



# Getting started with Habari OpenMQ Client

## *Version 1.8*

---

### Trademarks

Habari is a registered trademark of Michael Justin and is protected by the laws of Germany and other countries. Oracle and Java are registered trademarks of Oracle and/or its affiliates. Embarcadero, the Embarcadero Technologies logos and all other Embarcadero Technologies product or service names are trademarks, service marks, and/or registered trademarks of Embarcadero Technologies, Inc. and are protected by the laws of the United States and other countries. Microsoft, Windows, Windows NT, and/or other Microsoft products referenced herein are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other brands and their products are trademarks of their respective holders.

Contents

Introduction.....

About Habari OpenMQ Client.....

Quick Start.....

Download and Installation.....

GlassFish v3 configuration.....

Dependencies.....

Requirements.....

TCP/IP Communication Libraries.....

Communication Adapter Configuration.....

Introduction.....

The JMS API Programming Model.....

Tutorials.....

Habari Quick Start Tutorial.....

Online Tutorials.....

Connections and Sessions.....

Step by Step Example.....

Transacted Sessions.....

Failover Support.....

Destinations.....

Introduction.....

Create a new Destination.....

Producer and Consumer.....

Message Producer.....

Message Consumer.....

JMS Selectors.....

Text Messages.....

Sending Text Messages.....

Receive Text Messages.....

Bytes Messages.....

Creation.....

Sending.....

**Object Messages.....31**

**Introduction..... 31**

**Message Transformers in Habari OpenMQ Client.....31**

**Durable Subscriptions.....34**

**Description..... 34**

**Example Applications .....35**

**ConsumerTool..... 35**

**ProducerTool..... 41**

**Message Options.....45**

**JMS Standard Properties.....45**

**User Defined Properties.....46**

**Useful Units.....47**

**BTStreamHelper..... 47**

**BTJavaPlatform..... 47**

**Known Limitations.....48**

**Sessions..... 48**

**Messages..... 48**

**Security..... 49**

**References.....50**

**Habari OpenMQ Client License.....51**

**Third Party Library Licenses.....53**

**Synapse..... 53**

**Indy BSD License..... 53**

**SuperObject..... 54**

**Log4D..... 54**

**NativeXml..... 55**

**Release Notes.....56**

**Version 1.8..... 56**

**Version 1.7..... 57**

**Version 1.6..... 57**

**Version 1.5..... 58**

**Version 1.4..... 59**

**Version 1.3..... 59**

**Version 1.2.....**

**60**

**Version 1.1.....**

**60**

**Version 1.0.....**

**61**

**Index.....**

**62**

# Introduction

---

## About Habari OpenMQ Client

Habari OpenMQ Client is a Delphi library for OpenMQ. With Habari OpenMQ Client, Delphi developers can build integrated solutions, connecting cross language clients and protocols from C, Delphi, and Java using the peer-to-peer or the publish and subscribe communication model. The library uses a plug-in style architecture for communication libraries. It supports OpenMQ version 4.4 and 4.5, Delphi 6 to XE and Free Pascal, and follows the specification of the JMS API.

## How Can I Use It?

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

- **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.
- **Application Server Integration:** Open Message Queue is a key component of the GlassFish v2.1.1 and GlassFish v3 Application Server.
- **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.
- **Inter-process Communication:** applications can use point-to-point messages to exchange information between each other even if the receiver currently is not running.

## Example Illustrations

### Habari for shared business logic

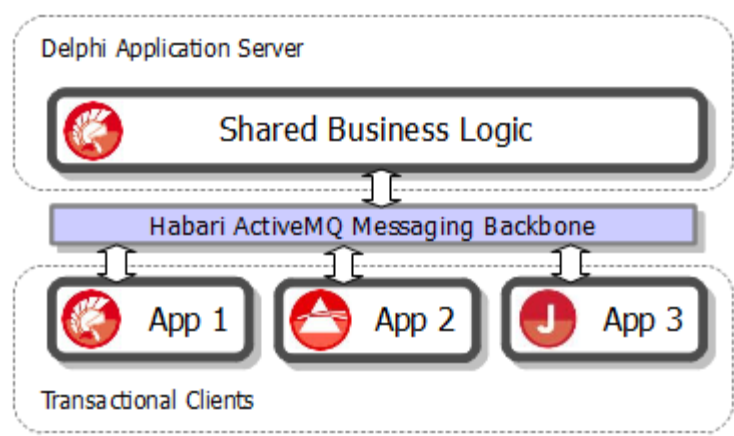


Illustration 1: Shared Business Logic

Similar to SOAP or REST servers, Delphi software systems can use Habari to provide business logic to other processes.

Documents and messages (including objects, serialized using JSON or XML) can be exchanged and secured by **client-side acknowledgment** and **transactional sessions**.

Habari in a network of Delphi applications

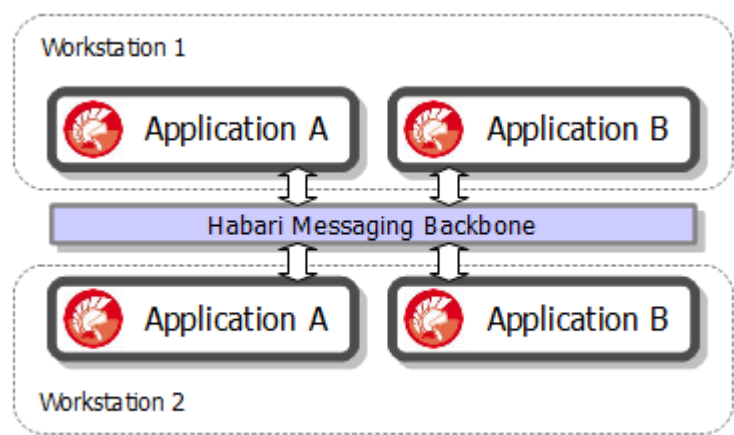


Illustration 2: Peer to Peer Communication

This illustration shows different Delphi applications running in a local network, using Habari client libraries to implement **Interprocess communication**: applications use point-to-point messages to exchange information between each other even if the receiver currently is not running.

Using the **publish/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.

Habari in a load balancing solution

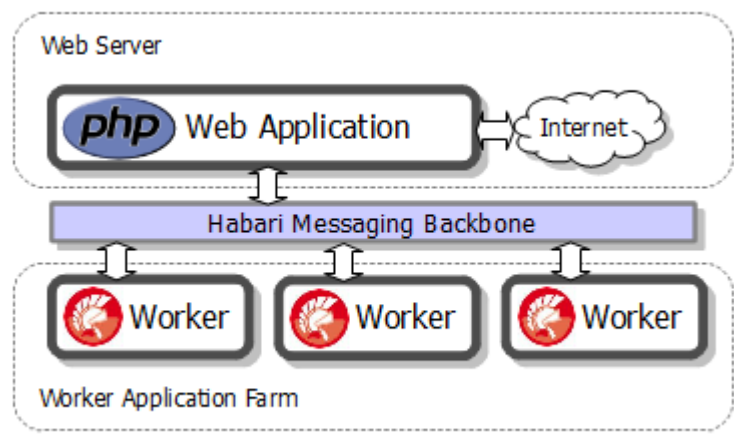


Illustration 3: Load Balancing

In this example, a PHP web application sends data to the message queue. The Habari communication layer in the Delphi worker applications takes care of receiving and acknowledging incoming messages.

Using the point-to-point or queuing model, many 'worker' applications can be installed on different computers. Every new message sent to the **message queue** will be delivered only to one client. The message broker will keep messages until they are expired or delivered to a client.

Habari JMS Client Libraries - Feature Matrix

Feature	Habari ActiveMQ Client	Habari HornetQ Client	Habari OpenMQ Client
JMS Message Types			
TextMessage / BytesMessage	✓ / ✓	✓ / ✓	✓ / ✓
ObjectMessage / MapMessage	✓ / ✓	✓ / -	✓ / -
JMS Features			
Temporary Queues	✓	-	✓
Durable Topics	✓	✓	✓
Transacted Sessions	✓	✓	✓
Stomp JMS Bindings			
Message Expiration (JMSExpiration)	✓	✓	✓
Message Priority Level (JMSPriority)	✓	✓	✓
Persistent Message Flag (JMSDeliveryMode)	✓	✓	✓
Message Selector: SQL-92 / XPath	✓ / ✓	✓ / -	✓ / -
No-Local Flag	✓	✓	✓
Object Serialization			
JSON libraries: SuperObject / IkJSON	✓ / ✓	✓ / -	✓ / -
XML libraries: NativeXml / OmniXML	✓ / ✓	- / ✓	✓ / ✓
Advanced Features			
Failover Transport	✓	✓	✓
Log4D logging library	✓	✓	✓
Object Exchange: Delphi-to-Delphi / Cross-Language	✓ / ✓	✓ / -	✓ / -
Included Demo Projects	16	10	11
Compiler Compatibility			
Delphi 6 – XE / Free Pascal 2.4.2 +	✓ / ✓	✓ / ✓	✓ / ✓
Supported Communication Adapters			
Internet Direct (Indy) / Synapse	✓ / ✓	✓ / ✓	✓ / ✓



## Quick Start

---

### Download and Installation

The OpenMQ 4.4 web page is located at <https://mq.dev.java.net/4.4.html>.

The OpenMQ 4.5 web page is located at <https://mq.dev.java.net/4.5.html>.

Note that this installation documentation uses the Windows installer, however the Habari OpenMQ Client library would work with installations of OpenMQ on other operating system platforms as well.

Installation steps for OpenMQ 4.5:

1. download the binary image file with installer (for Microsoft Windows x86) from the OpenMQ web page
2. unpack the installer
3. navigate to the directory `openmq4_5-installer`
4. start the installer script `installer.vbs`

When the installer application is started, a graphical application will display allow you to install and configure Message Queue.

### Start the Broker

You are ready to start the OpenMQ server now via the `mq/bin/imqbrokerd` command, which will run the broker and display log messages in a window

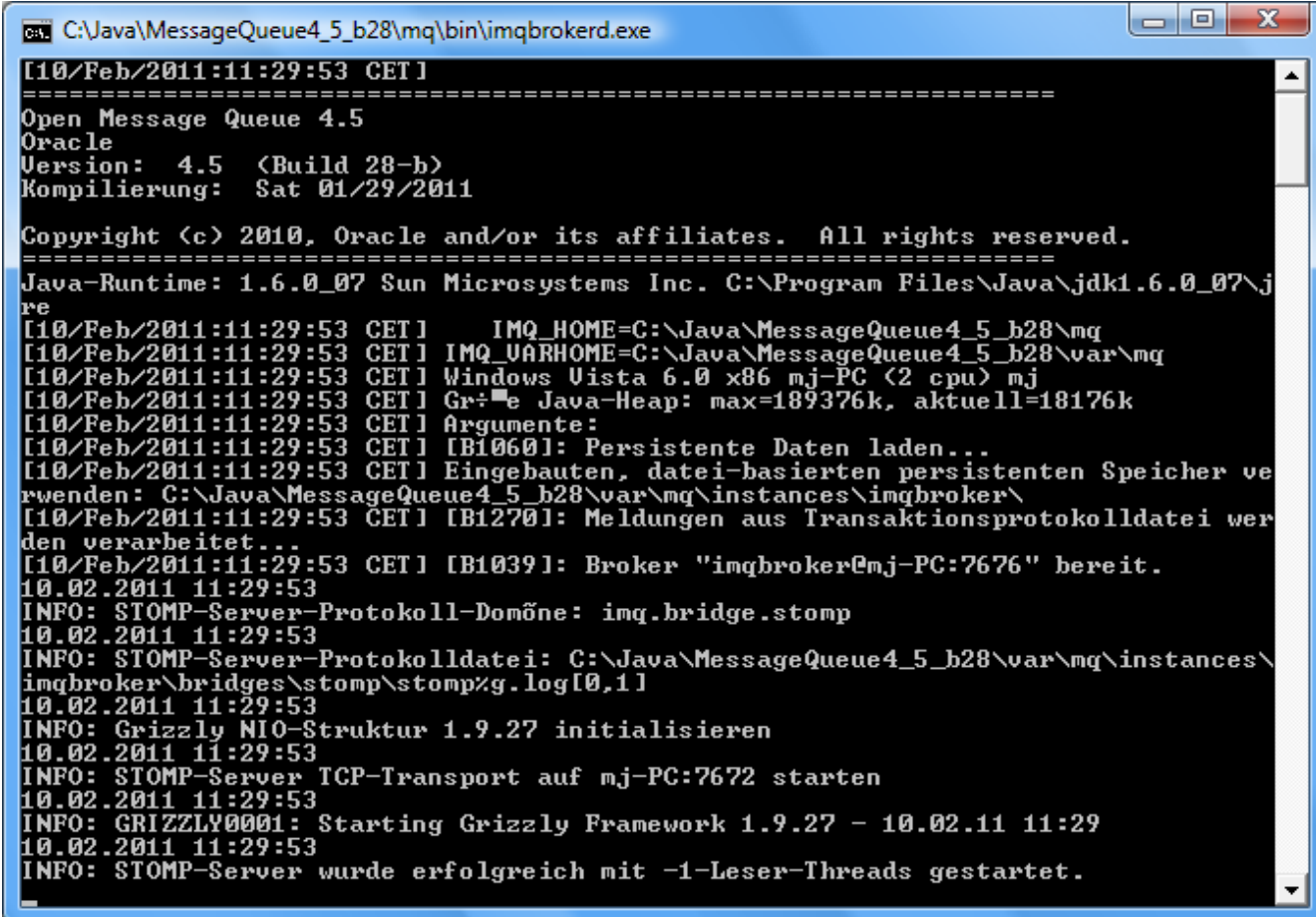
### Configuration

Now you will need to enable Stomp support in the broker and configure a user name and password for the broker bridge.

1. stop the broker
2. navigate to the configuration folder `.../instances/<instance name>/props` (this folder will be created when the broker started), for example `C:\MessageQueue-4.5\var\mq\instances\imqbroker\props`
3. open the configuration file `config.properties` with a text editor
4. add the following lines to configure the Stomp adapter with a test admin account:

```
imq.bridge.admin.user=admin
imq.bridge.admin.password=admin
imq.bridge.activelist=stomp
imq.bridge.enabled=true
```

Save the file. You are ready to start the broker now with Stomp support.



```
C:\Java\MessageQueue4_5_b28\mq\bin\imqbrokerd.exe
[10/Feb/2011:11:29:53 CET]
=====
Open Message Queue 4.5
Oracle
Version: 4.5 <Build 28-b>
Kompilierung: Sat 01/29/2011

Copyright (c) 2010, Oracle and/or its affiliates. All rights reserved.
=====
Java-Runtime: 1.6.0_07 Sun Microsystems Inc. C:\Program Files\Java\jdk1.6.0_07\j
re
[10/Feb/2011:11:29:53 CET] IMQ_HOME=C:\Java\MessageQueue4_5_b28\mq
[10/Feb/2011:11:29:53 CET] IMQ_UARHOME=C:\Java\MessageQueue4_5_b28\var\mq
[10/Feb/2011:11:29:53 CET] Windows Vista 6.0 x86 mj-PC (2 cpu) mj
[10/Feb/2011:11:29:53 CET] Größte Java-Heap: max=189376k, aktuell=18176k
[10/Feb/2011:11:29:53 CET] Argumente:
[10/Feb/2011:11:29:53 CET] [B1060]: Persistente Daten laden...
[10/Feb/2011:11:29:53 CET] Eingebauten, datei-basierten persistenten Speicher ve
rwenden: C:\Java\MessageQueue4_5_b28\var\mq\instances\imqbroker\
[10/Feb/2011:11:29:53 CET] [B1270]: Meldungen aus Transaktionsprotokolldatei wer
den verarbeitet...
[10/Feb/2011:11:29:53 CET] [B1039]: Broker "imqbroker@mj-PC:7676" bereit.
10.02.2011 11:29:53
INFO: STOMP-Server-Protokoll-Domäne: imq.bridge.stomp
10.02.2011 11:29:53
INFO: STOMP-Server-Protokolldatei: C:\Java\MessageQueue4_5_b28\var\mq\instances\
imqbroker\bridges\stomp\stomp%g.log[0,1]
10.02.2011 11:29:53
INFO: Grizzly NIO-Struktur 1.9.27 initialisieren
10.02.2011 11:29:53
INFO: STOMP-Server TCP-Transport auf mj-PC:7672 starten
10.02.2011 11:29:53
INFO: GRIZZLY0001: Starting Grizzly Framework 1.9.27 - 10.02.11 11:29
10.02.2011 11:29:53
INFO: STOMP-Server wurde erfolgreich mit -1-Leser-Threads gestartet.
```

As you can see, the Stomp bridge is running and using TCP port 7672. The Habari OpenMQ Client library will use port 7672 by default.

## Test the Stomp connection

To test the Stomp connection, you can use the ProducerTool demo application in the directory <Habari>\demo\producertool.

If you start the ProducerTool without command line parameters, it will send 10 messages to the broker. For a description of the parameters, please see the readme.txt file in the ProducerTool directory or the chapter ProducerTool on page 41.

## Adding user accounts

The command line tool imqusermgr can be used for user administration. Example:

```
imqusermgr add -u name -p pass
```

---

## GlassFish v3 configuration

In the default installation of GlassFish v3, OpenMQ is not started automatically when the broker starts, so the message broker configuration file for the default domain "domain1" (glassfish/domain1/imq/instances/imqbroker/props/config.properties for example) is not present yet.

Follow these steps to initialize OpenMQ in GlassFish v3:

- **only for GlassFish v3.0** (no longer required in GlassFish v3.1): edit the domain configuration file domain.xml in the config folder of the default domain glassfish\domains\domain1\config to deactivate lazy initialization:

```
<jms-service default-jms-host="default_JMS_host" type="EMBEDDED">
  <jms-host host="localhost" name="default_JMS_host" lazy-init="false" />
</jms-service>
```

- start GlassFish
- run `asadmin jms-ping`
- edit the `config.properties` file to activate Stomp
- restart GlassFish

# Dependencies

---

## Requirements

### Development Environment

- Embarcadero Delphi 6 or higher
- Free Pascal

### Message Broker

- Open Message Queue 4.4 or 4.5
- Java Runtime Environment 1.6

### TCP/IP Communication Library

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

### Internet Direct (Indy)

Subversion repository access:

<https://svn.atozed.com:444/svn/Indy10/trunk>

Inofficial snapshots:

<http://indy.fulgan.com/ZIP>

### Synapse

Subversion repository access:

<https://synalist.svn.sourceforge.net/svnroot/synalist/trunk/>

---

## 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 XE and Free Pascal.

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

- Indy 10.5.8

#### Synapse

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

The library has been tested with these versions of Synapse:

- Release 39

# 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 run time.

## 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 run time by setting the adapter class (either by its name or by its type).

## Available Communication Adapters

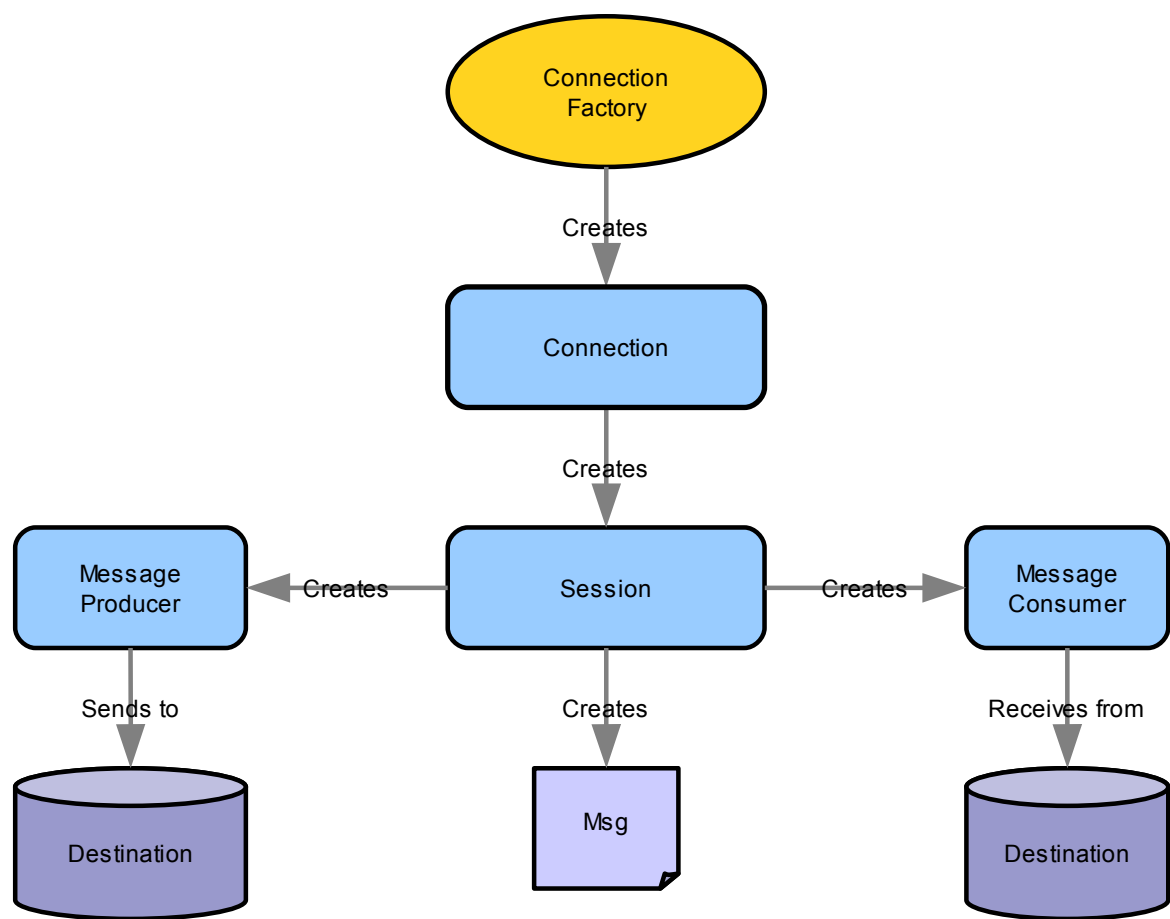
The Habari OpenMQ Client libraries includes two adapters for TCP/IP libraries, one for Indy (Internet Direct) and one for Synapse.

## Overview: Adapter classes and units

Adapter	Unit name
TBTCommAdapterIndy	BTCommAdapterIndy
TBTCommAdapterSynapse	BTCommAdapterSynapse
TBTCommAdapterIndySSL (beta)	BTCommAdapterIndySSL

## The JMS API Programming Model

The Sun online documentation contains a description of the JMS API Programming model:  
<http://download.oracle.com/javase/5/tutorial/doc/bnceh.html>



The JMS API Programming Model: Overview

# Tutorials

## Habari Quick Start Tutorial

This tutorial provides a very simple and quick introduction to the Habari client library by walking you through the creation of a simple "Hello World" application. Once you are done with this tutorial, you will have a general knowledge of how to create and run Habari applications.

This tutorial takes less than 10 minutes to complete.

**To complete this tutorial, you need the following software and resources:**

<u>Software or Resource</u>	<u>Version required</u>
Delphi	2009 <sup>1</sup>
Synapse	Revision 39
OpenMQ	Version 4.4

## Setting up the project

To create a new project:

1. Start the Delphi IDE.
2. In the IDE, choose File > New > VCL Forms Application – Delphi
3. Choose Project > Options ... to open the Project Options dialog
4. In the options tree on the left, select 'Delphi Compiler'
5. Add the source directory of Habari and the Synapse source directory to the 'Search path'
6. Choose Ok to close the Project Options dialog
7. Save the project as HelloOpenMQ

Now the project is created and saved.

You should see the main form in the GUI designer now.

## Adding code to the project

To use the Habari client library, you need to add the required units to the source code.

8. Switch to Code view (F12)

1 Delphi 6 to XE are supported, only the IDE related steps are different



9. Add the required units to the interface uses list:

```
uses
  BTJMSConnection,
  BTJMSInterfaces,
  BTCommAdapterSynapse,
  Windows, Messages, SysUtils, Variants, Classes, Graphics, Controls, Forms,
  Dialogs;
```

10. Compile and save the project.

11. Switch to Design view (F12), go to the Tool palette (Ctrl+Alt+P) and select TButton, add a Button to the form.

12. Double click on the new button to jump to the Button Click handler

13. Add the following code to send the message:

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
begin
  Connection := TBTJMSConnection.MakeConnection('admin', 'admin',
    'stomp://localhost:7672');
  Connection.Start;
  Session := Connection.CreateSession(False, amClientAcknowledge);
  Destination := Session.CreateQueue('TEST.DEFAULT');
  Producer := Session.CreateProducer(Destination);
  Producer.Send(Session.CreateTextMessage('Hello world!'));
  Connection.Close;
end;
```

14. Add a second button and double click on the new button to jump to the Button Click handler

15. Add the following code to receive and display the message:

```
procedure TForm1.Button2Click(Sender: TObject);
var
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Consumer: IMessageConsumer;
  Msg: ITextMessage;
begin
  Connection := TBTJMSConnection.MakeConnection('admin', 'admin',
    'stomp://localhost:7672');
  Connection.Start;
  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('TEST.DEFAULT');
  Consumer := Session.CreateConsumer(Destination);
  Msg := Consumer.Receive(1000) as ITextMessage;
  if Assigned(Msg) then
    ShowMessage(Msg.Text);
```

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

16. Compile and save the project

## Run the demo

- Launch OpenMQ
- Start the application
- Click on Button 1 to send the message to the OpenMQ queue
- Click on Button 2 to receive the message and display it

You can run two instances of the application at the same time, and also on different computers if the IP address of the message broker is used instead of localhost.

## Next steps

You now know how to accomplish some of the most common programming tasks for Habari. The next chapters provide details about the basic interfaces which are the building blocks for message broker clients.

---

## Online Tutorials

### Delphi integration with the GlassFish v3 application server

[This tutorial](#) will guide you through the creation of a simple web application for GlassFish V3 which uses a Servlet to send messages to a message queue on the embedded ActiveMQ broker.

<https://mikejustin.fogbugz.com/default.asp?W11>

The [second part of the tutorial](#) will guide you through the creation of a simple EJB application for GlassFish v3 which uses a Message Driven Bean to receive messages from a message queue on the embedded OpenMQ broker. The Delphi ProducerTool application sends messages to the message queue.

<https://mikejustin.fogbugz.com/default.asp?W12>

# 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,  
    BTJMSConnection,  
    BTJMSInterfaces;  
...
```

### 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.

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

## 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.

## Failover Support

The Failover transport layers reconnect logic on top of the Stomp transport.<sup>2</sup>

The Failover configuration syntax allows you to specify any number of composite URIs. The Failover transport randomly chooses one of the composite URI and attempts to establish a connection to it. If it does not succeed, a new connection is established to one of the other URIs in the list.

Example for a failover URI:

```
failover:(stomp://primary:61613,stomp://secondary:61613)
```

## Transport Options

Option Name	Default Value	Description
initialReconnectDelay	10	How long to wait before the first reconnect attempt (in ms)
maxReconnectDelay	30000	The maximum amount of time we ever wait between reconnect attempts (in ms)
backOffMultiplier	2	The exponent used in the exponential backoff attempts
maxReconnectAttempts	0	If not 0, then this is the maximum number of reconnect attempts before an error is sent back to the client
randomize	True	use a random algorithm to choose the the URI to use for reconnect from the list provided

2 <http://activemq.apache.org/failover-transport-reference.html>

**Example URI:**

```
failover:(tcp://localhost:61616,tcp://remotehost:61616)?  
initialReconnectDelay=100&maxReconnectAttempts=10
```

**Example code:**

```
with TBTJMSConnectionFactory.Create('failover:  
(stomp://primary:61616,stomp://localhost:61613)?maxReconnectAttempts=3') do  
try  
  Conn := CreateConnection;  
  Conn.Start;  
  Conn.Stop;  
  Conn.Close;  
finally  
  Free;  
end;
```

# Destinations

---

## Introduction

The JMS API supports two models:<sup>3</sup>

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.

---

## 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);
```

---

<sup>3</sup> Java Message Service. (2007, November 21). In Wikipedia, The Free Encyclopedia. [http://en.wikipedia.org/wiki/Java\\_Message\\_Service](http://en.wikipedia.org/wiki/Java_Message_Service)

The queue can then be used to send or receive messages using implementations of the `IMessageProducer` and `IMessageConsumer` interfaces. The methods of a queue are defined in the `IQueue` interface and the parent interface `IDestination`. (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. The methods of a topic are defined in the `ITopic` interface and the parent interface `IDestination`. (See next chapter for an example).



## 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.

See also:

[JMS Message Listeners](#) in "The JMS API Programming Model"

[Interface `MessageListener`](#) in "Java Message Service (JMS) API"

---

## 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.

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`.

See also:

[JMS Message Selectors](#) in "The JMS API Programming Model"

[Interface `MessageConsumer`](#) in "Java Message Service (JMS) API"

## Text Messages

---

### Sending Text Messages

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

```
program SendOneMessage;

{$APPTYPE CONSOLE}

uses
  SysUtils,
  BTCommAdapterIndy,
  BTJMSConnection,
  BTJMSInterfaces;

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

begin
  Connection := TBTJMSConnection.MakeConnection('user', 'pass', '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.

## 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
  procedure OnMessage(const Message: IMessage);
end;
```

## Synchronous Receive

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

Example:

```
while I < EXPECTED do
begin
  TextMessage := Consumer.Receive(1000) as ITextMessage;
  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.

## Receive and ReceiveNoWait

There are three different methods for synchronous receive:

- |                         |  |
|-------------------------|--|
| <b>Receive</b>          | The Receive method with no arguments will block (wait until a message is available).   |
| <b>Receive(Timeout)</b> | The Receive method with a timeout parameter will wait for the given time in milliseconds. If no message arrived, it will return nil. |
| <b>ReceiveNoWait</b>    | The ReceiveNowait method will return immediately. If no message arrived, it will return nil.   |

# Bytes Messages

---

## Creation

```
var
  Msg: IBytesMessage;
begin
  ..
  Producer := Session.CreateProducer(OutQueue);
  Msg := Session.CreateBytesMessage;
```

---

## Sending

### Reading Binary Content using BTStreamHelper

The BTStreamHelper unit contains the procedure LoadBytesFromStream which can be used to read a file into a BytesMessage. Example:

```
// create the message
Msg := Session.CreateBytesMessage;

// open a file
FS := TFileStream.Create('filename.dat', fmOpenRead);

try
  // read the file bytes into the message
  LoadBytesFromStream(Msg, FS);

  Size := Length(Msg.Content);

  // display message content size
  WriteLn(IntToStr(Size) + ' Bytes');

finally
  FS.Free;
end;
```

# Object Messages

## Introduction

### Object Serialization

Object serialization is the process of saving an object's state to a sequence of bytes, as well as the process of rebuilding those bytes into a live object at some future time.<sup>4</sup> In messaging applications, object serialization is required to transfer objects between clients, but also to store objects on the broker if they are declared persistent.

## Message Transformers in Habari OpenMQ Client

Transformation	Message Type	Library	Unit
<b>XML</b>	<b>ObjectMessage</b>	<b>OmniXML</b>	BTMessageTransformerXMLOmni
<b>JSON</b>	<b>ObjectMessage</b>	<b>SuperObject</b>	BTMessageTransformerJSONSuperObject

Table 1: Message Transformer Implementations

## Memory Management

### Outgoing Objects

The message transformer will not free objects which have been sent. To release the memory, the application has to explicitly free them when they are no longer used.

### Incoming Objects

The message transformer will create an object instance when a object message has been received. To avoid memory leaks, the application must free this instance when it is no longer in use.

### Assign a Message Transformer

To insert a object decoder / encoder in the message processing chain, create a message transformer instance and assign it to the connection's MessageTransformer property.

The constructor of message transformers for object exchange takes one argument, which is the class of the serialized object. In this example, SamplePojo is the class.

4 <http://java.sun.com/developer/technicalArticles/Programming/serialization/>

```
Connection: IConnection;
...

with (Connection as IMessageTransformerSupport) do
begin
  MessageTransformer := TBTMessageTransformerXMLOmni.Create(SamplePojo);
end;

...
Connection.Start;
```

With version 1.8 and newer, you can also use the helper procedure `SetTransformer` in unit `BTJMSSConnection`:

```
Connection: IConnection;
...

SetTransformer(Connection, TBTMessageTransformerXMLOmni.Create(SamplePojo));

...
Connection.Start;
```

## Create and Send an ObjectMessage

1. create a `IObjectMessage` instance using `ISession#CreateObjectMessage`
2. send the object message to the broker using `IMessageProducer#Send`

```
ObjectMessage := Session.CreateObjectMessage(Instance);
Producer.Send(ObjectMessage);
```

## Complete Example using NativeXml

From `ObjectExchangeTests.pas`.

Send:

```
procedure TObExTestCase.TestXMLNative;
var
  ObjectMessage: IObjectMessage;
  Obj: SamplePojo;
begin
  // send
  Connection := TBTJMSSConnection.MakeConnection;
  try
    SetTransformer(Connection, TBTMessageTransformerXMLNative.Create(SamplePojo));
    Connection.Start;
    Session := Connection.CreateSession(False, amAutoAcknowledge);
    Destination := Session.CreateQueue('TOOL.OBJECT.XML');
    Producer := Session.CreateProducer(Destination);
    Obj := SamplePojo.Create;
    try
      Obj.messageText := 'test';
      Obj.messageNo := 0;
      ObjectMessage := Session.CreateObjectMessage(Obj);
```



```
    ObjectMessage.SetStringProperty(SH_TRANSFORMATION + '-custom',  
        TRANSFORMER_ID_OBJECT_XML); // required for "Delphi Only" object exchange  
    Producer.Send(ObjectMessage);  
finally  
    Obj.Free;  
end;  
finally  
    Connection.Close;  
end;
```

Receive:

```
Connection := TBTJMSConnection.MakeConnection;  
try  
    SetTransformer(Connection, TBTMessageTransformerXMLNative.Create(SamplePojo));  
    Connection.Start;  
    Session := Connection.CreateSession(False, amClientAcknowledge);  
    Destination := Session.CreateQueue('TOOL.OBJECT.XML');  
    Consumer := Session.CreateConsumer(Destination);  
    ObjectMessage := Consumer.Receive(1000) as IObjectMessage;  
    if Assigned(ObjectMessage) then  
        begin  
            ObjectMessage.Acknowledge;  
            Obj := ObjectMessage.GetObject as SamplePojo;  
            try  
                CheckEquals('test', Obj.messageText);  
                CheckEquals(0, Obj.messageNo);  
            finally  
                Obj.Free;  
            end;  
        end;  
    end;  
finally  
    Connection.Close;  
end;  
end;
```

# Durable Subscriptions

---

## Description

If a client needs to receive all the messages published on a topic, including the ones published while the subscriber is inactive, it uses a durable TopicSubscriber. The JMS provider retains a record of this durable subscription and insures that all messages from the topic's publishers are retained until they are acknowledged by this durable subscriber or they have expired.<sup>5</sup> The combination of the clientId and durable subscriber name uniquely identifies the durable topic subscription. After you restart your program and re-subscribe, the Broker will know which messages you need that were published while you were away.

## Creation

The Session interface contains the CreateDurableSubscriber method which creates a durable subscriber to the specified topic. A JMS durable subscriber MessageConsumer is created with a unique JMS clientId and durable subscriber name. Only **one** thread can be actively consuming from a given logical topic subscriber.

**Note:** For durable topic subscriptions you must specify the same clientId on the connection and subscriptionName on the subscribe.

## Example

With the ProducerTool and ConsumerTool demo applications, you can send messages to a durable topic:

```
ProducerTool --MessageCount=1000 --Topic --Persistent -Subject=test-durable
```

and receive them from a client:

```
ConsumerTool --MaximumMessages=1000 --Topic --Subject=test-durable --Durable  
--ClientID=12345 --ConsumerName=12345 -Verbose
```

---

<sup>5</sup> <http://download.oracle.com/javaee/5/api/javax/jms/TopicSession.html>

# Example Applications

## ConsumerTool

The ConsumerTool demo sends test messages to the broker. It is configurable by command line parameters, all are optional:

<b>AckMode</b>	Acknowledgement mode, possible values are: CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE or SESSION_TRANSACTED
<b>ClientId</b>	client id for durable subscriber
<b>ConsumerName</b>	name of the message consumer - for durable subscriber
<b>Durable</b>	true: use a durable subscriber
<b>MaximumMessages</b>	expected number of messages
<b>Password</b>	password
<b>PauseBeforeShutDown</b>	true: wait for key press
<b>ReceiveTimeout</b>	0: asynchronous receive, > 0: consume messages while they continue to be delivered within the given time out
<b>SleepTime</b>	time to sleep after asynchronous receive
<b>Subject</b>	queue or topic name
<b>Topic</b>	true: topic false: queue
<b>Transacted</b>	true: transacted session
<b>URL</b>	server url
<b>User</b>	user name
<b>Verbose</b>	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(const Conn: IConnection;
      const Session: ISession;
      const Consumer: IMessageConsumer); overload;

    procedure ConsumeMessagesAndClose(const Conn: IConnection;
      const Session: ISession;
      const Consumer: IMessageConsumer; const 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,
    BTMgmtInterfaces,
    StrUtils, SysUtils;

{ TConsumerTool }

constructor TConsumerTool.Create;
begin
    ConsumerName := 'Habari';
    FAckMode := amClientAcknowledge;
    MaximumMessages := 10;
    Password := BTJMSConnectionFactory.DEFAULT_PASSWORD;
    Subject := 'TOOL.DEFAULT';
    URL := BTJMSConnectionFactory.DEFAULT_BROKER_URL;
    User := BTJMSConnectionFactory.DEFAULT_USER;
    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;
                        end
                    else
                        begin
                            Msg := '';
                        end
                end
            else
                begin
                    Msg := 'Unknown message type';
                end
            end;
        except
            Msg := 'Exception occurred';
        end;
    end;
end;
```

```
        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;

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

end;

procedure TConsumerTool.ConsumeMessagesAndClose(const Conn: IConnection; const
    Session:
    ISession; const 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;
        end;

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

```
procedure TConsumerTool.ConsumeMessagesAndClose(const Conn: IConnection; const
  Session:
  ISession; const Consumer: IMessageConsumer; const 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: IConnectionFactory;
  Connection: IConnection;
  Destination: IDestination;
  LibraryInfoProvider: IClientLibraryInfoProvider;
  LibInfo: IClientLibraryInfo;
begin
  TCommandLineSupport.Configure(Self);

  Running := True;

  ConnectionFactory := TBTJMSConnectionFactory.Create(User, Password, URL);
  if Supports(ConnectionFactory, IClientLibraryInfoProvider, LibraryInfoProvider) then
    begin
      LibInfo := LibraryInfoProvider.ClientLibraryInfo;
      WriteLn(LibInfo.LibraryName + ' ' + LibInfo.LibraryVersion
        + ' ' + LibInfo.LibraryCopyright);
    end;

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

  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
  begin
    Consumer.SetMessageListener(Self);
    while True do
      Sleep(10); // run forever
    end
  else
    ConsumeMessagesAndClose(Connection, Session, Consumer, ReceiveTimeOut);
  end;
end;

end.
```



## ProducerTool

The ProducerTool demo is configurable by command line parameters, all are optional:

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

## 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;
        FPassword: string;
        FUser: string;

        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 Password: string read FPassword write FPassword;
        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 User: string read FUser write FUser;
        property Verbose: Boolean read FVerbose write FVerbose;

    end;

implementation

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

{ TProducerTool }

constructor TProducerTool.Create;
begin
    MessageCount := 10;
    MessageSize := 255;
    Password := BTJMSConnectionFactory.DEFAULT_PASSWORD;
    Subject := 'TOOL.DEFAULT';
```

```
URL := BTJMSConnectionFactory.DEFAULT_BROKER_URL;
User := BTJMSConnectionFactory.DEFAULT_USER;
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
  ConnectionFactory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
  LibraryInfoProvider: IClientLibraryInfoProvider;
  LibInfo: IClientLibraryInfo;
begin
  TCommandLineSupport.Configure(Self);

  ConnectionFactory := TBTJMSConnectionFactory.Create(User, Password, URL);
  if Supports(ConnectionFactory, IClientLibraryInfoProvider, LibraryInfoProvider) then
  begin
    LibInfo := LibraryInfoProvider.ClientLibraryInfo;
    WriteLn(LibInfo.LibraryName + ' ' + LibInfo.LibraryVersion
      + ' ' + LibInfo.LibraryCopyright);
  end;

  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 := ConnectionFactory.CreateConnection;
  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
            Msg := CreateMessageText(I);
            TextMessage := Session.CreateTextMessage(Msg);
            if Verbose then
                begin
                    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.
```

# 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.<sup>6</sup>

### JMS properties for outgoing messages

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

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

### JMS properties for incoming messages

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

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

6 <http://download.oracle.com/javase/5/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:

- 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.

## Useful Units

---

### BTStreamHelper

This unit contains the procedure `LoadBytesFromStream` which can be used to read a file into a `BytesMessage`.

Example:

```
Msg := Session.CreateBytesMessage;

FS := TFileStream.Create('filename.dat', fmOpenRead);
try
  LoadBytesFromStream(Msg, FS);
  Size := Length(Msg.Content);
  WriteLn(IntToStr(Size) + ' Bytes');
finally
  FS.Free;
end;

Producer.Send(Msg);
```

---

### BTJavaPlatform

This unit contains some helper functions for Java dates. Java dates are `Int64` values based on the Unix date.

```
function JavaDateToTimeStamp(const JavaDate: Int64): TDateTime;
```

```
function TimeStampToJavaDate(const TimeStamp: TDateTime): Int64;
```

## Known Limitations

---

### Sessions

#### Acknowledgement Modes

Acknowledgment mode **"amDupsOkAcknowledge"** is unsupported.

Acknowledgment mode **"amAutoAcknowledge"** may cause message loss if you do not read all remaining messages in the Queue before closing the connection.

Background information: The Indy and Synapse libraries reads messages into an client-side buffer, and even when the client does not fetch the messages from the buffer (using one of the "Receive" methods), the server will handle them as 'delivered' and acknowledged. If the client reconnects, these messages will not be sent again.

---

### Messages

#### Message Types

OpenMQ only supports TextMessage and BytesMessage message types.

It is not possible to detect the message type for incoming messages (sent from the OpenMQ broker to the Stomp client) because there is no indicator in the message for the type. As a workaround, the Habari library for OpenMQ uses a non-standard Stomp header to indicate the message type:

- `messageType=text`
- `messageType=byte`

This header will be included in outgoing messages (from the Habari Stomp client to the message broker) and helps the receiver to identify the message type.

ObjectMessage support in the library is provided based on a message transformer architecture. However, native Java objects sent from Java clients will not work because OpenMQ will not send them to Stomp clients.

#### 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.

---

## Security

### Default account

The library currently uses admin / admin for the default user name / password.

# References

## Message Broker

Home page <https://mq.dev.java.net/>

## IDE

Embarcadero Delphi <http://www.embarcadero.com/products/delphi>  
Lazarus <http://www.lazarus.freepascal.org/>

## JMS

JMS Specification <http://www.oracle.com/technetwork/java/jms/index.html>

## Stomp

Project home <http://stomp.codehaus.org/>

## Communication Libraries

Synapse <http://www.synapse.ararat.cz>  
Internet Direct (Indy) <http://www.indyproject.org/>  
Indy Snapshot <http://indy.fulgan.com/ZIP>

## Logging

Log4D <http://log4d.sourceforge.net/>

## Habari OpenMQ Client License

Habari OpenMQ Client (c) 2009-2011 Michael Justin

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 habarisoft copyright notice that appears on the Software; (ii) you may not remove or alter any habarisoft copyright, trademark or other proprietary rights notice contained in any portion of habarisoft libraries, source code, Redistributables or other files that bear such a notice; (iii) habarisoft 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 habarisoft for such services or assistance; (iv) you will indemnify and hold habarisoft, 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 habarisoft'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 habarisoft libraries, source code, Redistributables and other files remain habarisoft's exclusive property. Regardless of any modifications that you make, you may not distribute any files (particularly habarisoft 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 merchantability 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

## Synapse

The following software may be included in this product: Ararat Synapse; Use of any of this software is governed by the terms of the license below:

```
| Copyright (c)1999-2008, Lukas Gebauer |
| 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 Lukas Gebauer 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 THE COPYRIGHT HOLDERS AND CONTRIBUTORS "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. |
|=====|
| The Initial Developer of the Original Code is Lukas Gebauer (Czech Republic). |
| Portions created by Lukas Gebauer are Copyright (c)1999-2008. |
| All Rights Reserved. |
```

## 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.

---

## SuperObject

```
*                               Super Object Toolkit
*
* Usage allowed under the restrictions of the Lesser GNU General Public License
* or alternatively the restrictions of the Mozilla Public License 1.1
*
* 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@gmail.com>
* Web site   : http://www.progdigy.com
*
* This unit is inspired from the json c lib:
*   Michael Clark <michael@metaparadigm.com>
*   http://oss.metaparadigm.com/json-c/
```

---

## Log4D

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/MPL-1.1.html>

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.

---

## NativeXml

Copyright (c) 2003 - 2011 Simdesign BV. All rights reserved.

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

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. 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.

THIS SOFTWARE IS PROVIDED BY SIMDESIGN BV "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 SIMDESIGN BV 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.

# Release Notes

---

## Version 1.8

Released June 14, 2011

### New

<b>Durable Subscribers</b>	If a client needs to receive all the messages published on a topic, including the ones published while the subscriber is inactive, it uses a durable TopicSubscriber. This version introduces support for Durable Subscribers. For details see chapter Durable Subscriptions
<b>NativeXml</b>	Support for the NativeXml open source XML parser library, class TBTMessageTransformerXMLNative for IObjectMessage
<b>OpenMQ 4.5</b>	Tested with OpenMQ version 4.4u1, 4.4u2 and 4.5
<b>/autolaunch</b>	If a command line parameter /autolaunch is specified, the DUnit test program will launch the OpenMQ brokers 4.4u1, 4.4u2 and 4.5 automatically in the test suite setup and terminate the broker in the teardown stage
<b>NoLocal</b>	If this parameter of ISession#CreateConsumer is set, it inhibits the delivery of messages published by its own connection.
<b>Transformer</b>	A helper method, SetTransformer, can be used to set a message transformer on a connection
<b>Object Exchange</b>	Unit tests for object messages (unit ObjectExchangeTests) have been added to the test suites, they replace the omnixml and superobject demo applications

### Changed

<b>Refactoring</b>	Refactored to new broker-independent units BTSerialIntf, BTSessionIntf instead of unit BTOMQInterfaces
<b>Frame Decoder</b>	The new unit BTStompDecoder contains a new Stomp Frame decoder implementation, it can be enabled with the conditional symbol HABARI_USE_TBYTES
<b>Performance Demo</b>	Shared code with Habari ActiveMQ Client and Habari OpenMQ Client, display transfer speed in msgs/s
<b>Indy 10.5.8</b>	Tested with Indy 10.5.8 revision 4639
<b>FPC 2.4.4</b>	Build tested with Free Pascal 2.4.4
<b>Logging</b>	Removed unused logging units (BTLog etc)



<b>TransformationId</b>	The client verifies if the transformation id of the message transformer matches the 'transformation' header of incoming object messages
<b>TTransformable</b>	Class TTransformable is deprecated
<b>Stomp headers</b>	Repeated headers will be ignored, only the first value will be used
<b>BTypes</b>	Uses System.SetString for TBytes -> RawByteString conversion and SysUtils.ToBytes for RawByteString -> TBytes conversion

---

## Version 1.7

Released March 8, 2011

### New

<b>OpenMQ 4.5b29</b>	Tested with OpenMQ version 4.4u1, 4.4u2 and 4.5b29
<b>Failover transport</b>	The Failover transport layers reconnect logic on top of the Stomp transport. The URL for a connection factory can be configured with failover:(uri1,...,uriN)?transportOptions
<b>CreateMessage</b>	Function Session#CreateMessage returns a IMessage object

### Changed

<b>ReadOneMessage</b>	The internal method ReadOneMessage in TBTCCommAdapterIndy has a new parameter, ATimeout
<b>OnVerifyPeer</b>	BTCCommAdapterIndySSL updated to use the method signature of current Indy version with an additional AError parameter
<b>Log4D</b>	Log4D updated to revision 37
<b>SuperObject</b>	SuperObject updated to revision 39
<b>GUI demo fix</b>	Fixed an AV which occurred when the demo program was compiled without assertions
<b>ERROR frame</b>	The body of ERROR frames is included in the exception message

---

## Version 1.6

Released December 14, 2010

### New

<b>IConnectionInfo</b>	New interface in BtMgmtInterfaces which provides connection state information
<b>OpenMQ 4.5b19</b>	Tested with OpenMQ version 4.5b19 (Milestone 6 build for GlassFish 3.1 project)

<b>Single Source</b>	All versions of the Habari JMS Client library share the source code for the basic demo applications ConsumerTool, ProducerTool, DelphiGUI and HabariChat
<b>FPC 2.4.2</b>	Tested with Free Pascal 2.4.2

Fixed

<b>Thread</b>	Fixed compiler warning about deprecated Thread methods Resume and Suspend
<b>Connection</b>	Fixed code to avoid a EIdConnClosedGracefully exception in BTStompCustomClient

Examples

<b>DelphiGUI logging</b>	Use Log4D for logging
<b>DelphiGUI dialog</b>	Added a Connection Factory configuration dialog
<b>DelphiGUI flicker</b>	Reduced flickering of the Delphi GUI demo application
<b>ExampleQueue</b>	DelphiGUI Demo uses default name 'ExampleQueue' for the JMS destination
<b>Log4D JMSAppender</b>	Provided an example implementation of a JMS log appender for the open source Log4D logging framework (unit LogJMSAppender)

Version 1.5

Released October 14, 2010

New

<b>Delphi XE</b>	Ready for Delphi XE
<b>Log4D Library</b>	The Log4D logging library is now included and used when HABARI_LOGGING is defined
<b>Unit Tests</b>	Unit tested with Open Message Queue version 4.4u1, 4.4u2 and 4.5 b16
<b>Library Information</b>	The Connection Factory class now implements a new interface, IClientLibraryInfoProvider
<b>Indy 10.5.8</b>	Tested with Indy 10.5.8

Changed

<b>doxygen</b>	Updated to doxygen 1.7.1, fixes
<b>Subscription Header</b>	The library now removes the Stomp header 'subscription' which is included in incoming messages. It is added by the broker so that the client knows which subscription the message relates to.

<b>Documentation</b>	Documentation updates and fixes
----------------------	---------------------------------

# Version 1.4

Released March 30, 2010

## New

<b>Logging Switch</b>	If the HABARI_LOGGING compiler condition is set, logging code will be included. This will reduce code size in production and also improve performance
<b>IPv6</b>	Prepared for IPv6 on Linux
<b>UMS Monitor demo</b>	A new demo application shows how message broker information can be retrieved over HTTP and the imqums web application

## Changed

<b>Synapse</b>	Improved performance for Synapse communication adapter
<b>Free Pascal</b>	Tested with Free Pascal 2.5.1
<b>SuperObject</b>	Update to SuperObject 1.2.4

# Version 1.3

Released January 12, 2010

## New

<b>GlassFish v3</b>	The documentation includes configuration information for OpenMQ embedded in the GlassFish v3 application server and links to online tutorials which describe Delphi and Java integration with GlassFish and the NetBeans IDE
---------------------	--

## Changed

<b>GlassFish support</b>	Tested with GlassFish v2.1.1 and GlassFish v3
<b>OpenMQ version</b>	Tested with OpenMQ build 4.4 u1 Final released Dec 4, 2009
<b>SuperObject version</b>	Updated to SuperObject 1.2.2 released Dec 22, 2009
<b>Linux</b>	Tested on Ubuntu 9.10 with Free Pascal 2.2
<b>Multithreading demo</b>	The performance test demo application can create up to 20 threads

**Fixed**

**ConnectTimeout** Fixed default value for the ConnectTimeout property

---

**Version 1.2**

Released November 10, 2009

**New**

**ConnectTimeout** New property in BTJMSConnection and BTJMSConnectionFactory

**Changed**

**OpenMQ version** Tested with OpenMQ build 4.4u1 b3 released October 27, 2009. According to the release notes, this is release candidate 1 for planned inclusion in GlassFish v3. (There will probably be another release candidate) – see: <http://download.java.net/mq/openmq/4.4u1/b3-rc1/changes.html>

**Indy 10 version** Tested with Indy 10.5.7 revision revision 3865. Fixed Unicode conversion error in Indy communication adapter method ReadMessageBuffer (below Delphi 2009).

**Synapse version** Updated to release 39

---

**Version 1.1**

Released September 8, 2009

**New**

**ReceiveNoWait** MessageConsumer now provides three methods to read messages from a destination. The new method ReceiveNoWait is non-blocking (it will immediately return nil if there is no message), it replaces the old Receive method. Receive will now block until a message arrives.

**Delphi 2010** Tested with Delphi 2010 (all unit tests passed)

**Changed**

**OpenMQ version** Tested with OpenMQ build 4.4 b15 released August 14, 2009.

**Indy 10 version** Updated to Indy 10.5.6.

**Demo applications** Demo applications will report memory leaks on shutdown.

**Minor changes** JMS interface and Stomp command key cleanup.

---

# Version 1.0

Released July 7, 2009

# Index

## Reference

amAutoAcknowledge.....	48	JMSReplyTo.....	45
amDupsOkAcknowledge.....	48	JMSTimestamp.....	45
BTJMSConnection.....	32	LoadBytesFromStream.....	30
BTStreamHelper.....	30	Log4D.....	50
Connection.....	19	Message Consumer.....	25
connection factory.....	19	Message Producer.....	25
ConsumerTool.....	35	MessageListener.....	26, 28
CreateDurableSubscriber.....	34	MessageTransformer.....	31
CreateObjectMessage.....	32	NativeXml.....	55
Failover Support.....	21	Object Message.....	31
IConnection.....	19	OnMessage.....	26, 28
IDestination.....	24, <b>28</b>	point-to-point.....	23
IMessage.....	<b>28</b>	ProducerTool.....	41
IMessageConsumer.....	24, <b>28</b>	publish and subscribe.....	23
IMessageListener.....	<b>28</b>	Queue.....	23
IMessageProducer.....	24, 32	Receive.....	29
Internet Direct (Indy).....	12, <b>13</b> , 50	ReceiveNoWait.....	29
IQueue.....	24	SamplePojo.....	31
ISession.....	32	Session.....	20
ITopic.....	24	SetTransformer.....	32
JMS.....	50	Stomp.....	50
JMS Selector.....	26	SuperObject.....	31, 54
JMSCorrelationID.....	45	Synapse.....	12, <b>13</b> , 50
JMSDeliveryMode.....	45	Text Message.....	27
JMSExpiration.....	45	Topic.....	24
JMSMessageId.....	45	TopicSubscriber.....	34
JMSPriority.....	45	Transacted Sessions.....	21

**Illustrations**

Illustration 1: Shared Business Logic.....6

Illustration 2: Peer to Peer Communication.....7

Illustration 3: Load Balancing.....7