Solace JMS Integration with Spring Framework v4.0

Document Version 1.2

January 2015

This document is an integration guide for using Solace JMS as a JMS provider in the Spring Framework.

The Spring Framework provides a comprehensive programming and configuration model for modern Java-based enterprise applications - on any kind of deployment platform. A key element of Spring is infrastructural support at the application level: Spring focuses on the "plumbing" of enterprise applications so that teams can focus on application-level business logic, without unnecessary ties to specific deployment environments.

The Solace message router supports persistent and non-persistent JMS messaging with high throughput and low, consistent latency. Thanks to very high capacity and built-in virtualization, each Solace message router can replace dozens of software-based JMS brokers in multi-tenant deployments. Since JMS is a standard API, client applications connect to Solace like any other JMS broker so companies whose applications are struggling with performance or reliability issues can easily overcome them by upgrading to Solace's hardware.

Solace Systems®

http://www.solacesystems.com

Table of Contents

1 Overview

This document demonstrates how to integrate Solace Java Message Service (JMS) with the Spring Java Framework for production and consumption of JMS messages. The goal of this document is to outline best practices for this integration to enable efficient use of both the Spring Framework and Solace JMS.

The target audience of this document is developers using the Spring Framework with knowledge of both the Spring Java Framework and JMS in general. As such this document focuses on the technical steps required to achieve the integration. For detailed background on either Solace JMS or the Spring Framework refer to the referenced documents below.

This document is divided into the following sections to cover the Solace JMS integration with Spring Framework:

- Integrating with Spring Framework
- Performance Considerations
- Working with Solace High Availability
- Debugging Tips
- Advanced Topics including:
    - Using SSL Communication
    - Working with Solace Disaster Recovery

1.1 Related Documentation

These documents contain information related to the feature defined in this document

Document ID	Document Title	Document Source
[Solace-JMS-REF]	Solace Messaging API for JMS Developer Guide	Contact Solace Systems Support
[Solace-JMS-API]	Solace JMS API Online Reference Documentation	Contact Solace Systems Support
[Solace-FG]	Solace Messaging Platform – Feature Guide	Contact Solace Systems Support
[Solace-FP]	Solace Messaging Platform – Feature Provisioning	Contact Solace Systems Support
[Solace-CLI]	Solace Appliance Command Line Interface Reference	Contact Solace Systems Support
[Spring-REF]	Spring Framework Reference Documentation	http://docs.spring.io/spring/docs/4.0.3.RELEASE/spring-framework-reference/htmlsingle/
[Spring-API]	Spring Framework API Online Reference Documentation	http://docs.spring.io/spring/docs/4.0.3.RELEASE/javadoc-api/

Table 1 - Related Documents

2 Why Solace

There are many reasons why the Solace Messaging Router (Solace appliance) is the messaging platform of choice. The following is a summary. To learn more visit *http://www.solacesystems.com*.

Performance

Solace brings the fast, predictable performance of purpose-built hardware to application messaging and data movement.

- Solace message routers support 50-100x higher throughput than software solutions – millions of messages per second, and hundreds of thousands per second with fully guaranteed, sequential delivery.
- Solace offers low, predictable latency thanks to a pure hardware datapath that eliminates the variability introduced by software and operating systems.
- Thanks to integrated WAN optimization features like streaming compression, Solace supports 30 times more throughput than competitive messaging products over the same long-distance network.

Robustness

Solace's solution is the most reliable and resilient messaging platform available.

- Solace message routers offer high availability and disaster recovery without the need for 3rd party products, and fast failover times no other solution can match.
- The use of TCP distribution (instead of multicast) ensures an orderly, well-behaved system under load while still providing high performance.
- Patented techniques only possible in hardware ensure that the performance of publishers and high-speed consumers is never impacted by disconnected or slow consumers.

Simplicity

Solace simplifies the architecture and operation of your IT infrastructure.

- Solace supports all kinds of data movement and messaging qualities of service, including WAN distribution and web streaming, in a single platform, eliminating the complexity and fragility of integrating multiple platforms and bridging environments.
- A common API for all kinds of messaging makes it easy for your developers to build applications faster.
- Unified administration framework simplifies monitoring and management.

Savings

Solace typically helps customers reduce the TCO of their messaging infrastructure by 80% or more.

- Virtualization lets dozens of applications share each high-capacity message router which enables customers to replace existing messaging products and host hardware, with one compact and cost-effective platform.
- Support for many kinds of data movement lets Built-in functionality like WAN optimization, web messaging, high availability and disaster recovery eliminates the need for many third-party add-on products.
- Easier operations and superior robustness make Solace less expensive to maintain than software alternatives.
- Solace message routers are bought as a one-time capital expense, without restrictive per-CPU licenses or complicated ELAs

3 Integrating with Spring Framework

The general Spring Framework support for JMS integration is outlined in detail in the [Spring-REF]. There are many ways to integrate the Spring Framework and Solace JMS. The configuration outlined in this document makes use of Spring messaging resource caching and JNDI object caching to achieve the desired integration with Solace.

In order to illustrate the Spring Framework integration, the following sections will highlight the required Spring Framework configuration changes and provide snippets of sample code for sending and receiving messages. The full Spring XML configuration file (SolResource.xml) and `MessageProducer` and `MessageConsumer` Java code can be found in the Section Appendix - Configuration and Java Source Reference.

Spring supports several ways of configuring containers. The following sections use the XML-based configuration but integration with Solace JMS would be equally supported using annotation-based configuration. For more information on Spring container configuration see [Spring-REF] Section 4.12 - Java-based container configuration.

This integration guide demonstrates how to configure a Spring application to send and receive JMS messages using a shared JMS connection. Accomplishing this requires completion of the following steps.

- Step 1 - Configuration of the Solace Appliance.
- Step 2 – Configuring the Spring Framework to connect to the Solace appliance.
- Step 3 – Configuring the Spring Application to send messages using Solace JMS.
- Step 4 – Configuring the Spring Application to receive messages using Solace JMS.

3.1 Description of Resources Required

This integration guide will demonstrate creation of Solace resources and configuration of the Spring Framework resources. This section outlines the resources that are created and used in the subsequent sections.

3.1.1 Solace Resources

The following Solace appliance resources are required.

Resource	Value	Description
Solace appliance IP:Port	__IP:Port__	The IP address and port of the Solace appliance message backbone. This is the address clients use when connecting to the Solace appliance to send and receive message. This document uses a value of __IP:PORT__.
Message VPN	Solace_Spring_VPN	A Message VPN, or virtual message broker, to scope the integration on the Solace appliance.
Client Username	spring_user	The client username.
Client Password	spring_password	Optional client password.
Solace Queue	Q/requests	Solace destination of messages produced and consumed
JNDI Connection Factory	JNDI/CF/spring	The JNDI Connection factory for controlling Solace JMS connection properties
JNDI Queue Name	JNDI/Q/requests	The JNDI name of the queue used in the samples

Table 2 – Solace Configuration Resources

3.1.2 Spring Framework Configuration Resources

The following Spring container configuration is referenced in the integration steps. These items are explained in detail in the [Spring-REF]. The Section 8.1 SolResources.xml contains the full Spring configuration file for these resources and how each of these resources relates to integration with Solace is explained in the subsequent sections as these resources are introduced.

Resource	Value
org.springframework.jndi.JndiTemplate	solaceJndiTemplate
org.springframework.jndi.JndiObjectFactoryBean	solaceConnectionFactory
org.springframework.jms.connection.CachingConnectionFactory	solaceCachedConnectionFactory
org.springframework.jndi.JndiObjectFactoryBean	destination
org.springframework.jms.core.JmsTemplate	jmsTemplate
org.springframework.jms.listener.DefaultMessageListenerContainer	jmsContainer

Table 3 – Spring Configuration Resources

3.2 Step 1 – Configuring the Solace Appliance

The Solace appliance needs to be configured with the following configuration objects at a minimum to enable JMS to send and receive messages within the Spring Framework.

- A Message VPN, or virtual message broker, to scope the integration on the Solace appliance.
- Client connectivity configurations like usernames and profiles
- Guaranteed messaging endpoints for receiving messages.
- Appropriate JNDI mappings enabling JMS clients to connect to the Solace appliance configuration.

For reference, the CLI commands in the following sections are from SolOS version 6.2 but will generally be forward compatible. For more details related to Solace appliance CLI see [Solace-CLI]. Wherever possible, default values will be used to minimize the required configuration. The CLI commands listed also assume that the CLI user has a Global Access Level set to Admin. For details on CLI access levels please see [Solace-FG] section “User Authentication and Authorization”.

Also note that this configuration can also be easily performed using SolAdmin, Solace’s GUI management tool. This is in fact the recommended approach for configuring a Solace appliance. This document uses CLI as the reference to remain concise.

3.2.1 Creating a Message VPN

This section outlines how to create a message-VPN called “Solace_Spring_VPN” on the Solace appliance with authentication disabled and 2GB of message spool quota for Guaranteed Messaging. This message-VPN name is required in the Spring configuration when connecting to the Solace messaging appliance. In practice appropriate values for authentication, message spool and other message-VPN properties should be chosen depending on the end application’s use case.

```
(config)# create message-vpn Solace_Spring_VPN
(config-msg-vpn)# authentication
(config-msg-vpn-auth)# user-class client
(config-msg-vpn-auth-user-class)# basic auth-type none
(config-msg-vpn-auth-user-class)# exit
(config-msg-vpn-auth)# exit
(config-msg-vpn)# no shutdown
(config-msg-vpn)# exit
(config)#
(config)# message-spool message-vpn Solace_Spring_VPN
(config-message-spool)# max-spool-usage 2000
(config-message-spool)# exit
(config)#
```

3.2.2 Configuring Client Usernames & Profiles

This section outlines how to update the default client-profile and how to create a client username for connecting to the Solace appliance. For the client-profile, it is important to enable guaranteed messaging for JMS messaging and transacted sessions if using transactions.

The chosen client username of “spring_user” will be required by the Spring Framework when connecting to the Solace appliance.

```
(config)# client-profile default message-vpn Solace_Spring_VPN
(config-client-profile)# message-spool allow-guaranteed-message-receive
(config-client-profile)# message-spool allow-guaranteed-message-send
(config-client-profile)# message-spool allow-transacted-sessions
(config-client-profile)# exit
(config)#
(config)# create client-username spring_user message-vpn Solace_Spring_VPN
(config-client-username)# acl-profile default
(config-client-username)# client-profile default
(config-client-username)# no shutdown
(config-client-username)# exit
(config)#
```

3.2.3 Setting up Guaranteed Messaging Endpoints

This integration guide shows receiving messages within the Spring Framework from a single JMS Queue. For illustration purposes, this queue is chosen to be an exclusive queue with a message spool quota of 2GB matching quota associated with the message VPN. The queue name chosen is “Q/requests”.

```
(config)# message-spool message-vpn Solace_Spring_VPN
(config-message-spool)# create queue Q/requests
(config-message-spool-queue)# access-type exclusive
(config-message-spool-queue)# max-spool-usage 2000
(config-message-spool-queue)# permission all delete
(config-message-spool-queue)# no shutdown
(config-message-spool-queue)# exit
(config-message-spool)# exit
(config)#
```

3.2.4 Setting up Solace JNDI References

To enable the JMS clients to connect and look up the Queue destination required by Spring, there are two JNDI objects required on the Solace appliance:

- A connection factory: JNDI/CF/spring
- A queue destination: JNDI/Q/requests

They are configured as follows:

```
(config)# jndi message-vpn Solace_Spring_VPN
(config-jndi)# create connection-factory JNDI/CF/spring
(config-jndi-connection-factory)# property-list messaging-properties
(config-jndi-connection-factory-pl)# property default-delivery-mode persistent
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# property-list transport-properties
(config-jndi-connection-factory-pl)# property direct-transport false
(config-jndi-connection-factory-pl)# property "reconnect-retry-wait" "3000"
(config-jndi-connection-factory-pl)# property "reconnect-retries" "20"
(config-jndi-connection-factory-pl)# property "connect-retries-per-host" "5"
(config-jndi-connection-factory-pl)# property "connect-retries" "1"
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# exit
(config-jndi)#
(config-jndi)# create queue JNDI/Q/requests
(config-jndi-queue)# property physical-name Q/requests
(config-jndi-queue)# exit
(config-jndi)#
(config-jndi)# no shutdown
(config-jndi)# exit
(config)#
```

3.3 Step 2 – Spring Framework – Connecting

The following configuration is required to successfully establish a connection from Spring to the Solace appliance.

The example configuration below uses XML-based container configuration to illustrate the integration. The "___IP:PORT___" should be updated to reference the actual Solace appliance message-backbone VRF IP.

In Solace JMS, the "java.naming.security.principal" often uses the format <username>@<message-vpn>. This allows specification of the Solace appliance client username ("spring_user") and message-vpn ("Solace_Spring_VPN") created in the previous section. Both of these items are mandatory in order to connect to the Solace appliance.

The "java.naming.security.credentials" is optional and provides the Solace appliance client password for use when authenticating with the Solace appliance. In this example a password is not used and so this parameter is left commented in the configuration. For further details on authentication see Section 7.1.

```
<bean id="solaceJndiTemplate" class="org.springframework.jndi.JndiTemplate"
  lazy-init="default" autowire="default">
  <property name="environment">
    <map>
      <entry key="java.naming.provider.url" value="smf://___IP:PORT___" />
      <entry key="java.naming.factory.initial"
             value="com.solacesystems.jndi.SolJNDIInitialContextFactory" />
      <entry key="java.naming.security.principal"
             value="spring_user@Solace_Spring_VPN" />
<!-- <entry key="java.naming.security.credentials"
             value="spring_password" /> -->
    </map>
  </property>
</bean>

<bean id="solaceConnectionFactory"
  class="org.springframework.jndi.JndiObjectFactoryBean"
  lazy-init="default" autowire="default">
  <property name="jndiTemplate" ref="solaceJndiTemplate" />
  <property name="jndiName" value="JNDI/CF/spring" />
</bean>

<bean id="solaceCachedConnectionFactory"
  class="org.springframework.jms.connection.CachingConnectionFactory">
  <property name="targetConnectionFactory" ref="solaceConnectionFactory" />
  <property name="sessionCacheSize" value="10" />
</bean>
```

The following table explains each bean configuration and its purpose when connecting to the Solace appliance.

Bean Id	Description
solaceJndiTemplate	This template outlines general connection details for reaching the Solace JNDI hosted on the Solace appliance. The Solace JNDI is used to look up parameters for client connections and for destinations.
solaceConnectionFactory	This references a specific connection factory within the Solace JNDI that will be used when creating new connections. The value for “jndiName” is the connection factory name as configured in the Solace JNDI. In the previous section this was configured as “JNDI/CF/spring”
solaceCachedConnectionFactory	The cached connection factory allows for re-use of the Solace connection when sending messages. For efficient integration within the Spring Framework, it is essential that connection caching be enabled and configured correctly. There are more details on this in Section 4 Performance Considerations including discussion of the `sessionCacheSize` attribute. It is this connection factory that is used by the producer and consumer clients when connecting.

Table 4 - Solace Connection Configuration

3.4 Step 3 – Spring Framework – Sending Messages to Solace

In general the `JmsTemplate` is a convenient and recommended way to send messages from within the Spring Framework. The `JmsTemplate` contains several methods for sending messages including methods where the target JMS destination can be specified at send time or alternatively an option where no destination is provided which uses the default JMS producer destination on send. Details of the `JmsTemplate` are covered in the [Spring-REF], Section Java Message Service.

3.4.1 Configuration

The configuration below is used by the message producer to send JMS messages to the Solace appliance.

```
<bean id="destination" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiTemplate" ref="solaceJndiTemplate" />
  <property name="jndiName" value="JNDI/Q/requests" />
</bean>

<bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="solaceCachedConnectionFactory" />
  <property name="defaultDestination" ref="destination" />
  <property name="deliveryPersistent" value="true" />
  <property name="explicitQosEnabled" value="true" />
</bean>

<bean id="messageProducer"
  class="com.solacesystems.integration.spring.MessageProducer">
  <property name="jmsTemplate" ref="jmsTemplate" />
</bean>
```

The following table explains the configuration and its purpose when publishing to the Solace appliance.

Bean Id	Description
destination	This configuration defines a JMS destination for use in sending. The destination is found in JNDI by looking up the name “JNDI/Q/requests” which was previously configured on the Solace appliance JNDI as a queue destination.
jmsTemplate	The jmsTemplate is the core component of the Spring framework integration with JMS. It contains the reference to the connection factory, a default destination for sending and parameters for customizing the JMS producer from within the Spring framework. A full list of available parameters is documented in [Spring-API].
messageProducer	The message producer is a reference to the producer Java code that will be used to send messages to the Solace appliance. This configuration connects that Java code to the correct `JmsTemplate` configuration.

Table 5 - Solace Publish Configuration

3.4.2 Message Producer Java Code

The following code sample illustrates basic message publishing from within the Spring Framework. This code will create a simple JMS Text message with contents “test” and send this to the Solace appliance using the default destination of the `JmsTemplate`.

```
public class MessageProducer {
  private JmsTemplate jmsTemplate;

  public void sendMessages() throws JMSException {
    getJmsTemplate().send(new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        Message message = session.createTextMessage("test");
        return message;
      }
    });
  }

  public JmsTemplate getJmsTemplate() {
    return jmsTemplate;
  }

  public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
  }
}
```

The publishing code could be run using a simple `main()`. The following assumes that the XML configuration required is present in the file SolResources.xml. This file can be found in Section Appendix - Configuration and Java Source Reference. The `ClassPathXmlApplicationContext` is used to lookup the XML configuration on the class path and create a standalone Spring XML application context. This application context is then used to lookup the message producer bean and send 10 messages over a cached JMS connection.

```
public static void main(String[] args) throws JMSException {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] { "SolResources.xml" });
    MessageProducer producer = (MessageProducer) context.getBean("messageProducer");
    for (int i = 0; i < 10; i++) {
      producer.sendMessages();
    }
    context.close();
  }
```

See Section 4 Performance Considerations for some further performance considerations related to sending messages.

3.5 Step 4 – Spring Framework – Receiving Messages from Solace

The recommended method of receiving messages is through the `DefaultMessageListenerContainer` following the recommendation found in the [Spring-REF], Section “Asynchronous Reception – Message-Driven POJOs”.

3.5.1 Configuration

The configuration outlined below enables receiving of messages via a `DefaultMessageListenerContainer` listener container with message processing being handled by the message consumer as outlined in Section 3.5.2 Message Consumer Java Code.

```
<bean id="destination" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiTemplate" ref="solaceJndiTemplate" />
  <property name="jndiName" value="JNDI/Q/requests" />
</bean>

<bean id="messageConsumer"
  class="com.solacesystems.integration.spring.MessageConsumer">
</bean>

<bean id="jmsContainer"
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="connectionFactory" ref="solaceCachedConnectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageConsumer" />
</bean>
```

The following table explains the configuration and its purpose when receiving messages from the Solace appliance.

Bean Id	Description
destination	This configuration defines a JMS destination that the consumer will bind to for receiving messages. For the purposes of demonstrating integration, it is the same destination as used when sending messages. The destination is found in JNDI by looking up the name “JNDI/Q/requests” which was previously configured on the Solace appliance JNDI as a queue destination.
messageConsumer	This configuration identifies the POJO code responsible for processing an incoming JMS message.
jmsContainer	The JmsContainer links the MessageConsumer with a JMS destination and JMS cached connection using the `DefaultMessageListenerContainer` from the Spring Framework. This enables messages to be correctly received and processed efficiently within the Spring Framework.

Table 6 - Solace Receive Configuration

3.5.2 Message Consumer Java Code

The following is an example of receiving messages using the `MessageListener` interface and the `DefaultMessageListenerContainer` using the consumer configuration. The callback will print the message text for all received messages for the purposes of an example.

```
public class MessageConsumer implements MessageListener {

  public void onMessage(Message message) {
    // Application specific handling code would follow.
    // For this example print the topic of each message
    try {
      System.out.println("Received message on destination: " +
                         message.getJMSDestination().toString());
    } catch (JMSException ex) {
      throw new RuntimeException(ex);
    }
  }
}
```

Similar to the message producer, the message consumer can be run using a simple `main()` as follows. This method will establish the JMS connection and listen for messages over the single JMS connection.

```
  public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] { "SolResources.xml" });
    MessageConsumer consumer = (MessageConsumer) context.getBean("messageConsumer");
  }
```

Note that it is also possible to use the `MessageListenerAdapter` provided by Spring in order to avoid using an interface when receiving messages. This is outlined in the [Spring-REF] in further detail. In general it makes sense to select the method for receiving messages that most closely matches with the existing behavior of the application. It does not affect the message reception from the Solace appliances.

4 Performance Considerations

The standard JMS API allows clients to send and receive persistent messages at high rates if used efficiently. In order to use the Solace JMS API efficiently, some JMS objects should be cached. The Spring Framework makes it possible to create these objects up front and cache them for re-use. This section outlines how to tune the Spring Framework configuration to properly re-use the following JMS Objects:

- Connection
- Session
- MessageProducer
- MessageConsumer
- Destination

Failure to correctly cache these objects can result in a new connection being established to the Solace appliance for each message sent. This results in low overall performance and is not a recommended method of operating. It is possible to detect this scenario by monitoring the Solace event logs for frequent client connection and disconnection events.

4.1 Caching JMS Connections

Section 3 “Integrating with Spring Framework” outlines the required configuration to enable Connection, Session, MessageProducer and MessageConsumer caching. In Spring, this object caching is controlled by the CachingConnectionFactory. A CachingConnectionFactory contains a single JMS Connection which is reused across all JmsTemplates. In order to enable session caching within the JMS Connection, the sessionCacheSize parameter must be set to specify the number of JMS Session objects to cache for reuse.

One behavior worth noting is that as outlined in the Spring documentation, if the pool of cached sessions is fully utilized and a further request for a cached `Session` is made, then the requested `Session` will be created and disposed on demand. If you couple this behavior with the fact that the default value is 1 for `sessionCacheSize` it is important to configure this to a value that is applicable to the end application to avoid the undesirable behaviour of create and dispose on each call to send a message during periods of high demand. This value should be set to the maximum concurrency required by the application.

The following configuration sample illustrates how to set the various `CachingConnectionFactory` cache sizes.

```
<bean id="SolaceCachedConnectionFactory"
      class="org.springframework.jms.connection.CachingConnectionFactory">
  <property name="targetConnectionFactory" ref="SolaceConnectionFactory" />
  <property name="sessionCacheSize" value="10" />
</bean>
```

4.2 Resolving and Caching JMS Destinations on Send

When working with Solace JMS and using the Solace Appliance as the JNDI provider, it is also important to know that each JNDI lookup of a destination will result in a JNDI request to and response from the Solace appliance. As such, for efficient integration with Solace JMS, destinations should be cached and reused as much as possible. This is very important for producers to consider when sending messages.

There are three options for JMS Destination resolution within the Spring framework. The following sections outline performance considerations for each option. The first option is to use the default destination within the `JmsTemplate`. This is the simplest option. There are also two options for resolving destinations dynamically within the application. The

`DynamicDestinationResolver` is the default option for dynamic destinations and does not make use of JNDI for resolving destinations. The `JndiDestinationResolver` enables dynamic destinations to be resolved using JNDI.

4.2.1 Using the default Destination of a producer

One common way to configure applications is to have a separate producer for each destination. In this case a producer is equivalent to a destination. If the application is configured in this way, then the Spring framework makes it simple to configure the default JMS destination as part of the `JmsTemplate` configuration. The `JmsTemplate` will look up the destination in JNDI and on `MessageProducer` creation and then it will be reused during publish.

This configuration is what is demonstrated earlier in this document. The following is how to set a default destination of a producer through configuration.

```
<bean id="destination" class="org.springframework.jndi.JndiObjectFactoryBean">
  <property name="jndiTemplate" ref="SolaceJndiTemplate" />
  <property name="jndiName" value="JNDI/Q/requests" />
</bean>

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

4.2.2 Resolving destinations dynamically

The `DynamicDestinationResolver` allows destinations to be resolved using the JMS provider's specific String to JMS destination mapping. This is very efficient and bypasses JNDI. Therefore destination names provided by the application must be the physical destinations used by the JMS broker. The [Spring-REF] document contains details related to `DynamicDestinationResolver` in Section "Using Spring JMS" subsection "Destination Management".

In the case of Solace JMS this will be the physical queue or topic name. The [Solace-JMS-API] has an appendix which details the Topic and Queue syntax used by Solace appliances.

The following is an example of how to configure a `DynamicDestinationResolver`.

```
<bean id="dynamicDestinationResolver"
  class="org.springframework.jms.support.destination.DynamicDestinationResolver" />

<bean id="dynamicJmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="solaceCachedConnectionFactory" />
   <property name="destinationResolver" ref="dynamicDestinationResolver" />
</bean>

<bean id="dynamicMessageProducer"
  class="com.solacesystems.integration.spring.DynamicMessageProducer">
  <property name="jmsTemplate" ref="dynamicJmsTemplate" />
</bean>
```

And to send messages, the following Java code will use the `DynamicDestinationResolver` to send messages to a topic of "`T/dynamic/topic`".

```
public class DynamicMessageProducer {
  private JmsTemplate jmsTemplate;

  public void sendMessages() throws JMSException {
    getJmsTemplate().send("T/dynamic/topic", new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        Message message = session.createTextMessage("test");
        return message;
      }
    });
  }

  public JmsTemplate getJmsTemplate() {
    return jmsTemplate;
  }

  public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
  }
}
```

4.2.3 Looking up destinations in JNDI with caching

The `JndiDestinationResolver` allows destinations to be resolved dynamically using JNDI. Because a JNDI lookup is an expensive request, the `JndiDestinationResolver` also allows for caching of destinations through the `setCache()` method. When using this destination resolver with Solace JMS, it is very important to enable destination caching for Solace JMS to work effectively. By default this is enabled in the Spring Framework.

This destination resolver would be a good option for applications that use a group of destinations and send large numbers of messages across this group of destinations. In this scenario, the JNDI destination lookup would occur once for each unique destination and then subsequent publishes would use the destination from the local cache avoiding the cost of the JNDI lookup.

If the application needs to send a small number of messages per destination across a large topic space then this destination resolver would not be a good choice because effectively this would translate into a JNDI lookup per send which is inefficient. A better choice in that specific scenario would be the `DynamicDestinationResolver`.

The following is an example of how to configure a `JndiDestinationResolver`.

```
<bean id="jndiDestinationResolver"
  class="org.springframework.jms.support.destination.JndiDestinationResolver">
  <property name="cache" value="true" />
  <property name="jndiTemplate" ref="solaceJndiTemplate"/>
</bean>

<bean id="jndiJmsTemplate" class="org.springframework.jms.core.JmsTemplate">
  <property name="connectionFactory" ref="solaceCachedConnectionFactory" />
   <property name="destinationResolver" ref="jndiDestinationResolver" />
</bean>

<bean id="jndiMessageProducer"
  class="com.solacesystems.integration.spring.JndiMessageProducer">
  <property name="jmsTemplate" ref="jndiJmsTemplate" />
</bean>
```

And to send messages, the following Java code will use the `JndiDestinationResolver` to send messages to a JNDI destination of "`JNDI/T/requests`".

```
public class JndiMessageProducer {
  private JmsTemplate jmsTemplate;

  public void sendMessages() throws JMSException {
    getJmsTemplate().send("JNDI/T/requests", new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        Message message = session.createTextMessage("test");
        return message;
      }
    });
  }

  public JmsTemplate getJmsTemplate() {
    return jmsTemplate;
  }

  public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
  }
}
```

5 Working with Solace High Availability (HA)

The [Solace-JMS-REF] section “Establishing Connection and Creating Sessions” provides details on how to enable the Solace JMS connection to automatically reconnect to the standby appliance in the case of a HA failover of a Solace appliance. By default Solace JMS connections will reconnect to the standby appliance in the case of an HA failover.

In general the Solace documentation contains the following note regarding reconnection:

> Note: When using HA redundant appliances, a fail-over from one appliance to its mate will typically occur in under 30 seconds, however, applications should attempt to reconnect for at least five minutes.

In section 3.2.4 Setting up Solace JNDI References, the Solace CLI commands correctly configured the required JNDI properties to reasonable values. These commands are repeated here for completeness.

```
config)# jndi message-vpn Solace_Spring_VPN
(config-jndi)# create connection-factory JNDI/CF/spring
(config-jndi-connection-factory)# property-list transport-properties
(config-jndi-connection-factory-pl)# property "reconnect-retry-wait" "3000"
(config-jndi-connection-factory-pl)# property "reconnect-retries" "20"
(config-jndi-connection-factory-pl)# property "connect-retries-per-host" "5"
(config-jndi-connection-factory-pl)# property "connect-retries" "1"
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# exit
(config-jndi)# exit
(config)#
```

6 Debugging Tips for Solace JMS API Integration

The key component for debugging integration issues with the Solace JMS API is the API logging that can be enabled. How to enable logging in the Solace API is described below.

6.1 How to enable Solace JMS API logging

The [Spring-REF] has details on logging within Spring. Since the Solace JMS API also makes use of the Jakarta Commons Logging API (JCL), configuring the Solace JMS API logging is very similar to configuring any other Spring Framework logging. The following example shows how to enable debug logging in the Solace JMS API using log4j.

One note to consider is that since the Solace JMS API has a dependency on the Solace Java API (JCSMP) both of the following logging components should be enabled and tuned when debugging to get full information. For example to set both to debug level:

```
log4j.category.com.solacesystems.jms=DEBUG
log4j.category.com.solacesystems.jcsmp=DEBUG
```

As outlined in the [Spring-REF] there are multiple options for enabling logging within the Spring Framework. By default info logs will be written to the consol. This section will focus on using log4j as the logging library and tuning Solace JMS API logs using the log4j properties. Therefore in order to enable Solace JMS API logging, a user must do two things:

- Put Log4j on the classpath.
- Create a log4j.properties configuration file in the root folder of the classpath

Below is an example Log4j properties file that will enable debug logging within the Solace JMS API.

```
log4j.rootCategory=INFO, stdout
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=%d{ABSOLUTE} %5p %t %c{2}:%L - %m%n
log4j.category.com.solacesystems.jms=DEBUG
log4j.category.com.solacesystems.jcsmp=DEBUG
```

With this you can get output in a format similar to the following which can help in understanding what is happening within the Solace JMS API.

```
14:35:01,171 DEBUG main client.ClientRequestResponse:75 - Starting request timer (SMP-
EstablishP2pSub) (10000 ms)
14:35:01,171 DEBUG Context_2_ReactorThread client.ClientRequestResponse:83 - Stopping
request timer (SMP-EstablishP2pSub)
14:35:01,173  INFO main jms.SolConnection:151 - Connection created.
14:35:01,173  INFO main connection.CachingConnectionFactory:298 - Established shared JMS
Connection: com.solacesystems.jms.SolConnection@ca3f2d
14:35:01,180  INFO main jms.SolConnection:327 - Entering start()
14:35:01,180  INFO main jms.SolConnection:338 - Leaving start() : Connection started.
14:35:01,180  INFO jmsContainer-1 jms.SolConnection:252 - Entering createSession()
```

7 Advanced Topics

7.1 Authentication

JMS Client authentication is handled by the Solace appliance. The Solace appliance supports a variety of authentications schemes as described in [Solace-FG] in the Section “Client Authentication and Authorization”. The required JMS authentication properties can be set in the `JndiTemplate` configuration depending on which authentication scheme is being used. The following example shows how to enable basic authentication using a username of “spring_user” and password of “spring_password”.

```
<bean id="solaceJndiTemplate" class="org.springframework.jndi.JndiTemplate"
    lazy-init="default" autowire="default">
    <property name="environment">
      <map>
        <entry key="java.naming.provider.url" value="smf://___IP:PORT___" />
        <entry key="java.naming.factory.initial"
               value="com.solacesystems.jndi.SolJNDIInitialContextFactory" />
        <entry key="java.naming.security.principal"
               value="spring_user@Solace_Spring_VPN" />
        <entry key="java.naming.security.credentials"
               value="spring_password" />
      </map>
    </property>
  </bean>
```

7.2 Using SSL Communication

This section outlines how to update the Solace appliance and Spring Framework configuration to switch the client connection to using secure connections with the Solace appliance. For the purposes of illustration, this section uses a server certificate on the Solace appliance and basic client authentication. It is possible to configure Solace JMS to use client certificates instead of basic authentication. This is done using configuration steps that are very similar to those outlined in this document. The [Solace-FP] and [Solace-JMS-REF] outline the extra configuration items required to switch from basic authentication to client certificates.

To change a Spring application from using a plain text connection to a secure connection, first the Solace appliance configuration must be updated as outlined in Section 7.2.1 and the Solace JMS configuration within the Spring Framework must be updated as outlined in Section 7.2.2.

7.2.1 Configuring the Solace Appliance

To enable secure connections to the Solace appliance, the following configuration must be updated on the Solace appliance.

- Server Certificate
- TLS/SSL Service Listen Port
- Enable TLS/SSL over SMF in the Message VPN

The following sections outline how to configure these items.

7.2.1.1 Configure the Server Certificate

Before, starting, here is some background detail on the server certificate required by the Solace appliance. This is from the [Solace-FP] section “Setting a Server Certificate”

> To enable the exchange of information through TLS/SSL-encrypted SMF service, you must set the TLS/SSL server certificate file that the Solace appliance is to use. This server certificate is presented to a client during the TLS/SSL handshakes. A server certificate used by an appliance must be an x509v3 certificate and it must include a private key. The server certificate and key use an RSA algorithm for private key generation, encryption and decryption, and they both must be encoded with a Privacy Enhanced Mail (PEM) format.
>
> The single server certificate file set for the appliance can have a maximum chain depth of three (that is, the single certificate file can contain up to three certificates in a chain that can be used for the certificate verification).

To configure the server certificate, first copy the server certificate to the Solace appliance. For the purposes of this example, assume the server certificate file is named “mycert.pem”.

```
# copy sftp://[<username>@]<ip-addr>/<remote-pathname>/mycert.pem /cert
<username>@<ip-addr>'s password:
#
```

Then set the server certificate for the Solace appliance.

```
(config)# ssl server-certificate mycert.pem
(config)#
```

7.2.1.2 Configure TLS/SSL Service Listen Port

By default, the Solace appliance accepts secure messaging client connections on port 55443. If this port is acceptable then no further configuration is required and this section can be skipped. If a non-default port is desired, then follow the steps below. Note this configuration change will disrupt service to all clients of the Solace appliance and should therefore be performed during a maintenance window when this client disconnection is acceptable. This example assumes that the new port should be 55403.

```
(config)# service smf
(config-service-smf)# shutdown
All SMF and WEB clients will be disconnected.
Do you want to continue (y/n)? y
(config-service-smf)# listen-port 55403 ssl
(config-service-smf)# no shutdown
(config-service-smf)# exit
(config)#
```

7.2.1.3 Enable TLS/SSL within the Message VPN

By default within Solace message VPNs both the plain-text and SSL services are enabled. If the Message VPN defaults remain unchanged, then this section can be skipped. However, if within the current application VPN, this service has

been disabled, then for secure communication to succeed it should be enabled. The steps below show how to enable SSL within the SMF service to allow secure client connections from the Spring Framework.

```
(config)# message-vpn Solace_Spring_VPN
(config-msg-vpn)# service smf
(config-msg-vpn-service-smf)# ssl
(config-msg-vpn-service-ssl)# no shutdown
(config-msg-vpn-service-ssl)# exit
(config-msg-vpn-service-smf)# exit
(config-msg-vpn-service)# exit
(config-msg-vpn)# exit
(config)#
```

7.2.2 Configuring the Spring Framework

The configuration in the Spring Framework requires updating the `SolaceJndiTemplate` bean in two ways.

- Updating the provider URL to specify the protocol as secure (smfs)
- Adding the required parameters for the secure connection

7.2.2.1 Updating the provider URL

In order to signal to the Solace JMS API that the connection should be a secure connection, the protocol must be updated in the URI scheme. The Solace JMS API has a URI format as follows:

```
<URI Scheme>://[username]:[password]@<IP address>[:port]
```

Recall from Section 3.3, originally, the “java.naming.provider.url” was as follows:

```
<entry key="java.naming.provider.url" value="smf://___IP:PORT___" />
```

This specified a URI scheme of “smf” which is the plaint-text method of communicating with the Solace appliance. This should be updated to “smfs” to switch to secure communication giving you the following configuration:

```
<entry key="java.naming.provider.url" value="smfs://___IP:PORT___" />
```

7.2.2.2 Adding SSL Related Configuration

Additionally, the Solace JMS API must be able to validate the server certificate of the Solace appliance in order to establish a secure connection. To do this, the following trust store parameters need to be provided.

First the Solace JMS API must be given a location of a trust store file so that it can verify the credentials of the Solace appliance server certificate during connection establishment. This parameter takes a URL or Path to the trust store file.

```
<entry key="Solace_JMS_SSL_TrustStore" value="___Path_or_URL___" />
```

It is also required to provide a trust store password. This password allows the Solace JMS API to validate the integrity of the contents of the trust store. This is done through the following parameter.

```
<entry key="Solace_JMS_SSL_TrustStorePassword" value="___Password___" />
```

There are multiple formats for the trust store file. By default Solace JMS assumes a format of Java Key Store (JKS). So if the trust store file follows the JKS format then this parameter may be omitted. Solace JMS supports two formats for the trust store: “jks” for Java Key Store or “pkcs12”. Setting the trust store format is done through the following parameter.

```
<entry key="Solace_JMS_SSL_TrustStoreFormat" value="jks" />
```

And finally, the authentication scheme must be selected. Solace JMS supports the following authentication schemes for secure connections:

- AUTHENTICATION_SCHEME_BASIC
- AUTHENTICATION_SCHEME_CLIENT_CERTIFICATE

This integration example will use basic authentication. So the required parameter is as follows:

```
<entry key="Solace_JMS_Authentication_Scheme" value="AUTHENTICATION_SCHEME_BASIC" />
```

7.2.2.3 Spring Configuration Bean

The following example bean outlines all of the required Spring Framework configuration changes to the `SolaceJndiTemplate` where the user should substitute appropriate values for:

- ___IP:PORT___
- ___Path_or_URL___
- ___Password___

```
<bean id="SolaceJndiTemplate" class="org.springframework.jndi.JndiTemplate"
  lazy-init="default" autowire="default">
  <property name="environment">
    <map>
      <entry key="java.naming.provider.url" value="smfs://___IP:PORT___" />
      <entry key="java.naming.factory.initial"
             value="com.solacesystems.jndi.SolJNDIInitialContextFactory" />
      <entry key="java.naming.security.principal" value="spring_user" />
      <entry key="Solace_JMS_VPN" value="Solace_Spring_VPN" />

      <!-- SSL Related Configuration -->
      <entry key="Solace_JMS_Authentication_Scheme"
             value="AUTHENTICATION_SCHEME_BASIC" />
      <entry key="Solace_JMS_SSL_TrustStore" value="___Path_or_URL___" />
      <entry key="Solace_JMS_SSL_TrustStoreFormat" value="jks" />
      <entry key="Solace_JMS_SSL_TrustStorePassword" value="___Password___" />
    </map>
  </property>
</bean>
```

7.3 Working with the Solace Disaster Recovery Solution

The [Solace- FG] section “Data Center Replication” contains a sub-section on “Application Implementation” which details items that need to be considered when working with Solace’s Data Center Replication feature. This integration guide will show how the following items required to have a Spring application successfully connect to a backup data center using the Solace Data Center Replication feature.

- Configuring a Host List within the Spring Framework
- Configuring JMS Reconnection Properties within Solace JNDI
- Disaster Recovery Behavior Notes

7.3.1 Configuring a Host List within the Spring Framework

As described in [Solace-FG], the host list provides the address of the backup data center. This is configured within the Spring Framework through the `java.naming.provider.url` which is set within the `solaceJndiTemplate` bean in the sample configuration as follows:

```
<bean id="solaceJndiTemplate" class="org.springframework.jndi.JndiTemplate"
  lazy-init="default" autowire="default">
  <property name="environment">
    <map>
      <entry key="java.naming.provider.url"
             value="smf://__IP_active_site:PORT__,smf://__IP_standby_site:PORT__" />
      <!-- ... Snip remaining configuration ... -->
    </map>
  </property>
</bean>
```

For the `java.naming.provider.url` both the active site IP address and standby site IP address are provided. When connecting, the Solace JMS connection will first try the active site and if it is unable to successfully connect to the active site, then it will try the standby site. This is discussed in much more detail in the referenced Solace documentation.

7.3.2 Configuring reasonable JMS Reconnection Properties within Solace JNDI

In order to enable applications to successfully reconnect to the standby site in the event of a data center failure, it is required that the Solace JMS connection be configured to attempt connection reconnection for a sufficiently long time to enable the manual switch-over to occur. This time is application specific depending on individual disaster recovery procedures and can range from minutes to hours depending on the application. In general it is best to tune the reconnection by changing the “reconnect retries” parameter within the Solace JNDI to a value large enough to cover the maximum time to detect and execute a disaster recovery switch over. If this time is unknown, it is also possible to use a value of “-1” to force the Solace JMS API to reconnect indefinitely.

The reconnect retries is tuned in the Solace appliance CLI as follows:

```
config)# jndi message-vpn Solace_Spring_VPN
(config-jndi)# connection-factory JNDI/CF/spring
(config-jndi-connection-factory)# property-list transport-properties
(config-jndi-connection-factory-pl)# property "reconnect-retries" "-1"
(config-jndi-connection-factory-pl)# exit
(config-jndi-connection-factory)# exit
(config-jndi)# exit
(config)#
```

7.3.3 Disaster Recovery Behavior Notes

When a disaster recovery switch-over occurs, the Solace JMS API must establish a new connection to the Solace appliances in the standby data center. Because this is a new connection there are some special considerations worth noting. The [Solace-FG] contains the following notes:

> Java and JMS APIs
>
> For client applications using the Java or JMS APIs, any sessions on which the clients have published Guaranteed messages will be destroyed after the switch-over. To indicate the disconnect and loss of publisher flow:

- The Java API will generate an exception from the `JCSMPStreamingPublishCorrelatingEventHandler.handleErrorEx()` that contains a subcode of `JCSMPErrorResponseSubcodeEx.UNKNOWN_FLOW_NAME`.
- The JMS API will generate an exception from the `javax.jms.ExceptionListener` that contains the error code `SolJMSErrorCodes.EC_UNKNOWN_FLOW_NAME_ERROR`.

Upon receiving these exceptions the client application will know to create a new session.

After a new session is established, the client application can republish any Guaranteed messages that had been sent but not acked on the previous session, as these message might not have been persisted and replicated.

To avoid out-of-order messages, the application must maintain an unacked list that is added to before message publish and removed from on receiving an ack from the appliance. If a connection is re-established to a different host in the hostlist, the unacked list must be resent before any new messages are published.

Note: When sending persistent messages using the JMS API, a producer's send message will not return until an acknowledgment is received from the appliance. Once received, it is safe to remove messages from the unacked list.

Alternatively, if the application has a way of determining the last replicated message—perhaps by reading from a last value queue—then the application can use that to determine where to start publishing.

For integration with Spring, it's important to consider this interaction in the context of a Spring `JmsTemplate` and `DefaultMessageListenerContainer`.

7.3.3.1 Receiving Messages via a DefaultMessageListenerContainer

There is no special processing required during a disaster recovery switch-over specifically for applications receiving messages. After successfully reconnecting to the standby site, it is possible that the application will receive some duplicate messages. The application should apply normal duplicate detection handling for these messages.

7.3.3.2 Sending Messages via a JmsTemplate

For Spring applications that are sending messages, there is nothing specifically required to reconnect the Solace JMS connection. However, any messages that were in the process of being sent will receive an error from the Spring Framework and must be retransmitted as possibly duplicated. The error received by the applications will be one of the following two errors:

- org.springframework.jms.IllegalStateException
- org.springframework.jms.UncategorizedJmsException

The following code shows an example of handling these exceptions using the Java producer sample. In practice, application code will already have publisher failure handling in the code and this failure handling code should be called when either of the two JMS exceptions is seen.

```
public static void main(String[] args) throws JMSException {
  ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] { "SolResources.xml" });
  MessageProducer producer = (MessageProducer) context.getBean("messageProducer");
  for (int i = 0; i < 10; i++) {
    try {
      producer.sendMessages();
    } catch (org.springframework.jms.IllegalStateException |
             UncategorizedJmsException |
             JMSException e) {
      // Handle publish failures appropriately.
      e.printStackTrace();
      producer = (MessageProducerForHATests) context.getBean("messageProducerForHA");
    }
  }
  context.close();
}
```

8 Appendix - Configuration and Java Source Reference

8.1 SolResources.xml

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:jms="http://www.springframework.org/schema/jms"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
   http://www.springframework.org/schema/beans/spring-beans.xsd
   http://www.springframework.org/schema/jms
   http://www.springframework.org/schema/jms/spring-jms-3.1.xsd">

  <!-- The section is used to configure all of the JNDI Initial Context information
    to connect in the JMS JNDI Provider -->
  <bean id="solaceJndiTemplate" class="org.springframework.jndi.JndiTemplate"
    lazy-init="default" autowire="default">
    <property name="environment">
      <map>
        <entry key="java.naming.provider.url" value="smf://___IP:PORT___" />
        <entry key="java.naming.factory.initial"
               value="com.solacesystems.jndi.SolJNDIInitialContextFactory" />
        <entry key="java.naming.security.principal"
               value="spring_user@Solace_Spring_VPN" />
 <!-- <entry key="java.naming.security.credentials"
               value="spring_password" /> -->
      </map>
    </property>
  </bean>
  <bean id="solaceConnectionFactory"
class="org.springframework.jndi.JndiObjectFactoryBean"
    lazy-init="default" autowire="default">
    <property name="jndiTemplate" ref="solaceJndiTemplate" />
    <property name="jndiName" value="JNDI/CF/spring" />
  </bean>
  <bean id="solaceCachedConnectionFactory"
    class="org.springframework.jms.connection.CachingConnectionFactory">
    <property name="targetConnectionFactory" ref="solaceConnectionFactory" />
    <property name="sessionCacheSize" value="10" />
  </bean>
```

```
  <bean id="destination" class="org.springframework.jndi.JndiObjectFactoryBean">
    <property name="jndiTemplate" ref="solaceJndiTemplate" />
    <property name="jndiName" value="JNDI/Q/requests" />
  </bean>
  <bean id="jmsTemplate" class="org.springframework.jms.core.JmsTemplate">
    <property name="connectionFactory" ref="solaceCachedConnectionFactory" />
    <property name="defaultDestination" ref="destination" />
    <property name="deliveryPersistent" value="true" />
    <property name="explicitQosEnabled" value="true" />
  </bean>
  <bean id="messageProducer"
        class="com.solacesystems.integration.spring.MessageProducer">
    <property name="jmsTemplate" ref="jmsTemplate" />
  </bean>
  <bean id="messageConsumer"
        class="com.solacesystems.integration.spring.MessageConsumer">
  </bean>
  <bean id="jmsContainer"
    class="org.springframework.jms.listener.DefaultMessageListenerContainer">
    <property name="connectionFactory" ref="solaceCachedConnectionFactory" />
    <property name="destination" ref="destination" />
    <property name="messageListener" ref="messageConsumer" />
  </bean>
</beans>
```

8.2 MessageProducer.java

```
package com.solacesystems.integration.spring;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.context.support.ClassPathXmlApplicationContext;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class MessageProducer {
  private JmsTemplate jmsTemplate;

  public void sendMessages() throws JMSException {
```

```
    getJmsTemplate().send(new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        Message message = session.createTextMessage("test");
        return message;
      }
    });
  }

  public JmsTemplate getJmsTemplate() {
    return jmsTemplate;
  }

  public void setJmsTemplate(JmsTemplate jmsTemplate) {
    this.jmsTemplate = jmsTemplate;
  }

  public static void main(String[] args) throws JMSException {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] { "SolResources.xml" });
    MessageProducer producer = (MessageProducer) context.getBean("messageProducer");
    for (int i = 0; i < 10; i++) {
      producer.sendMessages();
    }
    context.close();
  }
}
```

8.3 MessageConsumer.java

```
package com.solacesystems.integration.spring;

import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.MessageListener;
import javax.jms.TextMessage;

import org.springframework.context.support.ClassPathXmlApplicationContext;

public class MessageConsumer implements MessageListener {

  public void onMessage(Message message) {
```

```
    // Application specific handling code would follow.
    // For this example print the topic of each message
    try {
      System.out.println("Received message on destination: " +
                         message.getJMSDestination().toString());
    } catch (JMSException ex) {
      throw new RuntimeException(ex);
    }
  }

  public static void main(String[] args) {
    ClassPathXmlApplicationContext context = new ClassPathXmlApplicationContext(
        new String[] { "SolResources.xml" });
    MessageConsumer consumer = (MessageConsumer) context.getBean("messageConsumer");
  }
}
```