1.0.0 M1

Copyright © 2004-2008 Mark Pollack, Mark Fisher, Oleg Zhurakousky

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

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 an overview of Spring AMQP and the underlying concepts.

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. It part also includes a chapter on integration with Erlang and a chapter about the sample applications.

Chapter 1. Using Spring AMQP

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

1.1. AMQP Abstractions

The Spring AMQP project consists of a few assemblies. The assembly `Spring.Messaging.Amqp` contains 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. As a result, 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.Messaging.Amqp.Rabbit`'. For the M1 release RabbitMQ and Apache Qpid (2 versions) have been used to vet these base abstractions.

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

1.1.1. Message

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

```
public class Message
{
    private readonly IMessageProperties messageProperties;

    private readonly byte[] body;

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

    public byte[] Body
    {
        get { return body; }
    }

    public IMessageProperties MessageProperties
    {
        get { return messageProperties; }
    }

}
```

The IMessageProperties 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 val)` method.

1.1.2. Exchange

The IExchange interface represents an AMQP Exchange, which is what a Message Producer sends to. Each Exchange within a virtual host of a broker will have a unique name as well as a few other properties:

```
    public interface IExchange
    {
        string Name { get; }

        ExchangeType ExchangeType { get; }

        bool Durable { get; }

        bool AutoDelete { get;  }

        IDictionary Arguments { get;  }

    }
```

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

Note

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

1.1.3. Queue

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

```
public class Queue  {

   private readonly string name;

   private volatile bool durable;

   private volatile bool exclusive;

   private volatile bool autoDelete;

   private volatile IDictionary arguments;

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

   // Property setter and getters  omitted for brevity
}
```

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

1.1.4. Binding

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

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

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

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

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

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

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

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

```
Binding b = BindingBuilder.From(someQueue).To(someTopicExchange).With("foo.*");
```

By itself, an instance of the Binding class is just holding the data about a connection. In other words, it is not an "active" component. However, as you will see later in Section 1.7, “Configuring the broker”, Binding instances can be used by the IAmqpAdmin interface to actually trigger the binding actions on the broker.

The interface IAmqpTemplate is also defined within the Spring.Messaging.Amqp assembly. As one of the main components involved in actual AMQP messaging, it is discussed in detail in its own section (see Section 1.3, “AmqpTemplate”).

1.2. Connection and Resource Management

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

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

```
SingleConnectionFactory connectionFactory = new SingleConnectionFactory("somehost");
connectionFactory.UserName = "guest";
connectionFactory.Password = "guest";

IConnection connection = connectionFactory.CreateConnection();
```

When using XML, the configuration might look like this:

```
<object id="ConnectionFactory" type="Spring.Messaging.Amqp.Rabbit.Connection.SingleConnectionFactory, Spring.Messaging.Amqp.
```

```
  <property name="Username" value="guest"/>
  <property name="Password" value="guest"/>
</object>
```

Note

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

1.3. AmqpTemplate

As with many other high-level abstractions provided by the Spring Framework and related projects, Spring AMQP provides a "template" that plays a central role. The interface that defines the main operations is called IAmqpTemplate. 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 one complete implementation: RabbitTemplate but the QpidTemplate has some of its methods implemented in M1. In the examples that follow, you will often see usage of an "`IAmqpTemplate`", 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 IAmqpTemplate 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. The IRabbitOperations interface contains additional send and execute methods that rely on specific RabbitMQ AMQP channel classes.

1.4. Sending messages

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

```
void Send(MessageCreatorDelegate messageCreator);

void Send(string routingkey, MessageCreatorDelegate messageCreator);

void Send(string exchange, string routingKey, MessageCreatorDelegate messageCreatorDelegate);
```

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:

```
template.Send("marketData.topic", "quotes.nasdaq.FOO", channel => new Message(Encoding.UTF8.GetBytes("12.34"), somePropertie
```

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.Exchange = "marketData.topic";
amqpTemplate.Send("quotes.nasdaq.FOO", channel => new Message(Encoding.UTF8.GetBytes("12.34"), someProperties) );
```

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

```
amqpTemplate.Exchange = "marketData.topic";
amqpTemplate.RoutingKey ="quotes.nasdaq.FOO";
amqpTemplate.Send(channel => new Message(Encoding.UTF8.GetBytes("12.34"), 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(new SingleConnectionFactory()); // using default no-name Exchange
template.Send("queue.helloWorld", channel => 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.RoutingKey = "queue.helloWorld";       // but we'll always send to this Queue
template.Send(channel => new Message(Encoding.UTF8.GetBytes("Hello World"), someProperties) );
```

1.5. Receiving messages

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

1.5.1. Synchronous Reception

The IAmqpTemplate 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();

Message Receive(string queueName);
```

1.5.2. Asynchronous Reception

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

```
    public interface IMessageListener
    {
        void OnMessage(Message message);
    }
```

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

```
    public interface IChannelAwareMessageListener
    {
        void OnMessage(Message message, IModel model);
    }
```

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 POCO" support. When using the adapter, you only need to provide a reference to the instance that the adapter itself should invoke.

```
IMessageListener 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.ConnectionFactory = rabbitConnectionFactory;
container.Queue = "some.queue";
container.MessageListener = someListener;
```

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

```
  <object name="MessageListenerContainer" type="Spring.Messaging.Amqp.Rabbit.Listener.SimpleMessageListenerContainer, Spring
    <property name="ConnectionFactory" ref="RabbitConnectionFactory"/>
    <property name="Queue" value="some.queue"/>
    <property name="MessageListener" ref="SomeListener"/>
  </object>
```

1.6. Message Converters

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

```
    public interface IMessageConverter
    {
        Message ToMessage(object obj, IMessagePropertiesFactory messagePropertiesFactory);

        object FromMessage(Message message);
    }
```

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

```
        void ConvertAndSend(object message);
```

```
        void ConvertAndSend(string routingKey, object message);

        void ConvertAndSend(string exchange, string routingKey, object message);

        void ConvertAndSend(object message, MessagePostProcessorDelegate messagePostProcessorDelegate);

        void ConvertAndSend(string routingKey, object message, MessagePostProcessorDelegate messagePostProcessorDelegate);

        void ConvertAndSend(string exchange, string routingKey, object message, MessagePostProcessorDelegate messagePostProc
```

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();

        object ReceiveAndConvert(string queueName);
```

1.6.1. SimpleMessageConverter

The default implementation of the IMessageConverter 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, and simple byte arrays.

1.6.1.1. Converting From a Message

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

In the next two sections, we'll explore some alternatives for passing rich domain object content without relying on .NET (byte[])serialization.

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

1.6.1.2. Converting To a Message

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

1.6.2. JsonMessageConverter

One rather common approach to object serialization that is flexible and portable across different languages and platforms is JSON (JavaScript Object Notation). An implementation is available and can be configured on any RabbitTemplate instance to override its usage of the SimpleMessageConverter default.

```
<object name="RabbitTemplate" type="Spring.Messaging.Amqp.Rabbit.Core.RabbitTemplate, Spring.Messaging.Amqp.Rabbit">
  <property name="ConnectionFactory" ref="ConnectionFactory"/>
  <property name="MessageConverter">
    <object type="Spring.Messaging.Amqp.Support.Converter.JsonMessageConverter, Spring.Messaging.Amqp">
      <property name="TypeMapper" ref="CustomTypeMapper"/>
    </object>
  </property>
</object>
```

1.7. Configuring the broker

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

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

```
    public interface IAmqpAdmin
    {

        void DeclareExchange(IExchange exchange);

        void DeleteExchange(string exchangeName);

        Queue DeclareQueue();

        void DeclareQueue(Queue queue);

        void DeleteQueue(string queueName);

        void DeleteQueue(string queueName, bool unused, bool empty);

        void PurgeQueue(string queueName, bool noWait);

        void DeclareBinding(Binding binding);

    }
```

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

Note

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

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

```
<object id="ConnectionFactory" type="Spring.Messaging.Amqp.Rabbit.Connection.SingleConnectionFactory, Spring.Messaging.Amqp.
    <constructor-arg value="localhost"/>
    <property name="username" value="guest"/>
    <property name="password" value="guest"/>
</object>


<object name="AmqpAdmin" type="Spring.Messaging.Amqp.Rabbit.Core.RabbitAdmin, Spring.Messaging.Amqp.Rabbit">
  <property name="ConnectionFactory" ref="ConnectionFactory"/>
</object>
```

There is also a more extensive set of administration operations available that are specific to the RabbitMQ broker. Thesse are in the interface IRabbitBrokerOperations and are implemented in the class RabbitBrokerAdmin. The implementation uses an Erlang interopability library to make Erlang RPC calls to the server. The functionality mimics what is available in rabbitmqctl.bat.

```
    public interface IRabbitBrokerOperations : IAmqpAdmin
    {
```

```
        void RemoveBinding(Binding binding);

        RabbitStatus Status { get; }

        IList<QueueInfo> Queues { get; }

        // User management

        void AddUser(string username, string password);

        void DeleteUser(string username);

        void ChangeUserPassword(string username, string newPassword);

        IList<string> ListUsers();

        void StartBrokerApplication();

        void StopBrokerApplication();

        /// <summary>
        /// Starts the node. NOT YET IMPLEMENTED!
        /// </summary>
        void StartNode();

        void StopNode();

        void ResetNode();

        void ForceResetNode();

        // NOTE THE OPERATIONS BELOW ARE NOT YET IMPLEMENTED IN M1

        // VHost management

        int AddVhost(string vhostPath);

        int DeleteVhost(string vhostPath);

        // permissions

        void SetPermissions(string username, Regex configure, Regex read, Regex write);

        void SetPermissions(string username, Regex configure, Regex read, Regex write, string vhostPath);

        void ClearPermissions(string username);

        void ClearPermissions(string username, string vhostPath);

        List<string> ListPermissions();

        List<string> ListPermissions(string vhostPath);

        List<string> ListUserPermissions(string username);
    }
```

You instantiate an instance of RabbitBrokerAdmin by passing an Spring Rabbit IConnectionFactory reference to its constructor. Please refer to the API docs for the contents of the RabbitStatus and QueueInfo classes.

Chapter 2. Erlang integration

2.1. Introduction

There is an open source project located on github called Erlang.NET. It provides to .NET what JInterface provides to Java users, namely a means to communicate with an Erlang process. The API is very low level and rather tedious to use. The Spring Elang project makes accessing functions in Erlang from .NET easy, often they can be one liners.

2.2. Communicating with Erlang processes

TODO

2.2.1. Executing RPC

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

```
public interface IErlangOperations
{
    T Execute<T>(ConnectionCallbackDelegate<T> action);

    OtpErlangObject ExecuteErlangRpc(string module, string function, OtpErlangList args);

    OtpErlangObject ExecuteErlangRpc(string module, string function, params OtpErlangObject[] args);

    OtpErlangObject ExecuteRpc(string module, string function, params object[] args);

    object ExecuteAndConvertRpc(string module, string function, IErlangConverter converterToUse,
                                params object[] args);

    // Sweet!
    object ExecuteAndConvertRpc(string module, string function, params object[] args);
}
```

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 Erlang.NET 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'

```
    public void DeleteUser(string username)
    {
        erlangTemplate.ExecuteAndConvertRpc("rabbit_access_control", "delete_user", encoding.GetBytes(username));
    }
```

The 'encoding' field is simply and instance of ASCIIEncoding.

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

2.2.2. ErlangConverter

The IErlangConverter interface is shown below

```
public interface IErlangConverter
```

```
    {
        /// <summary>
        /// Convert a .NET object to a Erlang data type.
        /// </summary>
        OtpErlangObject ToErlang(object objectToConvert);

        /// <summary>
        /// Convert from an Erlang data type to a .NET data type.
        /// </summary>
        object FromErlang(OtpErlangObject erlangObject);

        /// <summary>
        /// The return value from executing the Erlang RPC.
        /// </summary>
        object FromErlangRpc(string module, string function, OtpErlangObject erlangObject);
    }
```

The provided implementation is SimpleErlangConverter which is used by default with ErlangTemplate and handles all basic types.

Chapter 3. Sample Applications

3.1. Introduction

The Spring AMQP project includes two sample applications. The first is a simple "Hello World" example that demonstrates both synchronous and asynchronous message reception. It provides an excellent starting point for acquiring an understanding of the essential components. The second sample is based on a simplified stock-trading use case to demonstrate the types of interaction that would be common in real world applications. In this chapter, we will provide a quick walk-through of each sample so that you can focus on the most important components. The samples are available in the distribution in the main solution file.

3.2. Hello World

The Hello World sample demonstrates both synchronous and asynchronous message reception.

3.2.1. Synchronous Example

Within the HelloWorld solution folder navigate to the Spring.Amqp.HelloWorld.BrokerConfiguration class. Run the "Program.cs" main application there in order to create a new queue declartion named "`hello.world.queue`" on the broker.

The HelloWorld/Sync solution folder has a project named Spring.Amqp.HelloWorldProducer. The Spring XML configuration for creating the RabbitTemplate instance is shown below

```
<objects xmlns="http://www.springframework.net">

  <object id="ConnectionFactory" type="Spring.Messaging.Amqp.Rabbit.Connection.SingleConnectionFactory, Spring.Messaging.Amq

  </object>

  <object id="RabbitTemplate" type="Spring.Messaging.Amqp.Rabbit.Core.RabbitTemplate, Spring.Messaging.Amqp.Rabbit">
    <constructor-arg ref="ConnectionFactory"/>
    <!-- The queue will be bound to the default direct exchange unless specified otherwise -->
    <property name="Queue" value="hello.world.queue"/>
    <property name="RoutingKey" value="hello.world.queue"/>
  </object>

</objects>
```

This is identical to the configuration of the Consumer application.

Looking back at the "rabbitTemplate" object definition 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 Program.cs file in the Producer project It contains a main() method where the Spring ApplicationContext is created.

```
        static void Main(string[] args)
        {
            using (IApplicationContext ctx = ContextRegistry.GetContext())
            {
                IAmqpTemplate amqpTemplate = (IAmqpTemplate) ctx.GetObject("RabbitTemplate");
                log.Info("Sending hello world message.");
                amqpTemplate.ConvertAndSend("Hello World");
                log.Info("Hello world message sent.");
```

```
            }

            Console.WriteLine("Press 'enter' to exit.");
            Console.ReadLine();
        }
```

As you can see in the example above, an instance of the IAmqpTemplate interface is retrieved and used for sending a Message. Since the client code should rely on interfaces whenever possible, the type is IAmqpTemplate rather than RabbitTemplate. Even though this is just a simple example, 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 IMessageConverter 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 project. It actually shares the same configuration as the producer project. The Consumer code is basically a mirror image of the Producer, calling ReceiveAndConvert() rather than ConvertAndSend().

```
        static void Main(string[] args)
        {
            using (IApplicationContext ctx = ContextRegistry.GetContext())
            {
                IAmqpTemplate amqpTemplate = (IAmqpTemplate)ctx.GetObject("RabbitTemplate");
                log.Info("Synchronous pull");
                String message = (String) amqpTemplate.ReceiveAndConvert();
                if (message == null)
                {
                    log.Info("[No message present on queue to receive.]");
                }
                else
                {
                    log.Info("Received: " + message);
                }
            }

            Console.WriteLine("Press 'enter' to exit.");
            Console.ReadLine();
        }
```

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

3.2.2. Asynchronous Example

Now that we've walked through the synchronous Hello World sample, it's time to move on to a slightly more advanced but significantly more powerful option. With a few modifications, the Hello World sample can provide an example of asynchronous reception, a.k.a. *Message-driven POCOs*. In fact, there is a proejct that provides exactly that in HelloWorld/Async solution folder.

Once again, we will start with the sending side. Open the ProducerConfiguration class and notice that it creates a "connectionFactory" and "rabbitTemplate" object definition. 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.

```
<objects xmlns="http://www.springframework.net">

  <object id="ConnectionFactory" type="Spring.Messaging.Amqp.Rabbit.Connection.SingleConnectionFactory, Spring.Messaging.Amq

  </object>

  <object id="RabbitTemplate" type="Spring.Messaging.Amqp.Rabbit.Core.RabbitTemplate, Spring.Messaging.Amqp.Rabbit">
    <constructor-arg ref="ConnectionFactory"/>
```

```
    <!-- The queue will be bound to the default direct exchange unless specified otherwise -->
    <property name="RoutingKey" value="hello.world.queue"/>
  </object>

</objects>
```

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).

```
    class Program
    {
        private static readonly ILog log = LogManager.GetLogger(typeof(Program));

        static void Main(string[] args)
        {
            using (IApplicationContext ctx = ContextRegistry.GetContext())
            {
                IAmqpTemplate amqpTemplate = (IAmqpTemplate)ctx.GetObject("RabbitTemplate");
                int i = 0;
                while (true)
                {
                    amqpTemplate.ConvertAndSend("Hello World " + i++);
                    log.Info("Hello world message sent.");
                    Thread.Sleep(3000);
                }
            }
        }
    }
```

Now, let's turn to the receiving side. To emphasize the Message-driven POCO 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)
        {
            Console.WriteLine("Received: " + text);
        }
    }
```

Clearly, that *is* a POCO. 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 Application.xml configuration file. You can see the POCO declared there.

```
<objects xmlns="http://www.springframework.net">

  <object id="ConnectionFactory" type="Spring.Messaging.Amqp.Rabbit.Connection.SingleConnectionFactory, Spring.Messaging.Amq

  </object>

  <object id="MessageListenerContainer" type="Spring.Messaging.Amqp.Rabbit.Listener.SimpleMessageListenerContainer, Spring.M
    <property name="ConnectionFactory" ref="ConnectionFactory"/>
    <property name="Queue" value="hello.world.queue"/>
    <property name="ConcurrentConsumers" value="5"/>
    <property name="MessageListener" ref="MessageListenerAdapter"/>
  </object>


  <object id="MessageListenerAdapter" type="Spring.Messaging.Amqp.Rabbit.Listener.Adapter.MessageListenerAdapter, Spring.Mes
    <property name="HandlerObject" ref="HelloWorldHandler"/>
  </object>

  <object id="HelloWorldHandler" type="Spring.Amqp.HelloWorld.Consumer.Async.HelloWorldHandler, Spring.Amqp.HelloWorld.Consu

  </object>
```

```
</objects>
```

You can start the Producer and Consumer in any order, and you should see messages being sent and received every 3 seconds.

To run the application make sure that you select the properties of the top level solution and select "Multiple Startup Project" option. Pick the producer and consumer applications"

3.3. Stock Trading

TODO

Part III. Other Resources

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

Chapter 4. Further Reading

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

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

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

Bibliography

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