Spring JMS

Spring JMS

- Spring provides a JMS abstraction framework
  - that simplifies the use of the JMS API
  - shields the user from differences between the JMS 1.0.2 and 1.1 APIs

- JMS can be roughly divided into two areas of functionality
  - production and consumption of messages
- The JmsTemplate class is used for message production and synchronous message reception
- For asynchronous reception similar to Java EE's message-driven bean style, Spring provides a number of message listener containers that are used to create Message-Driven POJOs (MDPs).

JMS Packages

- org.springframework.jms.core
  - provides the core functionality for using JMS
  - It contains JMS template classes that simplifies the use of the JMS by handling the creation and release of resources

JMS Packages

- org.springframework.jms.support
  - provides JMSException translation functionality
  - The translation converts the checked JMSException hierarchy to a mirrored hierarchy of unchecked exceptions

JMS Packages

- **org.springframework.jms.support.converter**
  - provides a MessageConverter abstraction to convert between Java objects and JMS messages

- **org.springframework.jms.support.destination**
  - provides various strategies for managing JMS destinations, such as providing a service locator for destinations stored in JNDI

- **org.springframework.jms.connection**
  - provides an implementation of the ConnectionFactory suitable for use in standalone applications
  - It also contains an implementation of Spring's PlatformTransactionManager for JMS

Using Spring JMS

JmsTemplate

- Two implementations of the JmsTemplate class are provided
  - The class **JmsTemplate** uses the JMS 1.1 API
  - and the subclass **JmsTemplate102** uses the JMS 1.0.2 API

JmsTemplate

- Code that uses the JmsTemplate only needs to implement callback interfaces giving them a clearly defined contract
- The **MessageCreator** callback interface creates a message given a Session provided by the calling code in JmsTemplate

Connections

- The JmsTemplate requires a reference to a ConnectionFactory
- Spring provides an implementation of the ConnectionFactory interface, SingleConnectionFactory, that will return the same Connection on all createConnection calls and ignore calls to close
- This is useful for testing and standalone environments

Destination Management

- Destinations, like ConnectionFactories, are JMS administered objects that can be stored and retrieved in JNDI
- JNDI factory class **JndiObjectFactoryBean** can be used to perform dependency injection on your object's references to JMS destinations
- A **JndiDestinationResolver** is also provided that acts as a service locator for destinations contained in JNDI

Message Listener Containers

- One of the most common uses of JMS messages in the EJB world is to drive message-driven beans (MDBs).
- Spring offers a solution to create message-driven POJOs (MDPs) in a way that does not tie a user to an EJB container

Message Listener Containers

- A subclass of AbstractMessageListenerContainer is used to receive messages from a JMS message queue and drive the MDPs that are injected into it
- The AbstractMessageListenerContainer is responsible for all threading of message reception and dispatch into the MDPs for processing
- A message listener container is the intermediary between an MDP and a messaging provider, and takes care of registering to receive messages, participating in transactions, resource acquisition and release, exception conversion and suchlike.

AbstractMessageListenerContainer

- There are three subclasses of AbstractMessageListenerContainer
  - SimpleMessageListenerContainer
  - DefaultMessageListenerContainer
  - ServerSessionMessageListenerContainer

Sending a Message

```
public class JmsQueueSender {

  private JmsTemplate jmsTemplate;
  private Queue queue;

  public void setConnectionFactory(ConnectionFactory cf) {
    this.jmsTemplate = new JmsTemplate102(cf, false);
  }

  public void setQueue(Queue queue) {
    this.queue = queue;
  }

  public void simpleSend() {
    this.jmsTemplate.send(this.queue, new MessageCreator() {
      public Message createMessage(Session session) throws JMSException {
        return session.createTextMessage("hello queue world");
      }
    });
  }
}
```

Using Message Converters

- In order to facilitate the sending of domain model objects, the JmsTemplate has various send methods that take a Java object as an argument for a message's data content
- The overloaded methods convertAndSend and receiveAndConvert in JmsTemplate delegate the conversion process to an instance of the MessageConverter interface
- This interface defines a simple contract to convert between Java objects and JMS messages
- The default implementation SimpleMessageConverter supports conversion between String and TextMessage, byte[] and BytesMesssage, and java.util.Map and MapMessage
- By using the converter, you and your application code can focus on the business object that is being sent or received via JMS and not be concerned with the details of how it is represented as a JMS message

```
public void sendWithConversion() {
  Map m = new HashMap();
  m.put("Name", "Mark");
  m.put("Age", new Integer(47));
  jmsTemplate.convertAndSend("testQueue", m, new
      MessagePostProcessor() {
    public Message postProcessMessage(Message message) throws
      JMSException {
      message.setIntProperty("AccountID", 1234);
      message.setJMSCorrelationID("123-00001");
      return message;
    }
  });
}
```

- **This results in a message of the form:**

```
MapMessage={
  Header={
    ... standard headers ...
    CorrelationID={123-00001}
  }
  Properties={
    AccountID={Integer:1234}
  }
  Fields={
    Name={String:Mark}
    Age={Integer:47}
  }
}
```

Receiving a message

- Synchronous Reception
  - While JMS is typically associated with asynchronous processing, it is possible to consume messages synchronously
- Asynchronous Reception - Message-Driven POJOs
  - In a fashion similar to a Message-Driven Bean (MDB) in the EJB world, the Message-Driven POJO (MDP) acts as a receiver for JMS messages

MDP

- The one restriction on an MDP is that it must implement the javax.jms.MessageListener interface
- Please also be aware that in the case where your POJO will be receiving messages on multiple threads, it is important to ensure that your implementation is thread-safe

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

public class ExampleListener implements MessageListener {

  public void onMessage(Message message) {
    if (message instanceof TextMessage) {
      try {
        System.out.println(((TextMessage) message).getText());
      } catch (JMSException ex) {
        throw new RuntimeException(ex);
      }
    } else {
      throw new IllegalArgumentException("Message must be of type TextMessage");
    }
  }
}
```

create a message listener container

```
<!-- this is the Message Driven POJO (MDP) -->
<bean id="messageListener" class="jmsexample.ExampleListener" />

<!-- and this is the message listener container -->
<bean id="listenerContainer"
  class="org.springframework.jms.listener.DefaultMessageListenerContainer">
  <property name="concurrentConsumers" value="5"/>
  <property name="connectionFactory" ref="connectionFactory" />
  <property name="destination" ref="destination" />
  <property name="messageListener" ref="messageListener" />
</bean>
```

Spring RMI

Spring RMI

- Spring features integration classes for remoting support using various technologies
- The remoting support eases the development of remote-enabled services, implemented by your usual (Spring) POJOs

Support for RMI

- *Remote Method Invocation (RMI)*
  - Through the use of the **RmiProxyFactoryBean** and the **RmiServiceExporter** Spring supports both traditional RMI (with java.rmi.Remote interfaces and java.rmi.RemoteException) and transparent remoting via RMI invokers

Exposing services using RMI

- **RmiServiceExporter**
- Using the RmiServiceExporter, we can expose the interface of our AccountService object as RMI object
- The interface can be accessed by using RmiProxyFactoryBean, or via plain RMI in case of a traditional RMI service
- The RmiServiceExporter explicitly supports the exposing of any non-RMI services via RMI invokers

The Example classes

```
public class Account implements Serializable{
 private String name;
 public String getName();
 public void setName(String name) { this.name =
    name;
 }
}
```

```
public interface RemoteAccountService extends Remote {

 public void insertAccount(Account account) throws
     RemoteException;

public List getAccounts(String name) throws
     RemoteException;

 }
```

```
public interface AccountService {
public void insertAccount(Account account);
 public List getAccounts(String name);
}
```

- we first have to set up our service in the Spring container

```
<bean id="accountService" class="example.AccountServiceImpl">
<!-- any additional properties, maybe a DAO? -->
</bean>
```

- Next we'll have to expose our service using the RmiServiceExporter:

```
<bean
     class="org.springframework.remoting.rmi.RmiServiceExporter"
     >

 <!-- does not necessarily have to be the same name as the bean
     to be exported -->

 <property name="serviceName" value="AccountService"/>

<property name="service" ref="accountService"/> <property
     name="serviceInterface" value="example.AccountService"/>

<!-- defaults to 1099 --> <property name="registryPort"
     value="1199"/>

</bean>
```

Linking in the service at the client

- Our client is a simple object using the AccountService to manage accounts

```
public class SimpleObject {

private AccountService accountService; public void
    setAccountService(AccountService accountService)
    {

 this.accountService = accountService;

 }

}
```

- To link in the service on the client, we'll create a separate Spring container, containing the simple object and the service linking configuration bits

```
<bean class="example.SimpleObject"> <property
    name="accountService" ref="accountService"/>

 </bean>

 <bean id="accountService"
    class="org.springframework.remoting.rmi.RmiProxyFactoryB
    ean">

 <property name="serviceUrl"
    value="rmi://HOST:1199/AccountService"/>

<property name="serviceInterface"
    value="example.AccountService"/>

 </bean>
```