Message Queues 1.0

Summary: This document provides basic information about the extension providing **Message Queues** support for JAVA.

What's new?

Changes in 1.0.0-beta2 & 1.0.0-funcrel:

- Support for Spring AMQP RabbitMQ with SpringBoot for Direct, Topic, Fanout exchanges along with Default exchange
- Support for RabbitMQ with SpringBoot when the listener is configured using Message Listener Adapter or annotations.
- Support for ActiveMQ when JNDI is used to store Queuename.
- Support for ActiveMQ with SpringBoot(when queue is autowired in different file).

Changes in 1.0.0-beta1:

- Support for ActiveMQ with Spring JMS (XML based configuration and Annotation based configuration)
- Support for RabbitMQ with Spring AMQP (XML based configuration) for Direct, Topic, Fanout exchanges along with Default exchange
- Support for RabbitMQ with SpringBoot(Annotation based configuration) for Default exchange

Changes in 1.0.0-alpha1 & 1.0.0-alpha2:

- Support ActiveMQ with OpenWire + JMS for Plain Java
- Support RabbitMQ with AMQP + SLF4J for Plain JAVA for only Default exchange

Description

This extension provides support for **Message Queues for Plain Java and Spring (XML based configuration and Annotation based configuration)**.

In what situation should you install this extension?

This extension should be installed when analyzing a Java project containing **Message Queue** applications, and wanting to view a transaction consisting of queue call and queue receive objects with their corresponding links. This version supports **Plain Java and Spring (XML based configuration and Annotation based configuration)** for both **ActiveMQ** and **RabbitMQ**.

Supported Message Queue versions

The following table displays the supported versions matrix:

Message Queue	Version	Support
ActiveMQ	5.15.3	OpenWire + JMS
ActiveMQ	5.15.3	Spring + JMS with XML based configuration
ActiveMQ	5.15.3	JMS with SpringBoot

RabbitMQ	3.6.9	AMQP + SLF4J
RabbitMQ	3.6.9	Spring AMQP + Spring Rabbit with XML based configuration
RabbitMQ	3.6.9	Spring AMQP with SpringBoot

CAST AIP Compatibility

This extension is compatible with:

CAST AIP release	Supported
8.3.x	✓
8.2.x	✓
8.1.x	✓
8.0.x	✓
7.3.4 and all higher 7.3.x releases	✓

Supported DBMS servers

This extension is compatible with the following DBMS servers:

CAST AIP release	CSS	Oracle	Microsoft
All supported releases	✓	✓	✗

Prerequisities

✓	An installation of any compatible release of CAST AIP (see table above)

Dependencies with other extensions

Some CAST extensions require the presence of other CAST extensions in order to function correctly. The **Message Queue** extension requires that the following other CAST extensions are also installed:

- Web Services Linker
- **CAST AIP Internal extension** ((internal technical extension)

> Note that when using the **CAST Extension Downloader** to download the extension and the **Manage Extensions** interface in **CAST Server Manager** to install the extension, any dependent extensions are **automatically** downloaded and installed for you. You do not need to do anything.

Download and installation instructions

Please see:

- Download an extension
- Install an extension

> The latest release status of this extension can be seen when downloading it from the CAST Extend server.

Packaging, delivering and analyzing your source code

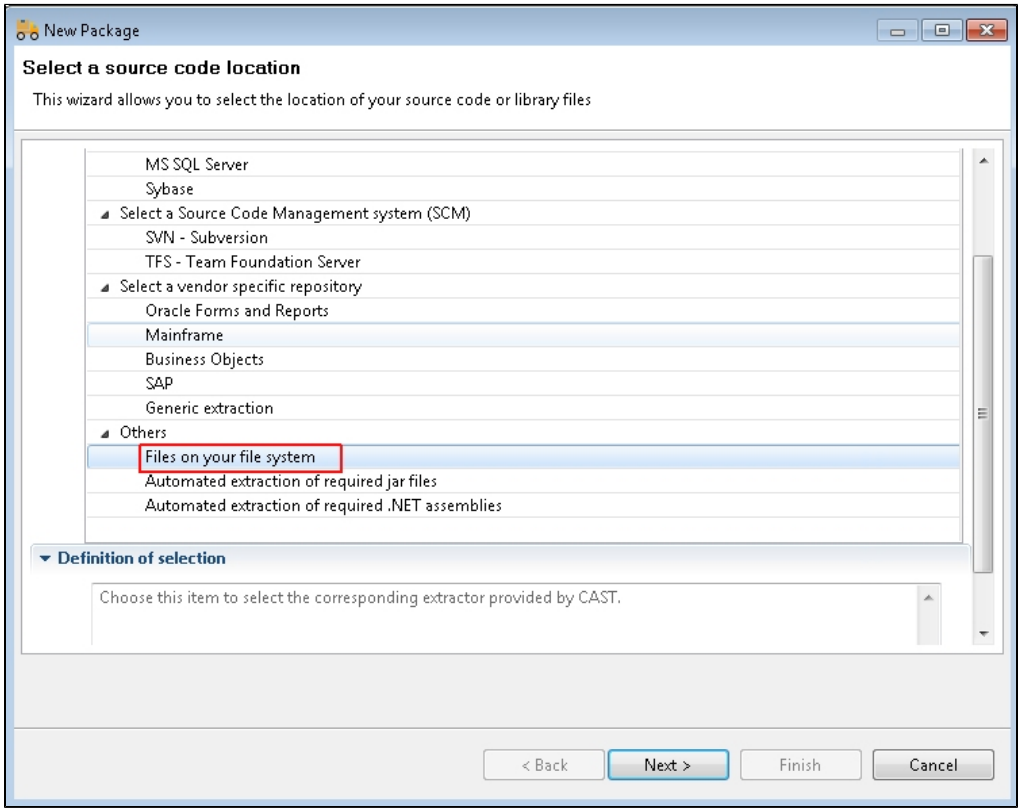
Once the extension is installed, no further configuration changes are required before you can package your source code and run an analysis. The process of packaging, delivering and analyzing your source code is as follows:

Packaging and delivery

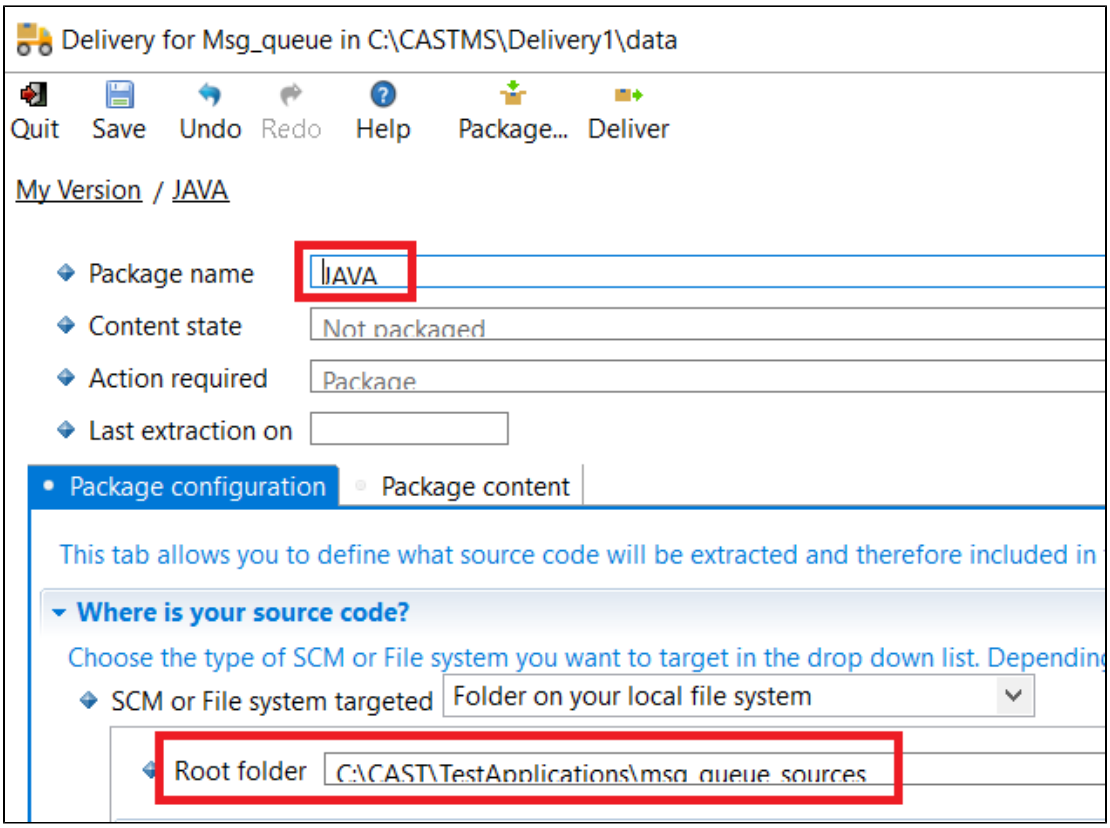
Note that the **Message Queues** extension does not contain any CAST Delivery Manager Tool **discoverers or extractors**, therefore, no "Message Queue" projects will be detected by the DMT. You therefore need to manually create an Analysis Unit in the CAST Management Studio - this is explained below.

Using the CAST Delivery Manager Tool:

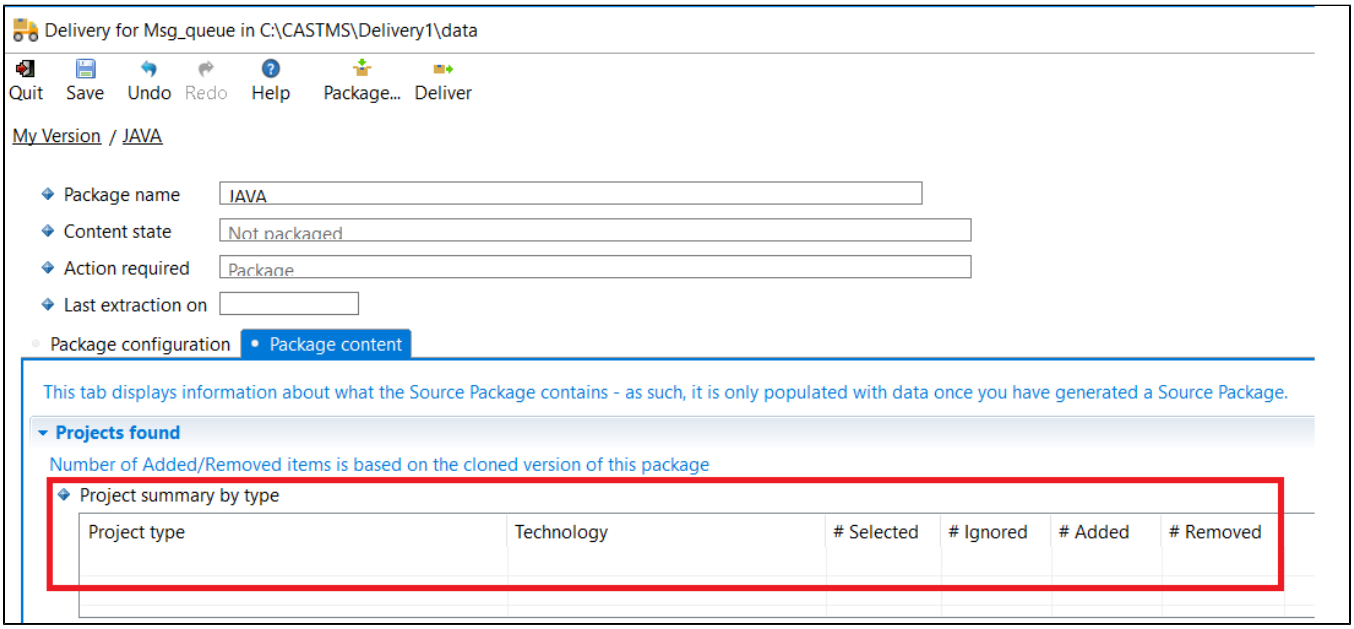
- Create a new **Version**
- Create a new **Package** for your Message Queue source code using the **Files on your file system** option:



- Define a **name** for the package and the **root folder** of your Application source code:



- Run the **Package action**: the CAST Delivery Manager Tool will **not** find any "projects" related to the Message Queue application source code - this is the **expected behavior**. However, if your Java related source code is part of a larger application, then other projects may be found during the package action.

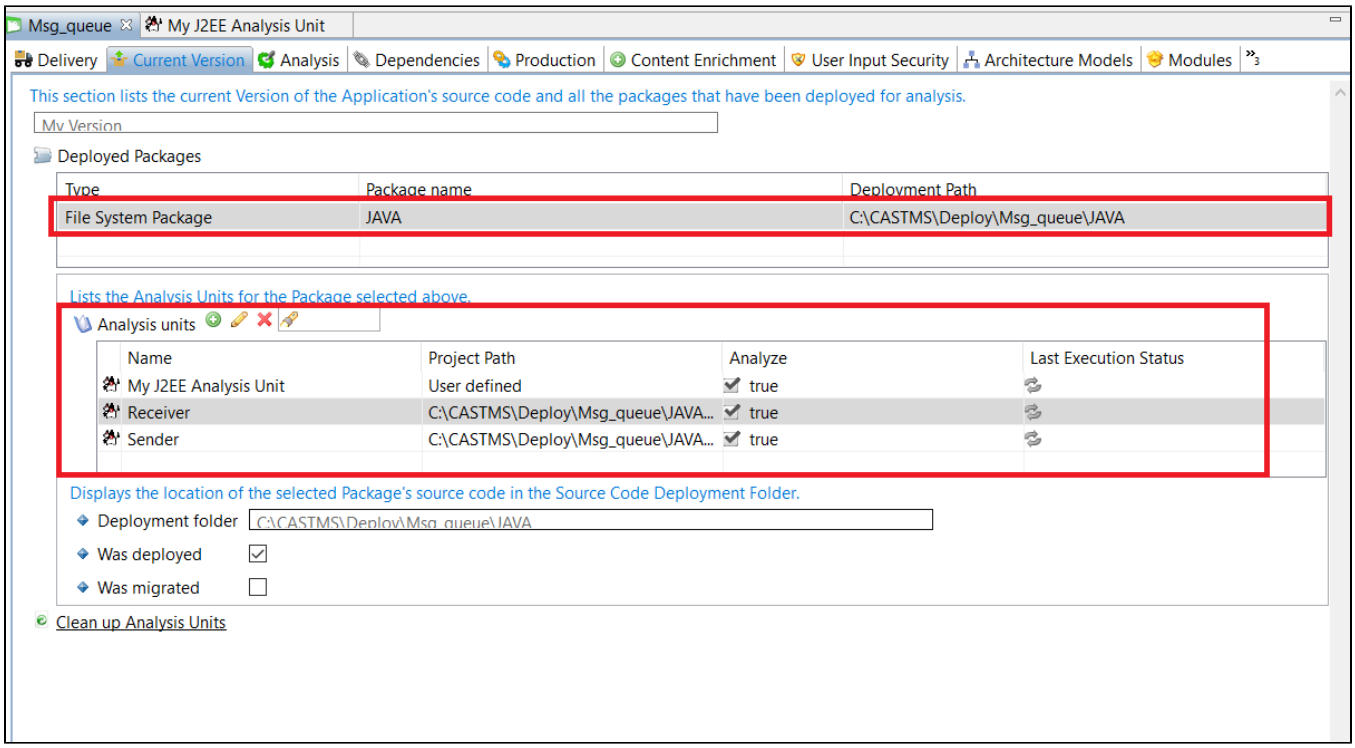


- Deliver the **Version**

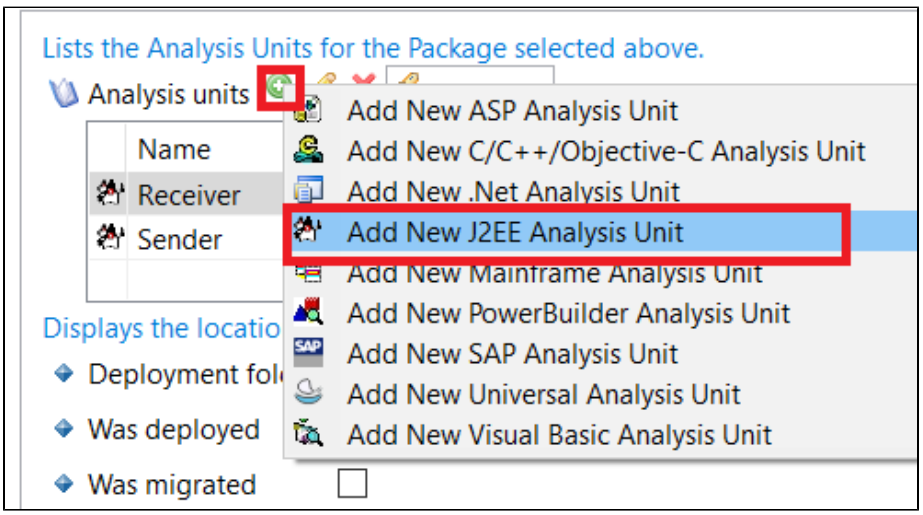
Analyzing

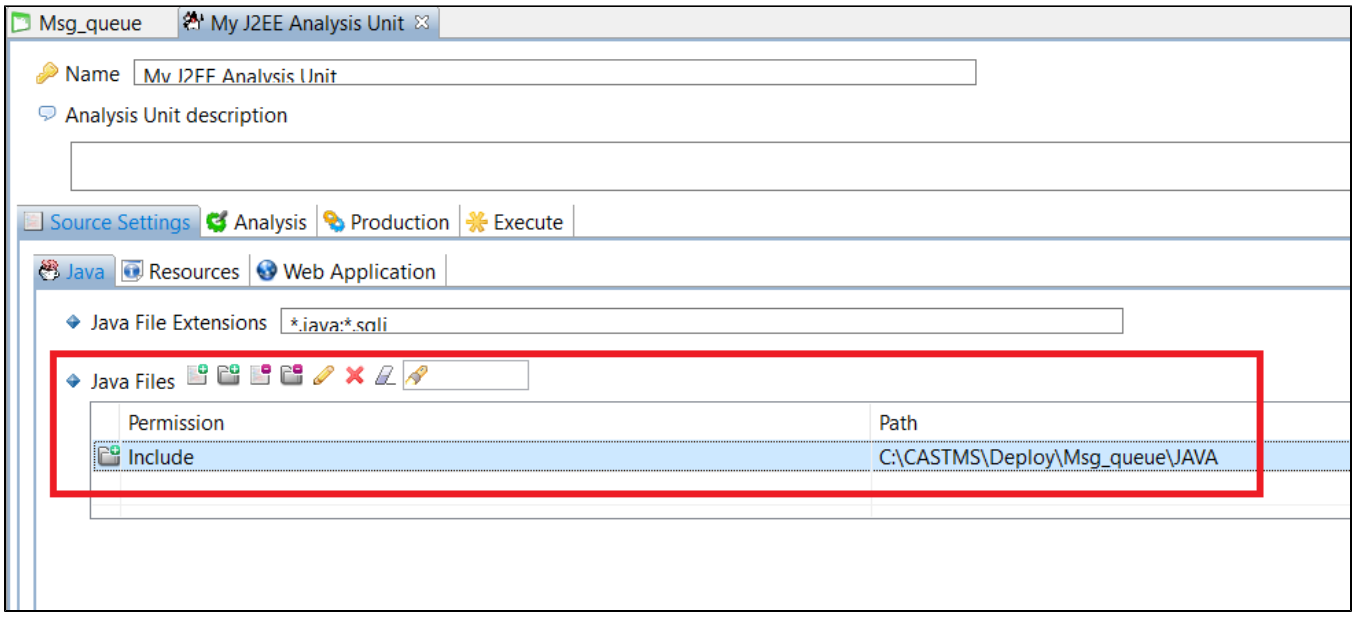
Using the CAST Management Studio:

- Accept and deploy the **Version** in the CAST Management Studio. No **Analysis Units** will be created automatically relating to the Java source code - this is the **expected behavior**. However, if your Message Queue related source code is part of a larger application, then other Analysis Units may be created automatically:



- In the **Current Version tab**, add a new Analysis Unit specifically for your Java source code containing Message Queues, selecting the **Add new J2EE Analysis Unit** option:

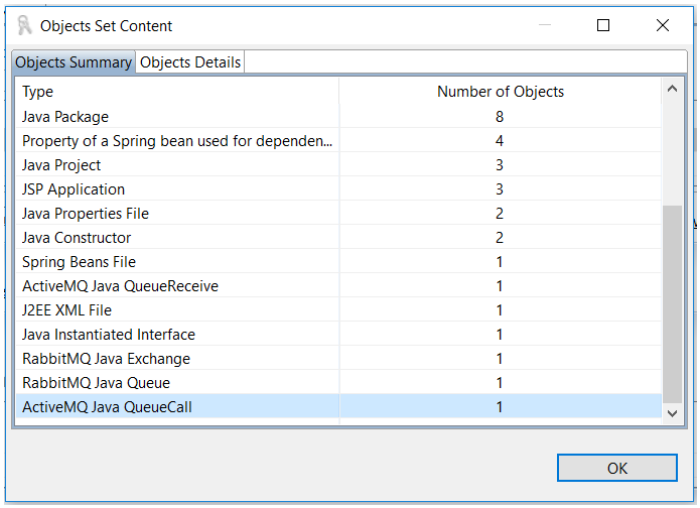




- Run a **test analysis** on the Analysis Unit before you generate a **new snapshot**.

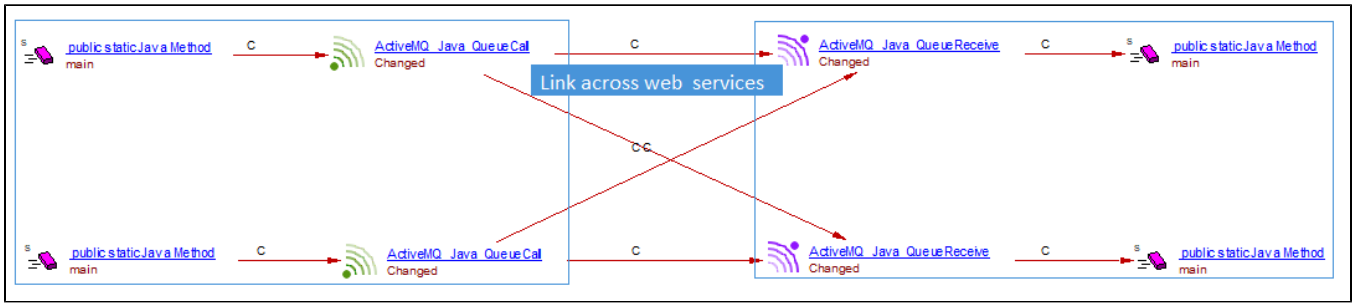
What results can you expect?

Once the analysis/snapshot generation has completed, you can view the results in the normal manner:

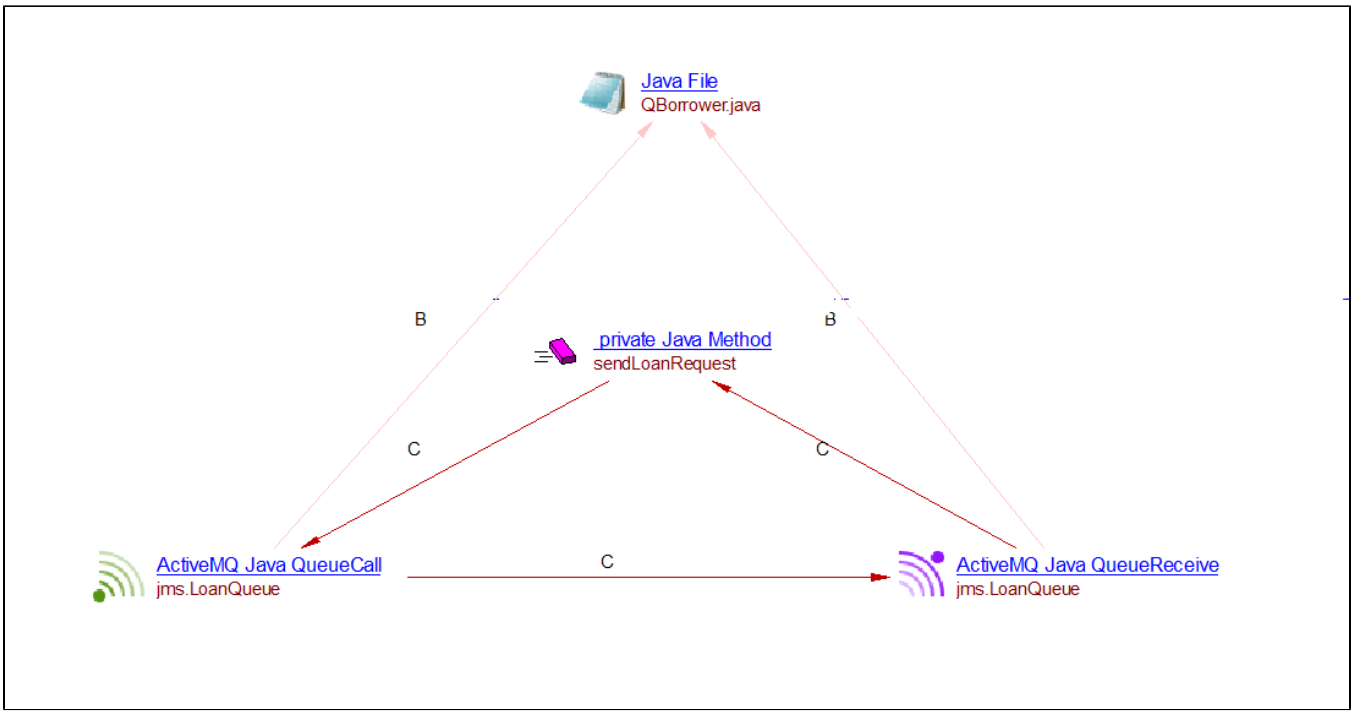


Below are the transactions obtained after analysis:

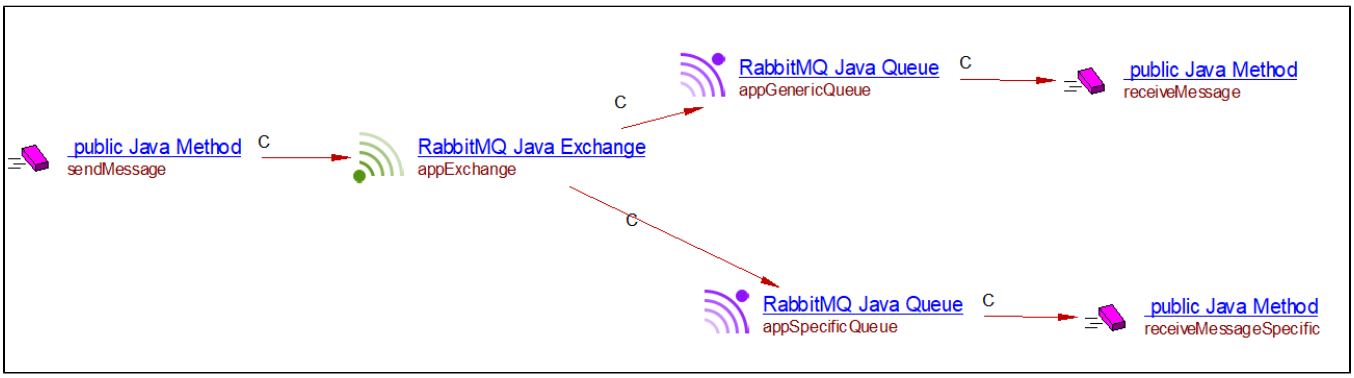
Many-to-Many : ActiveMQ



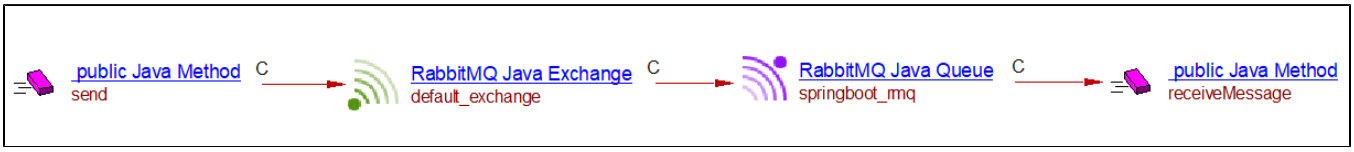
Producer/Consumer in same file:ActiveMQ



One to Many: RabbitMQ Topic Exchange bound to two Queues



RabbitMQ : Sender using Default Exchange to send message to Queue



Objects

The following specific objects are displayed in CAST Enlighten:

Icons	Description
	ActiveMQ Producer Queue Call, RabbitMQ Exchange
	ActiveMQ Consumer Queue Receive, RabbitMQ Queue

Links

For ActiveMQ**, Call link** is created between:

- Producer method object and Queue Call object, at the analyser level
- Consumer method object and Queue Receive object, at the analyser level
- Queue Call object and Queue Receive object, at the Application level by Web Services Linker

For RabbitMQ, **Call link** is created between:

- Producer method object and RabbitMQ Exchange object, at the analyser level
- Consumer method object and RabbitMQ Queue object, at the analyser level/application level
- RabbitMQ Exchange object and RabbitMQ Queue object, at the application level by Web Services Linker

ActiveMQ

Example of ActiveMQ Producer(Spring-XML)

```
 <bean id="destination" class="org.apache.activemq.command.ActiveMQQueue">
 <constructor-arg index="0" value="queue_temp"></constructor-arg>
</bean>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
 <property name="connectionFactory" ref="jmsFactory"></property>
 <property name="defaultDestination" ref="destination" />
</bean>

<bean id="producer" class="me.andycheung.dev.springjms.producer.TestProducer">
 <property name="jmsTemplate" ref="jmsTemplate"/>
</bean>

<bean id="consumer" class="me.andycheung.dev.springjms.consumer.TestConsumer">
 <property name="jmsTemplate" ref="jmsTemplate"></property>
</bean>
```

Example of ActiveMQ Producer(Springboot) :

```
        public class OrderSender {
                        private static Logger log = LoggerFactory.getLogger(OrderSender.class);
                        public static final String ORDER_QUEUE = "Sender_Queue";
                         @Autowired
                            private JmsTemplate jmsTemplate;
                            public void send(Order myMessage) {
                                log.info("sending with convertAndSend() to queue <" + myMessage + ">");
                                jmsTemplate.convertAndSend(ORDER_QUEUE, myMessage); } }
```

Example of ActiveMQ Consumer(Springboot):

```
       public class OrderConsumer {
                       private static Logger log = LoggerFactory.getLogger(OrderConsumer.class);
                           Order received;
                           private CountDownLatch countDownLatch;
                           @JmsListener(destination = Consumer_QUEUE)
                           public void receiveMessage(@Payload Order order,@Headers MessageHeaders headers,
                                                      Message message, Session session){
                                                            received = order;
                                                            log.info("received <" + order + ">"); } }
```

Example of ActiveMQ- JNDI is used to store Queue

```
  public QBorrower() throws NamingException, JMSException {
          Context ctx=new InitialContext();
          QueueConnectionFactory connectionFactory=(QueueConnectionFactory)ctx.lookup("ConnectionFactory");
          queueConnection=connectionFactory.createQueueConnection();
          requestQueue=(Queue)ctx.lookup("jms.LoanRequestQueue");
          responseQueue=(Queue)ctx.lookup("jms.LoanResponseQueue");
          queueConnection.start();
          queueSession=queueConnection.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);
         }

 private void sendLoanRequest(double salary,double loanAmount) throws JMSException {
          MapMessage message=queueSession.createMapMessage();
          message.setDoubleProperty("salary", salary);
          message.setDoubleProperty("loanAmount", loanAmount);
          message.setJMSReplyTo(responseQueue);
          QueueSender sender=queueSession.createSender(requestQueue);
              QueueReceiver queueReceiver=queueSession.createReceiver(responseQueue);
           sender.send(message);
        }
```

RabbitMQ

Example of Spring AMQP RabbitMQ Producer:

```
         @Service
         public class CustomMessageSender {
             private static final Logger log = LoggerFactory.getLogger(CustomMessageSender.class);
             private final RabbitTemplate rabbitTemplate;
             @Autowired
             public CustomMessageSender(final RabbitTemplate rabbitTemplate) {
                 this.rabbitTemplate = rabbitTemplate;
             }
         @Scheduled(fixedDelay = 3000L)
         public void sendMessage() {
             final CustomMessage message = new CustomMessage("Hello there!", new Random().nextInt(50), false);
             log.info("Sending message...");
             rabbitTemplate.convertAndSend(MessagingApplication.EXCHANGE_NAME, MessagingApplication.
ROUTING_KEY, message);
          }  }
```

Example of Spring AMQP RabbitMQ Consumer:

```
    @Service
    public class CustomMessageListener {
        private static final Logger log = LoggerFactory.getLogger(CustomMessageListener.class);
        @RabbitListener(queues = MessagingApplication.QUEUE_GENERIC_NAME)
        public void receiveMessage(final Message message) {
            log.info("Received message as generic: {}", message.toString());
        }
        @RabbitListener(queues = MessagingApplication.QUEUE_SPECIFIC_NAME)
        public void receiveMessageSpecific(final CustomMessage customMessage) {
            log.info("Received message as specific class: {}", customMessage.toString());
       } }
```

Example of SpringBoot RabbitMQ Exchange-Queue Binding configuration:

```
public class MessagingApplication implements RabbitListenerConfigurer{
     public static final String EXCHANGE_NAME = "appExchange";
     public static final String QUEUE_GENERIC_NAME = "appGenericQueue";
     public static final String QUEUE_SPECIFIC_NAME = "appSpecificQueue";
     public static final String ROUTING_KEY = "messages.key";
     public static void main(String[] args) {
          SpringApplication.run(MessagingApplication.class, args);
     }
     @Bean
      public TopicExchange appExchange() {
          return new TopicExchange(EXCHANGE_NAME);
     }
     @Bean
     public Queue appQueueGeneric() {
          return new Queue(QUEUE_GENERIC_NAME);
     }
     @Bean
     public Queue appQueueSpecific() {
          return new Queue(QUEUE_SPECIFIC_NAME);
     }
     @Bean
     public Binding declareBindingGeneric() {
          return BindingBuilder.bind (appQueueGeneric()).to(appExchange()).with(ROUTING_KEY);
     }
     @Bean
     public Binding declareBindingSpecific() {
          return BindingBuilder.bind(appQueueSpecific()).to(appExchange()).with(ROUTING_KEY);
     }
```

Example of Spring AMQP RabbitMQ XML based configuration:

```
<rabbit:template id="amqpTemplate" exchange="myExchange" routing-key="foo.bar" /> <rabbit:queue name="myQueue"
/>
 <rabbit:topic-exchange name="myExchange">
   <rabbit:bindings>
           <rabbit:binding queue="myQueue" pattern="foo.*" />
   </rabbit:bindings>
</rabbit:topic-exchange>
<rabbit:listener-container connection-factory="connectionFactory">
   <rabbit:listener ref="consumer" method="listen" queue-names="myQueue" />
</rabbit:listener-container>
<bean id="consumer" class="com.baeldung.springamqp.consumer.Consumer" />
```

Limitations

The following cases are not handled:

- When the queue name is given at the runtime i.e. when **Queue name** is not initialized anywhere in the code and is given **dynamically** during the session/connection