Getting started with Habari ActiveMQ Client

Michael Justin

A short guide for the first steps with the JMS client library

Trademarks

Contents

Introduction

About Habari ActiveMQ Client

Habari ActiveMQ Client is a Delphi library for Apache ActiveMQ, the most popular and powerful open source Message Broker. With Habari ActiveMQ Client, Delphi developers can build integrated solutions, connecting cross language clients and protocols from Java, C, C++, C#, Ruby, Perl, Python, and PHP, using the peer-to-peer or the publish and subscribe communication model. The library uses the Stomp message protocol and a plug-in architecture for communication libraries (including SSL) and message transformers for XML and JSON object serialization. It supports Apache ActiveMQ versions 4.0 to 5.2, Delphi 6 to 2009 and Free Pascal, and follows the specification of the JMS API for Message Oriented Middleware.

How Can I Use It?

Here are some examples for software solutions built on top of a Message Broker like Apache ActiveMQ:

- **Intranet News Ticker Application**: using the publish and subscribe communication model, news can be delivered to all registered client applications. The message sender works like a broadcast station, and does not care if clients don't listen.
- **Load Balancing**: using the point-to-point or queuing model, many 'worker' applications can be installed on different computers. Every new message sent to the queue will be delivered only to one client. The server will keep messages until they are expired or delivered to a client.
- **Persistent Storage**: messages and objects can be stored in the Object Broker and retrieved even after a restart.
- **Interprocess Communication**: applications can use point-to-point messages to exchange information between each other even if the receiver currently is not running.

About Apache ActiveMQ

Apache ActiveMQ is the most popular and powerful open source Message Broker and Enterprise Integration Patterns provider. Apache ActiveMQ is fast, supports many Cross Language Clients and Protocols, comes with easy to use Enterprise

Integration Patterns and many advanced features while fully supporting JMS 1.1 and J2EE 1.4.

Apache ActiveMQ Features[1]

- Supports a variety of Cross Language Clients and Protocols from Java, C, C++, C#, Ruby, Perl, Python, PHP
    - OpenWire for high performance clients in Java, C, C++, C#
    - Stomp support so that clients can be written easily in C, Ruby, Perl, Python, PHP, ActionScript/Flash, Smalltalk to talk to ActiveMQ as well as any other popular Message Broker
- full support for the Enterprise Integration Patterns both in the JMS client and the Message Broker
- Supports many advanced features such as Message Groups, Virtual Destinations, Wildcards and Composite Destinations
- Fully supports JMS 1.1 and J2EE 1.4 with support for transient, persistent, transactional and XA messaging
- Spring Support so that ActiveMQ can be easily embedded into Spring applications and configured using Spring's XML configuration mechanism
- Tested inside popular J2EE servers such as Geronimo, JBoss 4, GlassFish and WebLogic
    - Includes JCA 1.5 resource adaptors for inbound & outbound messaging so that ActiveMQ should auto-deploy in any J2EE 1.4 compliant server
- Supports pluggable transport protocols such as in-VM, TCP, SSL, NIO, UDP, multicast, JGroups and JXTA transports
- Supports very fast persistence using JDBC along with a high performance journal
- Designed for high performance clustering, client-server, peer based communication
- REST API to provide technology agnostic and language neutral web based API to messaging
- Ajax to support web streaming support to web browsers using pure DHTML, allowing web browsers to be part of the messaging fabric
- CXF and Axis Support so that ActiveMQ can be easily dropped into either of these web service stacks to provide reliable messaging
- Can be used as an in memory JMS provider, ideal for unit testing JMS

---

[1] http://activemq.apache.org/index.html

Habari ActiveMQ Client License

This copyright applies to all source code, compiled code, documentation, graphics and auxiliary files, except those parts written by other people (which are normally copyright their authors).

GENERAL TERMS THAT APPLY TO COMPILED PROGRAMS AND REDISTRIBUTABLES

You may write and compile your own application programs using the library. You may reproduce and distribute, in executable form only, programs which you create using the library without additional license or fees, subject to all of the conditions in this statement.

The license granted in this statement for you to create your own compiled programs and distribute your programs and the Redistributables (if any) is subject to all of the following conditions: (i) all copies of the programs you create must bear a valid copyright notice, either your own or the betasoft copyright notice that appears on the Software; (ii) you may not remove or alter any betasoft copyright, trademark or other proprietary rights notice contained in any portion of betasoft libraries, source code, Redistributables or other files that bear such a notice; (iii) betasoft provides no warranty at all to any person, other than the Limited Warranty provided to the original purchaser of the Software, and you will remain solely responsible to anyone receiving your programs for support, service, upgrades, or technical or other assistance, and such recipients will have no right to contact betasoft for such services or assistance; (iv) you will indemnify and hold betasoft, its related companies and its suppliers, harmless from and against any claims or liabilities arising out of the use, reproduction or distribution of your programs; (v) your programs must be written using a licensed, registered copy of the Software; (vi) your programs must add primary and substantial functionality, and may not be merely a set or subset of any of the libraries (including runtime libraries), code, Redistributables or other files of the Software; (vii) regardless of any modifications which you make and regardless of how you might compile, link, or package your programs, the libraries (including runtime libraries), code, Redistributables, and/or other files of the Software (including any portions thereof) may not be used in programs created by your end users (i.e., users of your programs) and may not be further redistributed by your end users; and (viii) you may not use betasoft's or any of its suppliers' names, logos, or trademarks to market your programs, except to state that your program was written using the Software.

All betasoft libraries, source code, Redistributables and other files remain betasoft's exclusive property. Regardless of any modifications that you make, you may not distribute any files (particularly betasoft source code and other non-executable files).

LIMITED WARRANTY

No warranty of any sort, expressed or implied, is provided in connection with the library, including, but not limited to, implied warranties of merchantibility or fitness for a particular purpose. Any cost, loss or damage of any sort incurred owing to the malfunction or misuse of the library or the inaccuracy of the documentation or connected with the library in any other way whatsoever is solely the responsibility of the person who incurred the cost, loss or damage. Furthermore, any illegal use of the library is solely the responsibility of the person committing the illegal act. By using this program you accept these responsibilities, and give up any right to seek any damages against the authors in connection with this program.

Third Party Library Licenses

Indy BSD License

Copyright

Portions of this software are Copyright (c) 1993 - 2003, Chad Z. Hower (Kudzu) and the Indy Pit Crew - http://www.IndyProject.org/

License

Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

- Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
- Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation, about box and/or other materials provided with the distribution.
- No personal names or organizations names associated with the Indy project may be used to endorse or promote products derived from this software without specific prior written permission of the specific individual or organization.

THIS SOFTWARE IS PROVIDED BY Chad Z. Hower (Kudzu) and the Indy Pit Crew "AS IS'' AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN NO EVENT SHALL THE REGENTS OR CONTRIBUTORS BE LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.

lkJSON

```
  LkJSON v1.05

  25 jan 2009

* Copyright (c) 2006,2007,2008,2009 Leonid Koninin
* leon_kon@users.sourceforge.net
* All rights reserved.
*
* Redistribution and use in source and binary forms, with or without
* modification, are permitted provided that the following conditions are met:
*     * Redistributions of source code must retain the above copyright
*       notice, this list of conditions and the following disclaimer.
*     * Redistributions in binary form must reproduce the above copyright
*       notice, this list of conditions and the following disclaimer in the
*       documentation and/or other materials provided with the distribution.
*     * Neither the name of the <organization> nor the
*       names of its contributors may be used to endorse or promote products
*       derived from this software without specific prior written permission.
*
* THIS SOFTWARE IS PROVIDED BY Leonid Koninin ``AS IS'' AND ANY
* EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED
* WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE
* DISCLAIMED. IN NO EVENT SHALL Leonid Koninin BE LIABLE FOR ANY
* DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
* (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
* LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
* ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
* (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF THIS
* SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
```

SuperObject

```
 *                         Super Object Toolkit
 *
 * The contents of this file are subject to the Mozilla Public License Version
 * 1.1 (the "License"); you may not use this file except in compliance with the
 * License. You may obtain a copy of the License at http://www.mozilla.org/MPL
 *
 * Software distributed under the License is distributed on an "AS IS" basis,
 * WITHOUT WARRANTY OF ANY KIND, either express or implied. See the License for
 * the specific language governing rights and limitations under the License.
 *
 * Unit owner : Henri Gourvest <hgourvest@progdigy.com>
 *
 * This unit is inspired from the json c lib:
 *   Michael Clark <michael@metaparadigm.com>
 *   http://oss.metaparadigm.com/json-c/
```

Installation

Requirements

Development Environment

- CodeGear Delphi 6 or higher,

or

- Free Pascal

Message Broker

- Apache ActiveMQ 4 or higher
- IONA FUSE Message Broker

TCP/IP Communication Library

Recommended libraries:

- Internet Direct (Indy)
- Synapse

See the next chapter for a discussion of all communication libraries and a feature matrix.

JSON Serialization Library

A JSON library is required for JSON transformation used in object exchange.

- lkJSON - BSD licensed and (c) 2006,2007,2008 Leonid Koninin
- SuperObject - licensed under MPL 1.1 and (c) Henri Gourvest

This JSON library is still included but deprecated and unsupported

- JSON Toolkit - licensed under MPL 1.1 and (c) Henri Gourvest

SOAP Serialization Library

Sending and receiving of objects with SOAP requires Delphi 7 or higher.[2]

Sending and receiving of objects in Free Pascal requires the Web Service Toolkit or binary serialization.

TCP/IP Communication Libraries

Supported libraries

Internet Direct (Indy) 10

The communication adapter for Indy supports both GUI-based and console mode applications, and works with Delphi 6 to 2009 and Free Pascal.

The library has been tested with these versions of Internet Direct:

- Indy 10.2.3
- Indy 10.5.5 (Tiburon branch)

Synapse

The communication adapter for Synapse supports both GUI-based and console mode applications, and works with Delphi 6 to 2009 and Free Pascal.

The library has been tested with these versions of Synapse:

- V 38
- V 95 (with Delphi 2009 support)

---

[2] The library is designed to use new methods that were added to TRemotable (ObjectToSOAP and SOAPToObject.).

Communication Adapter Feature Matrix

	Features	Indy 10	Synapse
DELPHI	Send/receive **text** messages in GUI applications	Yes/Yes	Yes/Yes
	Send/receive **binary** messages in GUI applications	Yes/Yes	Yes/Yes
	Send/receive **text** messages in CONSOLE applications	Yes/Yes	Yes/Yes
	Send/receive **binary** messages in CONSOLE applications	Yes/Yes	Yes/Yes
FREE PASCAL	Send/receive **text** messages in GUI applications	Yes/Yes	Yes/Yes
	Send/receive **binary** messages in GUI applications	Yes/Yes	Yes/Yes
	Send/receive **text** messages in CONSOLE applications	Yes/Yes	Yes/Yes
	Send/receive **binary** messages in CONSOLE applications	Yes/Yes	Yes/Yes

Upgrades

If you upgrade from older versions, make a backup of your existing version and make sure that you also have a backup of your own source.

HabariExpress Example Components

Note to users of the HabariExpress example component: if you upgrade from old versions, component properties may have changed and this could cause error messages when you open existing projects with the new version installed.

Demo Source Code

The Delphi demo applications have been built using Delphi 6 and Delphi 2009, in some cases using third party libraries like JCL and TMS Unicode.

Delphi form files (DFM) are not backwards compatible, so opening them in previous versions of Delphi might fail.

Supported Message Brokers

Apache ActiveMQ

ActiveMQ supports the Stomp protocol and the Stomp – JMS mapping. This makes it easy to write a client in pure Ruby, Perl, Python or PHP for working with ActiveMQ.

The Habari ActiveMQ Client library uses the Stomp protocol, so the ActiveMQ Stomp connector has to be enabled.

In the Apache ActiveMQ default configuration, support for Stomp is already enabled.

Standard JMS message brokers

Connections to other JMS message brokers are possible (but not yet tested) using the StompConnect library available from Codehaus.

Starting ActiveMQ

Pre-Installation Requirements[3]

Hardware:

- 40 MB of free disk space for the ActiveMQ binary distribution.
- 200 MB of free disk space for the ActiveMQ source or developer's distributions.

Operating Systems:

- Windows: Windows XP SP2, Windows 2000.
- Unix: Ubuntu Linux, Powerdog Linux, MacOS, AIX, HP-UX, Solaris, or any Unix platform that supports Java.

Environment:

- Java Developer Kit (JDK) 1.5.x or greater for deployment and 1.5.x (Java 5) for compiling/building.
- The JAVA_HOME environment variable must be set to the directory where the JDK is installed, e.g., `c:\Program Files\jdk.1.5.0_07-87`.

Download the binary distribution

After downloading from http://activemq.apache.org/download.html and unpacking ActiveMQ, you are ready to start the messages broker.

Running the Broker

From the binary distribution you can run the Apache ActiveMQ server pretty easily via the bin/activemq command. e.g. from a shell type

```
cd bin
activemq
```

The Apache ActiveMQ broker should now have started.

---

[3] http://activemq.apache.org/version-5-getting-started.html

Monitoring ActiveMQ

There are various ways to monitor ActiveMQ. If you are on version 4.2 or later of ActiveMQ you can then monitor ActiveMQ using the Web Console by pointing your browser at

http://localhost:8161/admin

Or you can use the JMX support to view the running state of ActiveMQ.

Stopping ActiveMQ

For both Windows and Unix installations, terminate ActiveMQ by typing "CTRL-C" in the console or command shell in which it is running.

ActiveMQ Authentication Configuration

If you have modest authentication requirements (or just want to quickly set up your testing environment) you can use SimpleAuthenticationPlugin.

With this plugin you can define users and groups directly in the broker's XML configuration.[4]

Take a look at the following snippet for example:

---

[4] For more information see http://activemq.apache.org/security.html

```
<broker xmlns="http://activemq.apache.org/schema/core"
brokerName="localhost" dataDirectory="${activemq.base}/data">

  ...

  <plugins>

    <simpleAuthenticationPlugin>
      <users>
        <authenticationUser username="system" password="manager"
                            groups="users,admins"/>
        <authenticationUser username="user" password="password"
                        groups="users"/>
        <authenticationUser username="guest" password="password"
groups="guests"/>
      </users>
    </simpleAuthenticationPlugin>

  </plugins>

</broker>
```

Caveat:	The default activemq.xml configuration file comes with three optional and enabled elements: <commandAgent>, <camelContext>, and <jetty>. If you enable authentication & authorization services, these enabled elements will cause the broker to throw security-related exceptions. This is because these elements represent functionality that is essentially represented by clients that need to connect to the broker and the connections are made without security credentials. If you do not require the functionality behind these elements, disable or comment-out the elements.
ActiveMQ 5.1	The ActiveMQ Stomp connector supports password authentication only in versions since version 5.1.

Communication Adapter Configuration

Introduction

Habari uses communication adapters as an abstraction layer between the internal library and the TCP/IP library. These adapters are implemented using a common API, which allows to exchange them easily, even at runtime.

Installation of Communication Adapter classes

A communication adapter implementation can be prepared for usage by simply adding its Delphi unit to the project. Behind the scenes, the communication adapter will add itself to the communication adapter list in the BTAdapterRegistry unit. If more than one communication adapter is in the project, the first adapter class in the list will be the default adapter. (The methods of the adapter registry performs some checks, for example to prevent duplicate entries in the adapter list, and raise exceptions in case of errors)

No additional setup of communication adapters is required. At run time, the JMS connection class will pick the default adapter from this list.

The default adapter can be changed at runtime by setting the adapter class (either by its name or by its type).

Available Communication Adapters

The Habari ActiveMQ Client libraries includes two adapters for TCP/IP libraries, one for Indy (Internet Direct) and one for Synapse.

Indy (Internet Direct)

The Indy adapter requires Indy 10.5.5 for Delphi 2009 and Indy 10.2.3 for previous versions of Delphi.

Synapse

The Synapse adapter requires Synapse V 95 for Delphi 2009 and V 38 for previous version of Delphi.

Connections and Sessions

Step by Step Example

Add required units

Three units are required for this example

- a communication adapter unit (e.g. BTCommAdapterIndy)
- a connection factory unit (BTJMSConnectionFactory or BTJMSConnection)
- the unit containing the interface declarations (BTJMSInterfaces)

The SysUtils unit is necessary for the exception handling.

```
program SendOneMessage;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  BTCommAdapterIndy in '..\..\source\BTCommAdapterIndy.pas',
  BTJMSConnection in '..\..\source\BTJMSConnection.pas',
  BTJMSInterfaces in '..\..\source\BTJMSInterfaces.pas';
...
```

Creating a new Connection

To create a new connection,

- declare a variable of type IConnection
- use the helper method MakeConnection of the TBTJMSConnection class to create and configure a new connection with user name, password and the broker URL

or

- use an instance of TBTJMSConnectionFactory to create connections

Since IConnection is an interface type, the connection instance will be destroyed automatically if there are no more references to it in the program. Note that there is no call to Connection.Free in the source.

```
var
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
begin
  Connection := TBTJMSConnection.MakeConnection('', '', 'stomp://localhost');
  Connection.Start;
```

Local connection

If you just need a connection to the broker on the local computer using default port number and login credentials, you can call MakeConnection without parameters:

```
  Connection := TBTJMSConnection.MakeConnection;
```

Creating a Session

To create the communication session,

- declare a variable of type ISession
- use the helper method CreateSession of the connection, and specify if it is a transacted session, and the acknowledgement mode

Please check the API documentation for the different session types and acknowledgement modes.

Since ISession is an interface type, the session instance will be destroyed automatically if there are no more references to it in the program. Note that there is no call to Session.Free in the source.

```
  try
    Session := Connection.CreateSession(False, amAutoAcknowledge);
```

Using the Session

The Session variable is ready to use now. Destinations, producers and consumers will be covered in the next chapters.

```
    Destination := Session.CreateQueue('testqueue');
    Producer := Session.CreateProducer(Destination);
    Producer.Send(Session.CreateTextMessage('This is a test message'));
```

Closing a Connection

Finally, the application closes the connection. The client will disconnect from the message broker. Closing a connection also implicitly closes all open sessions.

```
  finally
    Connection.Close;
  end;
end.
```

Transacted Sessions

A session may be specified as transacted. Each transacted session supports a single series of transactions. Each transaction groups a set of message sends and a set of message receives into an atomic unit of work. In effect, transactions organize a session's input message stream and output message stream into series of atomic units. When a transaction commits, its atomic unit of input is acknowledged and its associated atomic unit of output is sent. If a transaction rollback is done, the transaction's sent messages are destroyed and the session's input is automatically recovered.

The content of a transaction's input and output units is simply those messages that have been produced and consumed within the session's current transaction.

A transaction is completed using either its session's Commit method or its session's Rollback method. The completion of a session's current transaction automatically begins the next. The result is that a transacted session always has a current transaction within which its work is done.

Destinations

Introduction

The JMS API supports two models:[5]

1. point-to-point or queuing model
2. publish and subscribe model

In the point-to-point or queuing model, a producer posts messages to a particular queue and a consumer reads messages from the queue. Here, the producer knows the destination of the message and posts the message directly to the consumer's queue. It is characterized by following:

- Only one consumer will get the message
- The producer does not have to be running at the time the receiver consumes the message, nor does the receiver need to be running at the time the message is sent
- Every message successfully processed is acknowledged by the receiver

The publish/subscribe model supports publishing messages to a particular message topic. Zero or more subscribers may register interest in receiving messages on a particular message topic. In this model, neither the publisher nor the subscriber know about each other. A good metaphor for it is anonymous bulletin board. The following are characteristics of this model:

- Multiple consumers can get the message
- There is a timing dependency between publishers and subscribers. The publisher has to create a subscription in order for clients to be able to subscribe. The subscriber has to remain continuously active to receive messages, unless it has established a durable subscription. In that case, messages published while the subscriber is not connected will be redistributed whenever it reconnects.

---

[5] Java Message Service. (2007, November 21). In Wikipedia, The Free Encyclopedia. http://en.wikipedia.org/wiki/Java_Message_Service

Create a new Destination

Queues

A queue can be created using the CreateQueue method of the Session. Example:

```
Destination := Session.CreateQueue('foo');
Consumer := Session.CreateConsumer(Destination);
```

The queue can then be used to send or receive messages using implementations of the IMessageProducer and IMessageConsumer interfaces. (See next chapter for an example)

Topics

A topic can be created using the CreateTopic method of the Session. Example:

```
Destination := Session.CreateTopic('bar');
Consumer := Session.CreateConsumer(Destination);
```

The topic can then be used to send or receive messages using implementations of the IMessageProducer and IMessageConsumer interfaces. (See next chapter for an example).

Destination Options

Destination Options are a way to provide extended configuration options to a JMS consumer without having to extend the JMS API. The options are encoded using URL query syntax in the destination name that the consumer is created on.[6]

Example:

```
Destination := Session.CreateQueue('foo?activemq.retroactive=true');
Consumer := Session.CreateConsumer(Destination);
```

activemq.dispatchAsync (boolean)

Should messages be dispatched synchronously or asynchronously from the producer thread for non-durable topics in the broker? For fast consumers set this to **false**. For slow consumers set it to **true** so that dispatching will not block fast consumers.

activemq.exclusive (boolean)

I would like to be an Exclusive Consumer on the queue.[7]

activemq.maximumPendingMessageLimit (int)

For Slow Consumer Handling on non-durable topics by dropping old messages - we can set a maximum-pending limit, such that once a slow consumer backs up to this high water mark we begin to discard old messages.[8]

activemq.prefetchSize (int)

Specifies the maximum number of pending messages that will be dispatched to the client. Once this maximum is reached no more messages are dispatched until the client acknowledges a message. Set to **1** for very fair distribution of messages across consumers where processing messages can be slow.

activemq.priority (byte)

Sets the priority of the consumer so that dispatching can be weighted in priority order.

---

[6] http://activemq.apache.org/destination-options.html

[7] http://activemq.apache.org/exclusive-consumer.html

[8] http://activemq.apache.org/slow-consumer-handling.html

activemq.retroactive (boolean)

A retroactive consumer is just a regular JMS consumer who indicates that at the start of a subscription every attempt should be used to go back in time and send any old messages (or the last message sent on that topic) that the consumer may have missed.[9]

---

[9] http://activemq.apache.org/retroactive-consumer.html

Producer and Consumer

Message Producer

A client uses a MessageProducer object to send messages to a destination. A MessageProducer object is created by passing a Destination object to a message-producer creation method supplied by a session.

Example:

```
...
Destination := Session.CreateQueue('foo');
Producer := Session.CreateProducer(Destination);
Producer.Send(Session.CreateTextMessage('Test message'));
...
```

A client can specify a default delivery mode, priority, and time to live for messages sent by a message producer. It can also specify the delivery mode, priority, and time to live for an individual message.

Message Consumer

A client uses a MessageConsumer object to receive messages from a destination. A MessageConsumer object is created by passing a Destination object to a message-consumer creation method supplied by a session.

Example:

```
...
Destination := Session.CreateQueue('foo');
Consumer := Session.CreateConsumer(Destination);
Consumer.MessageListener := Self;
...
```

A message consumer can be created with a message selector. A message selector allows the client to restrict the messages delivered to the message consumer to those that match the selector.

A client may either synchronously receive a message consumer's messages or have the consumer asynchronously deliver them as they arrive.

For synchronous receipt, a client can request the next message from a message consumer using one of its receive methods. There are several variations of receive that allow a client to poll or wait for the next message.

For asynchronous delivery, a client can register a MessageListener object with a message consumer. As messages arrive at the message consumer, it delivers them by calling the MessageListener's OnMessage method.

JMS Selectors

Selectors are a way of attaching a filter to a subscription to perform content based routing. Selectors are defined using SQL 92 syntax and typically apply to message headers; whether the standard properties available on a JMS message or custom headers you can add via the JMS code.[10]

Here is an example

```
JMSType = 'car' AND color = 'blue' AND weight > 2500
```

For more documentation on the detail of selectors see the reference on javax.jmx.Message[11].

ActiveMQ supports some JMS defined properties, as well as some ActiveMQ ones that the selector can use.

Note	The Stomp protocol used by Habari ActiveMQ Client only supports string type properties and operations in selectors.

Delphi example:

```
...
MessageConsumer := Session.CreateConsumer(Destination, 'foo = ''bar''');
...
```

Using XPath to filter messages

Apache ActiveMQ also supports XPath based selectors when working with messages containing XML bodies. To use an XPath selector use the following syntax

---

[10] See http://activemq.apache.org/selectors.html

[11] See http://java.sun.com/j2ee/1.4/docs/api/javax/jms/Message.html

```
XPATH '//title[@lang=''eng'']'
```

Note The standard installation of ActiveMQ does not include the Xalan JAR files which are necessary for XPATH evaluation. The files xalan.jar, xercesImpl.jar and xml-apis.jar need to be placed in the lib folder of ActiveMQ.

Delphi example:

```
...
MessageConsumer := Session.CreateConsumer(Destination,
  'XPATH ''//title[@lang="en"]''');
...
```

Text Messages

Sending a TextMessage

Source code for a simple application which sends a test message:

```
program SendOneMessage;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  BTCommAdapterIndy in '..\..\source\BTCommAdapterIndy.pas',
  BTJMSConnection in '..\..\source\BTJMSConnection.pas',
  BTJMSInterfaces in '..\..\source\BTJMSInterfaces.pas';

var
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;

begin
  Connection := TBTJMSConnection.MakeConnection('', '', 'stomp://localhost');
  Connection.Start;
  try
    Session := Connection.CreateSession(False, amAutoAcknowledge);
    WriteLn('Send a message');
    Destination := Session.CreateQueue('onemessage');
    Producer := Session.CreateProducer(Destination);
    Producer.Send(Session.CreateTextMessage('This is a test message'));
    WriteLn('Hit any key');
    ReadLn;
  finally
    Connection.Close;
  end;
end.
```

The unit BTCommAdapterIndy contains the Internet Direct (Indy) communication adapter class. By including this unit, it will register the adapter class with an internal list of all available communication adapters. By default, the first registered communication adapter will be used.

HabariExpress Example Component

In the following example, an instance of the component will be created at runtime and then used to send a text message to the queue 'test'.

Note that configuration of the HabariExpress component happens before setting the component to 'Active', and changing properties while the HabariExpress component is active will trigger an exception.

```
Msg: IMessage;
...
Habari := THabariExpress.Create(nil);

try
  // set the destination
  Habari.OptionsDestination.DestinationName := 'test';

  // Open the connection
  Habari.Active := True;

  // create text messsage
  Msg := Habari.Session.CreateTextMessage('Hello world');

  // send text
  Habari.MessageProducer.Send(Msg);

  // Close the connection
  Habari.Active := False;

finally
  Habari.Free;

end;
```

For text messages, there is also a simple Send method that takes a string parameter:

```
...

Habari.Send('Text message body');

...
```

Important note The Habari Express component is included as example code only and is unsupported. Some features of the Habari ActiveMQ Client library might not be available in Habari Express.

Receive Text Messages

Asynchronous receive

To receive text messages asynchronously, the client subscribes to a destination (which can be a queue or a topic) on the server.

The messages will be delivered to an event handler which has to be provided by the client.

```
var
  Destination: IDestination;
  Consumer: IMessageConsumer;

begin
 ...
  // create a destination queue
  Destination := Session.CreateQueue('test');

  // create a consumer
  Consumer := Session.CreateConsumer(Destination);

  // set the message listener
  Consumer.MessageListener := Self;
  ...
end;
```

The asynchronous MessageListener is an object which implements the IMessageListener interface.

This interface only contains one procedure, OnMessage:

```

  IMessageListener = interface(IInterface)
    procedure OnMessage(Message: IMessage);
  end;
```

Synchronous Receive

A MessageConsumer offers a Receive method which can be used to consume exactly one message at a time.

Example (from SubscriberDemo.dpr):

```
while I < EXPECTED do
begin
  TextMessage := ITextMessage(Consumer.Receive(1000));
  if Assigned(TextMessage) then
  begin
    Inc(I);
    TextMessage.Acknowledge;
    L.Info(Format('%d %s', [I, TextMessage.Text]));
  end;
end;
```

Compared with a MessageListener, the Receive method has the advantage that the application can stop consuming messages at any point in time (for example, after receiving 20 messages). With an asynchronous MessageListener, it is possible that the MessageConsumer will still receive some messages after calling the close method.

Binary Messages

Send Binary Messages

The GUI demo includes an option to send binary files as JMS messages.

The following code uses TFileStream and TStringStream to load the selected file into the memory and the Send method of the MessageProducer to transmit the file content.

```
procedure TDemoMainForm.SendFile(Sender: TObject);
var
  Destination: IDestination;
  Producer: IMessageProducer;
  FileStream: TFileStream;
  S: TStringStream;
  BytesMessage: IBytesMessage;
begin
  ...

  Producer := Session.CreateProducer(Destination);
  S := TStringStream.Create('');
  try
    FileStream := TFileStream.Create(OpenDialog1.FileName, fmOpenRead or
fmShareDenyWrite);
    try
      S.CopyFrom(FileStream, FileStream.Size);
      BytesMessage := Session.CreateBytesMessage;
      BytesMessage.Content := S.DataString;
      Producer.Send(BytesMessage);
    finally
      FileStream.Free;
    end;
  finally
    S.Free;
  end;
end;
```

Note that this procedure works only if the file size does not exceed the maximum size for a string.

Memory Streams

The following code converts a TMemoryStream instance to a string (given that the stream size does not exceed the maximum size for a string):

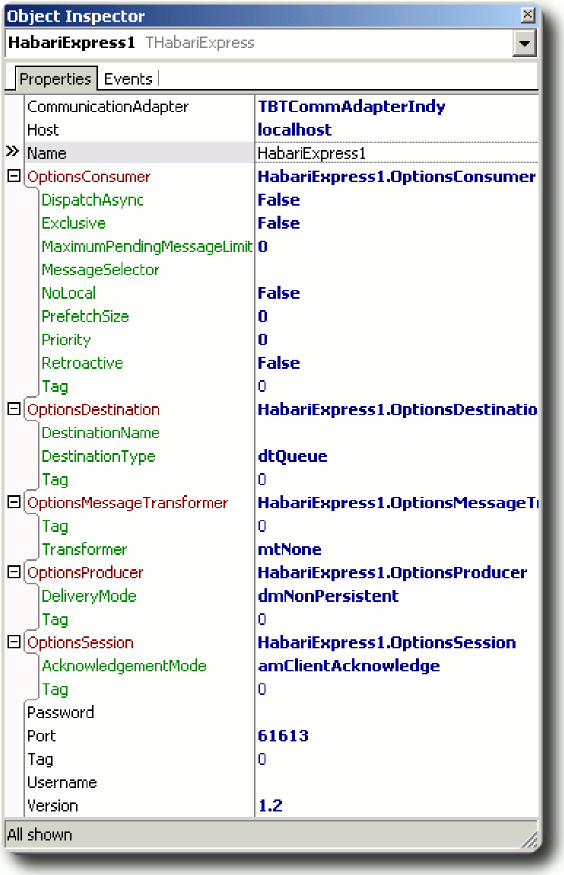
```
function MemoryStreamToString(Stream: TMemoryStream): string;
begin
  SetString(Result, PChar(Stream.Memory), Stream.Size);
end;
```

Example Components

The source includes two example components based on Habari ActiveMQ Client, with a limited features set. They can be used without installation in the Delphi IDE. If you want to install them in the IDE, you may use one of the package projects in the packages directory.

For example, packages\d120\dclHabariD120.dpk is the Delphi 2009 package, dclHabariD105.dpk is the Delphi 2007 package.

Note that these components are only example sources, they do not include all options which are available in the core Habari library. They are unsupported.



CommunicationAdapter	TBTCommAdapterIndy
Host	localhost
Name	HabariExpress1
OptionsConsumer	HabariExpress1.OptionsConsumer
DispatchAsync	False
Exclusive	False
MaximumPendingMessageLimit	0
MessageSelector	
NoLocal	False
PrefetchSize	0
Priority	0
Retroactive	False
Tag	0
OptionsDestination	HabariExpress1.OptionsDestinatio
DestinationName	
DestinationType	dtQueue
Tag	0
OptionsMessageTransformer	HabariExpress1.OptionsMessageT
Tag	0
Transformer	mtNone
OptionsProducer	HabariExpress1.OptionsProducer
DeliveryMode	dmNonPersistent
Tag	0
OptionsSession	HabariExpress1.OptionsSession
AcknowledgementMode	amClientAcknowledge
Tag	0
Password	
Port	61613
Tag	0
Username	
Version	1.2

HabariExpress component properties (version 1.2)

Example Applications

ConsumerTool

The ConsumerTool demo is based on the Java example class ConsumerTool.java in the ActiveMQ binary distribution.

It is configurable by command line parameters, all are optional:

AckMode	Acknowledgement mode, possible values are: CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE or SESSION_TRANSACTED
ClientId	client id for durable subscriber
ConsumerName	name of the message consumer - for durable subscriber
Durable	true: use a durable subscriber
MaximumMessages	expected number of messages
Password	password
PauseBeforeShutDown	true: wait for key press
ReceiveTimeOut	0: asynchronous receive, > 0: consume messages while they continue to be delivered within the given time out
SleepTime	time to sleep after asynchronous receive
Subject	queue or topic name
Topic	true: topic false: queue
Transacted	true: transacted session
URL	server url
User	user name
Verbose	verbose output

Source code:

```
unit ConsumerToolUnit;

interface

uses
  BTJMSInterfaces;

type
{$M+}
  TConsumerTool = class(TInterfacedObject, IMessageListener)
  private
    Session: ISession;
    Running: Boolean;
    Consumer: IMessageConsumer;
    ReplyProducer: IMessageProducer;

    FAckMode: TAcknowledgementMode;
    FURL: string;
    FTopic: Boolean;
    FSubject: string;
    FDurable: Boolean;
    FSleepTime: Integer;
    FMaximumMessages: Integer;
    FTransacted: Boolean;
    FVerbose: Boolean;
    FUser: string;
    FPassword: string;
    FClientId: string;
    FConsumerName: string;
    FReceiveTimeOut: Integer;
    FPauseBeforeShutdown: Boolean;

    function TargetType: string;
    function DurableString: string;

    procedure SetAckMode(const Value: string);

    procedure OnMessage(const Message: IMessage);
    procedure ConsumeMessagesAndClose(Conn: IConnection; Session:
ISession;
      Consumer: IMessageConsumer); overload;
    procedure ConsumeMessagesAndClose(Conn: IConnection; Session:
ISession;
      Consumer: IMessageConsumer; TimeOut: Integer); overload;

  public
    constructor Create;

    procedure Run;
  published
    property AckMode: string write SetAckMode;
    property ClientId: string read FClientId write FClientId;
    property ConsumerName: string read FConsumerName write
FConsumerName;
    property Durable: Boolean read FDurable write FDurable;
    property MaximumMessages: Integer read FMaximumMessages write
      FMaximumMessages;
    property Password: string read FPassword write FPassword;
    property PauseBeforeShutdown: Boolean read FPauseBeforeShutdown
write
      FPauseBeforeShutdown;
    property ReceiveTimeOut: Integer read FReceiveTimeOut write
FReceiveTimeOut;
    property SleepTime: Integer read FSleepTime write FSleepTime;
    property Subject: string read FSubject write FSubject;
    property Topic: Boolean read FTopic write FTopic;
    property Transacted: Boolean read FTransacted write FTransacted;
    property URL: string read FURL write FURL;
    property User: string read FUser write FUser;
    property Verbose: Boolean read FVerbose write FVerbose;

  end;

implementation

uses
  CommandLineSupport,
  BTCommAdapterIndy,
  BTJMSConnection,
  BTJMSConnectionFactory,
  StrUtils, SysUtils;

{ TConsumerTool }

constructor TConsumerTool.Create;
begin
  ConsumerName := 'James';
  MaximumMessages := 10;
  Subject := 'TOOL.DEFAULT';
  URL := BTJMSConnectionFactory.DEFAULT_BROKER_URL;
  Verbose := True;
end;

procedure TConsumerTool.SetAckMode(const Value: string);
begin
  if Value = 'CLIENT_ACKNOWLEDGE' then
    FAckMode := amClientAcknowledge
  else if Value = 'AUTO_ACKNOWLEDGE' then
    FAckMode := amAutoAcknowledge
  else if Value = 'SESSION_TRANSACTED' then
```

```
    FAckMode := amTransactional
end;

function TConsumerTool.TargetType: string;
begin
  if Topic then
    Result := 'topic'
  else
    Result := 'queue';
end;

function TConsumerTool.DurableString: string;
begin
  if Durable then
    Result := 'durable'
  else
    Result := 'non-durable';
end;

procedure TConsumerTool.OnMessage(const Message: IMessage);
var
  TxtMsg: ITextMessage;
  Msg: string;
begin
  try
    try
      if Supports(Message, ITextMessage, TxtMsg) then
      begin
        if Verbose then
        begin
          Msg := TxtMsg.Text;
          if Length(Msg) > 50 then
            Msg := Copy(Msg, 1, 50) + '...';
          WriteLn('Received: ' + Msg);
        end;
      end
      else
      begin
        if Verbose then
          WriteLn('Received: Message');
      end;

      if Message.JMSReplyTo <> nil then
      begin
        ReplyProducer.Send(Message.JMSReplyTo,
          Session.CreateTextMessage('Reply: ' + Message.JMSMessageID));
      end;

      if Transacted then
        Session.Commit
      else if FAckMode = amClientAcknowledge then
        Message.Acknowledge;
    except
      on E: Exception do
      begin
        WriteLn(E.Message);
      end;

    end;
  finally
    if SleepTime > 0 then
    begin
      Sleep(SleepTime);
    end;
  end;

end;

procedure TConsumerTool.ConsumeMessagesAndClose(Conn: IConnection;
Session:
  ISession; Consumer: IMessageConsumer);
var
  I: Integer;
  Message: IMessage;
begin
  WriteLn('We are about to wait until we consume: ' +
IntToStr(MaximumMessages)
    + ' message(s) then we will shutdown');

  I := 0;
  while (I < MaximumMessages) and Running do
  begin
    Message := Consumer.Receive(1000);
    if Message <> nil then
    begin
      Inc(I);
      OnMessage(Message);
    end;
  end;

  WriteLn('Closing connection');
  Consumer.Close;
  Session.Close;
  Conn.Close;
  if PauseBeforeShutdown then
  begin
    WriteLn('Press return to shut down');
    ReadLn;
  end;
end;

procedure TConsumerTool.ConsumeMessagesAndClose(Conn: IConnection;
Session:
  ISession; Consumer: IMessageConsumer; TimeOut: Integer);
var
  Message: IMessage;
begin
  WriteLn('We will consume messages while they continue to be delivered
within: '
    + IntToStr(Timeout) + ' ms, and then we will shutdown');

  Message := Consumer.Receive(Timeout);
  while (Message <> nil) do
  begin
    OnMessage(Message);
    Message := Consumer.Receive(Timeout);
  end;

  WriteLn('Closing connection');
  Consumer.Close;
  Session.Close;
  Conn.Close;
  if PauseBeforeShutdown then
  begin
    WriteLn('Press return to shut down');
    ReadLn;
  end;

end;

procedure TConsumerTool.Run;
var
  ConnectionFactory: TBTJMSConnectionFactory;
  Connection: IConnection;

  Destination: IDestination;
begin
  TCommandLineSupport.Configure(Self);

  Running := True;

  WriteLn('Connecting to URL: ' + URL);
  WriteLn('Consuming ' + TargetType + ': ' + Subject);
  WriteLn('Using a ' + DurableString + ' subscription');

  ConnectionFactory := TBTJMSConnectionFactory.Create(User, Password,
URL);
  Connection := ConnectionFactory.CreateConnection;
  if (Durable and (ClientId <> '')) then
  begin
    Connection.ClientID := ClientId;
  end;
  Connection.Start;

  // Create the session.
  Session := Connection.CreateSession(Transacted, FAckMode);

  // Create the Producer for the Destination.
  if Topic then
    Destination := Session.CreateTopic(Subject)
  else
    Destination := Session.CreateQueue(Subject);

  ReplyProducer := Session.createProducer(nil);
  ReplyProducer.setDeliveryMode(dmNonPersistent);

  if (Durable and Topic) then
    Consumer := Session.CreateDurableSubscriber(ITopic(Destination),
      ConsumerName)
  else
    Consumer := Session.CreateConsumer(Destination);

  if MaximumMessages > 0 then
  begin
    ConsumeMessagesAndClose(Connection, Session, Consumer);
  end
  else
  begin
    if ReceiveTimeOut = 0 then
      Consumer.SetMessageListener(Self)
    else
      ConsumeMessagesAndClose(Connection, Session, Consumer,
ReceiveTimeOut);
  end;

  Connection.Close;
  WriteLn('Done.');
end;

end.
```

ProducerTool

The ProducerTool demo is based on the Java example class ProducerTool.java in the ActiveMQ binary distribution.

It is configurable by command line parameters, all are optional:

MessageCount	number of messages
MessageSize	length of a message
Persistent	delivery mode persistent
SleepTime	pause between messages
Subject	destination name
TimeToLive	message expiration time
Topic	destination is a topic
Transacted	use a transaction
URL	message broker URL
Verbose	verbose output

The demo uses the CommandLineSupport helper unit to set these properties.

Source code:

```
unit ProducerToolUnit;

interface

uses
  BTJMSInterfaces;

type
{$M+}
  TProducerTool = class(TObject)
  private
    FURL: string;
    FMessageSize: Integer;
    FTopic: Boolean;
    FSubject: string;
    FPersistent: Boolean;
    FSleepTime: Integer;
    FTimeToLive: Integer;
    FMessageCount: Integer;
    FTransacted: Boolean;
    FVerbose: Boolean;

    function TargetType: string;
    function PersistentString: string;

    procedure SendLoop(const Session: ISession;
      const Producer: IMessageProducer);

  public
    constructor Create;

    procedure Run;

  published
    property MessageCount: Integer read FMessageCount write
FMessageCount;
    property MessageSize: Integer read FMessageSize write FMessageSize;
    property Persistent: Boolean read FPersistent write FPersistent;
    property SleepTime: Integer read FSleepTime write FSleepTime;
    property Subject: string read FSubject write FSubject;
    property TimeToLive: Integer read FTimeToLive write FTimeToLive;
    property Topic: Boolean read FTopic write FTopic;
    property Transacted: Boolean read FTransacted write FTransacted;
    property URL: string read FURL write FURL;
    property Verbose: Boolean read FVerbose write FVerbose;

  end;

implementation
```

```
uses
  CommandLineSupport,
  BTCommAdapterIndy,
  BTJMSConnection,
  BTJMSConnectionFactory,
  StrUtils, SysUtils;

{ TProducerTool }

constructor TProducerTool.Create;
begin
  MessageCount := 10;
  MessageSize := 255;
  Subject := 'TOOL.DEFAULT';
  URL := BTJMSConnectionFactory.DEFAULT_BROKER_URL;
  Verbose := True;
end;

function TProducerTool.TargetType: string;
begin
  if Topic then
    Result := 'topic'
  else
    Result := 'queue';
end;

function TProducerTool.PersistentString: string;
begin
  if Persistent then
    Result := 'persistent'
  else
    Result := 'non-persistent';
end;

procedure TProducerTool.Run;
var
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
begin
  TCommandLineSupport.Configure(Self);

  WriteLn('Connecting to URL: ' + URL);
  WriteLn('Publishing a Message with size ' + IntToStr(MessageSize) + '
to ' +
    TargetType + ': ' + Subject);
  WriteLn('Using ' + PersistentString + ' messages');
  WriteLn('Sleeping between publish ' + IntToStr(SleepTime) + ' ms');
  if TimeToLive <> 0 then
  begin
```

```
    WriteLn('Messages time to live ' + IntToStr(TimeToLive) + ' ms');
  end;

  Connection := TBTJMSConnection.MakeConnection;
  Connection.Start;

  // Create the session.
  Session := Connection.CreateSession(Transacted, amAutoAcknowledge);

  // Create the Producer for the Destination.
  if Topic then
    Destination := Session.CreateTopic(Subject)
  else
    Destination := Session.CreateQueue(Subject);

  // Create the producer.
  Producer := Session.CreateProducer(Destination);

  if Persistent then
    Producer.DeliveryMode := dmPersistent
  else
    Producer.DeliveryMode := dmNonPersistent;

  if TimeToLive <> 0 then
    Producer.TimeToLive := TimeToLive;

  SendLoop(Session, Producer);

  Connection.Close;
  WriteLn('Done.');
end;

procedure TProducerTool.SendLoop(const Session: ISession;
  const Producer: IMessageProducer);
var
  I: Integer;
  TextMessage: ITextMessage;
  Msg: string;

  function CreateMessageText(const Index: Integer): string;
  begin
    Result := 'Message: ' + IntToStr(Index) + ' sent at: ' +
DateTimeToStr(Now);

    if Length(Result) > MessageSize then
      Result := Copy(Result, 1, MessageSize)
    else
      Result := Copy(Result + DupeString(' ', MessageSize), 1,
MessageSize);
  end;

begin
```

```
  for I := 0 to MessageCount - 1 do
  begin
    TextMessage := Session.CreateTextMessage(CreateMessageText(I));
    if Verbose then
    begin
      Msg := TextMessage.Text;
      if Length(Msg) > 50 then
      begin
        Msg := Copy(Msg, 1, 50) + '...';
      end;
      WriteLn('Sending message: ' + Msg);
    end;
    Producer.Send(TextMessage);
    if Transacted then
    begin
      Session.Commit;
    end;
    Sleep(SleepTime);
  end;
end;

end.
```

Object Messages

Object messages and object exchange between Java and Delphi with Habari ActiveMQ Client is explained in detail in the document HabariObjectExchange.pdf

Message Options

JMS Standard Properties

API Documentation

JMS Standard properties are documented in more detail in the API documentation for the TBTMessage class. The are based on the JMS specification of the Message interface.[12]

JMS properties for outgoing messages

Messages sent by Habari ActiveMQ Client can set these JMS standard properties:

JMSCorrelationID	The correlation ID for the message.
JMSExpiration	The message's expiration value.
JMSDeliveryMode	Whether or not the message is persistent.
JMSPriority	The message priority level.
JMSReplyTo	The Destination object to which a reply to this message should be sent.

JMS properties for incoming messages

Messages received by Habari ActiveMQ Client may contain these JMS standard properties:

JMSCorrelationID	The correlation ID for the message.
JMSExpiration	The message's expiration value.
JMSDeliveryMode	Whether or not the message is persistent.
JMSPriority	The message priority level.
JMSTimestamp	The timestamp the broker added to the message.
JMSMessageId	The message ID which is set by the provider.
JMSReplyTo	The Destination object to which a reply to this message should be sent.

---

[12] http://java.sun.com/javaee/5/docs/api/javax/jms/Message.html

User Defined Properties

Supported Data Types

The Stomp protocol only supports string type properties.

Reserved Names

The following names are reserved Stomp header properties and can not be used as names for user defined properties:

- activemq.* (everything starting with activemq is a reserved name)
- login
- passcode
- transaction
- session
- message
- destination
- id
- ack
- selector
- type
- content-length
- correlation-id
- expires
- persistent
- priority
- reply-to
- message-id
- timestamp
- transformation
- client-id
- redelivered

The client library detects overwriting of Stomp defined message properties. It will raise an Exception if the application tries to send a message with a reserved property name.

ESB Integration Examples

Overview

This section will give you some step by step introductions to the integration of Habari ActiveMQ and Open Source ESB (Enterprise Service Bus) systems.

Apache ServiceMix: Basic example

Introduction

This section will show how Habari can be used in an environment using Apache ServiceMix 3.

The Basic example has successfully been tested with Apache ServiceMix 3.2.2-SNAPSHOT release.

ServiceMix configuration

To prepare the example for Habari ActiveMQ Client, add the Stomp connector to the servicemix.xml file in the folder examples/basic/src/main/resources:

```
<!-- message broker -->
<amq:broker id="broker" persistent="false">
  <amq:transportConnectors>
    <amq:transportConnector uri="tcp://localhost:61616" />
    <amq:transportConnector uri="stomp://localhost:61613" />
  </amq:transportConnectors>
</amq:broker>
```

Launch the basic demo

To launch the demo, build and run the example using Maven:

```
mvn jbi:embeddedServicemix
```

ServiceMix will use a Quartz timer to send messages to the topic with the name 'servicemix.source'.

Note	Apache ServiceMix uses an built-in ActiveMQ server. To avoid port conflicts, you should not launch a second instance of Apache ActiveMQ.

Receive the test messages using Habari

You can use the Habari ActiveMQ Client GUI demo to receive the JMS messages. To do this, connect to ActiveMQ on port 61613, start a session and create a new topic with the name **servicemix.source**.

If you click on subscribe, the log file will display the messages from ServiceMix.

About Apache ServiceMix

Apache ServiceMix is an open source ESB (Enterprise Service Bus) that combines the functionality of a Service Oriented Architecture (SOA) and an Event Driven Architecture (EDA) to create an agile, enterprise ESB.

Apache ServiceMix is an open source distributed ESB built from the ground up on the Java Business Integration (JBI) specification JSR 208 and released under the Apache license. The goal of JBI is to allow components and services to be integrated in a vendor independent way, allowing users and vendors to plug and play.

MULE: Echo Example

Introduction

This section will show how Habari can be used in an environment using Apache ActiveMQ 5.0 and MULE 1.4.3.

Apache ActiveMQ Configuration

Apache ActiveMQ has to be configured for Stomp and TCP support. To do this, open the configuration file activemq.xml and verify that it contains the following settings for the transport connectors:

```
<transportConnectors>
  <transportConnector name="openwire" uri="tcp://localhost:61616"
     discoveryUri="multicast://default"/>
  <transportConnector name="stomp"   uri="stomp://localhost:61613"/>
</transportConnectors>
```

If you launch ActiveMQ, check that the log file shows the successful startup of these connectors:

```
INFO  TransportServerThreadSupport   - Listening for connections at:
tcp://localhost:61616
INFO  TransportConnector             - Connector openwire Started
INFO  TransportServerThreadSupport   - Listening for connections at:
stomp://localhost:61613
INFO  TransportConnector             - Connector stomp Started
```

MULE configuration

Edit the echo.bat file in the echo example folder so that it uses the echo-config.xml configuration file:

```
call "%MULE_BASE%\bin\mule.bat" -config .\conf\echo-config.xml
```

Edit the echo-config.xml configuration file in the echo/conf folder. Add the JMS connector just after the system stream connector:

```
<!-- ActiveMQ configuration -->
<connector name="jmsConnector"
  className="org.mule.providers.jms.activemq.ActiveMqJmsConnector">
  <properties>
    <property name="connectionFactoryJndiName" value="ConnectionFactory"/>
    <property name="jndiInitialFactory"
     value="org.apache.activemq.jndi.ActiveMQInitialContextFactory"/>
    <property name="specification" value="1.1"/>
    <map name="connectionFactoryProperties">
      <property name = "brokerURL" value = "tcp://localhost:61616" />
    </map>
  </properties>
</connector>
```

Add the ActiveMQ queue "in.queue" to the list of inbound router endpoint addresses, and replace the outbound endpoint (System.out) with the "out.queue" endpint address:

```
<inbound-router>
  <endpoint address="stream://System.in"/>
  <endpoint address="vm://echo" />
  <endpoint address="jms://in.queue" />
</inbound-router>

...

<outbound-router>
  <router className="org.mule.routing.outbound.OutboundPassThroughRouter">
    <!-- endpoint address="stream://System.out"/ -->
    <endpoint address="jms://out.queue"/>
  </router>
</outbound-router>
```

Important Note that there can be only one endpoint in a OutboundPassThroughRouter

Additional setup of MULE

Check list:

- the ActiveMQ jar file has to be copied to the lib/user folder. All libraries (.jar files) in this folder will be added to the classpath before starting Mule.
- the environment variable MULE_HOME is required, it should point to the MULE installation folder

Launch the “echo” Demo

After successful completion of the MULE startup, the demo will ask you to enter text.

The text will be passed from MULE to the outbound endpoint, the ActiveMQ queue “out.queue”.

You can verify this using the ActiveMQ admin console (http://localhost:8161/admin).

You can also use the ActiveMQ admin console to send messages to the ActiveMQ queue “in.queue”, which is the inbound MULE endpoint. These messages will also be passed from MULE to the to the ActiveMQ queue “out.queue”.

Known Limitations

Communication Libraries

The communication adapters for OverByte ICS V6 and TClientSocket do not reliable support receiving of messages.

The ICS communication adapter uses Overbyte ICS V6 (for Delphi up to 2007). ICS V7 for Delphi 2009 is under development and might be supported in a future version of Habari.

The communication adapters based on OverByte ICS and TClientSocket do not support console applications, and can not be compiled under Free Pascal.

Location

The source code for unsupported TCP/IP communication libraries is located in the folder source/unsupported/commlib

Overbyte ICS V6

The communication adapter for ICS only works in GUI-based applications. It supports only message sending.

TClientSocket

The communication adapter for TClientSocket only works in GUI-based applications. It supports message sending only.

TClientSocket has been declared deprecated by Borland / CodeGear.

Sessions

Acknowledgement Modes

Acknowledgment mode “amDupsOkAcknowledge” is unsupported.

Destinations

Durable Subscriptions

Removing durable subscriptions is not supported.

Messages

Message Property Data Types

The Stomp protocol uses string type key/value lists for the representation of message properties. Regardless of the method used to set message properties (e.g. SetInt or SetDate), all message properties will be interpreted as Java Strings by the Message Broker.

As a side effect, the expressions in a Selector are limited to operations which are valid for strings.

Timestamp properties are converted to an Unix time stamp value, which is the internal representation in Java. But still, these values can not be used with date type expressions.

Multi Threading

The unit test suite contains multi threading tests, but there is no guarantee for error-free operation of the library in applications which make extensive use of multi threading.

SOAP Object Exchange

Delphi

Sending and receiving of objects requires Delphi 7 or higher. The library is designed to use methods that were added to TRemotable: ObjectToSOAP and SOAPToObject. These methods are available since Delphi 7.

Free Pascal

Sending and receiving of objects in FreePascal requires the Web Service Toolkit for binary serialization. The Habari ActiveMQ Client library distribution currently does not include examples for object exchange using Free Pascal.

References

Message Broker

Apache ActiveMQ	http://activemq.apache.org
IONA	http://open.iona.com/products/fuse-message-broker

IDE

CodeGear Delphi	http://www.codegear.com/delphi
Free Pascal	http://freepascal.org
Lazarus	http://www.lazarus.freepascal.org

JMS

JMS Spec (PDF)	http://java.sun.com/products/jms/docs.html

JSON

LkJSON	http://sourceforge.net/projects/lkjson
SuperObject	http://www.progdigy.com

SOAP

FPC Web Services	http://wiki.lazarus.freepascal.org/Web_Service_Toolkit

Stomp

In ActiveMQ	http://activemq.apache.org/stomp.html
Project home	http://stomp.codehaus.org/

Communication Libraries

Overbyte ICS	http://www.overbyte.be
Synapse	http://www.synapse.ararat.cz
Indy 10	http://www.indyproject.org
Indy 10 Snapshot	http://indy.fulgan.com/ZIP

XML libraries

OmniXML	http://www.omnixml.com/
XStream	http://xstream.codehaus.org/

Release Notes

Version 1.5

Released March 3, 2009

New

XML transformation	Support for object exchange using XML serialization, based on persistency helper methods in the OpenXML library (see example in xmljava folder)
ProducerTool demo	Command line tool which generates test messages, many configuration parameters (inspired by the ActiveMQ ProducerTool class)
ConsumerTool demo	Command line tool which consumes test messages, many configuration parameters (inspired by the ActiveMQ ConsumerTool class)

Fixed

BinaryMessage	Fixed a bug in the Indy communication adapter (Delphi 2009)
Message header	The FillCreatedMessage method now only copies user-defined Stomp headers to the properties of the incoming JMS message

Changed

Synapse exceptions	The Synapse adapter raises exceptions in case of connection failures (this is now consistent with the Indy implementation)
TBytes data type	All communication adapters use the TBytes data type in the StompTransmit method
Transformer	Registration of default transformers has been replaced by explicit creation, the transformer constructor parameter is the class of the serialized objects
SOAP transformer	The BTMessageTransformerSOAP unit is beta / experimental now

DelphiGUI demo	The demo application includes a new administration page which displays server, client, topic and queue information (see readme.txt for details about message broker configuration)
JSON Toolkit	Deprecated JSON Toolkit adapter has been deleted
beta folder	The folder <installdir>\source\beta contains beta versions of new units (note that these units are not guaranteed to be included in future versions)

Version 1.4

Released February 9, 2009

New

BrokerURL	The factory methods to create a JMS connection now use URI syntax. For example, the BrokerURL using the stomp or the stomp+ssl protocol would be 'stomp://localhost' or 'stomp+ssl://localhost:61612'
Durable Subscriber	Session.CreateDurableSubscriber method creates a durable subscriber to the specified topic
SSL Support	A new communication adapter with SSL support is included, TBTCommAdapterIndySSL
Send Timeout	The send timeout can be set using a new property of the JMS connection
Synapse Support	Delphi 2009 can be used with revision 95 of the Synapse library
lkJSON	Delphi 2009 can be used with version 1.05 of the lkJSON library (and USE_D2009 compiler switch)

Changed

Demo	SSL support has been added to the delphigui demo application
Renamed file	The BTConnectionFactory.pas unit has been renamed to BTJMSConnectionFactory.pas to avoid name conflicts with other Habari Client libraries
Performance	The performance of the Synapse based communication adapter has been improved
JMS Selectors	The manual now includes information about the usage of JMS Selectors in SQL and XPath syntax

JMSReplyTo	JMSReplyTo headers are now supported in for incoming messages
XPath support	The documentation has been updated to include the information about required XPath support libraries (JAR files)
Delphi 2009	Fixed all compiler warnings (except for third party libraries like SuperObject, lkJSON, Synapse)

Version 1.3

Released January 8, 2009

New

Transformer	Like communication adapters, all object message transformers now use a transformer registry. A message transformation unit is provided for every JSON and SOAP implementation library
JSON Support	JSON serialization is now supported in Delphi 2009 with the new SuperObject library
SOAP Support	SOAP message transformation adapter with demo application

Changed

Interface Parameters	The **const** keyword has been added to interface type parameters to avoid unnecessary reference counting
ActiveMQ 5.2	This release has been tested with the new release 5.2 of Apache ActiveMQ
ICS V6 RC 1	This release has been tested with ICS V6 RC1, the release candidate of the Internet Component Suite
ICS/TServerSocket	The source code for unsupported TCP/IP communication libraries is now located in the folder source/unsupported/commlib
Multi Threading	The DUnit test suite includes new tests for multi threaded usage of the core library

Version 1.2

Released September 6, 2008

New

Delphi 2009	The library compiles and runs in Delphi 2009. Unicode is supported in the message body and message property values. Note: JSON object transformation is not supported for Delphi 2009.
lkJSON Support	Support for the lkJSON library has been added. The default library for JSON transformation is json_toolkit. To activate lkJSON, add the compiler switch LKJSON.
Load Balancing	The demo source code now include a simple file based load balancing example.

Fixed

Expiration Time	The library used the local time zone to calculate the expiration time in the message expiration header. This has been changed to UTC.
Closed Connections	Closing a closed connection does not throw an EBTStompClientAlreadyDisconnectedError anymore.
Packages	The pre-built package files include the source path to the JSON library now.

Version 1.1

Released March 31, 2008

New

ObjectMessage	A new message type supports data exchange using the Apache ActiveMQ standard JSON object message transformation The property OptionsMessageTransformer has been added in HabariExpress to support JSON object message transformation
Subscription config	You can add custom headers to configure a subscription. (see Chapter 'Destinations') The property OptionsConsumer has been extended in HabariExpress to support subscription configuration

---

Version 1.0.1

Released March 11, 2008

New

Unicode properties	String type message properties now support Unicode
Palette bitmap	HabariExpress and HabariExpressAdmin now have palette bitmaps (component icons)

Fixed

Unicode body	Incoming text messages which used Unicode in the message body have not been converted back to WideString. This has been fixed.
Examples	The SoapTransfer example application has been fixed

Version 1.0

Released March 5, 2008

FAQ – Frequently Asked Questions

Compiler Errors

BTCommAdapterIndy.pas

The Delphi compiler stops at this line in BTCommAdapterIndy.pas

```
Result := IndyTCPClient.IOHandler.CheckForDataOnSource(50);
```

Reason	The CheckForDataOnSource method in the Indy library is a function in version 10.2.3. Check that you are using the version 10.2.3 of Indy.

Index

Reference