Spring AMQP - Reference Documentation

1.0.0.M2

Preface

The Spring AMQP project applies core Spring concepts to the development of AMQP-based messaging solutions. We provide a "template" as a high-level abstraction for sending and receiving messages. We also provide support for Message-driven POJOs. These libraries facilitate management of AMQP resources while promoting the use of dependency injection and declarative configuration. In all of these cases, you will see similarities to the JMS support in the Spring Framework. The project consists of both Java and .NET versions. This manual is dedicated to the Java version. For links to the .NET version's manual or any other project-related information visit the Spring AMQP project homepage.

Part I. Reference

This part of the reference documentation details the various components that comprise Spring AMQP. The main chapter covers the core classes to develop an AMQP application. It part also includes a chapter on integration with Erlang and a chapter about the sample applications.

Chapter 1. Using Spring AMQP

In this chapter, we will explore interface and classes that are the essential components for developing applications with Spring AMQP.

1.1. AMQP Abstractions

The Spring AMQP project consists of a handful of modules, each represented by a JAR in the distribution. These modules are: spring-amqp, spring-rabbit, spring-erlang, and spring-rabbit-admin. The 'spring-amqp' module contains the org.springframework.amqp.core package. Within that package, you will find the classes that represent the core AMQP "model". Our intention is to provide generic abstractions that do not rely on any particular AMQP broker implementation or client library. End user code will be more portable across vendor implementations as it can be developed against the abstraction layer only. These abstractions are then used implemented by broker-specific modules, such as 'spring-rabbit'. For the M1 release there is only a RabbitMQ implementation however the abstractions have been vetted in .NET using Apache Qpid in addition to RabbitMQ.

The overview here assumes that you are already familiar with the basics of the AMQP specification already. If you are not, then have a look at the resources listed in Part III, “Other Resources”

1.1.1. Message

The 0-8 and 0-9-1 AMQP specifications do not define an Message class or interface. Instead, when performing an operation such as ' `basicPublish` ', the content is passed as a byte-array argument and additional properties are passed in as separate arguments. Spring AMQP defines a Message class as part of a more general AMQP domain model representation. The purpose of the Message class is to simply encapsulate the body and properties within a single instance so that the API can in turn be simpler. The Message class definition is quite straightforward.

```
public class Message {

    private final MessageProperties messageProperties;

    private final byte[] body;

    public Message(byte[] body, MessageProperties messageProperties) {
        this.body = body;
        this.messageProperties = messageProperties;
    }

    public byte[] getBody() {
        return this.body;
    }

    public MessageProperties getMessageProperties() {
        return this.messageProperties;
}
```

The `MessageProperties` interface defines several common properties such as 'messageId', 'timestamp', 'contentType', and several more. Those properties can also be extended with user-defined 'headers' by calling the `setHeader(String key, Object value)` method.

1.1.2. Exchange

The `Exchange` interface represents an AMQP Exchange, which is what a Message Producer sends to. Each

Exchange within a virtual host of a broker will have a unique name as well as a few other properties:

```
public interface Exchange {

    String getName();

    ExchangeType getExchangeType();

    boolean isDurable();

    boolean isAutoDelete();

    Map<String, Object> getArguments();

}
```

As you can see, an Exchange also has a 'type' represented by the enumeration `ExchangeType`. The basic types are: `Direct`, `Topic` and `Fanout`. In the core package you will find implementations of the `Exchange` interface for each of those types. The behavior varies across these Exchange types in terms of how they handle bindings to Queues. A Direct exchange allows for a Queue to be bound by a fixed routing key (often the Queue's name). A Topic exchange supports bindings with routing patterns that may include the '*' and '#' wildcards for 'exactly-one' and 'zero-or-more', respectively. The Fanout exchange publishes to all Queues that are bound to it without taking any routing key into consideration. For much more information about Exchange types, check out Part III, “Other Resources”.

Note

The AMQP specification also requires that any broker provide a "default" Direct Exchange that has no name. All Queues that are declared will be bound to that default Exchange with their names as routing keys. You will learn more about the default Exchange's usage within Spring AMQP in Section 1.3, “AmqpTemplate”.

1.1.3. Queue

The `Queue` class represents the component from which a Message Consumer receives Messages. Like the various Exchange classes, our implementation is intended to be an abstract representation of this core AMQP type.

```
public class Queue  {

    private final String name;

    private volatile boolean durable;

    private volatile boolean exclusive;

    private volatile boolean autoDelete;

    private volatile Map<String, Object> arguments;

    public Queue(String name) {
        this.name = name;
    }

    // Getters and Setters omitted for brevity

```

Notice that the constructor takes the Queue name. Depending on the implementation, the admin template may provide methods for generating a uniquely named Queue. Such Queues can be useful as a "reply-to" address or other *temporary* situations. For that reason, the 'exclusive' and 'autoDelete' properties of an auto-generated Queue would both be set to 'true'.

1.1.4. Binding

Given that a producer sends to an Exchange and a consumer receives from a Queue, the bindings that connect Queues to Exchanges are critical for connecting those producers and consumers via messaging. In Spring AMQP, we define a `Binding` class to represent those connections. Let's review the basic options for binding Queues to Exchanges.

You can bind a Queue to a DirectExchange with a fixed routing key.

```
new Binding(someQueue, someDirectExchange, "foo.bar")
```

You can bind a Queue to a TopicExchange with a routing pattern.

```
new Binding(someQueue, someTopicExchange, "foo.*")
```

You can bind a Queue to a FanoutExchange with no routing key.

```
new Binding(someQueue, someFanoutExchange)
```

We also provide a `BindingBuilder` to facilitate a "fluent API" style.

```
Binding b = BindingBuilder.from(someQueue).to(someTopicExchange).with("foo.*");
```

> **Note**
>
> The BindingBuilder class is shown above for clarity, but this style works well when using a static import for the 'from()' method.

By itself, an instance of the Binding class is just holding the data about a connection. In other words, it is not an "active" component. However, as you will see later in Section 1.7, “Configuring the broker”, Binding instances can be used by the `AmqpAdmin` class to actually trigger the binding actions on the broker. Also, as you will see in that same section, the Binding instances can be defined using Spring's `@Bean`-style within `@Configuration` classes. There is also a convenient base class which further simplifies that approach for generating AMQP-related bean definitions and recognizes the Queues, Exchanges, and Bindings so that they will all be declared on the AMQP broker upon application startup.

The `AmqpTemplate` is also defined within the core package. As one of the main components involved in actual AMQP messaging, it is discussed in detail in its own section (see Section 1.3, “AmqpTemplate”).

1.2. Connection and Resource Management

Whereas the AMQP model we described in the previous section is generic and applicable to all implementations, when we get into the management of resources, the details are specific to the broker implementation. Therefore, in this section, we will be focusing on code that exists only within our "spring-rabbit" module since at this point, RabbitMQ is the only supported implementation.

The central component for managing a connection to the RabbitMQ broker is the `ConnectionFactory` interface. The responsibility of a `ConnectionFactory` implementation is to provide an instance of `com.rabbitmq.client.Connection`. The simplest implementation we provide is `SingleConnectionFactory` which establishes a single connection that can be shared by the application. Sharing of the connection is possible since the "unit of work" for messaging with AMQP is actually a "channel" (in some ways, this is similar to the relationship between a Connection and a Session in JMS). As you can imagine, the connection instance provides a `createChannel` method. When creating an instance of `SingleConnectionFactory`, the 'hostname' can be provided via the constructor. The 'username' and 'password' properties should be provided as well.

```
SingleConnectionFactory connectionFactory = new SingleConnectionFactory("somehost");
connectionFactory.setUsername("guest");
connectionFactory.setPassword("guest");

Connection connection = connectionFactory.createConnection();
```

When using XML, the configuration might look like this:

```
<bean id="cf" class="org.springframework.amqp.rabbit.connection.SingleConnectionFactory">
    <constructor-arg value="somehost"/>
    <property name="username" value="guest"/>
    <property name="password" value="guest"/>
</bean>
```

Note

You may also discover the `CachingConnectionFactory` implementation, but at this time, that code is considered *experimental* . We recommend sticking with `SingleConnectionFactory` for now as the caching implementation will most likely evolve. Support for failover of connections is also planned.

1.3. AmqpTemplate

As with many other high-level abstractions provided by the Spring Framework and related projects, Spring AMQP provides a "template" that plays a central role. The interface that defines the main operations is called `AmqpTemplate`. Those operations cover the general behavior for sending and receiving Messages. In other words, they are not unique to any implementation, hence the "AMQP" in the name. On the other hand, there are implementations of that interface that are tied to implementations of the AMQP protocol. Unlike JMS, which is an interface-level API itself, AMQP is a wire-level protocol. The implementations of that protocol provide their own client libraries, so each implementation of the template interface will depend on a particular client library. Currently, there is only a single implementation: `RabbitTemplate`. In the examples that follow, you will often see usage of an "AmqpTemplate", but when you look at the configuration examples, or any code excerpts where the template is instantiated and/or setters are invoked, you will see the implementation type (e.g. "RabbitTemplate").

As mentioned above, the `AmqpTemplate` interface defines all of the basic operations for sending and receiving Messages. We will explore Message sending and reception, respectively, in the two sections that follow.

1.4. Sending messages

When sending a Message, one can use any of the following methods:

```
void send(MessageCreator messageCreator) throws AmqpException;

void send(String routingKey, MessageCreator messageCreator) throws AmqpException;

void send(String exchange, String routingKey, MessageCreator messageCreator) throws AmqpException;
```

We can begin our discussion with the last method listed above since it is actually the most explicit. It allows an AMQP Exchange name to be provided at runtime along with a routing key. The last parameter is the callback that is responsible for actual creating of the Message instance. An example of using this method to send a Message might look this this:

```
amqpTemplate.send("marketData.topic", "quotes.nasdaq.FOO", new MessageCreator() {
    public Message createMessage() {
        return new Message("12.34".getBytes(), someProperties);
    }
});
```

The "exchange" property can be set on the template itself if you plan to use that template instance to send to the same exchange most or all of the time. In such cases, the second method listed above may be used instead. The following example is functionally equivalent to the previous one:

```
amqpTemplate.setExchange("marketData.topic");
amqpTemplate.send("quotes.nasdaq.FOO", new MessageCreator() {
    public Message createMessage() {
        return new Message("12.34".getBytes(), someProperties);
    }
});
```

If both the "exchange" and "routingKey" properties are set on the template, then the method accepting only the `MessageCreator` may be used:

```
amqpTemplate.setExchange("marketData.topic");
amqpTemplate.setRoutingKey("quotes.nasdaq.FOO");
amqpTemplate.send(new MessageCreator() {
    public Message createMessage() {
        return new Message("12.34".getBytes(), someProperties);
    }
});
```

A better way of thinking about the exchange and routing key properties is that the explicit method parameters will always override the template's default values. In fact, even if you do not explicitly set those properties on the template, there are always default values in place. In both cases, the default is an empty String, but that is actually a sensible default. As far as the routing key is concerned, it's not always necessary in the first place (e.g. a Fanout Exchange). Furthermore, a Queue may be bound to an Exchange with an empty String. Those are both legitimate scenarios for reliance on the default empty String value for the routing key property of the template. As far as the Exchange name is concerned, the empty String is quite commonly used because the AMQP specification defines the "default Exchange" as having no name. Since all Queues are automatically bound to that default Exchange (which is a Direct Exchange) using their name as the binding value, that second method above can be used for simple point-to-point Messaging to any Queue through the default Exchange. Simply provide the queue name as the "routingKey" - either by providing the method parameter at runtime:

```
RabbitTemplate template = new RabbitTemplate(); // using default no-name Exchange
template.send("queue.helloWorld", new MessageCreator() {
    public Message createMessage() {
        return new Message("Hello World".getBytes(), someProperties);
    }
});
```

Or, if you prefer to create a template that will be used for publishing primarily or exclusively to a single Queue, the following is perfectly reasonable:

```
RabbitTemplate template = new RabbitTemplate(); // using default no-name Exchange
template.setRoutingKey("queue.helloWorld"); // but we'll always send to this Queue
template.send(new MessageCreator() {
    public Message createMessage() {
        return new Message("Hello World".getBytes(), someProperties);
```

```
    }
});
```

1.5. Receiving messages

Message reception is always a bit more complicated than sending. The reason is that there are two ways to receive a Message. The simpler option is to poll for a single Message at a time with a synchronous, blocking method call. The more complicated yet more common approach is to register a listener that will receive Messages on-demand, asynchronously. We will look at an example of each approach in the next two sub-sections.

1.5.1. Synchronous Reception

The `AmqpTemplate` itself can be used for synchronous Message reception. There are two 'receive' methods available. As with the Exchange on the sending side, there is a method that requires a queue property having been set directly on the template itself, and there is a method that accepts a queue parameter at runtime.

```
Message receive() throws AmqpException;

Message receive(String queueName) throws AmqpException;
```

1.5.2. Asynchronous Reception

For asynchronous Message reception, a dedicated component other than the `AmqpTemplate` is involved. That component is a container for a Message consuming callback. We will look at the container and its properties in just a moment, but first we should look at the callback since that is where your application code will be integrated with the messaging system. There are a few options for the callback. The simplest of these is to implement the `MessageListener` interface:

```
public interface MessageListener {

    void onMessage(Message message);

}
```

If your callback logic depends upon the AMQP Channel instance for any reason, you may instead use the `ChannelAwareMessageListener`. It looks similar but with an extra parameter:

```
public interface ChannelAwareMessageListener<M extends Message> {

    void onMessage(M message, Channel channel) throws Exception;

}
```

If you prefer to maintain a stricter separation between your application logic and the messaging API, you can rely upon an adapter implementation that is provided by the framework. This is often referred to as "Message-driven POJO" support. When using the adapter, you only need to provide a reference to the instance that the adapter itself should invoke.

```
MessageListener listener = new MessageListenerAdapter(somePojo);
```

Now that you've seen the various options for the Message-listening callback, we can turn our attention to the container. Basically, the container handles the "active" responsibilities so that the listener callback can remain passive. The container is an example of a "lifecycle" component. It provides methods for starting and stopping. When configuring the container, you are essentially bridging the gap between an AMQP Queue and the `MessageListener` instance. You must provide a reference to the `ConnectionFactory` and the queue name or Queue instance(s) from which that listener should consume Messages. Here is the most basic example using the default implementation, `SimpleMessageListenerContainer` :

```
SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
container.setConnectionFactory(rabbitConnectionFactory);
container.setQueueName("some.queue");
container.setMessageListener(someListener);
```

As an "active" component, it's most common to create the listener container with a bean definition so that it can simply run in the background. This can be done via XML:

```
<bean class="org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer">
    <property name="connectionFactory" ref="rabbitConnectionFactory"/>
    <property name="queueName" value="some.queue"/>
    <property name="messageListener" ref="someListener"/>
</bean>
```

Or, you may prefer to use the @Configuration style which will look very similar to the actual code snippet above:

```
@Configuration
public class ExampleAmqpConfiguration {

    @Bean
    public MessageListenerContainer messageListenerContainer {
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
        container.setConnectionFactory(rabbitConnectionFactory());
        container.setQueueName("some.queue");
        container.setMessageListener(exampleListener());
        return container;
    }

    @Bean
    public ConnectionFactory rabbitConnectionFactory() {
        SingleConnectionFactory connectionFactory = new SingleConnectionFactory("localhost");
        connectionFactory.setUsername("guest");
        connectionFactory.setPassword("guest");
        return connectionFactory;
    }

    @Bean
    public MessageListener exampleListener() {
        return new MessageListener() {
            public void onMessage(Message message) {
                System.out.println("received: " + message)
            }
        }
    }
}
```

1.6. Message Converters

The `AmqpTemplate` also defines several methods for sending and receiving Messages that will delegate to a `MessageConverter`. The `MessageConverter` itself is quite straightforward. It provides a single method for each direction: one for converting *to* a Message and another for converting *from* a Message. Notice that when converting to a Message, you may also provide properties in addition to the object. The "object" parameter typically corresponds to the Message body.

```
public interface MessageConverter {

    Message toMessage(Object object, MessageProperties messageProperties)
            throws MessageConversionException;

```

```
    Object fromMessage(Message message) throws MessageConversionException;

}
```

The relevant Message-sending methods on the `AmqpTemplate` are listed below. They are simpler than the methods we discussed previously because they do not require the MessageCreator callback. Instead, the MessageConverter is responsible for "creating" each Message by converting the provided object to the byte array for the Message body and then adding any provided MessageProperties.

```
void convertAndSend(Object message) throws AmqpException;

void convertAndSend(String routingKey, Object message) throws AmqpException;

void convertAndSend(String exchange, String routingKey, Object message) throws AmqpException;

void convertAndSend(Object message, MessagePostProcessor messagePostProcessor) throws AmqpException;

void convertAndSend(String routingKey, Object message, MessagePostProcessor messagePostProcessor)
        throws AmqpException;

void convertAndSend(String exchange, String routingKey, Object message,
        MessagePostProcessor messagePostProcessor) throws AmqpException;
```

On the receiving side, there are only two methods: one that accepts the queue name and one that relies on the template's "queue" property having been set.

```
Object receiveAndConvert() throws AmqpException;

Object receiveAndConvert(String queueName) throws AmqpException;
```

1.6.1. SimpleMessageConverter

The default implementation of the `MessageConverter` strategy is called `SimpleMessageConverter`. This is the converter that will be used by an instance of RabbitTemplate if you do not explicitly configure an alternative. It handles text-based content, serialized Java objects, and simple byte arrays.

1.6.1.1. Converting From a Message

If the content type of the input Message begins with "text" (e.g. "text/plain"), it will also check for the content-encoding property to determine the charset to be used when converting the Message body byte array to a Java String. If no content-encoding property had been set on the input Message, it will use the "UTF-8" charset by default. If you need to override that default setting, you can configure an instance of `SimpleMessageConverter`, set its "defaultCharset" property and then inject that into a `RabbitTemplate` instance.

If the content-type property value of the input Message is set to "application/x-java-serialized-object", the `SimpleMessageConverter` will attempt to deserialize (rehydrate) the byte array into a Java object. While that might be useful for simple prototyping, it's generally not recommended to rely on Java serialization since it leads to tight coupling between the producer and consumer. Of course, it also rules out usage of non-Java systems on either side. With AMQP being a wire-level protocol, it would be unfortunate to lose much of that advantage with such restrictions. In the next two sections, we'll explore some alternatives for passing rich domain object content without relying on Java serialization.

For all other content-types, the `SimpleMessageConverter` will return the Message body content directly as a byte array.

1.6.1.2. Converting To a Message

When converting to a Message from an arbitrary Java Object, the `SimpleMessageConverter` likewise deals with byte arrays, Strings, and Serializable instances. It will convert each of these to bytes (in the case of byte arrays, there is nothing to convert), and it will set the content-type property accordingly. If the Object to be converted does not match one of those types, the Message body will be null.

1.6.2. JsonMessageConverter

As mentioned in the previous section, relying on Java serialization is generally not recommended. One rather common alternative that is more flexible and portable across different languages and platforms is JSON (JavaScript Object Notation). An implementation is available and can be configured on any `RabbitTemplate` instance to override its usage of the `SimpleMessageConverter` default.

```
<bean class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <property name="connectionFactory" ref="rabbitConnectionFactory"/>
    <property name="messageConverter">
        <bean class="org.springframework.amqp.support.converter.JsonMessageConverter">
            <!-- if necessary, override the DefaultClassMapper -->
            <property name="classMapper" ref="customClassMapper"/>
        </bean>
    </property>
</bean>
```

1.6.3. MarshallingMessageConverter

Yet another option is the `MarshallingMessageConverter`. It delegates to the Spring OXM library's implementations of the `Marshaller` and `Unmarshaller` strategy interfaces. You can read more about that library here. In terms of configuration, it's most common to provide the constructor argument only since most implementations of `Marshaller` will also implement `Unmarshaller`.

```
<bean class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <property name="connectionFactory" ref="rabbitConnectionFactory"/>
    <property name="messageConverter">
        <bean class="org.springframework.amqp.support.converter.MarshallingMessageConverter">
            <constructor-arg ref="someImplemenationOfMarshallerAndUnmarshaller"/>
        </bean>
    </property>
</bean>
```

1.7. Configuring the broker

The AMQP specification describes how the protocol can be used to configure Queues, Exchanges and Bindings on the broker. These operations which are portable from the 0.8 specification and higher are present in the AmqpAdmin interface in the org.springframework.amqp.core package. The RabbitMQ implementation of that class is RabbitAdmin located in the org.springframework.amqp.rabbit.core package. Any many configuration and management functions are broker specific and not included in the AMQP specification, the interface RabbitBrokerOperations and its implementation RabbitBrokerAdmin located in the org.springframework.amqp.rabbit.admin package is provided to fill that gap.

The AmqpAdmin interface is based on using the Spring AMQP domain abstractions and is shown below:

```
public interface AmqpAdmin {

    // Exchange Operations

```

```
    void declareExchange(Exchange exchange);

    void deleteExchange(String exchangeName);

    // Queue Operations

    Queue declareQueue();

    void declareQueue(Queue queue);

    void deleteQueue(String queueName);

    void deleteQueue(String queueName, boolean unused, boolean empty);

    void purgeQueue(String queueName, boolean noWait);

    // Binding Operations

    void declareBinding(Binding binding);

}
```

The declareQueue() method defined a queue on the broker whose name is automatically created. The additional properties of this auto-generated queue are exclusive=true, autoDelete=true, and durable=false.

Note

Removing a binding was not introduced until the 0.9 version of the AMQP spec.

The RabbitMQ implementation of this interface is RabbitAdmin which when configured using Spring XML would look like this:

```
<bean id="cf" class="org.springframework.amqp.rabbit.connection.SingleConnectionFactory">
    <constructor-arg value="localhost"/>
    <property name="username" value="guest"/>
    <property name="password" value="guest"/>
</bean>

<bean id="amqpAdmin" class="org.springframework.amqp.rabbit.core.RabbitAdmin">
    <constructor-arg ref="cf"/>
</bean>
```

The class AbstractAmqpConfiguration can also be used to configure an instance of the AmqpAmin interface. The base class AbstractAmqpConfiguration is located in the package org.springframework.amqp.config and is shown in part below.

```
@Configuration
public abstract class AbstractAmqpConfiguration implements ApplicationContextAware, SmartLifecycle {

  @Bean
  public abstract AmqpAdmin amqpAdmin();

  // the rest omitted for brevity.
}
```

The part that is omitted is related to the implementation of SmartLifecycle which is responsible for querying the container for all Queues, Exchanges, and Bindings that are defined in the DI container and declaring them to the broker. Thus if you are using the @Configuration style of configuration, you can simply create Queue, Exchange, and Binding bean definitions and they will be declared to the broker once the application has started.

If you are using RabbitMQ as your broker, the additional abstract @Configuration class can be used to bootstrap an implementation of AmqpAdmin as shown below

```
@Configuration
public abstract class AbstractRabbitConfiguration extends AbstractAmqpConfiguration {

  @Bean
  public abstract RabbitTemplate rabbitTemplate();

  @Bean
  public AmqpAdmin amqpAdmin() {
    this.amqpAdmin = new RabbitAdmin(rabbitTemplate().getConnectionFactory());
    return this.amqpAdmin;
  }
}
```

This leaves it up to you to provide an implementation of the rabbitTemplate method in your application specific subclass. For example, using the Stock sample application, there is the @Configuration class AbstractStockRabbitConfiguration which in turn has RabbitClientConfiguration and RabbitServerConfiguration subclasses. The code for AbstractStockRabbitConfiguration is show below

```
@Configuration
public abstract class AbstractStockAppRabbitConfiguration extends AbstractRabbitConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() {
        SingleConnectionFactory connectionFactory = new SingleConnectionFactory("localhost");
        connectionFactory.setUsername("guest");
        connectionFactory.setPassword("guest");
        return connectionFactory;
    }

    @Bean
    public RabbitTemplate rabbitTemplate() {
        RabbitTemplate template = new RabbitTemplate(connectionFactory());
        template.setMessageConverter(jsonMessageConverter());
        configureRabbitTemplate(template);
        return template;
    }

    @Bean
    public MessageConverter jsonMessageConverter() {
        return new JsonMessageConverter();
    }

    @Bean
    public TopicExchange marketDataExchange() {
        return new TopicExchange("app.stock.marketdata");
    }

    // additional code omitted for brevity

}
```

In the Stock application, the server is configured using the following @Configuration class

```
@Configuration
public class RabbitServerConfiguration extends AbstractStockAppRabbitConfiguration  {

    @Bean
    public Queue stockRequestQueue() {
        return new Queue("app.stock.request");
    }
}
```

This is the end of the whole inheritance chain of @Configuration classes. The end result is the the TopicExchange and Queue will be declared to the broker upon application startup. There is no binding of the TopicExchange to a queue in the server configuration, as that is done in the client application. The stock request queue however is automatically bound to the AMQP default exchange - this behavior is defined by the specification.

The client @Configuration class is a little more interesting and is show below.

```
@Configuration
public class RabbitClientConfiguration extends AbstractStockAppRabbitConfiguration {

    @Value("${stocks.quote.pattern}")
    private String marketDataRoutingKey;

    @Bean
    public Queue marketDataQueue() {
        return amqpAdmin().declareQueue();
    }

    /**
     * Binds to the market data exchange. Interested in any stock quotes.
     */
    @Bean
    public Binding marketDataBinding() {
        return BindingBuilder.from(
                marketDataQueue()).to(marketDataExchange()).with(marketDataRoutingKey);
    }

    // additional code omitted for brevity

}
```

The client is declaring another queue via the declareQueue() method on the AmqpAdmin, and it binds that queue to the market data exchange with a routing pattern that is externalized in a properties file.

1.8. Exception Handling

Many operations with the RabbitMQ Java client can throw checked Exceptions. For example, there are a lot of cases where IOExceptions may be thrown. The RabbitTemplate, SimpleMessageListenerContainer, and other Spring AMQP components will catch those Exceptions and convert into one of the Exceptions within our runtime hierarchy. Those are defined in the 'org.springframework.amqp' package, and AmqpException is the base of the hierarchy.

1.9. Transactions

The Spring Rabbit framework has support for automatic transaction management in the synchronous and asynchronous use cases with a number of different semantics that can be selected declaratively, as is familiar to existing users of Spring transactions. This makes many if not most common messaging patterns very easy to implement.

There are two ways to signal the desired transaction semantics to the framework. In both the `RabbitTemplate` and `SimpleMessageListenerContainer` there is a flag `channelTransacted` which, if true, tells the framework to use a transactional channel and to end all operations (send or receive) with a commit or rollback depending on the outcome, with an exception signalling a rollback. Another signal is to provide an external transaction with one of Spring's `PlatformTransactionManager` implementations as a context for the ongoing operation. If there is already a transaction in progress when the framework is sending or receiving a message, and the `channelTransacted` flag is true, then the commit or rollback of the messaging transaction will be deferred until the end of the current transaction. If the `channelTransacted` flag is false, then no transaction semantics apply to the messaging operation (it is auto-acked).

The `channelTransacted` flag is a configuration time setting: it is declared and processed once when the AMQP components are created, usually at application startup. The external transaction is more dynamic in principle because the system responds to the current Thread state at runtime, but in practice is often also a configuration setting, when the transactions are layed onto an application declaratively.

For synchronous use cases with `RabbitTemplate` the external transaction is provided by the caller, either declaratively or imperatively according to taste (the usual Spring transaction model). An example of a declarative approach (usually preferred because it is non-invasive), where the template has been configured with `channelTransacted=true`:

```
@Transactional
public void doSomething() {
    String incoming = rabbitTemplate.receiveAndConvert();
    // do some more database processing...
    String outgoing = processInDatabaseAndExtractReply(incoming);
    rabbitTemplate.convertAndSend(outgoing);
}
```

A String payload is received, converted and sent as a message body inside a method marked as @Transactional, so if the database processing fails with an exception, the incoming message will be returned to the broker, and the outgoing message will not be sent. This applies to any operations with the `RabbitTemplate` inside a chain of transactional methods (unless the `Channel` is directly manipulated to commit the transactiom early for instance).

For asynchronous use cases with `SimpleMessageListenerContainer` an external transaction has to be requested by the container when it sets up the listener. To signal that an external transaction is required the user provides an implementation of `PlatformTransactionManager` to the container when it is configured. For example:

```
@Configuration
public class ExampleExternalTransactionAmqpConfiguration {

    @Bean
    public MessageListenerContainer messageListenerContainer {
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
        container.setConnectionFactory(rabbitConnectionFactory());
        container.setTransactionManager(transactionManager());
        container.setChannelTransacted(true);
        container.setQueueName("some.queue");
        container.setMessageListener(exampleListener());
        return container;
    }

}
```

In the example above, the transaction manager is added as a dependency injected from another bean definition (not shown), and the `channelTransacted` flag is also set to true. The effect is that if the listener fails with an exception the transaction will be rolled back, and the message will also be returned to the broker. Significantly, if the transaction fails to commit (e.g. a database constraint error, or connectivity problem), then the AMQP transaction will also be rolled back, and message will be returned to the broker. This is sometimes known as a Best Efforts 1 Phase Commit, and is a very powerful pattern for reliable messaging. If the `channelTransacted` flag was set to false in the example above, which is the default, then the external transaction would still be provided for the listener, but all messaging operations would be auto-acked, so the effect is to commit the messaging operations even on a rollback of the business operation.

A note on rollback of received messages. AMQP transactions only apply to messages and acks sent to the broker, so when there is a rollback of a Spring transaction and a message has been received, wheat Spring AMQP has to do is not just rollback the transaction, but also manually reject the message (sort of a nack, but that's not what the specification calls it). Such messages (and any that are unacked when a channel is closed or aborts) go to the back of the queue on a Rabbit broker, and this behaviour is not what some users expect, especially if they come from a JMS background, so it's good to be aware of it. The re-queuing order is not mandated by the AMQP specification, but it makes the broker much more efficient, and also means that if it is under load there is a natural back off before the message can be consumed again.

1.10. JMX Monitoring

All the methods in the RabbitBrokerAdmin class are exposed through JMX through the use of the @ManagedOperations annotation. Please refer to the sample application for how to configure spring to export the RabbitBrokerAdmin.

Chapter 2. Erlang integration

2.1. Introduction

There is an open source project called JInterface that provides a way for Java applications to communicate with an Erlang process. The API is very low level and rather tedious to use and throws checked exceptions. The Spring Erlang module makes accessing functions in Erlang from Java easy, often they can be one liners.

2.2. Communicating with Erlang processes

TODO

2.2.1. Connection Management

TODO

2.2.2. Executing RPC

The interface ErlangOperations is the high level API for interacting with an Erlang process.

```
public interface ErlangOperations {

    <T> T execute(ConnectionCallback<T> action) throws OtpException;

    OtpErlangObject executeErlangRpc(String module, String function, OtpErlangList args)
            throws OtpException;

    OtpErlangObject executeErlangRpc(String module, String function, OtpErlangObject... args)
            throws OtpException;

    OtpErlangObject executeRpc(String module, String function, Object... args)
            throws OtpException;

    Object executeAndConvertRpc(String module, String function,
            ErlangConverter converterToUse, Object... args) throws OtpException;

    // Sweet!
    Object executeAndConvertRpc(String module, String function, Object... args)
            throws OtpException;

}
```

The class that implements this interface is called `ErlangTemplate`. There are a few convenience methods, most notably `executeAndConvertRpc`, as well as the `execute` method which gives you access to the 'native' API of the JInterface project. For simple functions, you can invoke `executeAndConvertRpc` with the appropriate Erlang module name, function, and arguments in a one-liner. For example, here is the implementation of the RabbitBrokerAdmin method 'DeleteUser'

```
@ManagedOperation
public void deleteUser(String username) {
    erlangTemplate.executeAndConvertRpc(
            "rabbit_access_control", "delete_user", username.getBytes());
}
```

As the JInterface library uses specific classes such as OtpErlangDouble, OtpErlangString to represent the primitive types in Erlang RPC calls, there is a converter class that works in concert with ErlangTemplate that knows how to translate from Java primitive types to their Erlang class equivalents. You can also create custom converters and register them with the ErlangTemplate to handle more complex data format translations.

2.2.3. ErlangConverter

The ErlangConverter interface is show below

```
public interface ErlangConverter {

    /**
     * Convert a Java object to a Erlang data type.
     * @param object the object to convert
     * @return the Erlang data type
     * @throws ErlangConversionException in case of conversion failure
     */
    OtpErlangObject toErlang(Object object) throws ErlangConversionException;

    /**
     * Convert from a Erlang data type to a Java object.
     * @param erlangObject the Elang object to convert
     * @return the converted Java object
     * @throws ErlangConversionException in case of conversion failure
     */
    Object fromErlang(OtpErlangObject erlangObject) throws ErlangConversionException;

    /**
     * The return value from executing the Erlang RPC.
     */
    Object fromErlangRpc(String module, String function, OtpErlangObject erlangObject)
            throws ErlangConversionException;
}
```

2.3. Exceptions

The JInterface checked exception hierarchy is translated into a parallel runtime exception hierarchy when executing operations through ErlangTemplate.

Chapter 3. Sample Applications

3.1. Introduction

The Spring AMQP project includes two sample applications. The first is a simple "Hello World" example that demonstrates both synchronous and asynchronous message reception. It provides an excellent starting point for acquiring an understanding of the essential components. The second sample is based on a stock-trading use case to demonstrate the types of interaction that would be common in real world applications. In this chapter, we will provide a quick walk-through of each sample so that you can focus on the most important components. The samples are available in the distribution, and they are both Maven-based, so you should be able to import them directly into any Maven-aware IDE (such as SpringSource Tool Suite).

3.2. Hello World

The Hello World sample demonstrates both synchronous and asynchronous message reception. You can import the 'spring-rabbit-helloworld' sample into the IDE and then follow the discussion below.

3.2.1. Synchronous Example

Within the 'src/main/java' directory, navigate to the 'org.springframework.amqp.helloworld' package. Open the HelloWorldConfiguration class and notice that it contains the @Configuration annotation at class-level and some @Bean annotations at method-level. This is an example of Spring's Java-based configuration. You can read more about that here.

You will see that the RabbitConfiguration class extends a framework-provided class called AbstractRabbitConfiguration. That forces it to implement the abstract rabbitTemplate() method while the base class itself then creates an 'amqpAdmin' bean. The "rabbitTemplate" bean in turn depends upon the bean that is created by the connectionFactory() method. There, we are providing the 'username' and 'password' properties as well as the 'hostname' constructor argument to an instance of SingleConnectionFactory.

```
@Bean
public ConnectionFactory connectionFactory() {
    SingleConnectionFactory connectionFactory = new SingleConnectionFactory("localhost");
    connectionFactory.setUsername("guest");
    connectionFactory.setPassword("guest");
    return connectionFactory;
}
```

The base class also provides a mechanism that will recognize any Exchange, Queue, or Binding bean definitions and then declare them on the broker. In fact, the "helloWorldQueue" bean that is generated in HelloWorldConfiguration is an example simply because it is an instance of Queue.

```
@Bean
public Queue helloWorldQueue() {
    return new Queue(this.helloWorldQueueName);
}
```

Looking back at the "rabbitTemplate" bean configuration, you will see that it has the helloWorldQueue's name set as its "queue" property (for receiving Messages) and for its "routingKey" property (for sending Messages).

Now that we've explored the configuration, let's look at the code that actually uses these components. First, open the Producer class from within the same package. It contains a main() method where the Spring ApplicationContext is created.

```
public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(RabbitConfiguration.class);
    AmqpTemplate amqpTemplate = context.getBean(AmqpTemplate.class);
    amqpTemplate.convertAndSend("Hello World");
    System.out.println("Sent: Hello World");
}
```

As you can see in the example above, the AmqpTemplate bean is retrieved and used for sending a Message. Since the client code should rely on interfaces whenever possible, the type is AmqpTemplate rather than RabbitTemplate. Even though the bean created in HelloWorldConfiguration is an instance of RabbitTemplate, relying on the interface means that this code is more portable (the configuration can be changed independently of the code). Since the convertAndSend() method is invoked, the template will be delegating to its MessageConverter instance. In this case, it's using the default SimpleMessageConverter, but a different implementation could be provided to the "rabbitTemplate" bean as defined in HelloWorldConfiguration.

Now open the Consumer class. It actually shares the same configuration base class which means it will be sharing the "rabbitTemplate" bean. That's why we configured that template with both a "routingKey" (for sending) and "queue" (for receiving). As you saw in Section 1.3, “AmqpTemplate”, you could instead pass the 'routingKey' argument to the send method and the 'queue' argument to the receive method. The Consumer code is basically a mirror image of the Producer, calling receiveAndConvert() rather than convertAndSend().

```
public static void main(String[] args) {
    ApplicationContext context = new AnnotationConfigApplicationContext(RabbitConfiguration.class);
    AmqpTemplate amqpTemplate = context.getBean(AmqpTemplate.class);
    System.out.println("Received: " + amqpTemplate.receiveAndConvert());
}
```

If you run the Producer, and then run the Consumer, you should see the message "Received: Hello World" in the console output.

3.2.2. Asynchronous Example

Now that we've walked through the synchronous Hello World sample, it's time to move on to a slightly more advanced but significantly more powerful option. With a few modifications, the Hello World sample can provide an example of asynchronous reception, a.k.a. *Message-driven POJOs*. In fact, there is a sub-package that provides exactly that: org.springframework.amqp.samples.helloworld.async.

Once again, we will start with the sending side. Open the ProducerConfiguration class and notice that it creates a "connectionFactory" and "rabbitTemplate" bean. This time, since the configuration is dedicated to the message sending side, we don't even need any Queue definitions, and the RabbitTemplate only has the 'routingKey' property set. Recall that messages are sent to an Exchange rather than being sent directly to a Queue. The AMQP default Exchange is a direct Exchange with no name. All Queues are bound to that default Exchange with their name as the routing key. That is why we only need to provide the routing key here.

```
public RabbitTemplate rabbitTemplate() {
    RabbitTemplate template = new RabbitTemplate(connectionFactory());
    template.setRoutingKey(this.helloWorldQueueName);
    return template;
}
```

Since this sample will be demonstrating asynchronous message reception, the producing side is designed to continuously send messages (if it were a message-per-execution model like the synchronous version, it would not be quite so obvious that it is in fact a message-driven consumer). The component responsible for sending messages continuously is defined as an inner class within the ProducerConfiguration. It is configured to execute every 3 seconds.

```
static class ScheduledProducer {

    @Autowired
    private volatile RabbitTemplate rabbitTemplate;

    private final AtomicInteger counter = new AtomicInteger();

    @Scheduled(fixedRate = 3000)
    public void sendMessage() {
        rabbitTemplate.convertAndSend("Hello World " + counter.incrementAndGet());
    }
}
```

You don't need to understand all of the details since the real focus should be on the receiving side (which we will cover momentarily). However, if you are not yet familiar with Spring 3.0 task scheduling support, you can learn more here. The short story is that the "postProcessor" bean in the ProducerConfiguration is registering the task with a scheduler.

Now, let's turn to the receiving side. To emphasize the Message-driven POJO behavior will start with the component that is reacting to the messages. The class is called HelloWorldHandler.

```
public class HelloWorldHandler {

    public void handleMessage(String text) {
        System.out.println("Received: " + text);
    }

}
```

Clearly, that *is* a POJO. It does not extend any base class, it doesn't implement any interfaces, and it doesn't even contain any imports. It is being "adapted" to the MessageListener interface by the Spring AMQP MessageListenerAdapter. That adapter can then be configured on a SimpleMessageListenerContainer. For this sample, the container is created in the ConsumerConfiguration class. You can see the POJO wrapped in the adapter there.

```
@Bean
public SimpleMessageListenerContainer listenerContainer() {
    SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
    container.setConnectionFactory(connectionFactory());
    container.setQueueName(this.helloWorldQueueName);
    container.setMessageListener(new MessageListenerAdapter(new HelloWorldHandler()));
    return container;
}
```

The SimpleMessageListenerContainer is a Spring lifecycle component and will start automatically by default. If you look in the Consumer class, you will see that its main() method consists of nothing more than a one-line bootstrap to create the ApplicationContext. The Producer's main() method is also a one-line bootstrap, since the component whose method is annotated with @Scheduled will also start executing automatically. You can start the Producer and Consumer in any order, and you should see messages being sent and received every 3 seconds.

3.3. Stock Trading

The Stock Trading sample demonstrates more advanced messaging scenarios than the Hello World sample. However, the configuration is very similar - just a bit more involved. Since we've walked through the Hello

World configuration in detail, here we'll focus on what makes this sample different. There is a server that pushes market data (stock quotes) to a Topic Exchange. Then, clients can subscribe to the market data feed by binding a Queue with a routing pattern (e.g. "app.stock.quotes.nasdaq.*"). The other main feature of this demo is a request-reply "stock trade" interaction that is initiated by the client and handled by the server. That involves a private "replyTo" Queue that is sent by the client within the order request Message itself.

The Server's core configuration is in the RabbitServerConfiguration class within the org.springframework.amqp.rabbit.stocks.config.server package. It extends the AbstractStockAppRabbitConfiguration. That is where the resources common to the Server and Client(s) are defined, including the market data Topic Exchange (whose name is 'app.stock.marketdata') and the Queue that the Server exposes for stock trades (whose name is 'app.stock.request'). In that common configuration file, you will also see that a JsonMessageConverter is configured on the RabbitTemplate.

The Server-specific configuration consists of 2 things. First, it configures the market data exchange on the RabbitTemplate so that it does not need to provide that exchange name with every call to send a Message. It does this within an abstract callback method defined in the base configuration class.

```
public void configureRabbitTemplate(RabbitTemplate rabbitTemplate) {
    rabbitTemplate.setExchange(MARKET_DATA_EXCHANGE_NAME);
}
```

Secondly, the stock request queue is declared. It does not require any explicit bindings in this case, because it will be bound to the default no-name exchange with its own name as the routing key. As mentioned earlier, the AMQP specification defines that behavior.

```
@Bean
public Queue stockRequestQueue() {
    return new Queue(STOCK_REQUEST_QUEUE_NAME);
}
```

Now that you've seen the configuration of the Server's AMQP resources, navigate to the 'org.springframework.amqp.rabbit.stocks' package under the 'src/test/java' directory. There you will see the actual Server class that provides a main() method. It creates an ApplicationContext based on the 'server-bootstrap.xml' config file. In there you will see the scheduled task that publishes dummy market data. That configuration relies upon Spring 3.0's "task" namespace support. The bootstrap config file also imports a few other files. The most interesting one is 'server-messaging.xml' which is directly under 'src/main/resources'. In there you will see the "messageListenerContainer" bean that is responsible for handling the stock trade requests. Finally have a look at the "serverHandler" bean that is defined in "server-handlers.xml" (also in 'src/main/resources'). That bean is an instance of the ServerHandler class and is a good example of a Message-driven POJO that is also capable of sending reply Messages. Notice that it is not itself coupled to the framework or any of the AMQP concepts. It simply accepts a TradeRequest and returns a TradeResponse.

```
public TradeResponse handleMessage(TradeRequest tradeRequest) { ... }
```

Now that we've seen the most important configuration and code for the Server, let's turn to the Client. The best starting point is probably RabbitClientConfiguration within the 'org.springframework.amqp.rabbit.stocks.config.client' package. Notice that it declares two queues without providing explicit names.

```
@Bean
public Queue marketDataQueue() {
    return amqpAdmin().declareQueue();
}

@Bean
public Queue traderJoeQueue() {
```

```
    return amqpAdmin().declareQueue();
}
```

Those are private queues, and unique names will be generated automatically. The first generated queue is used by the Client to bind to the market data exchange that has been exposed by the Server. Recall that in AMQP, consumers interact with Queues while producers interact with Exchanges. The "binding" of Queues to Exchanges is what instructs the broker to deliver, or route, messages from a given Exchange to a Queue. Since the market data exchange is a Topic Exchange, the binding can be expressed with a routing pattern. The RabbitClientConfiguration declares that with a Binding object, and that object is generated with the BindingBuilder's fluent API.

```
@Value("${stocks.quote.pattern}")
private String marketDataRoutingKey;

@Bean
public Binding marketDataBinding() {
    return BindingBuilder.from(
            marketDataQueue()).to(marketDataExchange()).with(marketDataRoutingKey);
}
```

Notice that the actual value has been externalized in a properties file ("client.properties" under src/main/resources), and that we are using Spring's @Value annotation to inject that value. This is generally a good idea, since otherwise the value would have been hardcoded in a class and unmodifiable without recompilation. In this case, it makes it much easier to run multiple versions of the Client while making changes to the routing pattern used for binding. Let's try that now.

Start by running org.springframework.amqp.rabbit.stocks.Server and then org.springframework.amqp.rabbit.stocks.Client. You should see dummy quotes for NASDAQ stocks because the current value associated with the 'stocks.quote.pattern' key in client.properties is 'app.stock.quotes.nasdaq.*'. Now, while keeping the existing Server and Client running, change that property value to 'app.stock.quotes.nyse.*' and start a second Client instance. You should see that the first client is still receiving NASDAQ quotes while the second client receives NYSE quotes. You could instead change the pattern to get all stocks or even an individual ticker.

The final feature we'll explore is the request-reply interaction from the Client's perspective. Recall that we have already seen the ServerHandler that is accepting TradeRequest objects and returning TradeResponse objects. The corresponding code on the Client side is RabbitStockServiceGateway in the 'org.springframework.amqp.rabbit.stocks.gateway' package. It delegates to the RabbitTemplate in order to send Messages.

```
public void send(TradeRequest tradeRequest) {
    getRabbitTemplate().convertAndSend(tradeRequest, new MessagePostProcessor() {
            public Message postProcessMessage(Message message) throws AmqpException {
                message.getMessageProperties().setReplyTo(new Address(defaultReplyToQueue));
                try {
                    message.getMessageProperties().setCorrelationId(
                            UUID.randomUUID().toString().getBytes("UTF-8"));
                }
                catch (UnsupportedEncodingException e) {
                    throw new AmqpException(e);
                }
                return message;
            }
        });
    }
```

Notice that prior to sending the message, it sets the "replyTo" address. It's providing the queue that was generated by the "traderJoeQueue" bean definition shown above. Here's the @Bean definition for the StockServiceGateway class itself.

```
@Bean
public StockServiceGateway stockServiceGateway() {
    RabbitStockServiceGateway gateway = new RabbitStockServiceGateway();
    gateway.setRabbitTemplate(rabbitTemplate());
    gateway.setDefaultReplyToQueue(traderJoeQueue());
    return gateway;
}
```

Note

We are planning to provide a 'sendAndReceive' as well as a 'convertSendAndReceive' directly in the AmqpTemplate in the Milestone 2 release of Spring AMQP. At that point, this client-side gateway implementation will be simplified considerably.

If you are no longer running the Server and Client, start them now. Try sending a request with the format of '100 TCKR'. After a brief artificial delay that simulates "processing" of the request, you should see a confirmation message appear on the Client.

Part II. Spring Integration - Reference

This part of the reference documentation details the integration with the Spring Integration project.

Chapter 4. Spring Integration AMQP Support

4.1. Introduction

The Spring Integration project will include AMQP Channel Adapters and Gateways that build upon the Spring AMQP project as soon as the Spring AMQP project has a GA release. For now, those adapters are under development in the Spring Integration sandbox. In Spring Integration, "Channel Adapters" are unidirectional (one-way) whereas "Gateways" are bidirectional (request-reply). Ultimately, we will be providing an inbound-channel-adapter, outbound-channel-adapter, inbound-gateway, and outbound-gateway. As of the time of the Spring AMQP 1.0 Milestone 1 release, the 2 Channel Adapters are available. As mentioned, they are still in the "sandbox" and as such are subject to change and should not be depended upon in a production environment. That said, if you check out the project, you should be able to build it with Maven and experiment for yourself.

When the AMQP adapters are part of an official Spring Integration release, the documentation will be available as part of the Spring Integration distribution. In the meantime, we will just provide a quick overview of the current state of that development here.

4.2. Inbound Channel Adapter

To receive AMQP Messages from a Queue, configure an <inbound-channel-adapter>

```
<amqp:inbound-channel-adapter queue-name="some.queue"
                              channel="fromAMQP"
                              connection-factory="rabbitConnectionFactory"/>
```

4.3. Outbound Channel Adapter

To send AMQP Messages to an Exchange, configure an <outbound-channel-adapter>. A 'routing-key' may optionally be provided in addition to the exchange name.

```
<amqp:outbound-channel-adapter channel="toAMQP"
                               exchange-name="some.exchange"
                               routing-key="foo"
                               amqp-template="rabbitTemplate"/>
```

4.4. Inbound Gateway

Coming Soon

4.5. Outbound Gateway

Coming Soon

Part III. Other Resources

In addition to this reference documentation, there exist a number of other resources that may help you learn about AMQP.

Chapter 5. Further Reading

For those who are not familiar with AMQP, the specification is actually quite readable. It is of course the authoritative source of information, and the Spring AMQP code should be very easy to understand for anyone who is familiar with the spec. Our current implementation of the RabbitMQ support is based on their 1.8.x version, and it officially supports AMQP 0.8. However, we recommend reading the 0.9.1 document. The differences are minor (mostly clarifications in fact), and the document itself is more readable.

There are many great articles, presentations, and blogs available on the RabbitMQ Getting Started page. Since that is currently the only supported implementation for Spring AMQP, we also recommend that as a general starting point for all broker-related concerns.

Finally, be sure to visit the Spring AMQP Forum if you have questions or suggestions. With this first milestone release, we are looking forward to a lot of community feedback!

Bibliography

[jinterface-00] Ericsson AB. *jinterface User Guide*. Ericson AB . 2000.