SJMS

SJMS Component

Available as of Camel 2.11

The Simple JMS Component, or SJMS, is a JMS client for use with Camel that uses well known best practices when it comes to JMS client creation and configuration. SJMS contains a brand new JMS client API written explicitly for Camel eliminating third party messaging implementations keeping it light and resilient. The following features is included:

- Standard Queue and Topic Support (Durable & Non-Durable)
- InOnly & InOut MEP Support
- Asynchronous Producer and Consumer Processing
- Internal JMS Transaction Support

Additional key features include:

- Plugable Connection Resource Management
- Session, Consumer, & Producer Pooling & Caching Management
- Batch Consumers and Producers
- Transacted Batch Consumers & Producers
- Support for Customizable Transaction Commit Strategies (Local JMS Transactions only)

Why the S in SJMS

S stands for Simple and Standard and Springless. Also camel-jms was already taken. 🙂

Maven users will need to add the following dependency to their `pom.xml` for this component:

```
<dependency>
    <groupId>org.apache.camel</groupId>
    <artifactId>camel-sjms</artifactId>
    <version>x.x.x</version>
    <!-- use the same version as your Camel core version -->
</dependency>
```

URI format

```
sjms:[queue:|topic:]destinationName[?options]
```

Where `destinationName` is a JMS queue or topic name. By default, the `destinationName` is interpreted as a queue name. For example, to connect to the queue, `FOO.BAR` use:

```
sjms:FOO.BAR
```

You can include the optional `queue:` prefix, if you prefer:

```
sjms:queue:FOO.BAR
```

To connect to a topic, you *must* include the `topic:` prefix. For example, to connect to the topic, `Stocks.Prices`, use:

```
sjms:topic:Stocks.Prices
```

You append query options to the URI using the following format, `?option=value&option=value&...`

Component Options and Configurations

The SJMS Component supports the following configuration options:

Option	Required	Default Value	Description
`connectionCount`		`1`	The maximum number of connections available to endpoints started under this component
`connectionFactory`	✓	`null`	A ConnectionFactory is required to enable the SjmsComponent. It can be set directly or set set as part of a ConnectionResource.
`connectionResource`		`null`	A ConnectionResource is an interface that allows for customization and container control of the ConnectionFactory. See Plugable Connection Resource Management for further details.
`headerFilterStrategy`		`DefaultJmsKeyFormatStrategy`	
`keyFormatStrategy`		`DefaultJmsKeyFormatStrategy`	**Camel 2.15.x or older**: See option below
`jmsKeyFormatStrategy`		`DefaultJmsKeyFormatStrategy`	**Camel 2.16:** Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: `default` and `passthrough`. The `default` strategy will safely marshal dots and hyphens (`.` and `-`). The `passthrough` strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of the`org.apache.camel.component.jms.JmsKeyFormatStrategy` and refer to it using the # notation.
`transactionCommitStrategy`		`null`	
`DestinationCreationStrategy`		`DefaultDestinationCreationStrategy`	**Camel 2.15.0**: Support to set the custom DestinationCreationStrategy on the SJMS Component.
`messageCreatedStrategy`			**Camel 2.16:** To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms. Message objects when Camel is sending a JMS message.
`completionPredicate`			**Camel 2.18:** The completion predicate, which causes batches to be completed when the predicate evaluates as true. The predicate can also be configured using the simple language using the string syntax. You may want to set the option eagerCheckCompletion to true to let the predicate match the incoming message, as otherwise it matches the aggregated message.
`eagerCheckCompletion`		`false`	**Camel 2.18:** Use eager completion checking which means that the completionPredicate will use the incoming Exchange. As opposed to without eager completion checking the completionPredicate will use the aggregated Exchange.

Below is an example of how to configure the SjmsComponent with its required ConnectionFactory provider. It will create a single connection by default and store it using the components internal pooling APIs to ensure that it is able to service Session creation requests in a thread safe manner.

```
SjmsComponent component = new SjmsComponent();
component.setConnectionFactory(new ActiveMQConnectionFactory("tcp://localhost:61616"));
getContext().addComponent("sjms", component);
```

For a SjmsComponent that is required to support a durable subscription, you can override the default ConnectionFactoryResource instance and set the **clientId** property.

```
ConnectionFactoryResource connectionResource = new ConnectionFactoryResource();
connectionResource.setConnectionFactory(new ActiveMQConnectionFactory("tcp://localhost:61616"));
connectionResource.setClientId("myclient-id");

SjmsComponent component = new SjmsComponent();
component.setConnectionResource(connectionResource);
component.setMaxConnections(1);
```

Producer Configuration Options

The SjmsProducer Endpoint supports the following properties:

Option	Default Value	Description
acknowledgementMode	AUTO_ACKNOWLEDGE	The JMS acknowledgement name, which is one of: SESSION_TRANSACTED, AUTO_ACKNOWLEDGE or DUPS_OK_ACKNOWLEDGE. CLIENT_ACKNOWLEDGE is not supported at this time.
consumerCount	1	**InOut only.** Defines the number of MessageListener instances that for response consumers.
exchangePattern	InOnly	Sets the Producers message exchange pattern.
namedReplyTo	null	**InOut only.** Specifies a named reply to destination for responses.
persistent	true	Whether a message should be delivered with persistence enabled.
producerCount	1	Defines the number of MessageProducer instances.
responseTimeOut	5000	**InOut only.** Specifies the amount of time an InOut Producer will wait for its response.
synchronous	true	Sets whether the Endpoint will use synchronous or asynchronous processing.
transacted	false	If the endpoint should use a JMS Session transaction.
ttl	-1	Disabled by default. Sets the Message time to live header.
prefillPool	true	**Camel 2.14:** Whether to prefill the producer connection pool on startup, or create connections lazy when needed.
allowNullBody	true	**Camel 2.15.1:** Whether to allow sending messages with no body. If this option is false and the message body is null, then an JMSException is thrown.
mapJmsMessage	true	**Camel 2.16:** Specifies whether Camel should auto map the received JMS message to an appropiate payload type, such as javax.jms.TextMessage to a String etc.
messageCreatedStrategy		**Camel 2.16:** To use the given MessageCreatedStrategy which are invoked when Camel creates new instances of javax.jms. Message objects when Camel is sending a JMS message.
jmsKeyFormatStrategy		**Camel 2.16:** Pluggable strategy for encoding and decoding JMS keys so they can be compliant with the JMS specification. Camel provides two implementations out of the box: default and passthrough. The default strategy will safely marshal dots and hyphens (. and -). The passthrough strategy leaves the key as is. Can be used for JMS brokers which do not care whether JMS header keys contain illegal characters. You can provide your own implementation of theorg.apache.camel.component.jms.JmsKeyFormatStrategy and refer to it using the # notation.
includeAllJMSXProperties		**Camel 2.16:** Whether to include all JMSXxxx properties when mapping from JMS to Camel Message. Setting this to true will include properties such as JMSXAppID, and JMSXUserID etc. **Note:** If you are using a custom headerFilterStrategy then this option does not apply.

Producer Usage

InOnly Producer - (Default)

The InOnly Producer is the default behavior of the SJMS Producer Endpoint.

```
from("direct:start")
    .to("sjms:queue:bar");
```

InOut Producer

To enable InOut behavior append the exchangePattern attribute to the URI. By default it will use a dedicated TemporaryQueue for each consumer.

```
from("direct:start")
    .to("sjms:queue:bar?exchangePattern=InOut");
```

You can specify a `namedReplyTo` though which can provide a better monitor point.

```
from("direct:start")
    .to("sjms:queue:bar?exchangePattern=InOut&namedReplyTo=my.reply.to.queue");
```

Consumers Configuration Options

The SjmsConsumer Endpoint supports the following properties:

Option	Default Value	Description
`acknowledgementMode`	`AUTO_ACKNOWLEDGE`	The JMS acknowledgement name, which is one of: `TRANSACTED`, `AUTO_ACKNOWLEDGE` or `DUPS_OK_ACKNOWLEDGE`. `CLIENT_ACKNOWLEDGE` is not supported at this time.
`consumerCount`	`1`	Defines the number of MessageListener instances.
`durableSubscriptionId`	`null`	Required for a durable subscriptions.
`exchangePattern`	`InOnly`	Sets the Consumers message exchange pattern.
`messageSelector`	`null`	Sets the message selector.
`synchronous`	`true`	Sets whether the Endpoint will use synchronous or asynchronous processing.
`transacted`	`false`	If the endpoint should use a JMS Session transaction.
`transactionBatchCount`	`1`	The number of exchanges to process before committing a local JMS transaction. The `transacted` property must also be set to true or this property will be ignored.
`transactionBatchTimeout`	`5000`	The amount of time a the transaction will stay open between messages before committing what has already been consumed. Minimum value is 1000ms.
`ttl`	`-1`	Disabled by default. Sets the Message time to live header.
`asyncStartListener`	`false`	Whether to startup the consumer message listener asynchronously, when starting a route. For example if a JmsConsumer cannot get a connection to a remote JMS broker, then it may block while retrying and/or failover. This will cause Camel to block while starting routes. By setting this option to true, you will let routes startup, while the JmsConsumer connects to the JMS broker using a dedicated thread in asynchronous mode. If this option is used, then beware that if the connection could not be established, then an exception is logged at WARN level, and the consumer will not be able to receive messages; You can then restart the route to retry.
`asyncStopListener`	`false`	Whether to stop the consumer message listener asynchronously, when stopping a route.

Consumer Usage

InOnly Consumer - (Default)

The InOnly Consumer is the default Exchange behavior of the SJMS Consumer Endpoint.

```
from("sjms:queue:bar")
    .to("mock:result");
```

InOut Consumer

To enable InOut behavior append the `exchangePattern` attribute to the URI.

```
from("sjms:queue:in.out.test?exchangePattern=InOut")
    .transform(constant("Bye Camel"));
```

Advanced Usage Notes

Plugable Connection Resource Management

SJMS provides JMS Connection resource management through built-in connection pooling. This eliminates the need to depend on third party API pooling logic. However there may be times that you are required to use an external Connection resource manager such as those provided by J2EE or OSGi containers. For this SJMS provides an interface that can be used to override the internal SJMS Connection pooling capabilities. This is accomplished through the ConnectionResource interface.

The ConnectionResource provides methods for borrowing and returning Connections as needed is the contract used to provide Connection pools to the SJMS component. A user should use when it is necessary to integrate SJMS with an external connection pooling manager.

It is recommended though that for standard ConnectionFactory providers you use the ConnectionFactoryResource implementation that is provided with SJMS as-is or extend as it is optimized for this component.

Below is an example of using the pluggable ConnectionResource with the ActiveMQ PooledConnectionFactory:

```
public class AMQConnectionResource implements ConnectionResource {
    private PooledConnectionFactory pcf;

    public AMQConnectionResource(String connectString, int maxConnections) {
        super();
        pcf = new PooledConnectionFactory(connectString);
        pcf.setMaxConnections(maxConnections);
        pcf.start();
    }

    public void stop() {
        pcf.stop();
    }

    @Override
    public Connection borrowConnection() throws Exception {
        Connection answer = pcf.createConnection();
        answer.start();
        return answer;
    }

    @Override
    public Connection borrowConnection(long timeout) throws Exception {
        // SNIPPED...
    }

    @Override
    public void returnConnection(Connection connection) throws Exception {
        // Do nothing since there isn't a way to return a Connection
        // to the instance of PooledConnectionFactory
        log.info("Connection returned");
    }
}
```

Then pass in the ConnectionResource to the SjmsComponent:

```
CamelContext camelContext = new DefaultCamelContext();
AMQConnectionResource pool = new AMQConnectionResource("tcp://localhost:33333", 1);
SjmsComponent component = new SjmsComponent();
component.setConnectionResource(pool);
camelContext.addComponent("sjms", component);
```

To see the full example of its usage please refer to the ConnectionResourceIT.

Session, Consumer, & Producer Pooling & Caching Management

Coming soon ...

Batch Message Support

The SjmsProducer supports publishing a collection of messages by creating an Exchange that encapsulates a List. This SjmsProducer will take then iterate through the contents of the List and publish each message individually.

If when producing a batch of messages there is the need to set headers that are unique to each message you can use the SJMS BatchMessage class. When the SjmsProducer encounters a BatchMessage List it will iterate each BatchMessage and publish the included payload and headers.

Below is an example of using the BatchMessage class. First we create a List of BatchMessages:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Then publish the List:

```
template.sendBody("sjms:queue:batch.queue", messages);
```

Customizable Transaction Commit Strategies (Local JMS Transactions only)

SJMS provides a developer the means to create a custom and plugable transaction strategy through the use of the TransactionCommitStrategy interface. This allows a user to define a unique set of circumstances that the SessionTransactionSynchronization will use to determine when to commit the Session. An example of its use is the BatchTransactionCommitStrategy which is detailed further in the next section.

Transacted Batch Consumers & Producers

The SjmsComponent has been designed to support the batching of local JMS transactions on both the Producer and Consumer endpoints. How they are handled on each is very different though.

The SjmsConsumer endpoint is a straitforward implementation that will process X messages before committing them with the associated Session. To enable batched transaction on the consumer first enable transactions by setting the `transacted` parameter to true and then adding the `transactionBatchCount` and setting it to any value that is greater than 0. For example the following configuration will commit the Session every 10 messages:

```
sjms:queue:transacted.batch.consumer?transacted=true&transactionBatchCount=10
```

If an exception occurs during the processing of a batch on the consumer endpoint, the Session rollback is invoked causing the messages to be redelivered to the next available consumer. The counter is also reset to 0 for the BatchTransactionCommitStrategy for the associated Session as well. It is the responsibility of the user to ensure they put hooks in their processors of batch messages to watch for messages with the JMSRedelivered header set to true. This is the indicator that messages were rolled back at some point and that a verification of a successful processing should occur.

A transacted batch consumer also carries with it an instance of an internal timer that waits a default amount of time (5000ms) between messages before committing the open transactions on the Session. The default value of 5000ms (minimum of 1000ms) should be adequate for most use-cases but if further tuning is necessary simply set the `transactionBatchTimeout` parameter.

```
sjms:queue:transacted.batch.consumer?transacted=true&transactionBatchCount=10&transactionBatchTimeout=2000
```

The minimal value that will be accepted is 1000ms as the amount of context switching may cause unnecessary performance impacts without gaining benefit.

The producer endpoint is handled much differently though. With the producer after each message is delivered to its destination the Exchange is closed and there is no longer a reference to that message. To make a available all the messages available for redelivery you simply enable transactions on a Producer Endpoint that is publishing BatchMessages. The transaction will commit at the conclusion of the exchange which includes all messages in the batch list. Nothing additional need be configured. For example:

```
List<BatchMessage<String>> messages = new ArrayList<BatchMessage<String>>();
for (int i = 1; i <= messageCount; i++) {
    String body = "Hello World " + i;
    BatchMessage<String> message = new BatchMessage<String>(body, null);
    messages.add(message);
}
```

Now publish the List with transactions enabled:

```
template.sendBody("sjms:queue:batch.queue?transacted=true", messages);
```

Additional Notes

Message Header Format

The SJMS Component uses the same header format strategy that is used in the Camel JMS Component. This plugable strategy ensures that messages sent over the wire conform to the JMS Message spec.

For the exchange.in.header the following rules apply for the header keys:

Keys starting with JMS or JMSX are reserved.
exchange.in.headers keys must be literals and all be valid Java identifiers (do not use dots in the key name).
Camel replaces dots & hyphens and the reverse when when consuming JMS messages:

- is replaced by *DOT* and the reverse replacement when Camel consumes the message.
- is replaced by *HYPHEN* and the reverse replacement when Camel consumes the message.
  See also the option jmsKeyFormatStrategy, which allows use of your own custom strategy for formatting keys.

For the exchange.in.header, the following rules apply for the header values:

Message Content

To deliver content over the wire we must ensure that the body of the message that is being delivered adheres to the JMS Message Specification. Therefore, all that are produced must either be primitives or their counter objects (such as Integer, Long, Character). The types, String, CharSequence, Date, BigDecimal and BigInteger are all converted to their toString() representation. All other types are dropped.

Clustering

When using InOut with SJMS in a clustered environment you must either use TemporaryQueue destinations or use a unique named reply to destination per InOut producer endpoint. Message correlation is handled by the endpoint, not with message selectors at the broker. The InOut Producer Endpoint uses Java Concurrency Exchangers cached by the Message JMSCorrelationID. This provides a nice performance increase while reducing the overhead on the broker since all the messages are consumed from the destination in the order they are produced by the interested consumer.

Currently the only correlation strategy is to use the JMSCorrelationId. The InOut Consumer uses this strategy as well ensuring that all responses messages to the included JMSReplyTo destination also have the JMSCorrelationId copied from the request as well.

Transaction Support

SJMS currently only supports the use of internal JMS Transactions. There is no support for the Camel Transaction Processor or the Java Transaction API (JTA).

Does Springless Mean I Can't Use Spring?

Not at all. Below is an example of the SJMS component using the Spring DSL:

```
<route
    id="inout.named.reply.to.producer.route">
    <from
        uri="direct:invoke.named.reply.to.queue" />
    <to
        uri="sjms:queue:named.reply.to.queue?namedReplyTo=my.response.queue&amp;exchangePattern=InOut" />
</route>
```

Springless refers to moving away from the dependency on the Spring JMS API. A new JMS client API is being developed from the ground up to power SJMS.