will be sent to Q1 (whose binding declared an interest in all of those header name/value pairs being present), Q2 (whose binding declared an interest in ANY of those header name/value pairs being present), and also to Q3 since some of the name/value pairs it mentions in its binding are present in the published message.

7.2.2 Event Processor

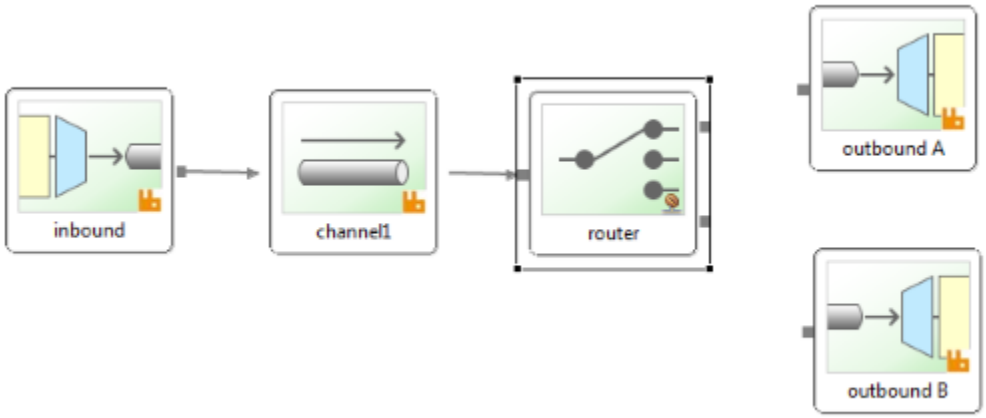
The event processing part of the PSEB will be responsible for implementing any integration logic necessary for one or more subscribers to receive notifications of published events. Some examples of this include:

- Filtering of a message based on its content
- Transformation of message content
- Conditional routing of a message based on its content
- Storing event information for logging and auditing purposes

The unifying characteristic of these examples is that they are independent, single-purpose tasks that can be carried out by small, discrete executables that intercept messages on the PSEB message broker as they travel to subscribers. These tasks interoperate with the PSEB message broker in exactly the same way that external clients do – by consuming their inputs from queues and publishing the results to exchanges – but with the important distinction that they are not directly accessible to external clients.

7.2.2.1 Event Processor Routing

Each event processor node will be developed using Spring Integration which provides an extension of the Spring programming model to support well-known enterprise integration patterns. It enables lightweight messaging inside the host application and supports integration with external systems via declarative adapters. Examples of these adapters include an AMQP inbound channel adapter that can consume messages from a configured queue and an AMQP outbound channel adapter for publishing messages to AMQP exchanges. Of greatest relevance in this discussion is that a single Spring Integration flow can contain multiple AMQP outbound channel adapters with each one referencing a different exchange. Hence, it is straightforward to construct a Spring Integration flow that will consume a message from RabbitMQ using its AMQP inbound adapter, process the message using one or more standard Spring Integration mediation components wired together using Spring Integration channels, and then send one or possibly more output messages out of the flow on one or more AMQP outbound channel adapters.

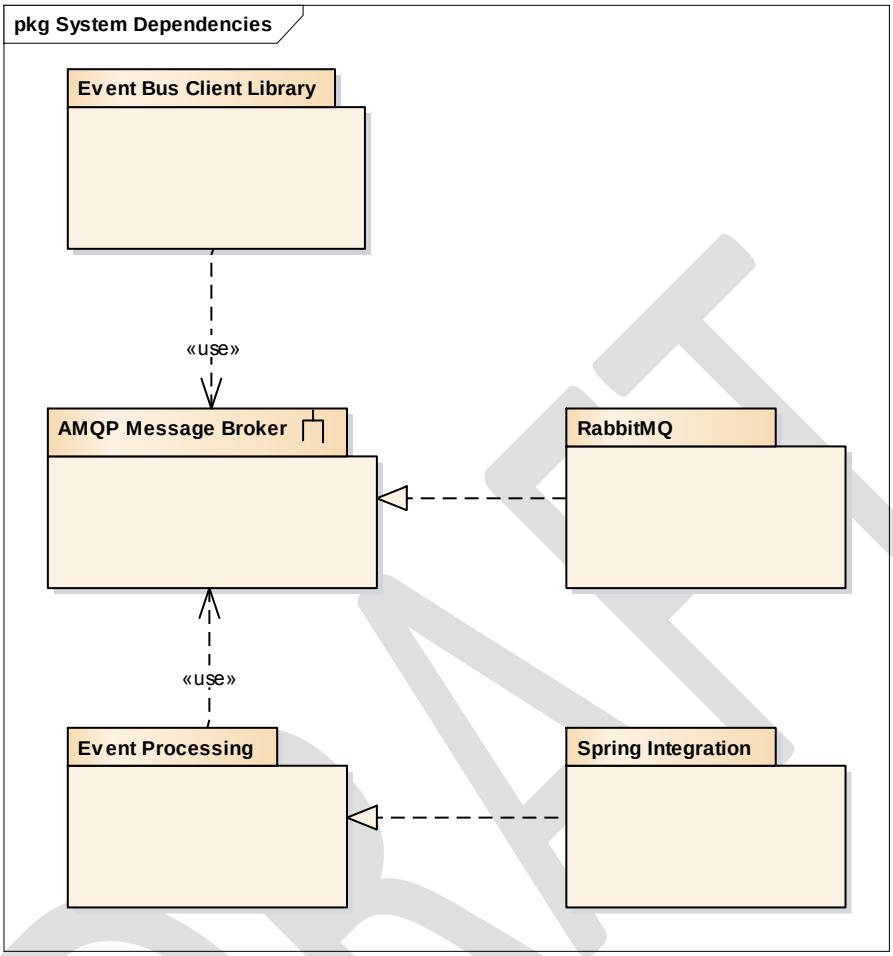


7.2.3 Client Library

Interfacing with an AMQP broker requires client applications to manage a number of protocol-specific concerns which are usually orthogonal to the main purpose of the application code. Examples include the construction of message objects; specifying the appropriate routing key information when carrying out a publishing operation; adding any custom header values to the message; handling acknowledgements or negative acknowledgements; sending confirmations when a message has been successfully read from a queue; and specifying what kind of rejection signal to send if a consumed message cannot be processed. To try and shield applications as much as possible from these low-level issues a library will be made available to clients that will provide an abstraction layer between them and the AMQP programming interface. External properties files will be used to supply configuration details relevant to a particular client application and the PSEB instance it wishes to use.

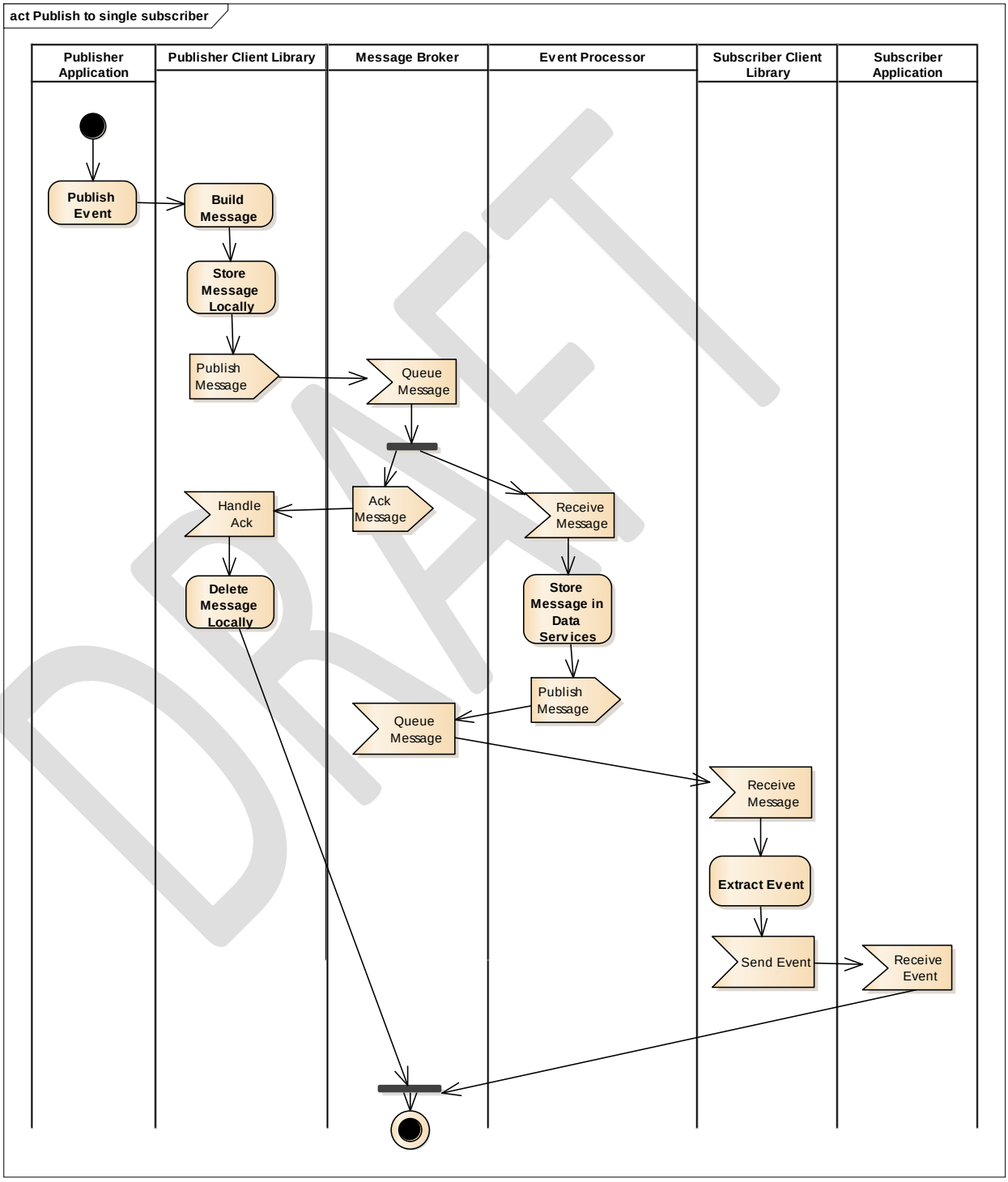
The client library will be responsible for handling message delivery acknowledgements from the message broker and, should errors occur in the form of negative acknowledgements (e.g. message received by the AMQP broker but the target exchange does not exist), be responsible for communicating those errors up the stack to the application code. Where "publisher confirms" is enabled on a sending channel (publisher confirms is a RabbitMQ optimisation on the basic AMQP where broker acknowledgements are sent back asynchronously) then the client library will be responsible for maintaining information about messages which have been published but not yet acknowledged.

7.3 Dependencies



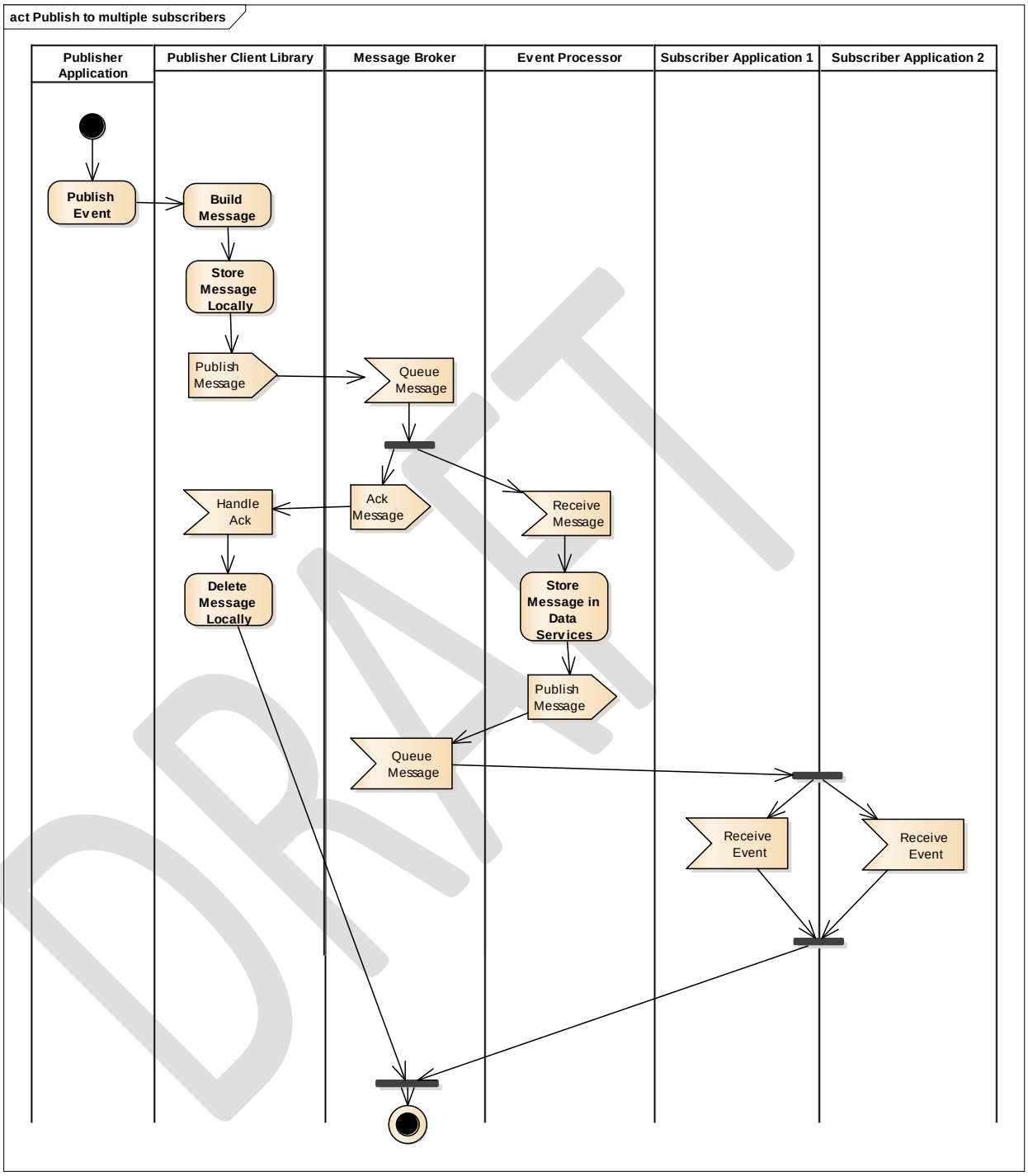
8. Process View

[This section describes the system's decomposition into lightweight processes (single threads of control) and heavyweight processes (groupings of lightweight processes). Organize the section by groups of processes that communicate or interact. Describe the main modes of communication between processes, such as message passing, interrupts, and rendezvous.]

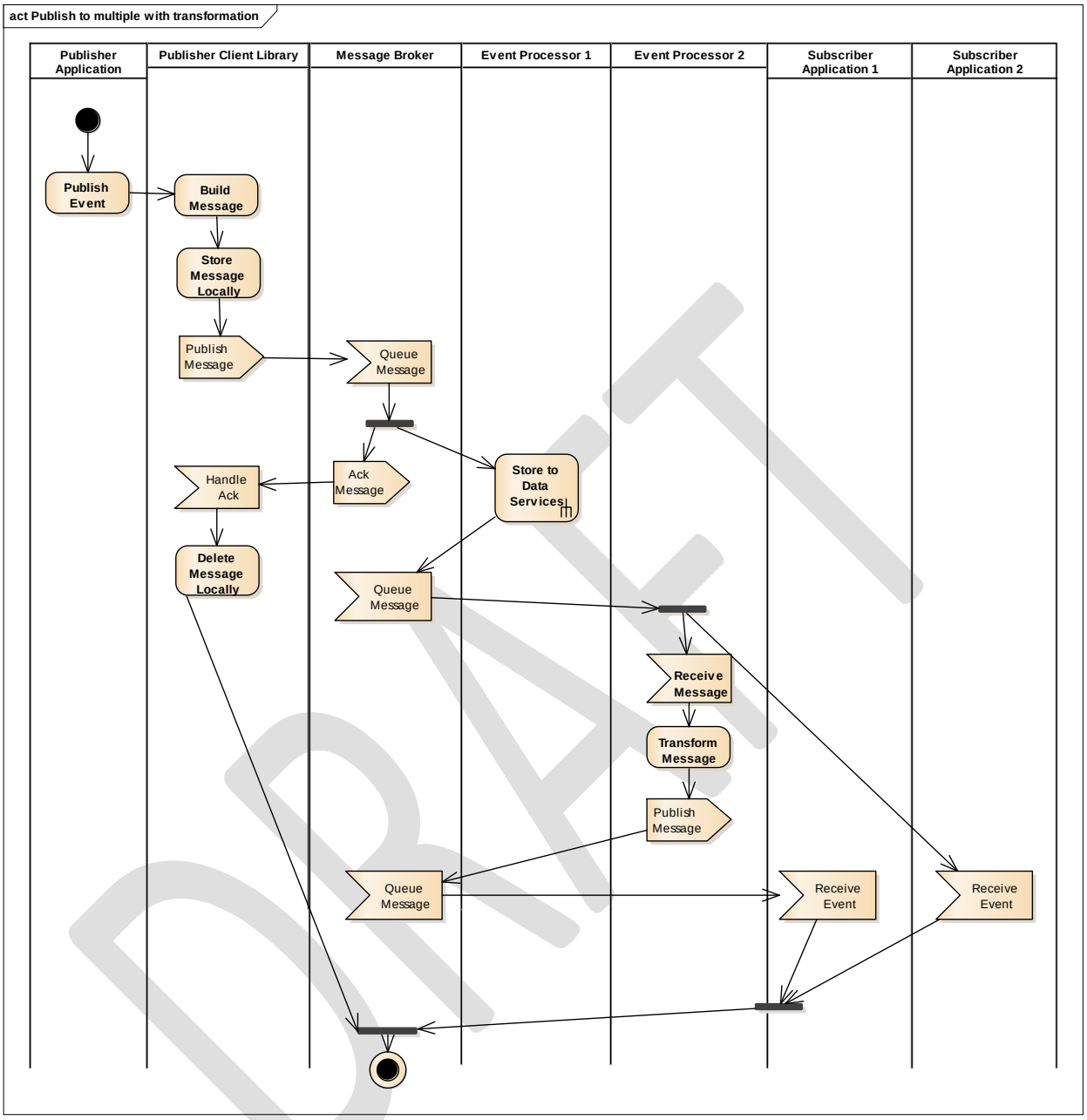


In all activity diagrams presented in this section of the document each published event is unconditionally stored into Data Services. This step is a pre-requisite of all further actions involved in getting events to message bus subscribers. When there is a failure storing an event to Data Services the event message will not be propagated further into the bus and instead be routed to an error queue for subsequent processing.

The following activity diagram sets out to show the process flow and communication where a single event is to be routed to more than one subscriber. For reasons of space, only two subscribers are shown here but the same principle will apply for more subscribers. Similarly, for space reasons the actions of the subscribers' respective client libraries are not shown. The receiving of the message from the message broker queue of interest, extraction of the payload, and subsequent passing of the payload to the subscribing application are implicit in this activity.


act Publish to multiple subscribers
Publisher Application
Publisher Client Library
Message Broker
Event Processor
Subscriber Application 1
Subscriber Application 2
Publish Event
Build Message
Store Message Locally
Publish Message
Queue Message
Ack Message
Handle Ack
Delete Message Locally
Receive Message
Store Message in Data Services
Publish Message
Queue Message
Receive Event
Receive Event

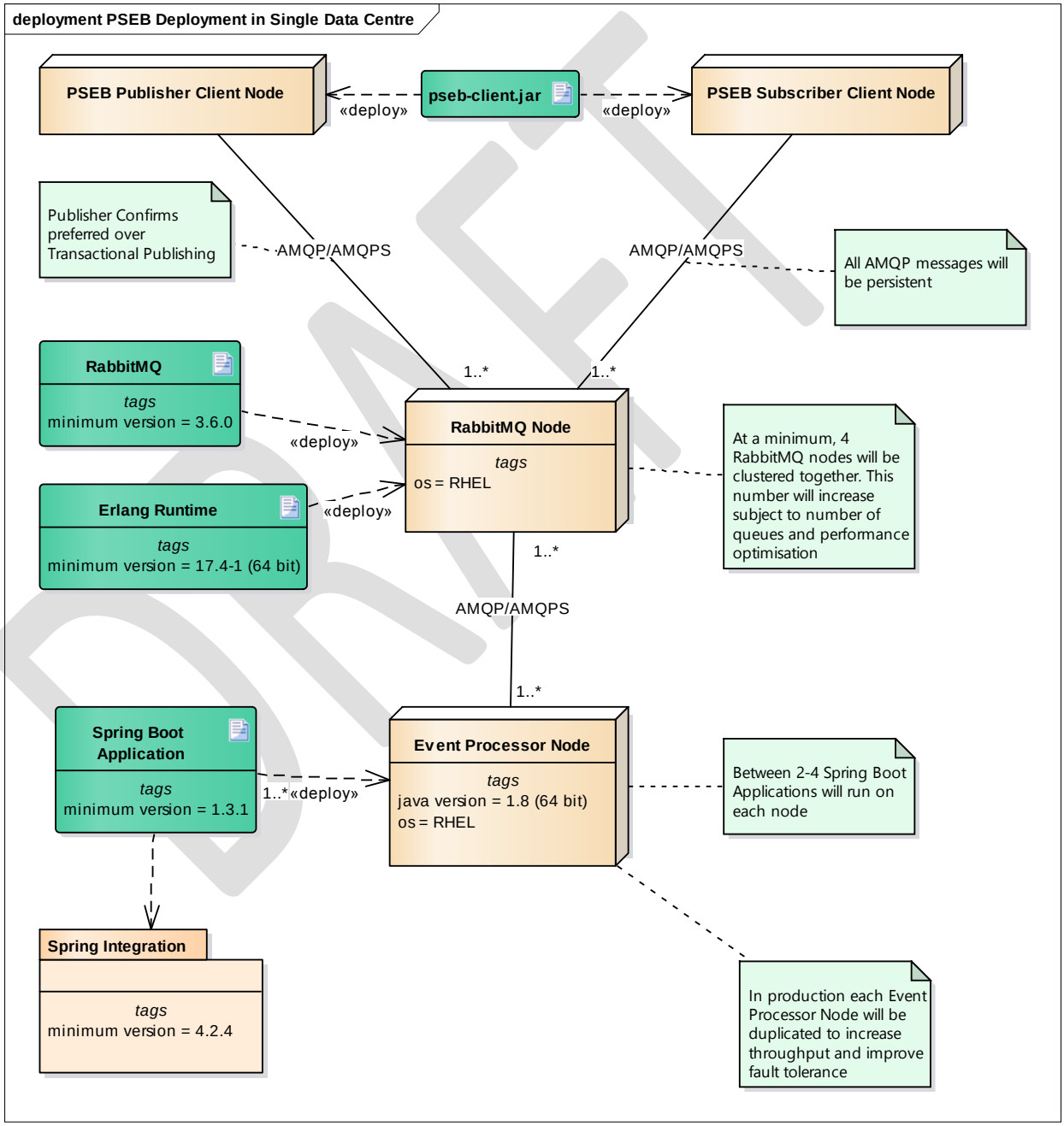
In the following activity diagram a further simplification is made with the Event Processor's storing of the event data encapsulated in a sub-activity (Store to Data Services). The main objective of the diagram is to show what actions can occur when one or more subscribers require additional processing of received event messages such as transformation into a format different to the published one. An event processor node – a small Spring Boot application developed with Spring Integration – will consume the message from a broker queue and carry out the required manipulation of it before publishing it back to the broker. This will be done in parallel to instances of the original (i.e. un-transformed) message being received by other subscriber applications. Where possible to do so, duplication of messages around the message bus will be done using RabbitMQ routing (i.e. bindings between queues and exchanges as discussed in the section on Message Broker Routing). Where the routing decisions depend on message payload then an Event Processor node will be required. See the section on Event Processor Routing for more information.


act Publish to multiple with transformation
Publisher Application
Publisher Client Library
Message Broker
Event Processor 1
Event Processor 2
Subscriber Application 1
Subscriber Application 2
Publish Event
Build Message
Store Message Locally
Publish Message
Queue Message
Ack Message
Handle Ack
Delete Message Locally
Store to Data Services
Queue Message
Receive Message
Transform Message
Publish Message
Queue Message
Receive Event
Receive Event

9. Deployment View

*[This section describes one or more physical network (hardware) configurations on which the software is deployed and run. It is a view of the Deployment Model. At a minimum for each configuration it should indicate the physical nodes (computers, CPUs) that execute the software and their interconnections (bus, LAN, point-to-point, and so on.) Also include a mapping of the processes of the **Process View** onto the physical nodes.]*

9.1 Deployment in a Single Data Centre



10. Implementation View

[This section describes the overall structure of the implementation model, the decomposition of the software into layers and subsystems in the implementation model, and any architecturally significant components.]

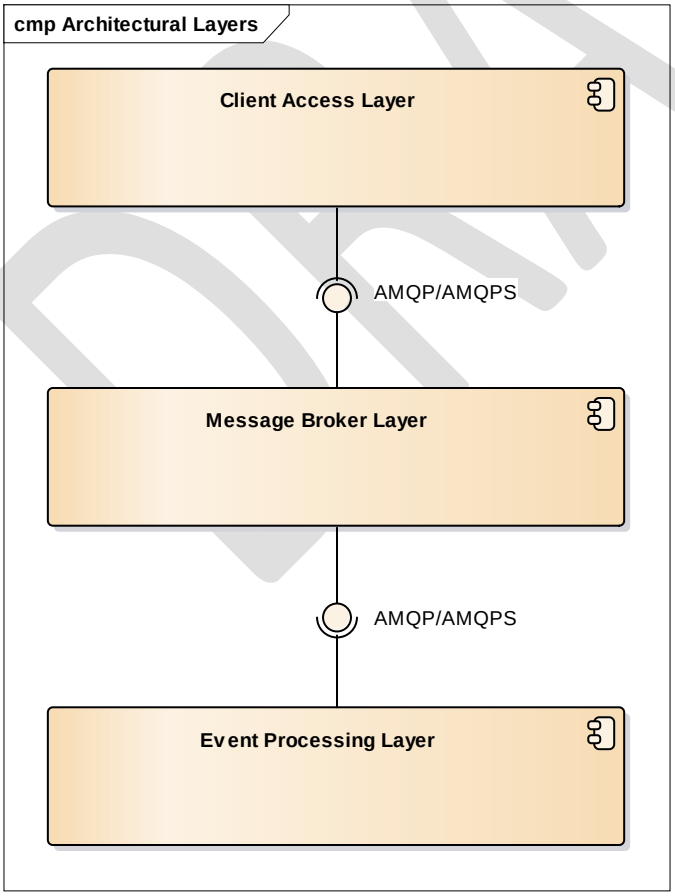
10.1 Overview

[This subsection names and defines the various layers and their contents, the rules that govern the inclusion to a given layer, and the boundaries between layers. Include a component diagram that shows the relations between layers.]

Three architectural layers have been identified in the PSEB:

- Client access layer – concerned with the publishing of events into the bus and the receiving of subscribed events from the bus
- Message broker layer – concerned with reliably transporting events between publishers and subscribers
- Event processing layer – concerned with executing any integration logic that may be required when integrating a publisher with its subscribers

Communications between the layers is two-way will use AMQP or, if the need emerges, AMQPS where all traffic is encrypted. This is illustrated in the below component diagram.



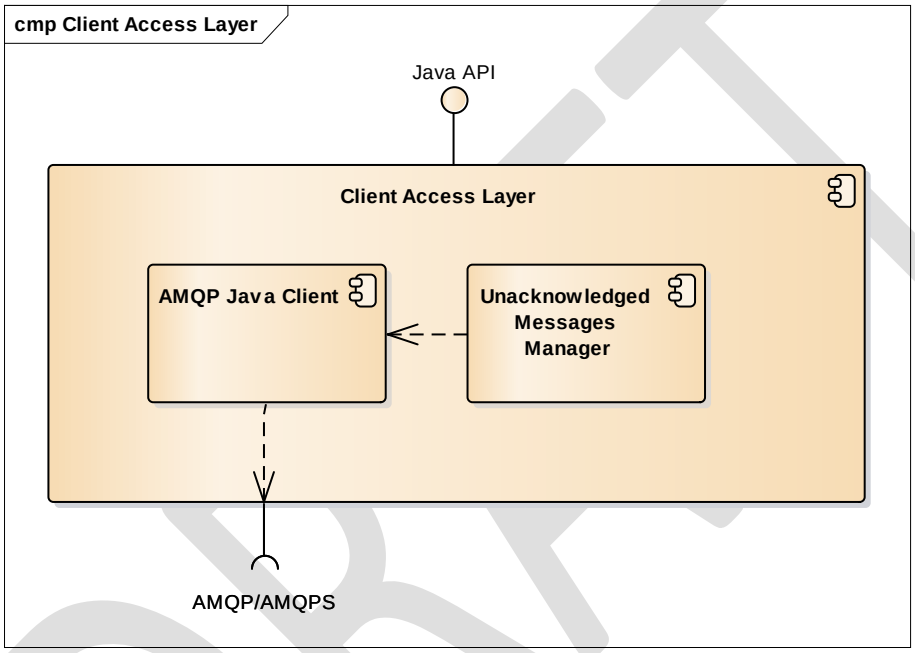
10.2 Layers

[For each layer, include a subsection with its name, an enumeration of the subsystems located in the layer, and a component diagram.]

10.2.1 Client Access Layer

This layer will be used by client applications to put and get event messages. It will include the following subsystems:

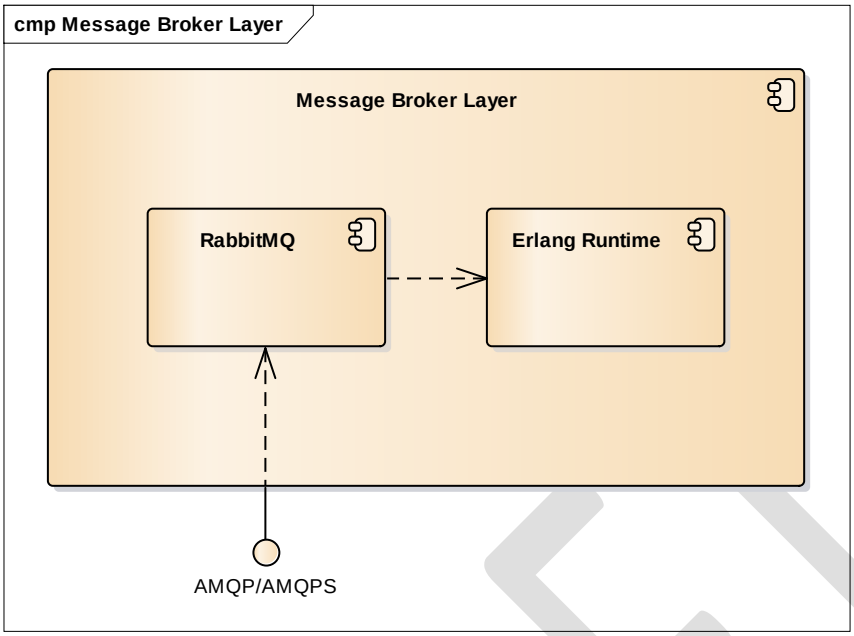
- AMQP Java client
- Manager of unacknowledged messages where Publisher Confirms is in use



10.2.2 Message Broker Layer

This layer will be capable of receiving AMQP messages from publisher clients, queueing them, and then routing them to queues where they can be consumed over AMQP by subscriber clients. Messages which fail to be successfully routed and consumed will be automatically sent to a pre-defined queue (i.e. a dead letter queue) for subsequent error processing. This layer will include the following subsystems:

- RabbitMQ message broker
- Erlang runtime



To provide the highest possible levels of fault tolerance and messaging throughput the message broker will cluster its RabbitMQ instances together.

10.2.3 Event Processing Layer

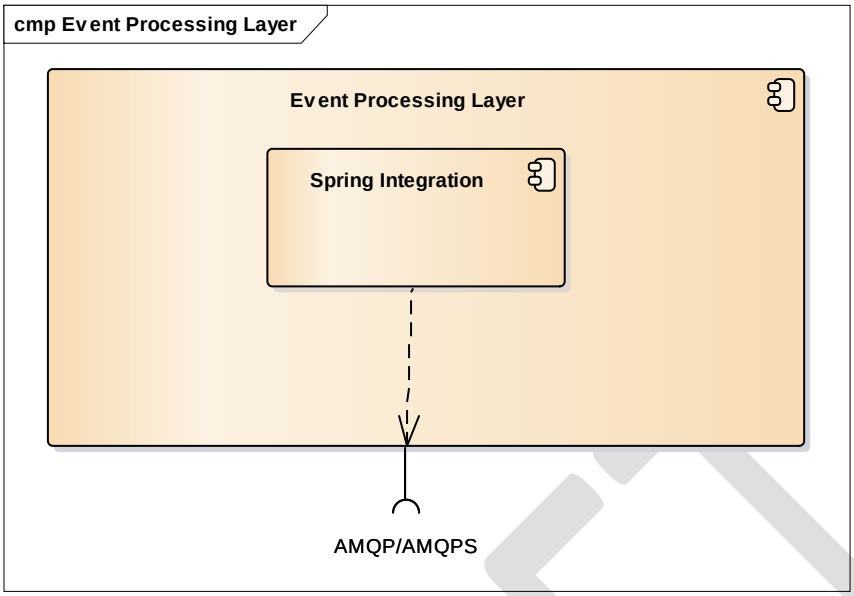
This layer will be used to carry out logical tasks that may be required to integrate an event with its subscribers. Examples of integration logical tasks include (but are not strictly limited to):

- Filtering out of messages depending on payload information
- Conditional routing of messages depending on payload information
- Transformation of the message into a different structure more suitable for a subscriber
- Splitting an event message into several other event messages
- Storing of vent message details (not necessarily the entire message) to data services

Precise details of the integration logic will vary according to the requirements set out by one or more subscribers to a specific event type.

This layer will not carry out application logic, only integration logic.

This layer includes the Spring Integration subsystem.


cmp Event Processing Layer
Event Processing Layer
Spring Integration
AMQP/AMQPS

11. Data View (optional)

[A description of the persistent data storage perspective of the system. This section is optional if there is little or no persistent data, or the translation between the Design Model and the Data Model is trivial.]

11.1 Event Structure

At time of writing the only known publisher to the PSEB is the Xchange business application which will publish dispute events and adjustment events. The structure of these events is still under development and will be included in a subsequent version of this document.

11.2 Event AMQP Headers

At time of writing it is envisaged that a significant amount of message routing inside the message broker layer could rely on using the AMQP headers exchange which supports arbitrary routing in RabbitMQ through use of the "headers" table in each message's properties. As explained in the section on Message Broker Routing, queues (or exchanges) bound to a headers exchange bind against a collection of key/value pairs to route on together with a binding argument that stipulates whether a routing match occurs if *any* of the message header values match the binding key/value collection or if all of the message header values need to match the binding key/value collection. This does not preclude a message from having additional headers set in it when it is published.

Using message headers results in self-describing messages at the properties (i.e. meta-data) level. Provided a clear and consistent set of header names and values is used across the PSEB when publishing messages and binding to headers exchanges then the potential exists for significant amounts of routing to be done without recourse to parsing message payloads.

An initial set of headers and their intended meaning is given below. This set comes from the Discover Event Specification version 2.0 controlled by Krishna Patil (EA). It is expected that this set will expand as the bus implementation progresses. Subsequent revisions of this document will update the below table.

Header Name	Description	Required/Optional and Cardinality	Examples
eventType	the primary classifier of a related set of events	Required, 1	String type. The set of event type identifiers will be different for each publishing application. It will need to be created prior to the PSEB launch and will require updating as more publishers are boarded over time.
eventActivity	the secondary classifier of a related set of events that is often needed to further qualify the event	Optional, 0..1	payments/oneTimePay, payments/directPay

eventOutcome	Verb in the past tense that describes the outcome of the event	Required, 1	approved, declined, opened, closed
eventMessageGUID	Identifier that helps established traceability of an event to its source. In the context of the DFSI Event Platform this is the JMS Id assigned to events on the Event Bus.	Required, 1	
eventAncestorID	This field provides event lineage in situations where events are generated from other events. When used, this field will be set to eventMessageGUID of the immediate ancestor event that this event was generated from.	Optional, 0..many	562a6f12-85c6-4b2c-b25d-1ff9a360083e
eventCorrelationID	This field is set to group the events to specific category (transaction domain, HTTP Session).	Optional, 0..1	
businessKeyType	Identifies the business key type that this event relates to. Examples of business key type are credit card account key, loan account key, customer email address, callID, customer, supplier, employee, merchant, etc.	Optional, 0..many	
businessKeyValue	Identifies the business key that this event relates	Optional, 0..many	

	to.		
rawEventSourceAppId	Identifier the application source where the raw event occurred.	Required, 1	DPS-SM, Account center, Orion
rawEventSourceChannel	Indicates the channel through which the event originated. Values are to be obtained from existing standard reference table	Required, 1	TRANSWITCH, any, na, chat, emails, webprivate, webpublic, accountManager, gateway
rawEventProcessorAppId	Identifies the application that process the raw event	Required, 1	authEventProcessor
rawEventSourceAppInstance	Application instance identifier to pinpoint the exact application instance generating the event.	Optional, 0..1	ServiceManager001
rawEventProcessorInstance	Identifies the device where the raw events are processed	Optional, 0..1	rwlpg999.rw.discoverfinancial.com
financialAgreementType	Indicates the financial agreement type that the event belongs to.	Required, 1	card, deposit, sl, hl, pl, dn, dci, pulse, any, na, paymentNetwork, bank
eventVersion	Identifies the event version. The event version may be changed to indicate any of the following 1) a change in topic 2) a change in the payload 3) change in the event header structure.	Required, 1	1.0
eventSpecVersion	Identifies the event message specification version	Required, 1	1.0
eventPayloadFormat	Identifies the event payload format.	Required, 1	xml, json, csv, etc..
resubmissionFlag	Indicates if the event was re-	Required, 1	true/false

	published on account of timeout or failure and could potentially be a duplicate.		
eventOccurrenceTime	Time when the event occurred in ISO 8601 format.	Required, 1	2013-07-26T05:29:32.905-0500

Note that message headers are intended to be used for custom purposes and are not intended to override any of the basic message properties used in RabbitMQ. These include “correlation-id”, “content-type”, “reply-to” and so on.

12. Quality

[A description of how the software architecture contributes to all capabilities (other than functionality) of the system: extensibility, reliability, portability, and so on. If these characteristics have special significance, such as safety, security or privacy implications, they must be clearly delineated.]

12.1 System Extensibility

RabbitMQ supports a plug-in mechanism which enables additional broker functionality over and above that offered in the core distribution. A good number of plugins ship with the core distribution (e.g. the Web based management console, cluster federation, etc.) but many more are available both from the RabbitMQ team and the wider community. It is also possible to develop custom plugins for edge case behaviour. An example of this is the custom plugin that will be required in order for RabbitMQ to authenticate and authorise clients using the CPP security service HTTP API.

In the event processing layer, Spring Integration provides a large palette of existing primitives to developers when constructing integration flows. Examples include filters, transformers, channel adapters, and routers. Where the out-of-the-box primitives are not sufficient, it is straightforward to develop message processing endpoints as annotated POJOs which are referred to from Spring Integration flows.

12.2 System Reliability

At the heart of the PSEB is the need for reliable messaging. In its role as bus message broker, RabbitMQ provides a number of behaviours which go towards meeting this objective.

- High Availability queues (HA queues). Each event bus queue will be created as an HA queue so that it is mirrored on at least one other RabbitMQ broker instance in the cluster. Even if one of the broker instances crashes the queue will continue to be available to clients. When the crashed node becomes available again it will be automatically sync'd from the running instance.
- Persistent messages. All event bus messages will be published as persistent mode so that they are stored on disk while they are held in in-memory queues. Should the RabbitMQ process crash then messages will still be available on server restart.
- Durable queues. All RabbitMQ queues in the event bus will be created as durable so that their queue definitions (as distinct from their contents) survive any broker crashes.
- Dead Letter Queueing (DLQ). Invalid messages (aka poison messages) or messages that cannot be processed because of temporary downstream errors will be routed to a pre-defined dead letter exchange from where it will be automatically sent to a dead letter queue for error processing. All RabbitMQ queues used by the event bus will be created with a dead letter exchange defined. No invalid messages or messages that cannot currently be processed will be silently discarded or otherwise lost. Similarly, no message will block the normal functionality of the bus. Messages which are detected as having been delivered too many times will be rejected back to their publisher with instructions not to place the message back in a queue. This will automatically cause the broker to route the message to a dead letter exchange.
- Unrouteable messages. In the event that a message published to a RabbitMQ exchange cannot be routed to a subscriber it will automatically be routed to a pre-defined alternate exchange from where it can ultimately be sent to a dead letter queue (DLQ) for error processing. All RabbitMQ exchanges used in the event bus will be created with an alternate exchange defined. No unrouteable messages will be silently discarded by the broker or otherwise lost.
- Automatic detection of message saturation from publishers. RabbitMQ has a feature called "TCP backpressure" which will stop publishers from oversaturating it with messages. This improves on the "channel flow" alternative offered in AMQP which simply requests that publishing clients stop publishing so fast and is often ignored.

In the production environments all event processor nodes (standalone Spring Boot applications developed using Spring Integration) will run in pairs consuming from the same queues. The instances will execute on separate hosts so that in the event of a server failure one processor will continue to service the queues.

12.3 System Portability

All software components of the PSEB are portable across different operating systems.

- The Erlang runtime is available across all of the popular operating systems
- RabbitMQ, which runs on Erlang, is available across all of the popular operating systems
- The event client library and the event node processors will run on a Java VM which is available for deployment across all of the popular operating systems

12.4 System Security

The PSEB will be capable of using AMQPS, the secure form of AMQP, for connections from clients to RabbitMQ clusters and connections between RabbitMQ clusters. With AMQPS all in-flight message data will flow encrypted. Similarly, the RabbitMQ management console will be configured to only accept HTTPS connections. In both cases SSL will be enabled through the creation of a signed server-side certificate using standard tooling (e.g. OpenSSL). Because bus clients will all be located inside the corporate firewall it will be sufficient to create self-signed certificates. Where a published event is designed to contain no sensitive data then the AMQP alternative will be an option.

Only persistent messages will be used in the PSEB. This means that where messages need to be placed in queues because there is currently no active consumer a copy of the message will also be stored out to disk. To prevent access to these persisted messages the disk locations where they are held will be protected using stringent UNIX file permissions in the same way that the contents of persisted WMQ messages are safeguarded in exiting Discover applications.

Where there is a case to keep a category of event message private from the rest of the bus resources RabbitMQ virtual hosts will be used. Broker resources such as users, exchanges, and queues created in the dedicated virtual host will be inaccessible to resources created in other virtual hosts.