Spring AMQP - Reference Documentation

1.3.3.RELEASE

Mark Pollack , Mark Fisher , Oleg Zhurakousky , Dave Syer , Gary Russell , Gunnar Hillert

Table of Contents

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. Introduction

This first part of the reference documentation is a high-level overview of Spring AMQP and the underlying concepts and some code snippets that will get you up and running as quickly as possible.

1. Quick Tour for the impatient

1.1 Introduction

This is the 5 minute tour to get started with Spring AMQP.

Prerequisites: install and run the RabbitMQ broker (http://www.rabbitmq.com/download.html). Then grab the spring-rabbit JAR and all its dependencies - the easiest way to do that is to declare a dependency in your build tool, e.g. for Maven:

```xml
<dependency>
  <groupId>org.springframework.amqp</groupId>
  <artifactId>spring-rabbit</artifactId>
  <version>1.3.3.RELEASE</version>
</dependency>
```

Very, Very Quick

Using plain, imperative Java to send and receive a message:

```java
ConnectionFactory connectionFactory = new CachingConnectionFactory();

AmqpAdmin admin = new RabbitAdmin(connectionFactory);
admin.declareQueue(new Queue("myqueue"));

AmqpTemplate template = new RabbitTemplate(connectionFactory);
template.convertAndSend("myqueue", "foo");

String foo = (String) template.receiveAndConvert("myqueue");
```

Note that there is a `ConnectionFactory` in the native Java Rabbit client as well. We are using the Spring abstraction in the code above. We are relying on the default exchange in the broker (since none is specified in the send), and the default binding of all queues to the default exchange by their name (hence we can use the queue name as a routing key in the send). Those behaviours are defined in the AMQP specification.

With XML Configuration

The same example as above, but externalizing the resource configuration to XML:

```java
ApplicationContext context =
    new GenericXmlApplicationContext("classpath:/rabbit-context.xml");
AmqpTemplate template = context.getBean(AmqpTemplate.class);

template.convertAndSend("myqueue", "foo");

String foo = (String) template.receiveAndConvert("myqueue");
```

```xml
<beans xmlns="http://www.springframework.org/schema/beans"
       xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
       xmlns:rabbit="http://www.springframework.org/schema/rabbit"
       xsi:schemaLocation="http://www.springframework.org/schema/rabbit
           http://www.springframework.org/schema/rabbit/spring-rabbit.xsd
           http://www.springframework.org/schema/beans
           http://www.springframework.org/schema/beans/spring-beans.xsd">

    <rabbit:connection-factory id="connectionFactory"/>

    <rabbit:template id="amqpTemplate" connection-factory="connectionFactory"/>

    <rabbit:admin connection-factory="connectionFactory"/>

    <rabbit:queue name="myqueue"/>

</beans>
```

The `<rabbit:admin/>` declaration by default automatically looks for beans of type `Queue`, `Exchange` and `Binding` and declares them to the broker on behalf of the user, hence there is no need to use that bean explicitly in the simple Java driver. There are plenty of options to configure the properties of the components in the XML schema - you can use auto-complete features of your XML editor to explore them and look at their documentation.

With Java Configuration

The same example again with the external configuration in Java:

```java
ApplicationContext context =
    new AnnotationConfigApplicationContext(RabbitConfiguration.class);
AmqpTemplate template = context.getBean(AmqpTemplate.class);

template.convertAndSend("myqueue", "foo");

String foo = (String) template.receiveAndConvert("myqueue");
```

```java
@Configuration
public class RabbitConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory connectionFactory =
            new CachingConnectionFactory("localhost");
        return connectionFactory;
    }

    @Bean
    public AmqpAdmin amqpAdmin() {
        return new RabbitAdmin(connectionFactory());
    }

    @Bean
    public RabbitTemplate rabbitTemplate() {
        return new RabbitTemplate(connectionFactory());
    }

    @Bean
    public Queue myQueue() {
       return new Queue("myqueue");
    }
}
```

2. What's New

2.1 Changes in 1.3 Since 1.2

Listener Concurrency

The listener container now supports dynamic scaling of the number of consumers based on workload, or the concurrency can be programmatically changed without stopping the container. See Section 3.12, "Listener Concurrency".

Listener Queues

The listener container now permits the queue(s) on which it is listening to be modified at runtime. Also, the container will now start if at least one of its configured queues is available for use. See Section 3.14, "Listener Container Queues"

This listener container will now redeclare any auto-delete queues during startup. See the section called "'auto-delete' Queues".

Consumer Priority

The listener container now supports consumer arguments, allowing the `x-priority` argument to be set. See the section called "Asynchronous Consumer" [23].

Exclusive Consumer

The `SimpleMessageListenerContainer` can now be configured with a single `exclusive` consumer, preventing other consumers from listening to the queue. See Section 3.13, "Exclusive Consumer".

Rabbit Admin

It is now possible to have the Broker generate the queue name, regardless of durable, autoDelete and exclusive settings. See Section 3.8, "Configuring the broker".

Direct Exchange Binding

Previously, omitting the `key` attribute from a `binding` element of a `direct-exchange` configuration caused the queue or exchange to be bound with an empty string as the routing key. Now it is bound with the the name of the provided `Queue` or `Exchange`. Users wishing to bind with an empty string routing key need to specify `key=""`.

AMQP Template

The `AmqpTemplate` now provides several synchronous `receiveAndReply` methods. These are implemented by the `RabbitTemplate`. For more information see Section 3.5, "Receiving messages".

The `RabbitTemplate` now supports configuring a `RetryTemplate` to attempt retries (with optional back off policy) for when the broker is not available. For more information see the section called "Adding Retry Capabilities".

Caching Connection Factory

The caching connection factory can now be configured to cache `Connection`s and their `Channel`s instead of using a single connection and caching just `Channel`s. See Section 3.2, "Connection and Resource Management".

Binding Arguments

The `<exchange>`'s `<binding>` now supports parsing of the `<binding-arguments>` sub-element. The `<headers-exchange>`'s `<binding>` now can be configured with a `key/value` attribute pair (to match on a single header) or with a `<binding-arguments>` sub-element, allowing matching on multiple headers; these options are mutually exclusive. See Section 3.8, "Configuring the broker" [33].

Routing Connection Factory

A new `SimpleRoutingConnectionFactory` has been introduced, to allow configuration of `ConnectionFactories` mapping to determine the target `ConnectionFactory` to use at runtime. See the section called "Routing Connection Factory".

MessageBuilder and MessagePropertiesBuilder

"Fluent APIs" for building messages and/or message properties is now provided. See the section called "Message Builder API".

RetryInterceptorBuilder

A "Fluent API" for building listener container retry interceptors is now provided. See the section called "Failures in Synchronous Operations and Options for Retry".

RepublishMessageRecoverer

This new `MessageRecoverer` is provided to allow publishing a failed message to another queue (including stack trace information in the header) when retries are exhausted. See the section called "Message Listeners and the Asynchronous Case".

Default Error Handler (Since 1.3.2)

A default `ConditionalRejectingErrorHandler` has been added to the listener container. This error handler detects message conversion problems (which are fatal) and instructs the container to reject the message to prevent the broker from continually redelivering the unconvertible message. See Section 3.9, "Exception Handling".

2.2 Changes to 1.2 Since 1.1

RabbitMQ Version

Spring AMQP now using RabbitMQ 3.1.x by default (but retains compatibility with earlier versions). Certain deprecations have been added for features no longer supported by RabbitMQ 3.1.x - federated exchanges and the `immediate` property on the `RabbitTemplate`.

Rabbit Admin

The `RabbitAdmin` now provides an option to allow exchange, queue, and binding declarations to continue when a declaration fails. Previously, all declarations stopped on a failure. By setting `ignore-`

declaration-exceptions, such exceptions are logged (WARN), but further declarations continue. An example where this might be useful is when a queue declaration fails because of a slightly different ttl setting would normally stop other declarations from proceeding.

The RabbitAdmin now provides an additional method getQueueProperties(). This can be used to determine if a queue exists on the broker (returns null for a non-existent queue). In addition, the current number of messages in the queue, as well as the current number of consumers is returned.

Rabbit Template

Previously, when using the ...sendAndReceive() methods were used with a fixed reply queue, two custom headers were used for correlation data and to retain/restore reply queue information. With this release, the standard message property correlationId is used by default, although the user can specifiy a custom property to use instead. In addition, nested replyTo information is now retained internally in the template, instead of using a custom header.

The immediate property is deprecated; users must not set this property when using RabbitMQ 3.0.x or greater.

JSON Message Converters

A Jackson 2.x MessageConverter is now provided, along with the existing converter that uses Jackson 1.x.

Automatic Declaration of Queues, etc

Previously, when declaring queues, exchanges and bindings, it was not possible to define which connection factory was used for the declarations, each RabbitAdmin would declare all components using its connection.

Starting with this release, it is now possible to limit declarations to specific RabbitAdmin instances. See the section called "Conditional Declaration".

AMQP Remoting

Facilities are now provided for using Spring Remoting techniques, using AMQP as the transport for the RPC calls. For more information see the section called "Spring Remoting with AMQP"

Requested Heart Beats

Several users have asked for the underlying client connection factory's requestedHeartBeats property to be exposed on the Spring AMQP CachingConnectionFactory. This is now available; previously, it was necessary to configure the AMQP client factory as a separate bean and provide a reference to it in the CachingConnectionFactory.

2.3 Changes to 1.1 Since 1.0

General

Spring-AMQP is now built using gradle.

Adds support for publisher confirms and returns.

Adds support for HA queues, and broker failover.

Adds support for Dead Letter Exchanges/Dead Letter Queues.

AMQP Log4j Appender

Adds an option to support adding a message id to logged messages.

Adds an option to allow the specification of a `Charset` name to be used when converting `String`s to `byte[]`.

Part II. 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. This part also includes a chapter on [integration with Erlang](#) and a chapter about the [sample applications](#).

3. Using Spring AMQP

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

3.1 AMQP Abstractions

Spring AMQP consists of a handful of modules, each represented by a JAR in the distribution. These modules are: spring-amqp, spring-rabbit and spring-erlang. 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 1.0 release there is only a RabbitMQ implementation however the abstractions have been validated in .NET using Apache Qpid in addition to RabbitMQ. Since AMQP operates at the protocol level in principle the RabbitMQ client can be used with any broker that supports the same protocol version, but we do not test any other brokers at present.

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 IV, "Other Resources"

Message

The 0-8 and 0-9-1 AMQP specifications do not define a 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.

```java
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.

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:

```java
public interface Exchange {

    String getName();

    String getExchangeType();

    boolean isDurable();

    boolean isAutoDelete();

    Map<String, Object> getArguments();

}
```

As you can see, an Exchange also has a 'type' represented by constants defined in `ExchangeTypes`. The basic types are: `Direct`, `Topic`, `Fanout`, and `Headers`. 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. For example, 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 these and the other Exchange types, check out Part IV, "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 3.3, "AmqpTemplate".

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.

```java
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;

    /**
     * The queue is durable, non-exclusive and non auto-delete.
     *
     * @param name the name of the queue.
     */
    public Queue(String name) {
        this(name, true, false, false);
    }

    // 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'.

> 🍃 **Note**
>
> See the section on queues in Section 3.8, "Configuring the broker" for information about declaring queues using namespace support, including queue arguments.

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.

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

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

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

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

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

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

```java
Binding b = BindingBuilder.bind(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 'bind()' 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 3.8, "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 3.3, "AmqpTemplate").

3.2 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 `org.springframework.amqp.rabbit.connection.Connection` which is a wrapper for `com.rabbitmq.client.Connection`. The only concrete implementation we provide is `CachingConnectionFactory` which, by default, establishes a single connection proxy 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. The `CachingConnectionFactory` implementation supports caching of those channels, and it maintains separate caches for channels based on whether they are transactional or not. When creating an instance of `CachingConnectionFactory`, the 'hostname' can be provided via the constructor. The 'username' and 'password' properties should be provided as well. If you would like to configure the size of the channel cache (the default is 1), you could call the `setChannelCacheSize()` method here as well.

Starting with *version 1.3*, the `CachingConnectionFactory` can be configured to cache connections as well as just channels. In this case, each call to `createConnection()` creates a new connection (or retrieves an idle one from the cache). Closing a connection returns it to the cache (if the cache size has not been reached). Channels created on such connections are cached too. The use of separate connections might be useful in some environments, such as consuming from an HA cluster, in conjunction with a load balancer, to connect to different cluster members.

Important

When the cache mode is `CONNECTION`, automatic declaration of queues etc. (See the section called "Automatic Declaration of Exchanges, Queues and Bindings") is NOT supported.

Also, at the time of writing, the `rabbitmq-client` library creates a fixed thread pool for each connection (5 threads) by default. When using a large number of connections, you should

consider setting a custom `executor` on the `CachingConnectionFactory`. Then, the same executor will be used by all connections and its threads can be shared. The executor's thread pool should be unbounded, or set appropriately for the expected utilization (usually, at least one thread per connection). If multiple channels are created on each connection then the pool size will affect the concurrency, so a variable (or simple cached) thread pool executor would be most suitable.

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

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

When using XML, the configuration might look like this:

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

Note

There is also a `SingleConnectionFactory` implementation which is only available in the unit test code of the framework. It is simpler than `CachingConnectionFactory` since it does not cache channels, but it is not intended for practical usage outside of simple tests due to its lack of performance and resilience. If you find a need to implement your own `ConnectionFactory` for some reason, the `AbstractConnectionFactory` base class may provide a nice starting point.

A `ConnectionFactory` can be created quickly and conveniently using the rabbit namespace:

```xml
<rabbit:connection-factory id="connectionFactory"/>
```

In most cases this will be preferable since the framework can choose the best defaults for you. The created instance will be a `CachingConnectionFactory`. Keep in mind that the default cache size for channels is 1. If you want more channels to be cached set a larger value via the 'channelCacheSize' property. In XML it would look like this:

```xml
<bean id="connectionFactory"
      class="org.springframework.amqp.rabbit.connection.CachingConnectionFactory">
    <constructor-arg value="somehost"/>
    <property name="username" value="guest"/>
    <property name="password" value="guest"/>
    <property name="channelCacheSize" value="25"/>
</bean>
```

And with the namespace you can just add the 'channel-cache-size' attribute:

```xml
<rabbit:connection-factory
    id="connectionFactory" channel-cache-size="25"/>
```

The default cache mode is CHANNEL, but you can configure it to cache connections instead; in this case, we use `connection-cache-size`:

```
<rabbit:connection-factory
    id="connectionFactory" cache-mode="CONNECTION" connection-cache-size="25"/>
```

Host and port attributes can be provided using the namespace

```
<rabbit:connection-factory
    id="connectionFactory" host="somehost" port="5672"/>
```

Alternatively, if running in a clustered environment, use the addresses attribute.

```
<rabbit:connection-factory
    id="connectionFactory" addresses="host1:5672,host2:5672"/>
```

Configuring the Underlying Client Connection Factory

The `CachingConnectionFactory` uses an instance of the Rabbit client `ConnectionFactory`; a number of configuration properties are passed through (`host`, `port`, `userName`, `password`, `requestedHeartBeat` for example) when setting the equivalent property on the `CachingConnectionFactory` . To set other properties (`clientProperties` for example), define an instance of the rabbit factory and provide a reference to it using the appropriate constructor of the `CachingConnectionFactory`. When using the namespace as described above, provide a reference to the configured factory in the `connection-factory` attribute.

```
<rabbit:connection-factory
      id="connectionFactory" connection-factory="rabbitConnectionFactory"/>
```

Routing Connection Factory

Starting with *version 1.3*, the `AbstractRoutingConnectionFactory` has been introduced. This provides a mechanism to configure mappings for several `ConnectionFactories` and determine a target `ConnectionFactory` by some `lookupKey` at runtime. Typically, the implementation checks a thread-bound context. For convenience, Spring AMQP provides the `SimpleRoutingConnectionFactory`, which gets the current thread-bound `lookupKey` from the `SimpleResourceHolder`:

```
<bean id="connectionFactory"
      class="org.springframework.amqp.rabbit.connection.SimpleRoutingConnectionFactory">
 <property name="targetConnectionFactories">
  <map>
   <entry key="#{connectionFactory1.virtualHost}" ref="connectionFactory1"/>
   <entry key="#{connectionFactory2.virtualHost}" ref="connectionFactory2"/>
  </map>
 </property>
</bean>

<rabbit:template id="template" connection-factory="connectionFactory" />
```

```
public class MyService {

    @Autowired
    private RabbitTemplate rabbitTemplate;

    public void service(String vHost, String payload) {
        SimpleResourceHolder.bind(rabbitTemplate.getConnectionFactory(), vHost);
        rabbitTemplate.convertAndSend(payload);
        SimpleResourceHolder.unbind(rabbitTemplate.getConnectionFactory());
    }

}
```

It is important to unbind the resource after use. For more information see the JavaDocs of `AbstractRoutingConnectionFactory`.

Publisher Confirms and Returns

Confirmed and returned messages are supported by setting the `CachingConnectionFactory`'s `publisherConfirms` and `publisherReturns` properties to 'true' respectively.
When these options are set, `Channel`s created by the factory are wrapped in an `PublisherCallbackChannel` which is used to facilitate the callbacks. When such a channel is obtained, the client can register a `PublisherCallbackChannel.Listener` with the `Channel`. The `PublisherCallbackChannel` implementation contains logic to route a confirm/return to the appropriate listener.

These features are explained further in the following sections.

⭐ **Tip**

For some more background information, please see the following blog post by the RabbitMQ team titled [Introducing Publisher Confirms](#).

3.3 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.

Adding Retry Capabilities

Starting with *version 1.3* you can now configure the `RabbitTemplate` to use a `RetryTemplate` to help with handling problems with broker connectivity. Refer to the [spring-retry](#) project for complete information; the following is just one example that uses an exponential back off policy and the default `SimpleRetryPolicy` which will make three attempts before throwing the exception to the caller.

Using the XML namespace:

```xml
<rabbit:template id="template" connection-factory="connectionFactory" retry-template="retryTemplate"/>

<bean id="retryTemplate" class="org.springframework.retry.support.RetryTemplate">
 <property name="backOffPolicy">
  <bean class="org.springframework.retry.backoff.ExponentialBackOffPolicy">
   <property name="initialInterval" value="500" />
   <property name="multiplier" value="10.0" />
   <property name="maxInterval" value="10000" />
  </bean>
 </property>
</bean>
```

Using `@Configuration`:

```java
@Bean
public AmqpTemplate rabbitTemplate();
  RabbitTemplate template = new RabbitTemplate(connectionFactory());
  RetryTemplate retryTemplate = new RetryTemplate();
  ExponentialBackOffPolicy backOffPolicy = new ExponentialBackOffPolicy();
  backOffPolicy.setInitialInterval(500);
  backOffPolicy.setMultiplier(10.0);
  backOffPolicy.setMaxInterval(10000);
  retryTemplate.setBackOffPolicy(backOffPolicy);
  template.setRetryTemplate(retryTemplate);
  return template;
}
```

Publisher Confirms and Returns

The `RabbitTemplate` implementation of `AmqpTemplate` supports Publisher Confirms and Returns.

For returned messages, the template's `mandatory` property must be set to 'true', and it requires a `CachingConnectionFactory` that has its `publisherReturns` property set to true (see the section called "Publisher Confirms and Returns"). Returns are sent to to the client by it registering a `RabbitTemplate.ReturnCallback` by calling `setReturnCallback(ReturnCallback callback)`. The callback must implement this method:

```java
void returnedMessage(Message message, int replyCode, String replyText,
          String exchange, String routingKey);
```

Only one `ReturnCallback` is supported by each `RabbitTemplate`.

For Publisher Confirms (aka Publisher Acknowledgements), the template requires a `CachingConnectionFactory` that has its `publisherConfirms` property set to true. Confirms are sent to to the client by it registering a `RabbitTemplate.ConfirmCallback` by calling `setConfirmCallback(ConfirmCallback callback)`. The callback must implement this method:

```
void confirm(CorrelationData correlationData, boolean ack);
```

The `CorrelationData` is an object supplied by the client when sending the original message. This is described further in the next section.

Only one `ConfirmCallback` is supported by a `RabbitTemplate`.

3.4 Sending messages

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

```
void send(Message message) throws AmqpException;

void send(String routingKey, Message message) throws AmqpException;

void send(String exchange, String routingKey, Message message) 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 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 Message("12.34".getBytes(), someProperties));
```

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

```
amqpTemplate.setExchange("marketData.topic");
amqpTemplate.setRoutingKey("quotes.nasdaq.FOO");
amqpTemplate.send(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 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 Message("Hello World".getBytes(), someProperties));
```

Message Builder API

Starting with *version 1.3*, a message builder API is provided by the `MessageBuilder` and `MessagePropertiesBuilder`; they provides a convenient "fluent" means of creating a message or message properties:

```
Message message = MessageBuilder.withBody("foo".getBytes())
 .setContentType(MessageProperties.CONTENT_TYPE_TEXT_PLAIN)
 .setMessageId("123")
 .setHeader("bar", "baz")
 .build();
```

or

```
MessageProperties props = MessagePropertiesBuilder.newInstance()
 .setContentType(MessageProperties.CONTENT_TYPE_TEXT_PLAIN)
 .setMessageId("123")
 .setHeader("bar", "baz")
 .build();
Message message = MessageBuilder.withBody("foo".getBytes())
 .andProperties(props)
 .build();
```

Each of the properties defined on the [MessageProperies](#) can be set. Other methods include `setHeader(String key, String value)`,`removeHeader(String key)`,`removeHeaders()`, and `copyProperties(MessageProperties properties)`. Each property setting method has a `set*IfAbsent()` variant. In the cases where a default initial value exists, the method is named `set*IfAbsentOrDefault()`.

Five static methods are provided to create an initial message builder:

```
public static MessageBuilder withBody(byte[] body) ❶

public static MessageBuilder withClonedBody(byte[] body) ❷

public static MessageBuilder withBody(byte[] body, int from, int to) ❸

public static MessageBuilder fromMessage(Message message) ❹

public static MessageBuilder fromClonedMessage(Message message) ❺
```

❶   The message created by the builder will have a body that is a direct reference to the argument.

❷   The message created by the builder will have a body that is a new array containing a copy of bytes in the argument.

❸   The message created by the builder will have a body that is a new array containing the range of bytes from the argument. See `Arrays.copyOfRange()` for more details.

❹   The message created by the builder will have a body that is a direct reference to the body of the argument. The argument's properties are copied to a new a `MessageProperties` object.

⑤   The message created by the builder will have a body that is a new array containing a copy of the argument's body. The argument's properties are copied to a new a `MessageProperties` object.

Three static methods are provided to create an initial message properties builder:

```
public static MessagePropertiesBuilder newInstance() ❶

public static MessagePropertiesBuilder fromProperties(MessageProperties properties) ❷

public static MessagePropertiesBuilder fromClonedProperties(MessageProperties
 properties) ❸
```

❶   A new message properties object is initialized with default values.
❷   The builder is initialized with, and `build()` will return, the provided properties object.
❸   The argument's properties are copied to a new a `MessageProperties` object.

Publisher Confirms

With the `RabbitTemplate` implementation of `AmqpTemplate`, each of the `send()` methods has an overloaded version that takes an additional `CorrelationData` object. When publisher confirms are enabled, this object is returned in the callback described in Section 3.3, "AmqpTemplate". This allows the sender to correlate a confirm (ack or nack) with the sent message.

Publisher Returns

When the template's `mandatory` property is 'true' returned messages are provided by the callback described in Section 3.3, "AmqpTemplate".

3.5 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.

Synchronous Consumer

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;
```

Just like in the case of sending messages, the `AmqpTemplate` has some convenience methods for receiving POJOs instead of `Message` instances, and implementations will provide a way to customize the `MessageConverter` used to create the `Object` returned:

```
Object receiveAndConvert() throws AmqpException;

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

Similar to `sendAndReceive` methods, beginning with *version 1.3*, the `AmqpTemplate` has several convenience `receiveAndReply` methods for synchronously receiving, processing and replying to messages:

```
<R, S> boolean receiveAndReply(ReceiveAndReplyCallback<R, S> callback)
    throws AmqpException;

<R, S> boolean receiveAndReply(String queueName, ReceiveAndReplyCallback<R, S> callback)
  throws AmqpException;

<R, S> boolean receiveAndReply(ReceiveAndReplyCallback<R, S> callback,
 String replyExchange, String replyRoutingKey) throws AmqpException;

<R, S> boolean receiveAndReply(String queueName, ReceiveAndReplyCallback<R, S> callback,
 String replyExchange, String replyRoutingKey) throws AmqpException;

<R, S> boolean receiveAndReply(ReceiveAndReplyCallback<R, S> callback,
  ReplyToAddressCallback<S> replyToAddressCallback) throws AmqpException;

<R, S> boolean receiveAndReply(String queueName, ReceiveAndReplyCallback<R, S> callback,
    ReplyToAddressCallback<S> replyToAddressCallback) throws AmqpException;
```

The `AmqpTemplate` implementation takes care of the 'receive' and 'reply' phases. In most cases you should provide only an implementation of `ReceiveAndReplyCallback` to perform some business logic for the received message and build a reply object or message, if needed. Note, a `ReceiveAndReplyCallback` may return `null`. In this case no reply is sent and `receiveAndReply` works like the `receive` method. This allows the same queue to be used for a mixture of messages, some of which may not need a reply.

Automatic message (request and reply) conversion is applied only if the provided callback is not an instance of `ReceiveAndReplyMessageCallback` - which provides a raw message exchange contract.

The `ReplyToAddressCallback` is useful for cases requiring custom logic to determine the `replyTo` address at runtime against the received message and reply from the `ReceiveAndReplyCallback`. By default, `replyTo` information in the request message is used to route the reply.

The following is an example of POJO-based receive and reply...

```
boolean received =
        this.template.receiveAndReply(ROUTE, new ReceiveAndReplyCallback<Order, Invoice>()
 {

                public Invoice handle(Order order) {
                        return processOrder(order);
                }
        });
if (received) {
        log.info("We received an order!");
}
```

Asynchronous Consumer

For asynchronous Message reception, a dedicated component (not 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 {
    void onMessage(Message 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.setQueueNames("some.queue");
container.setMessageListener(new MessageListenerAdapter(somePojo));
```

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:

```
<rabbit:listener-container connection-factory="rabbitConnectionFactory">
    <rabbit:listener queues="some.queue" ref="somePojo" method="handle"/>
</rabbit:listener-container>
```

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

```java
@Configuration
public class ExampleAmqpConfiguration {

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

    @Bean
    public ConnectionFactory rabbitConnectionFactory() {
        CachingConnectionFactory connectionFactory =
            new CachingConnectionFactory("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);
            }
        };
    }
}
```

Starting with *RabbitMQ Version 3.2*, the broker now supports consumer priority (see http://www.rabbitmq.com/blog/2013/12/16/using-consumer-priorities-with-rabbitmq/). This is enabled by setting the x-priority argument on the consumer. The SimpleMessageListenerContainer now supports setting consumer arguments:

```java
container.setConsumerArguments(Collections. <String, Object> singletonMap("x-priority",
 Integer.valueOf(10)));
```

For convenience, the namespace provides the priority attribute on the listener element:

```xml
<rabbit:listener-container connection-factory="rabbitConnectionFactory">
    <rabbit:listener queues="some.queue" ref="somePojo" method="handle" priority="10" />
</rabbit:listener-container>
```

Starting with *version 1.3* the queue(s) on which the container is listening can be modified at runtime; see Section 3.14, "Listener Container Queues".

'auto-delete' Queues

When a container is configured to listen to auto-delete queue(s), the queue is removed by the broker when the container is stopped (last consumer is cancelled). Before *version 1.3*, the container could not be restarted because the queue was missing; the RabbitAdmin only automatically redeclares queues etc, when the connection is closed/opens, which does not happen when the container is stopped/started.

Starting with *version 1.3*, the container will now use a RabbitAdmin to redeclare any missing queues during startup.

You can also use conditional declaration (the section called "Conditional Declaration") together with an `auto-startup="false"` admin to defer queue declaration until the container is started.
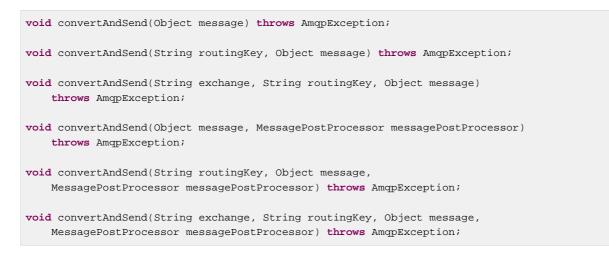
```xml
<rabbit:queue id="otherAnon" declared-by="containerAdmin" />

<rabbit:direct-exchange name="otherExchange" auto-delete="true" declared-
by="containerAdmin">
 <rabbit:bindings>
  <rabbit:binding queue="otherAnon" key="otherAnon" />
 </rabbit:bindings>
</rabbit:direct-exchange>

<rabbit:listener-container id="container2" auto-startup="false">
 <rabbit:listener id="listener2" ref="foo" queues="otherAnon" admin="containerAdmin" />
</rabbit:listener-container>

<rabbit:admin id="containerAdmin" connection-factory="rabbitConnectionFactory"
 auto-startup="false" />
```

In this case, the queue and exchange are declared by `containerAdmin` which has `auto-startup="false"` so the elements are not declared during context initialization. Also, the container is not started for the same reason. When the container is later started, it uses it's reference to `containerAdmin` to declare the elements.

3.6 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.

```java
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 `Message` instance. 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;
```

> ### Note
>
> The `MessageListenerAdapter` mentioned in the section called "Asynchronous Consumer" also uses a `MessageConverter`.

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.

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.

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.

JsonMessageConverter and Jackson2JsonMessageConverter

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). Two implementations are available and can be configured on any `RabbitTemplate` instance to override its usage of the `SimpleMessageConverter` default. The `JsonMessageConverter` which uses the `org.codehaus.jackson` 1.x library and `Jackson2JsonMessageConverter` which uses the `com.fasterxml.jackson` 2.x library.

```xml
<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>
```

```xml
<bean class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <property name="connectionFactory" ref="rabbitConnectionFactory"/>
    <property name="messageConverter">

  <bean class="org.springframework.amqp.support.converter.Jackson2JsonMessageConverter">
            <!-- if necessary, override the DefaultClassMapper -->
            <property name="classMapper" ref="customClassMapper"/>
        </bean>
    </property>
</bean>
```

As shown above, the `JsonMessageConverter` and `Jackson2JsonMessageConverter` uses a `DefaultClassMapper` by default. Type information is added to (and retrieved from) the `MessageProperties`. If an inbound message does not contain type information in the `MessageProperties`, but you know the expected type, you can configure a static type using the `defaultType` property

```xml
<bean id="jsonConverterWithDefaultType"
      class="o.s.amqp.support.converter.JsonMessageConverter">
    <property name="classMapper">
        <bean class="org.springframework.amqp.support.converter.DefaultClassMapper">
            <property name="defaultType" value="foo.PurchaseOrder"/>
        </bean>
    </property>
</bean>
```

```
<bean id="jsonConverterWithDefaultType"
      class="o.s.amqp.support.converter.Jackson2JsonMessageConverter">
    <property name="classMapper">
        <bean class="org.springframework.amqp.support.converter.DefaultClassMapper">
            <property name="defaultType" value="foo.PurchaseOrder"/>
        </bean>
    </property>
</bean>
```

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>
```

3.7 Request/Reply Messaging

The `AmqpTemplate` also provides a variety of `sendAndReceive` methods that accept the same argument options that you have seen above for the one-way send operations (exchange, routingKey, and Message). Those methods are quite useful for request/reply scenarios since they handle the configuration of the necessary "reply-to" property before sending and can listen for the reply message on an exclusive Queue that is created internally for that purpose.

Similar request/reply methods are also available where the `MessageConverter` is applied to both the request and reply. Those methods are named `convertSendAndReceive`. See the Javadoc of `AmqpTemplate` for more detail.

By default, a new temporary queue is used for each reply. However, a single reply queue can be configured on the template, which can be more efficient, and also allows you to set arguments on that queue. In this case, however, you must also provide a <reply-listener/> sub element. This element provides a listener container for the reply queue, with the template being the listener. All of the Section 3.11, "Message Listener Container Configuration" attributes allowed on a <listener-container/> are allowed on the element, except for connection-factory and message-converter, which are inherited from the template's configuration.

```
<rabbit:template id="amqpTemplate"
        connection-factory="connectionFactory" reply-queue="replies">
    <rabbit:reply-listener/>
</rabbit:template>
```

While the container and template share a connection factory, they do not share a channel and therefore requests and replies are not performed within the same transaction (if transactional).

Message Correlation With A Reply Queue

When using a fixed reply queue, it is necessary to provide correlation data so that replies can be correlated to requests. See RabbitMQ Remote Procedure Call (RPC). By default, the standard `correlationId` property will be used to hold the correlation data. However, if you wish to use a custom propertry to hold correlation data, you can set the `correlation-key` attribute on the <rabbit-template/>. Explicitly setting the attribute to `correlationId` is the same as omitting the attribute. Of course, the client and server must use the same header for correlation data.

> 🎾 **Note**
>
> Spring AMQP version 1.1 used a custom property `spring_reply_correlation` for this data. If you wish to revert to this behavior with the current version, perhaps to maintain compatibility with another application using 1.1, you must set the attribute to `spring_reply_correlation`.

Reply Listener Container

When using a fixed reply queue, a `SimpleListenerContainer` is used to receive the replies; with the `RabbitTemplate` being the `MessageListener`. When defining a template with the `<rabbit:template/>` namespace element, as shown above, the parser defines the container and wires in the template as the listener.

> 🎾 **Note**
>
> When the template does not use a fixed `replyQueue`, a listener container is not needed.

If you define your `RabbitTemplate` as a `<bean/>`, or using an `@Configuration` class to define it as an `@Bean`, or when creating the template programmatically, you will need to define and wire up the reply listener container yourself. If you fail to do this, the template will never receive the replies and will eventually time out and return null as the reply to a call to a `sendAndReceive` method.

> ➡ **Important**
>
> When wiring the reply listener and template yourself, it is important to ensure that the template's `replyQueue` and the container's `queues` (or `queueNames`) properties refer to the same queue. The template inserts the reply queue into the outbound message `replyTo` property.

The following are examples of how to manually wire up the beans.

```xml
<bean id="amqpTemplate" class="org.springframework.amqp.rabbit.core.RabbitTemplate">
    <constructor-arg ref="connectionFactory" />
    <property name="exchange" value="foo.exchange" />
    <property name="routingKey" value="foo" />
    <property name="replyQueue" ref="replyQ" />
    <property name="replyTimeout" value="600000" />
</bean>

<bean class="org.springframework.amqp.rabbit.listener.SimpleMessageListenerContainer">
    <constructor-arg ref="connectionFactory" />
    <property name="queues" ref="replyQ" />
    <property name="messageListener" ref="amqpTemplate" />
</bean>

<rabbit:queue id="replyQ" name="my.reply.queue" />
```

```
    @Bean
    public RabbitTemplate amqpTemplate() {
        RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory());
        rabbitTemplate.setMessageConverter(msgConv());
        rabbitTemplate.setReplyQueue(replyQueue());
        rabbitTemplate.setReplyTimeout(60000);
        return rabbitTemplate;
    }

    @Bean
    public SimpleMessageListenerContainer replyListenerContainer() {
        SimpleMessageListenerContainer container = new SimpleMessageListenerContainer();
        container.setConnectionFactory(connectionFactory());
        container.setQueues(replyQueue());
        container.setMessageListener(amqpTemplate());
        return container;
    }

    @Bean
    public Queue replyQueue() {
        return new Queue("my.reply.queue");
    }
```

A complete example of a `RabbitTemplate` wired with a fixed reply queue, together with a "remote" listener container that handles the request and returns the reply is shown in this test case.

Spring Remoting with AMQP

The Spring Framework has a general remoting capability, allowing Remote Procedure Calls (RPC) using various transports. Spring-AMQP supports a similar mechanism with a `AmqpProxyFactoryBean` on the client and a `AmqpInvokerServiceExporter` on the server. This provides RPC over AMQP. On the client side, a `RabbitTemplate` is used as described above; on the server side, the invoker (configured as a `MessageListener`) receives the message, invokes the configured service, and returns the reply using the inbound message's `replyTo` information.

The client factory bean can be injected into any bean (using its `serviceInterface`); the client can then invoke methods on the proxy, resulting in remote execution over AMQP.

Note

With the default `MessageConverter`s, the method paramters and returned value must be instances of `Serializable`.

On the server side, the `AmqpInvokerServiceExporter` has both `AmqpTemplate` and `MessageConverter` properties. Currently, the template's `MessageConverter` is not used. If you need to supply a custom message converter, then you should provide it using the `messageConverter` property. On the client side, a custom message converter can be added to the `AmqpTemplate` which is provided to the `AmqpProxyFactoryBean` using its `amqpTemplate` property.

Sample client and server configurations are shown below.

```
<bean id="client"
 class="org.springframework.amqp.remoting.client.AmqpProxyFactoryBean">
 <property name="amqpTemplate" ref="template" />
 <property name="serviceInterface" value="foo.ServiceInterface" />
</bean>

<rabbit:connection-factory id="connectionFactory" />

<rabbit:template id="template" connection-factory="connectionFactory" reply-timeout="2000"
 routing-key="remoting.binding" exchange="remoting.exchange" />

<rabbit:admin connection-factory="connectionFactory" />

<rabbit:queue name="remoting.queue" />

<rabbit:direct-exchange name="remoting.exchange">
 <rabbit:bindings>
  <rabbit:binding queue="remoting.queue" key="remoting.binding" />
 </rabbit:bindings>
</rabbit:direct-exchange>
```

```
<bean id="listener"
 class="org.springframework.amqp.remoting.service.AmqpInvokerServiceExporter">
 <property name="serviceInterface" value="foo.ServiceInterface" />
 <property name="service" ref="service" />
 <property name="amqpTemplate" ref="template" />
</bean>

<bean id="service" class="foo.ServiceImpl" />

<rabbit:connection-factory id="connectionFactory" />

<rabbit:template id="template" connection-factory="connectionFactory" />

<rabbit:queue name="remoting.queue" />

<rabbit:listener-container connection-factory="connectionFactory">
 <rabbit:listener ref="listener" queue-names="remoting.queue" />
</rabbit:listener-container>
```

Important

The `AmqpInvokerServiceExporter` can only process properly formed messages, such as those sent from the `AmqpProxyFactoryBean`. If it receives a message that it cannot interpret, a serialized `RuntimeException` will be sent as a reply. If the message has no `replyToAddress` property, the message will be rejected and permanently lost if no Dead Letter Exchange has been configured.

3.8 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.

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

```java
public interface AmqpAdmin {

    // Exchange Operations

    void declareExchange(Exchange exchange);

    void deleteExchange(String exchangeName);

    // Queue Operations

    Queue declareQueue();

    String 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);

    void removeBinding(Binding binding);

    Properties getQueueProperties(String queueName);

}
```

The no-arg declareQueue() method defines a queue on the broker whose name is automatically generated. The additional properties of this auto-generated queue are `exclusive=true`, `autoDelete=true`, and `durable=false`.

The `declareQueue(Queue queue)` method takes a `Queue` object and returns the name of the declared queue. This is useful if you wish the broker to generate the queue's name. This is in contrast to an `AnonymousQueue` where the framework generates a unique (`UUID`) name and sets `durable` to `false` and `exlusive, autoDelete` to `true`. If the provided `Queue`'s `name` property is an empty String, the Broker declares the queue with a generated name and that name is returned to the caller. The `Queue` object itself is not changed. This functionality can only be used programmatically by invoking the `RabbitAdmin` directly. It is not supported for auto-declaration by the admin by defining a queue declaratively in the application context. A `<rabbit:queue/>` with an empty, or missing, `name` will always create an `AnonymousQueue`. This is because the name will change if redeclared due to a connection failure. Declarative queues must have fixed names because they might be referenced elsewhere in the context, for example, in a listener:

```xml
<rabbit:listener-container>
    <rabbit:listener ref="listener" queue-names="#{someQueue.name}" />
</rabbit:listener-container>
```

See the section called "Automatic Declaration of Exchanges, Queues and Bindings".

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

```xml
<rabbit:connection-factory id="connectionFactory"/>

<rabbit:admin id="amqpAdmin" connection-factory="connectionFactory"/>
```

When the `CachingConnectionFactory` cache mode is `CHANNEL` (the default), the `RabbitAdmin` implementation does automatic lazy declaration of `Queues`, `Exchanges` and `Bindings` declared in the same `ApplicationContext`. These components will be declared as son as a `Connection` is opened to the broker. There are some namespace features that make this very convenient, e.g. in the Stocks sample application we have:

```xml
<rabbit:queue id="tradeQueue"/>

<rabbit:queue id="marketDataQueue"/>

<fanout-exchange name="broadcast.responses"
                 xmlns="http://www.springframework.org/schema/rabbit">
    <bindings>
        <binding queue="tradeQueue"/>
    </bindings>
</fanout-exchange>

<topic-exchange name="app.stock.marketdata"
                 xmlns="http://www.springframework.org/schema/rabbit">
    <bindings>
        <binding queue="marketDataQueue" pattern="${stocks.quote.pattern}"/>
    </bindings>
</topic-exchange>
```

In the example above we are using anonymous Queues (actually internally just Queues with names generated by the framework, not by the broker) and refer to them by ID. We can also declare Queues with explicit names, which also serve as identifiers for their bean definitions in the context. E.g.

```xml
<rabbit:queue name="stocks.trade.queue"/>
```

⭐ Tip

You can provide both an *id* and a *name* attribute. This allows you to refer to the queue (for example in a binding) by an id that is independent of the queue name. It also allows standard Spring features such as property placeholders, and SpEL expressions for the queue name; these features are not available when using the name as the bean identifier.

Queues can be configured with additional arguments, for example, 'x-message-ttl' or 'x-ha-policy'. Using the namespace support, they are provided in the form of a Map of argument name/argument value pairs, using the <rabbit:queue-arguments> element.

```xml
<rabbit:queue name="withArguments">
    <rabbit:queue-arguments>
        <entry key="x-ha-policy" value="all"/>
    </rabbit:queue-arguments>
</rabbit:queue>
```

By default, the arguments are assumed to be strings. For arguments of other types, the type needs to be provided.

```xml
<rabbit:queue name="withArguments">
    <rabbit:queue-arguments value-type="java.lang.Long">
        <entry key="x-message-ttl" value="100"/>
    </rabbit:queue-arguments>
</rabbit:queue>
```

When providing arguments of mixed types, the type is provided for each entry element:

```
<rabbit:queue name="withArguments">
    <rabbit:queue-arguments>
        <entry key="x-message-ttl">
            <value type="java.lang.Long">100</value>
        </entry>
        <entry key="x-ha-policy" value="all"/>
    </rabbit:queue-arguments>
</rabbit:queue>
```

With Spring Framework 3.2 and later, this can be declared a little more succinctly:

```
<rabbit:queue name="withArguments">
    <rabbit:queue-arguments>
        <entry key="x-message-ttl" value="100" value-type="java.lang.Long"/>
        <entry key="x-ha-policy" value="all"/>
    </rabbit:queue-arguments>
</rabbit:queue>
```

⇨ Important

The RabbitMQ broker will not allow declaration of a queue with mismatched arguments. For example, if a `queue` already exists with no `time to live` argument, and you attempt to declare it with, say, `key="x-message-ttl" value="100"`, an exception will be thrown.

By default, the `RabbitAdmin` will immediately stop processing all declarations when any exception occurs; this could cause downstream issues - such as a *listener container* failing to initialize because another queue (defined after the one in error) is not declared.

This behavior can be modified by setting the `ignore-declaration-failures` attribute to `true` on the `RabbitAdmin`. This option instructs the `RabbitAdmin` to log the exception, and continue declaring other elements.

Starting with *version 1.3* the HeadersExchange can be configured to match on multiple headers; you can also specify whether any or all headers must match:

```
<rabbit:headers-exchange name="headers-test">
 <rabbit:bindings>
  <rabbit:binding queue="bucket">
   <rabbit:binding-arguments>
    <entry key="foo" value="bar"/>
    <entry key="baz" value="qux"/>
    <entry key="x-match" value="all"/>
   </rabbit:binding-arguments>
  </rabbit:binding>
 </rabbit:bindings>
</rabbit:headers-exchange>
```

To see how to use Java to configure the AMQP infrastructure, look at the Stock sample application, where there is the `@Configuration` class `AbstractStockRabbitConfiguration` which in turn has RabbitClientConfiguration and RabbitServerConfiguration subclasses. The code for AbstractStockRabbitConfiguration is shown below

```
@Configuration
public abstract class AbstractStockAppRabbitConfiguration {

    @Bean
    public ConnectionFactory connectionFactory() {
        CachingConnectionFactory connectionFactory =
            new CachingConnectionFactory("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 shown 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
     * that match its routing key.
     */
    @Bean
    public Binding marketDataBinding() {
        return BindingBuilder.bind(
                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.

Conditional Declaration

By default, all queues, exchanges, and bindings are declared by all   RabbitAdmin instances (that have auto-startup="true") in the application context.

Note

Starting with the 1.2 release, it is possible to conditionally declare these elements. This is particularly useful when an application connects to multiple brokers and needs to specify with which broker(s) a particular element should be declared.

The classes representing these elements implement Declarable which has two methods: shouldDeclare() and getDeclaringAdmins(). The RabbitAdmin uses these methods to determine whether a particular instance should actually process the declarations on its Connection.

The properties are available as attributes in the namespace, as shown in the following examples.

```
<rabbit:admin id="admin1" connection-factory="CF1" />

<rabbit:admin id="admin2" connection-factory="CF2" />

<rabbit:queue id="declaredByBothAdminsImplicitly" />

<rabbit:queue id="declaredByBothAdmins" declared-by="admin1, admin2" />

<rabbit:queue id="declaredByAdmin1Only" declared-by="admin1" />

<rabbit:queue id="notDeclaredByAny" auto-declare="false" />

<rabbit:direct-exchange name="direct" declared-by="admin1, admin2">
 <rabbit:bindings>
  <rabbit:binding key="foo" queue="bar"/>
 </rabbit:bindings>
</rabbit:direct-exchange>
```

> **Note**
>
> The `auto-declare` attribute is `true` by default and if the `declared-by` is not supplied (or is empty) then all `RabbitAdmin`s will declare the object (as long as the admin's `auto-startup` attribute is true; the default).

Similarly, you can use Java-based `@Configuration` to achieve the same effect. In this example, the components will be declared by `admin1` but not `admin2`:

```
@Bean
public RabbitAdmin admin() {
 RabbitAdmin rabbitAdmin = new RabbitAdmin(cf1());
 rabbitAdmin.afterPropertiesSet();
 return rabbitAdmin;
}

@Bean
public RabbitAdmin admin2() {
 RabbitAdmin rabbitAdmin = new RabbitAdmin(cf2());
 rabbitAdmin.afterPropertiesSet();
 return rabbitAdmin;
}

@Bean
public Queue queue() {
 Queue queue = new Queue("foo");
 queue.setAdminsThatShouldDeclare(admin());
 return queue;
}

@Bean
public Exchange exchange() {
 DirectExchange exchange = new DirectExchange("bar");
 exchange.setAdminsThatShouldDeclare(admin());
 return exchange;
}

@Bean
public Binding binding() {
 Binding binding = new Binding("foo", DestinationType.QUEUE, exchange().getName(), "foo",
 null);
 binding.setAdminsThatShouldDeclare(admin());
 return binding;
}
```

3.9 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.

When a listener throws an exception, it is wrapped in a `ListenerExecutionFailedException` and, normally the message is rejected and requeued by the broker. Setting `defaultRequeueRejected` to false will cause messages to be discarded (or routed to a dead letter exchange). As discussed in the section called "Message Listeners and the Asynchronous Case", the listener can throw an `AmqpRejectAndDontRequeueException` to conditionally control this behavior.

However, there is a class of errors where the listener cannot control the behavior. When a message that cannot be converted is encountered (for example an invalid `content_encoding` header), the `MessageConversionException` is thrown before the message reaches user code. With `defaultRequeueRejected` set to `true` (default), such messages would be redelivered over and over. Before *version 1.3.2*, users needed to write a custom ErrorHandler, as discussed in Section 3.9, "Exception Handling" to avoid this situation.

Starting with *version 1.3.2*, the default `ErrorHandler` is now a `ConditionalRejectingErrorHandler` which will reject (and not requeue) messages that fail with a `MessageConversionException`. An instance of this error handler can be configured with a `FatalExceptionStrategy` so users can provide their own rules for conditional message rejection, e.g. a delegate implementation to the `BinaryExceptionClassifier` from Spring Retry (the section called "Message Listeners and the Asynchronous Case"). In addition, the `ListenerExecutionFailedException` now has a `failedMessage` property which can be used in the decision. If the `FatalExceptionStrategy.isFatal()` method returns `true`, the error handler throws an `AmqpRejectAndDontRequeueException`. The default FatalExceptionStrategy logs a warning message.

3.10 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 signaling 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 layered 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 transaction early for instance).

For asynchronous use cases with `SimpleMessageListenerContainer` if an external transaction is needed it 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 SimpleMessageListenerContainer 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 the 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, what 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). The action taken on message rejection is independent of transactions and depends on the `defaultRequeueRejected` property (default `true`). For more information about rejecting failed messages, see the section called "Message Listeners and the Asynchronous Case".

For more information about RabbitMQ transactions, and their limitations, refer to RabbitMQ Broker Semantics.

> 🍃 **Note**
>
> Prior to *RabbitMQ 2.7.0*, such messages (and any that are unacked when a channel is closed or aborts) went to the back of the queue on a Rabbit broker, since 2.7.0, rejected messages go to the front of the queue, in a similar manner to JMS rolled back messages.

Using the RabbitTransactionManager

The RabbitTransactionManager is an alternative to executing Rabbit operations within, and synchronized with, external transactions. This Transaction Manager is an implementation of the PlatformTransactionManager interface and should be used with a single Rabbit ConnectionFactory.

> **➡ Important**
>
> This strategy is not able to provide XA transactions, for example in order to share transactions between messaging and database access.

Application code is required to retrieve the transactional Rabbit resources via `ConnectionFactoryUtils.getTransactionalResourceHolder(ConnectionFactory, boolean)` instead of a standard `Connection.createChannel()` call with subsequent Channel creation. When using Spring's <u>RabbitTemplate</u>, it will autodetect a thread-bound Channel and automatically participate in it.

With Java Configuration you can setup a new RabbitTransactionManager using:

```java
@Bean
public RabbitTransactionManager rabbitTransactionManager() {
    return new RabbitTransactionManager(connectionFactory);
}
```

If you prefer using XML configuration, declare the following bean in your XML Application Context file:

```xml
<bean id="rabbitTxManager"
      class="org.springframework.amqp.rabbit.transaction.RabbitTransactionManager">
    <property name="connectionFactory" ref="connectionFactory"/>
</bean>
```

3.11 Message Listener Container Configuration

There are quite a few options for configuring a `SimpleMessageListenerContainer` related to transactions and quality of service, and some of them interact with each other.

The table below shows the container property names and their equivalent attribute names (in parentheses) when using the namespace to configure a `<rabbit:message-listener-container/>`
. Some properties are not exposed by the namespace; indicated by `N/A` for the attribute.

Table 3.1. Configuration options for a message listener container

Property (Attribute)	Description
channelTransacted (channel-transacted)	Boolean flag to signal that all messages should be acknowledged in a transaction (either manually or automatically)
acknowledgeMode (acknowledge)	`NONE` = no acks will be sent (incompatible with `channelTransacted=true`). RabbitMQ calls this "autoack" because the broker assumes all messages are acked without any action from the consumer. MANUAL = the listener must acknowledge all messages by calling `Channel.basicAck()`. AUTO = the container will acknowledge the message automatically, unless the `MessageListener` throws an exception. Note that `acknowledgeMode` is complementary to channelTransacted - if the channel is transacted then the

Property (Attribute)	Description
	broker requires a commit notification in addition to the ack. This is the default mode. See also `txSize`.
transactionManager (transaction-manager)	External transaction manager for the operation of the listener. Also complementary to channelTransacted - if the `Channel` is transacted then its transaction will be synchronized with the external transaction.
prefetchCount (prefetch)	The number of messages to accept from the broker in one socket frame. The higher this is the faster the messages can be delivered, but the higher the risk of non-sequential processing. Ignored if the `acknowledgeMode` is NONE. This will be increased, if necessary, to match the `txSize`.
shutdownTimeout (N/A)	When a container shuts down (e.g. if its enclosing `ApplicationContext` is closed) it waits for in-flight messages to be processed up to this limit. Defaults to 5 seconds. After the limit is reached, if the channel is not transacted messages will be discarded.
txSize (transaction-size)	When used with `acknowledgeMode` AUTO, the container will attempt to process up to this number of messages before sending an ack (waiting for each one up to the receive timeout setting). This is also when a transactional channel is committed. If the `prefetchCount` is less than the `txSize`, it will be increased to match the `txSize`.
receiveTimeout (receive-timeout)	The maximum time to wait for each message. If acknowledgeMode=NONE this has very little effect - the container just spins round and asks for another message. It has the biggest effect for a transactional `Channel` with `txSize > 1`, since it can cause messages already consumed not to be acknowledged until the timeout expires.
autoStartup (auto-startup)	Flag to indicate that the container should start when the `ApplicationContext` does (as part of the `SmartLifecycle` callbacks which happen after all beans are initialized). Defaults to true, but set it to false if your broker might not be available on startup, and then call `start()` later manually when you know the broker is ready.
phase (phase)	When autoStartup is true, the lifecycle phase within which this container should start and stop. The lower the value the earlier this container will start and the later it will stop. The default is Integer.MAX_VALUE meaning the container will start as late as possible and stop as soon as possible.

Property (Attribute)	Description
adviceChain (advice-chain)	An array of AOP Advice to apply to the listener execution. This can be used to apply additional cross cutting concerns such as automatic retry in the event of broker death. Note that simple re-connection after an AMQP error is handled by the `CachingConnectionFactory`, as long as the broker is still alive.
taskExecutor (task-executor)	A reference to a Spring TaskExecutor (or standard JDK 1.5+ Executor) for executing listener invokers. Default is a SimpleAsyncTaskExecutor, using internally managed threads.
errorHandler (error-handler)	A reference to an ErrorHandler strategy for handling any uncaught Exceptions that may occur during the execution of the MessageListener. Default: `ConditionalRejectingErrorHandler`
concurrentConsumers (concurrency)	The number of concurrent consumers to initially start for each listener. See Section 3.12, "Listener Concurrency".
maxConcurrentConsumers (max-concurrency)	The maximum number of concurrent consumers to start, if needed, on demand. Must be greater than or equal to 'concurrentConsumers'. See Section 3.12, "Listener Concurrency".
startConsumerMinInterval (min-start-interval)	The time in milliseconds which must elapse before each new consumer is started on demand. See Section 3.12, "Listener Concurrency". Default 10000 (10 seconds).
stopConsumerMinInterval (min-stop-interval)	The time in milliseconds which must elapse before a consumer is stopped, since the last consumer was stopped, when an idle consumer is detected. See Section 3.12, "Listener Concurrency". Default 60000 (1 minute).
consecutiveActiveTrigger (min-consecutive-active)	The minimum number of consecutive messages received by a consumer, without a receive timeout occurring, when considering starting a new consumer. Also impacted by 'txSize'. See Section 3.12, "Listener Concurrency". Default 10.
consecutiveIdleTrigger (min-consecutive-idle)	The minimum number of receive timeouts a consumer must experience before considering stopping a consumer. Also impacted by 'txSize'. See Section 3.12, "Listener Concurrency". Default 10. .
connectionFactory (connection-factory)	A reference to the connectionFactory; when configuring using the XML namespace, the default referenced bean name is "rabbitConnectionFactory".

Property (Attribute)	Description
defaultRequeueRejected (requeue-rejected)	Determines whether messages that are rejected because the listener threw an exception should be requeued or not. Default 'true'.
recoveryInterval (recovery-interval)	Determines the time in milliseconds between attempts to start a consumer if it fails to start for non-fatal reasons. Default '5000'.
exclusive (exclusive)	Determines whether the single consumer in this container has exclusive access to the queue(s). The concurrency of the container must be 1 when this is true. If another consumer has exclusive access, the container will attempt to recover the consumer, according to the `recovery-interval`. When using the namespace, this attribute appears on the <rabbit:listener/> element along with the queue names. Default 'false'.
rabbitAdmin (admin)	When a listener container listens to at least one auto-delete queue and it is found to be missing during startup, the container uses a `RabbitAdmin` to declare the queue and any related bindings and exchanges. If such elements are configured to use conditional declaration (see the section called "Conditional Declaration"), the container must use the admin that was configured to declare those elements. Specify that admin here; only required when using auto-delete queues with conditional declaration. If you do not wish the auto-delete queue(s) to be declared until the container is started, set `auto-startup` to `false` on the admin. Defaults to a `RabbitAdmin` that will declare all non-conditional elements.

3.12 Listener Concurrency

By default, the listener container will start a single consumer which will receive messages from the queue(s).

When examining the table in the previous section, you will see a number of properties/attributes that control concurrency. The simplest is `concurrentConsumers`, which simply creates that (fixed) number of consumers which will concurrently process messages.

Prior to *version 1.3.0*, this was the only setting available and the container had to be stopped and started again to change the setting.

Since *version 1.3.0*, you can now dynamically adjust the `concurrentConsumers` property. If it is changed while the container is running, consumers will be added or removed as necessary to adjust to the new setting.

In addition, a new property `maxConcurrentConsumers` has been added and the container will dynamically adjust the concurrency based on workload. This works in conjunction

with four additional properties: `consecutiveActiveTrigger`, `startConsumerMinInterval`, `consecutiveIdleTrigger`, `stopConsumerMinInterval`. With the default settings, the algorithm to increase consumers works as follows:

If the `maxConcurrentConsumers` has not been reached and an existing consumer is active for 10 consecutive cycles AND at least 10 seconds has elapsed since the last consumer was started, a new consumer is started. A consumer is considered active if it received at least one message in `txSize` * `receiveTimeout` milliseconds.

With the default settings, the algorithm to decrease consumers works as follows:

If there are more than `concurrentConsumers` running and a consumer detects 10 consecutive timeouts (idle) AND the last consumer was stopped at least 60 seconds ago, a consumer will be stopped. The timeout depends on the `receiveTimeout` and the `txSize` properties. A consumer is considered idle if it receives no messages in `txSize` * `receiveTimeout` milliseconds. So, with the default timeout (1 second) and a `txSize` of 4, stopping a consumer will be considered after 40 seconds of idle time (4 timeouts correspond to 1 idle detection).

Note

Practically, consumers will only be stopped if the whole container is idle for some time. This is because the broker will share its work across all the active consumers.

3.13 Exclusive Consumer

Also starting with *version 1.3*, the listener container can be configured with a single exclusive consumer; this prevents other containers from consuming from the queue(s) until the current consumer is cancelled. The concurrency of such a container must be 1.

When using exclusive consumers, other containers will attempt to consume from the queue(s) according to the `recoveryInterval` property, and log a WARNing if the attempt fails.

3.14 Listener Container Queues

Version 1.3 introduced a number of improvements for handling multiple queues in a listener container.

The container must be configured to listen on at least one queue; this was the case previously too, but now queues can be added and removed at runtime. The container will recycle (cancel and re-create) the consumers when any pre-fetched messages have been processed. See methods `addQueues`, `addQueueNames`, `removeQueues` and `removeQueueNames`. When removing queues, at least one queue must remain.

A consumer will now start if any of its queues are available - previously the container would stop if any queues were unavailable. Now, this is only the case if none of the queues are available. If not all queues are available, the container will attempt to passively declare (and consume from) the missing queue(s) every 60 seconds.

Also, if a consumer receives a cancel from the broker (for example if a queue is deleted) the consumer will attempt to recover and the recovered consumer will continue to process messages from any other configured queues. Previously a cancel on one queue cancelled the entire consumer and eventually the container would stop due to the missing queue.

If you wish to permanently remove a queue, you should update the container before or after deleting to queue, to avoid future attempts to consume from it.

3.15 Resilience: Recovering from Errors and Broker Failures

Some of the key (and most popular) high-level features that Spring AMQP provides are to do with recovery and automatic re-connection in the event of a protocol error or broker failure. We have seen all the relevant components already in this guide, but it should help to bring them all together here and call out the features and recovery scenarios individually.

The primary reconnection features are enabled by the `CachingConnectionFactory` itself. It is also often beneficial to use the `RabbitAdmin` auto-declaration features. In addition, if you care about guaranteed delivery, you probably also need to use the `channelTransacted` flag in `RabbitTemplate` and `SimpleMessageListenerContainer` and also the `AcknowledgeMode.AUTO` (or manual if you do the acks yourself) in the `SimpleMessageListenerContainer`.

Automatic Declaration of Exchanges, Queues and Bindings

The `RabbitAdmin` component can declare exchanges, queues and bindings on startup. It does this lazily, through a `ConnectionListener`, so if the broker is not present on startup it doesn't matter. The first time a `Connection` is used (e.g. by sending a message) the listener will fire and the admin features will be applied. A further benefit of doing the auto declarations in a listener is that if the connection is dropped for any reason (e.g. broker death, network glitch, etc.) they will be applied again the next time they are needed.

Note

Queues declared this way must have fixed names; either explicitly declared, or generated by the framework for `AnonymousQueue`s. Anonymous queues are non-durable, exclusive, and auto-delete.

Important

Automatic declaration is only performed when the `CachingConnectionFactory` cache mode is `CHANNEL` (the default). This limitation exists because exlusive and auto-delete queues are bound to the connection.

Failures in Synchronous Operations and Options for Retry

If you lose your connection to the broker in a synchronous sequence using `RabbitTemplate` (for instance), then Spring AMQP will throw an `AmqpException` (usually but not always `AmqpIOException`). We don't try to hide the fact that there was a problem, so you have to be able to catch and respond to the exception. The easiest thing to do if you suspect that the connection was lost, and it wasn't your fault, is to simply try the operation again. You can do this manually, or you could look at using Spring Retry to handle the retry (imperatively or declaratively).

Spring Retry provides a couple of AOP interceptors and a great deal of flexibility to specify the parameters of the retry (number of attempts, exception types, backoff algorithm etc.). Spring AMQP also provides some convenience factory beans for creating Spring Retry interceptors in a convenient form for AMQP use cases, with strongly typed callback interfaces for you to implement custom recovery logic. See the Javadocs and properties of `StatefulRetryOperationsInterceptor` and `StatelessRetryOperationsInterceptor` for more detail. Stateless retry is appropriate if there is no transaction or if a transaction is started inside the retry callback. Note that stateless retry is simpler to configure and analyse than stateful retry, but it is not usually appropriate if there is an ongoing transaction which must be rolled back or definitely is going to roll back. A dropped connection in the middle of a

transaction should have the same effect as a rollback, so for reconnection where the transaction is started higher up the stack, stateful retry is usually the best choice.

Starting with *version 1.3*, a builder API is provided to aid in assembling these interceptors using Java (or in `@Configuration` classes), for example:

```java
@Bean
public StatefulRetryOperationsInterceptor interceptor() {
 return RetryInterceptorBuilder.stateful()
   .maxAttempts(5)
   .backOffOptions(1000, 2.0, 10000) // initialInterval, multiplier, maxInterval
   .build();
}
```

Only a subset of retry capabilities can be configured this way; more advanced features would need the configuration of a `RetryTemplate` as a Spring bean. See the [Spring Retry Javadocs](#) for complete information about available policies and their configuration.

Message Listeners and the Asynchronous Case

If a `MessageListener` fails because of a business exception, the exception is handled by the message listener container and then it goes back to listening for another message. If the failure is caused by a dropped connection (not a business exception), then the consumer that is collecting messages for the listener has to be cancelled and restarted. The `SimpleMessageListenerContainer` handles this seamlessly, and it leaves a log to say that the listener is being restarted. In fact it loops endlessly trying to restart the consumer, and only if the consumer is very badly behaved indeed will it give up. One side effect is that if the broker is down when the container starts, it will just keep trying until a connection can be established.

Business exception handling, as opposed to protocol errors and dropped connections, might need more thought and some custom configuration, especially if transactions and/or container acks are in use. Prior to 2.8.x, RabbitMQ had no definition of dead letter behaviour, so by default a message that is rejected or rolled back because of a business exception can be redelivered ad infinitum. To put a limit in the client on the number of re-deliveries, one choice is a `StatefulRetryOperationsInterceptor` in the advice chain of the listener. The interceptor can have a recovery callback that implements a custom dead letter action: whatever is appropriate for your particular environment.

Another alternative is to set the container's rejectRequeued property to false. This causes all failed messages to be discarded. When using RabbitMQ 2.8.x or higher, this also facilitates delivering the message to a Dead Letter Exchange.

Or, you can throw a `AmqpRejectAndDontRequeueException`; this prevents message requeuing, regardless of the setting of the `defaultRequeueRejected` property.

Often, a combination of both techniques will be used. Use a `StatefulRetryOperationsInterceptor` in the advice chain, where it's `MessageRecover` throws an `AmqpRejectAndDontRequeueException`. The `MessageRecover` is called when all retries have been exhausted. The default `MessageRecoverer` simply consumes the errant message and emits a WARN message. In which case, the message is ACK'd and won't be sent to the Dead Letter Exchange, if any.

Starting with *version 1.3*, a new `RepublishMessageRecoverer` is provided, to allow publishing of failed messages after retries are exhausted:

```
@Bean
RetryOperationsInterceptor interceptor() {
 return RetryInterceptorBuilder.stateless()
   .withMaxAttempts(5)
   .setRecoverer(new RepublishMessageRecoverer(amqpTemplate(), "bar", "baz"))
   .build();
}
```

Exception Classification for Retry

Spring Retry has a great deal of flexibility for determining which exceptions can invoke retry. The default configuration will retry for all exceptions. Given that user exceptions will be wrapped in a `ListenerExecutionFailedException` we need to ensure that the classification examines the exception causes. The default classifier just looks at the top level exception.

Since *Spring Retry 1.0.3*, the `BinaryExceptionClassifier` has a property `traverseCauses` (default `false`). When `true` it will traverse exception causes until it finds a match or there is no cause.

To use this classifier for retry, use a `SimpleRetryPolicy` created with the constructor that takes the max attempts, the `Map` of `Exception`s and the boolean (traverseCauses), and inject this policy into the `RetryTemplate`.

3.16 Debugging

Spring AMQP provides extensive logging, especially at `DEBUG` level.

If you wish to monitor the AMQP protocol between the application and broker, you could use a tool such as WireShark, which has a plugin to decode the protocol. Alternatively the RabbitMQ java client comes with a very useful class `Tracer`. When run as a `main`, by default, it listens on port 5673 and connects to port 5672 on localhost. Simply run it, and change your connection factory configuration to connect to port 5673 on localhost. It displays the decoded protocol on the console. Refer to the `Tracer` javadocs for more information.

4. Erlang integration

4.1 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.

4.2 Communicating with Erlang processes

Executing RPC

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

```java
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'

```java
@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 and 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.

ErlangConverter

The ErlangConverter interface is shown below.

```java
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 Erlang 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;
}
```

4.3 Exceptions

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

5. Sample Applications

5.1 Introduction

The Spring AMQP Samples 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 both Maven-based, so you should be able to import them directly into any Maven-aware IDE (such as SpringSource Tool Suite).

5.2 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.

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.

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

The configuration also contains an instance of `RabbitAdmin`, which by default looks for any beans of type Exchange, Queue, or Binding and then declares 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 3.3, "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.

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.

5.3 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.bind(
        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;
}
```

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 III. Spring Integration - Reference

This part of the reference documentation provides a quick introduction to the AMQP support within the Spring Integration project.

6. Spring Integration AMQP Support

6.1 Introduction

The [Spring Integration](#) project includes AMQP Channel Adapters and Gateways that build upon the Spring AMQP project. Those adapters are developed and released in the Spring Integration project. In Spring Integration, "Channel Adapters" are unidirectional (one-way) whereas "Gateways" are bidirectional (request-reply). We provide an inbound-channel-adapter, outbound-channel-adapter, inbound-gateway, and outbound-gateway.

Since the AMQP adapters are part of the Spring Integration release, the documentation will be available as part of the Spring Integration distribution. As a taster, we just provide a quick overview of the main features here.

6.2 Inbound Channel Adapter

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

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

6.3 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"/>
```

6.4 Inbound Gateway

To receive an AMQP Message from a Queue, and respond to its reply-to address, configure an <inbound-gateway>.

```
<amqp:inbound-gateway request-channel="fromAMQP"
                      reply-channel="toAMQP"
                      queue-names="some.queue"
                      connection-factory="rabbitConnectionFactory"/>
```

6.5 Outbound Gateway

To send AMQP Messages to an Exchange and receive back a response from a remote client, configure an <outbound-gateway>. A 'routing-key' may optionally be provided in addition to the exchange name.

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

Part IV. Other Resources

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

7. 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 2.8.x version, and it officially supports AMQP 0.8 and 0.9.1. We recommend reading the 0.9.1 document.

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 GA release, we are looking forward to a lot of community feedback!

Bibliography

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