Table of Contents

[1 Introduction 1](#_Toc492624188)

[2 Process Model 3](#_Toc492624189)

[3 Object Model 9](#_Toc492624190)

[4 Resource Definitions 10](#_Toc492624191)

[5 API Definitions 10](#_Toc492624192)

# Introduction

The primary purpose of the Poplin "Messaging" shared service is to enable an executable, open source, standard mechanism for State Medicaid Agencies (SMA) to provide message capabilities between services in their enterprise architecture. An independent messaging service provides an abstract communication layer between services to help decouple them from each other.

Microservice-based architectures typically utilize several services working collaboratively to implement a business use case. These architectures require services that are able to communicate effectively and efficiently with each other. The Messaging Service in Poplin provides that functionality.

Poplin defines this shared service in a way that does not dictate tools or technologies. It does, however, provide a fully executable open source technology stack that SMAs can use in whole or in part to realize their service messaging capability.

This document specifies the business processes achieved through the Messaging Service, the objects that are used, and defines the resources and APIs that can be used by client software.

An example use case for the messaging shared service is shown in Figure 1, demonstrating how messages and a message broker can be used to coordinate the activities of several services to achieve a final result.

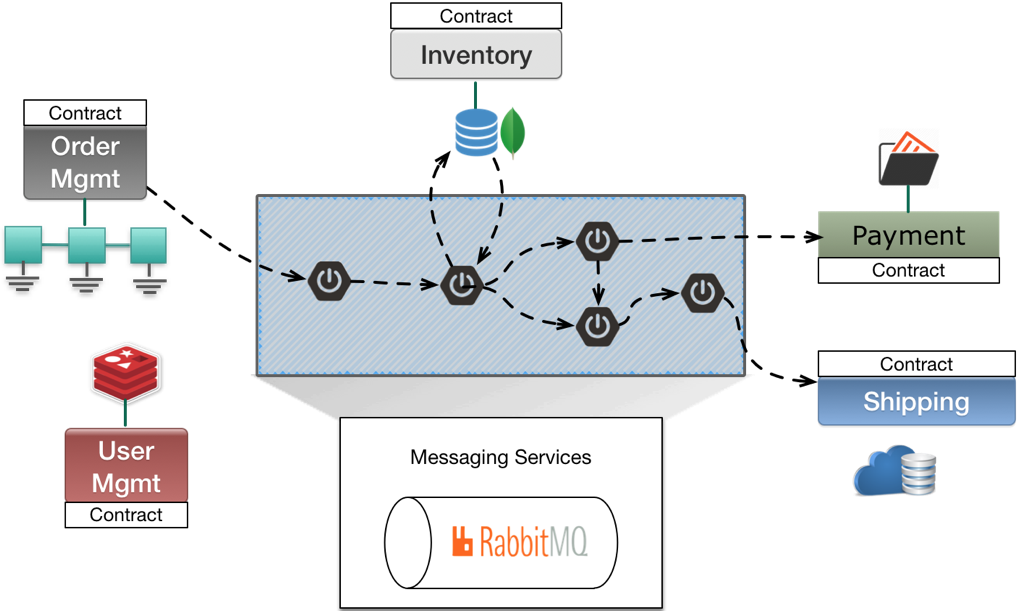


Figure : Example Use Case for Messaging[[1]](#footnote-1)

# Poplin Reference Implementation

The Poplin reference implementation uses RabbitMQ, a widely accepted existing open source message brokering technology, to implement messaging. RabbitMQ was originally written by Rabbit Technologies and is now owned by VMWare. RabbitMQ is currently used by small- as well as large-scale operations, supports a wide variety of message patterns (such as publish/subscribe, request/response, and push-pull) that are typically used in Medicaid Enterprise Systems, uses industry standard message formats, has a plug-in architecture to support future messaging formats, has a RESTful API, and a strong open source community with commercial support.

RabbitMQ supports three standard message formats through a common set of APIs:

* **Advanced Message Queuing Protocol (AMQP)** – The focus of AMQP is reliability and interoperability across vendors and is an open standard designed to replace prior proprietary protocols. AMQP provides a broad feature set, including reliable queuing, publish and subscribe with topics, routing options, transactions, and security. Google, JP Morgan, and NASA use AMQP in large scale installations processing billions of messages per day[[2]](#footnote-2). For more information about AMQP, visit [amqp.org](http://amqp.org/).
* **Message Queue Telemetry Transport (MQTT)** – Developed by IBM, MQTT is an open standard focused on operating in a small footprint that can better support mobile and Internet of Things (IoT) applications. MQTT implementations are small and payloads are in a compact binary format. IBM and Facebook Messenger use MQTT. More information about MQTT can be found at [mqtt.org](http://mqtt.org/).
* **Simple/Streaming Text Oriented Messaging Protocol (STOMP)** – STOMP is the HTTP of messaging formats in that it is text-based. It is simpler than AMQP, but more verbose than MQTT. Producers send messages with a “destination” string, which can represent a topic, exchange, or queue. Consumers can then subscribe to these destinations. STOMP also supports security, message acknowledgement, and transactions. It is an open standard and commonly used in interactive web applications and through WebSockets. STOMP is discussed in further detail at [stomp.github.com](http://stomp.github.com/).

RabbitMQ can also support additional messaging formats through a plug-in architecture.

AMQP is a mature messaging standard and provides the broadest set of features that can be important for Medicaid Enterprise Systems. This service definition will focus on AMQP-based functionality.

# Process Model

In its basic form, a messaging broker provides a message queue between the producer (P) of the message and consumer (C). The message queue provides first in, first out delivery of messages to the client.



Figure 2: Basic Messaging[[3]](#footnote-3)

Through the use of work queues, multiple consumers can retrieve messages from the same producer (e.g. from several instances of the same consuming service).



Figure 3: Multiple Consumers[[4]](#footnote-4)

A message broker can be added to create message exchanges with bindings to route messages to particular queues, providing flexibility in how messages are delivered from producers to consumers.

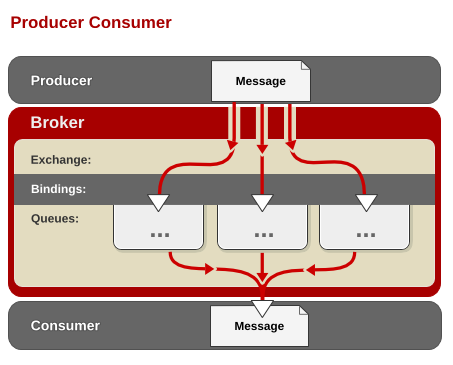


Figure 4: Producer / Consumer Model for Message Brokering[[5]](#footnote-5)

# Fanout Exchange (Broadcasting)

A producer may want to broadcast an event or information to many consumers. The fanout exchange supported by the message broker providers this functionality, where a single message from the producer is multiplexed on to several message queues delivering to multiple consumers.

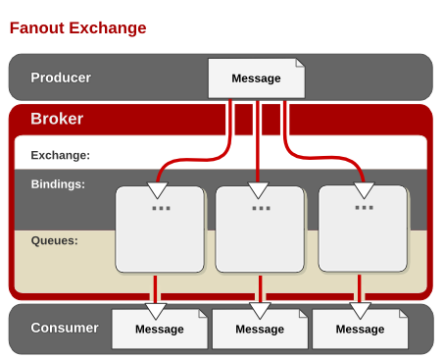


Figure : Fanout Exchange Messaging[[6]](#footnote-6)

# Routing

A producer may want to send a message to a particular queue through a routing key. The queue with the binding key that matches the routing key receives the message.

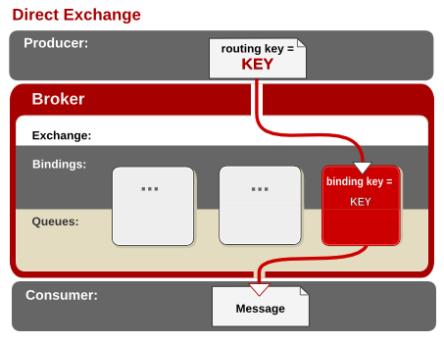
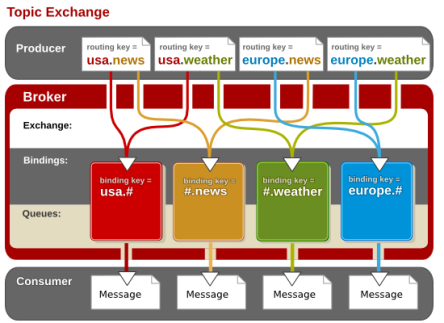


Figure 6: Direct Exchange Messaging[[7]](#footnote-7)

# Topics

The producer may want to organize messages by “topic” area and specify a different set of recipients for each topic. Consumers who are included as topic recipients receive messages for that particular topic.



*Figure 7: Topic Exchange Messaging[[8]](#footnote-8)*

# Publish and Subscribe

The publish and subscribe messaging pattern is an extension of the topic message pattern. In this pattern, clients indicate their interest in a topic by subscribing to the topic. When a producer publishes a message for that topic, all consuming subscribers receive that message.



Figure : Publish and Subscribe Messaging[[9]](#footnote-9)

# Request / Response

The producer may want to send a message and receive confirmation that the message was received by a consumer. This is useful for managing transactions between services where coordination needs to be precise. In this case the message is placed on the appropriate queue as before, but when the consumer retrieves the message from the queue, a confirmation reply is placed in a reply queue back to the producer.

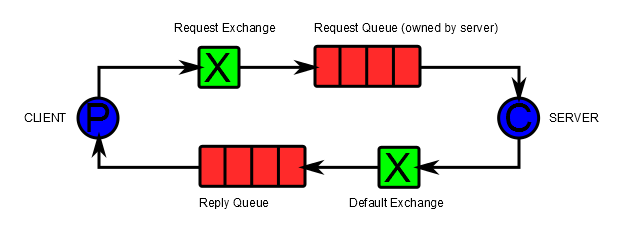


Figure 9: Request/Response Messaging[[10]](#footnote-10)

# Object Model

The diagram in Figure 9 shows the class architecture and relationships involved in consuming messages from RabbitMQ.

# 

Figure 10: AMQP Class Architecture[[11]](#footnote-11)

An article by Michael Meisinger of the Ocean Observatories Institute, which uses AMQP to exchange messages, has documented the object mode for AMQP at [confluence.oceanobservatories.org/display/syseng/CIAD+COI+TV+AMQP](http://confluence.oceanobservatories.org/display/syseng/CIAD+COI+TV+AMQP).

# Resource Definitions

The resource definitions or RabbitMQ are included with the project's API documentation, which can be found at [www.rabbitmq.com/resources/specs/amqp-xml-doc0-9-1.pdf](https://www.rabbitmq.com/resources/specs/amqp-xml-doc0-9-1.pdf).

# API Definitions

The RabbitMQ project provides documentation on how to use the base API with the RabbitMQ server and is covered separately at the following locations:

* Overall documentation:

[www.rabbitmq.com/documentation.html](https://www.rabbitmq.com/documentation.html)

* API reference for RabbitMQ:

[rawcdn.githack.com/rabbitmq/rabbitmq-management/rabbitmq\_v3\_6\_11/priv/www/api/index.html](https://rawcdn.githack.com/rabbitmq/rabbitmq-management/rabbitmq_v3_6_11/priv/www/api/index.html)

Client libraries have been written in several languages to simplify interactions with the RabbitMQ service. Use of these libraries can be a quicker, more effective, and often more standard way of using RabbitMQ.

* Client documentation for Java, .NET, and Erlang:

[www.rabbitmq.com/clients.html](https://www.rabbitmq.com/clients.html)

* Development libraries and tools for other languages:

[www.rabbitmq.com/devtools.html](https://www.rabbitmq.com/devtools.html)

1. <https://content.pivotal.io/blog/messaging-patterns-for-event-driven-microservices> [↑](#footnote-ref-1)
2. <http://amqp.org/about/examples> [↑](#footnote-ref-2)
3. <https://www.rabbitmq.com/getstarted.html> [↑](#footnote-ref-3)
4. <https://www.rabbitmq.com/getstarted.html> [↑](#footnote-ref-4)
5. <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.1/html/Messaging_User_Guide/chap-Messaging_User_Guide-Exchanges.html>. [↑](#footnote-ref-5)
6. <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.1/html/Messaging_User_Guide/chap-Messaging_User_Guide-Exchanges.html>. [↑](#footnote-ref-6)
7. <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.1/html/Messaging_User_Guide/chap-Messaging_User_Guide-Exchanges.html>. [↑](#footnote-ref-7)
8. <https://access.redhat.com/documentation/en-US/Red_Hat_Enterprise_MRG/1.1/html/Messaging_User_Guide/chap-Messaging_User_Guide-Exchanges.html>. [↑](#footnote-ref-8)
9. <https://docs.oracle.com/cd/E19575-01/819-3669/bnced/index.html> [↑](#footnote-ref-9)
10. <https://dzone.com/articles/rabbitmq-request-response> [↑](#footnote-ref-10)
11. <https://insidethecpu.com/2015/05/22/microservices-with-c-and-rabbitmq/> [↑](#footnote-ref-11)