Getting started with

Habari STOMP Client for RabbitMQ

Version 7.0
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.

Trademarks

Habari is a trademark or registered trademark of Michael Justin in Germany and/or other countries. Android is a trademark of Google Inc. Use of this trademark is subject to Google Permissions. The Android robot is reproduced or modified from work created and shared by Google and used according to terms described in the Creative Commons 3.0 Attribution License. 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. IBM and WebSphere are trademarks of International Business Machines Corporation in the United States, other countries, or both. HornetQ, WildFly, JBoss and the JBoss logo are registered trademarks or trademarks of Red Hat, Inc. Mac OS is a trademark of Apple Inc., registered in the U.S. and other countries. Oracle, WebLogic and Java are registered trademarks of Oracle and/or its affiliates. Pivotal, RabbitMQ and the RabbitMQ logo are trademarks and/or registered trademarks of GoPivotal, Inc. in the United States and/or other countries. Other brands and their products are trademarks of their respective holders.

Errors and omissions excepted. Specifications subject to change without notice.
# Contents

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Broker-specific information</td>
<td>7</td>
</tr>
<tr>
<td>Installation</td>
<td>8</td>
</tr>
<tr>
<td>Requirements</td>
<td>8</td>
</tr>
<tr>
<td>Development Environment</td>
<td>8</td>
</tr>
<tr>
<td>TCP/IP Communication Library</td>
<td>8</td>
</tr>
<tr>
<td>Test Suites</td>
<td>8</td>
</tr>
<tr>
<td>Installation steps</td>
<td>8</td>
</tr>
<tr>
<td>Communication Adapters</td>
<td>9</td>
</tr>
<tr>
<td>Introduction</td>
<td>9</td>
</tr>
<tr>
<td>Configuration of communication adapters</td>
<td>9</td>
</tr>
<tr>
<td>Registration of communication adapter class</td>
<td>9</td>
</tr>
<tr>
<td>Available communication adapters</td>
<td>10</td>
</tr>
<tr>
<td>Limitations of the Synapse communication adapter class</td>
<td>10</td>
</tr>
<tr>
<td>The Programming Model</td>
<td>11</td>
</tr>
<tr>
<td>New simplified API</td>
<td>11</td>
</tr>
<tr>
<td>Tutorials</td>
<td>12</td>
</tr>
<tr>
<td>Quick Start Tutorial</td>
<td>12</td>
</tr>
<tr>
<td>Setting up the project</td>
<td>12</td>
</tr>
<tr>
<td>Adding code to the project</td>
<td>12</td>
</tr>
<tr>
<td>Run the demo</td>
<td>14</td>
</tr>
<tr>
<td>Check for memory leaks</td>
<td>14</td>
</tr>
<tr>
<td>Tutorial source code</td>
<td>14</td>
</tr>
<tr>
<td>Connection Factory</td>
<td>15</td>
</tr>
<tr>
<td>Overview</td>
<td>15</td>
</tr>
<tr>
<td>Creation and configuration</td>
<td>15</td>
</tr>
<tr>
<td>Connection URL parameters</td>
<td>17</td>
</tr>
<tr>
<td>Heart-beating Support</td>
<td>17</td>
</tr>
<tr>
<td>Failover Support</td>
<td>17</td>
</tr>
<tr>
<td>Failover Transport Options</td>
<td>17</td>
</tr>
<tr>
<td>Receipt Support</td>
<td>18</td>
</tr>
<tr>
<td>SUBSCRIBE Receipt</td>
<td>19</td>
</tr>
<tr>
<td>UNSUBSCRIBE Receipt</td>
<td>19</td>
</tr>
<tr>
<td>SEND Receipt</td>
<td>19</td>
</tr>
<tr>
<td>DISCONNECT Receipt</td>
<td>20</td>
</tr>
<tr>
<td>Connections and Sessions</td>
<td>21</td>
</tr>
<tr>
<td>Connections use Stomp 1.2 by default</td>
<td>21</td>
</tr>
<tr>
<td>Step-by-Step Example</td>
<td>21</td>
</tr>
<tr>
<td>Overview</td>
<td>21</td>
</tr>
<tr>
<td>Add required units</td>
<td>21</td>
</tr>
<tr>
<td>Creating a new Connection</td>
<td>22</td>
</tr>
<tr>
<td>Connection URL Parameters</td>
<td>22</td>
</tr>
<tr>
<td>Creating a Session</td>
<td>22</td>
</tr>
</tbody>
</table>
Broker-specific information

For broker-specific notes, please read chapter Broker-specific notes on page 78 ff.
Installation

Requirements

Development Environment

- **Embarcadero Delphi** 2009 Update 4 or higher
- or -
  - **Free Pascal** 3.2.0 or higher

TCP/IP Communication Library

- **Internet Direct (Indy) 10.6** (recommended)
- or -
  - **Synapse** Release 40 (deprecated)

Test Suites

- The DUnit test suite requires the Delphi 2009 version of DUnit for compilation.
- The FPCUnit test suite requires Lazarus 2.0.12 or newer to run.

Installation steps

The installer application will guide you through the installation process.
By default Habari STOMP Client for RabbitMQ will be installed in the folder

C:\Users\Public\Documents\Habarisoft\habari-rabbitmq-7.0

---

1 Only release 40 of Ararat Synapse is used for Habari Client library development and tests
Breaking Changes in Version 7.0

Major changes
Some class declarations moved to a new unit.

• Unit BTJMSTypes has been removed
  ◦ Class EConnectionFailedException moved to unit BTTypes
  ◦ Class EMessageNotWriteableException moved to unit BTTypes
  ◦ Class EJMSException was renamed to EMQException and moved to unit BTTypes
  ◦ Class TBTDestination moved to new unit BTDestination

Potentially breaking changes

• If the server does not response with a valid STOMP message to the CONNECT message, the client raises an exception (EConnectionFailedException) and closes the socket connection.

Minor changes
In addition, all code related to the old map / object transformation API has been removed. (It was already commented out in version 6.12).
Communication Adapters

Introduction
Habari STOMP Client for RabbitMQ uses communication adapters as an abstraction layer for the TCP/IP library. All connections create their own internal instance of the adapter class.

Configuration of communication adapters
No configuration is required for the communication adapters. Applications specify communication and connection options in URL parameters or connection class properties or connection factory settings.

Registration of communication adapter class
A communication adapter implementation can be prepared for usage by simply adding its Delphi unit to the project.

Code example

```pascal
program ClientUsingIndy;

uses
  BTCommAdapterIndy, // use Internet Direct (Indy)
  BTConnectionFactory, BTJMSInterfaces,
  SysUtils;

...'''

Behind the scenes, the communication adapter class will register itself with the communication adapter manager in the BTAdapterRegistry unit.

Default adapter class
Applications typically use only one of the available communication adapter classes for all connections.

The library allows to register two or more adapter classes and switch at run-time, using methods in the adapter registry in unit BTAdapterRegistry - this feature is mainly for tests and demonstration purposes.

If more than one communication adapter is in the project, the first adapter class in the list will be the default adapter class. Example:
Code example

```pascal
program ClientUsingIndyOrSynapse;
uses
  BTCommAdapterIndy, // use Internet Direct (Indy) as default adapter class
  BTCommAdapterSynapse, // and register the Synapse adapter class
  BTConnectionFactory, BTJMSInterfaces,
  SysUtils;
...```

The default adapter class can be changed at run-time by setting the adapter class either by its name or by its class type.

Available communication adapters

The library includes two adapter classes for TCP/IP libraries, one for Indy (Internet Direct) and one for Synapse.

<table>
<thead>
<tr>
<th>Adapter Class</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBTCCommAdapterIndy</td>
<td>BTCommAdapterIndy</td>
</tr>
<tr>
<td>TBTCCommAdapterSynapse</td>
<td>BTCommAdapterSynapse</td>
</tr>
</tbody>
</table>

Table 1: Communication Adapters

Limitations of the Synapse communication adapter class

- The Synapse library does not support the ConnectTimeout property in synchronous socket operation mode, as connect timeouts are handled by the operating system. Indy uses a background thread to abort the connect operation.\(^2\)
- Release 40 of Ararat Synapse is used for Habari Client library development and tests. This is the last announced release, dated April 24, 2012. This release is compatible with Delphi versions before XE4\(^3\). If you use a newer release of Ararat Synapse, please let me know if you encounter any API incompatibilities or other problems.


The Programming Model

Habari Client libraries use a programming model which is based on message producers and message consumers, sessions, connections and connection factories.

The basic API is the same for all library versions to allow easy migration between supported message brokers (with the exception of broker-specific features).

Illustration 1: Programming Model

New simplified API

See also: section Simplified API on page 44.
Quick Start Tutorial

This tutorial provides a very simple and quick introduction to Habari STOMP Client for RabbitMQ 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.

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 STOMP Client for RabbitMQ and the Indy source directories to the 'Search path'
6. Choose Ok to close the Project Options dialog
7. Save the project as HelloMQ

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 STOMP Client for RabbitMQ library, you need to add the required units to the source code.

8. Switch to Code view (F12)
9. Add the required units to the interface uses list:

```
uses
   BTConnectionFactory,
   BTJMSInterfaces,
```

Code example
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:

```pascal
procedure TForm1.Button1Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
begin
  Factory := TBTConnectionFactory.Create('stomp://localhost');
  Connection := Factory.CreateConnection;
  Connection.Start;
  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMQ');
  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:

```pascal
procedure TForm1.Button2Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Consumer: IMessageConsumer;
  Msg: ITextMessage;
begin
  Factory := TBTConnectionFactory.Create('stomp://localhost');
  Connection := Factory.CreateConnection;
  Connection.Start;
  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMQ');
  Consumer := Session.CreateConsumer(Destination);
  Consumer.Terminated := false;
  while True do
  begin
    if Consumer.TryReceive(Msg) then
    begin
      Label1.Caption := Message.MSG;--
      break;
    end;
  end;
end;
```

BTCommAdapterIndy,
// auto-generated unit references
Windows, Messages, SysUtils, ...

Code example

```
procedure TForm1.Button1Click(Sender: TObject);
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
begin
  Factory := TBTConnectionFactory.Create('stomp://localhost');
  Connection := Factory.CreateConnection;
  Connection.Start;
  Session := Connection.CreateSession(False, amAutoAcknowledge);
  Destination := Session.CreateQueue('HelloMQ');
  Producer := Session.CreateProducer(Destination);
  Producer.Send(Session.CreateTextMessage('Hello world!'));
  Connection.Close;
end;
```
15. Destination := Session.CreateQueue('HelloMQ');
    Consumer := Session.CreateConsumer(Destination);
    Msg := Consumer.Receive(1000) as ITextMessage;
    if Assigned(Msg) then
        ShowMessage(Msg.Text)
    else
        ShowMessage('Error: no message received');
    Connection.Close;
end;

16. Compile and save the project

**Run the demo**

- Launch the message broker
- Start the application
- Click on Button 1 to send the message to the 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.

**Check for memory leaks**

To verify that the program does not cause memory leaks, insert a line in the project file HelloMQ.dpr:

```
Code example
program HelloMQ;
uses
    Forms,
    Unit1 in 'Unit1.pas' {Form1};
{$R *.res}
begin
    ReportMemoryLeaksOnShutdown := True; // check for memory leaks
    Application.Initialize;
    Application.MainFormOnTaskbar := True;
    Application.CreateForm(TForm1, Form1);
    Application.Run;
end.
```

**Tutorial source code**

The tutorial source code is included in the demo folder. It does not include a .proj file so you still need to add the Habari and Indy source paths to the project options.
Connection Factory

Overview
A connection factory is an object which holds all information required for the creation of a connection objects.

A factory instance is created and configured only once. It then may be used to create actual connection objects when needed. For example, a worker thread may create the connection factory once at program start-up and use it to create a new connection object whenever a connection failure occurred.

Creation and configuration
The code example below shows a helper function which creates a connection factory, and returns it using the interface type IConnectionFactory.

The factory will be freed automatically when there are no more references to it.

Code example

```pascal
function TExample.CreateConfiguredFactory: IConnectionFactory;
var
  Factory: IConnectionFactory;
begnin
  // -----------------------------------------------------------
  // create an instance
  // -----------------------------------------------------------

  // -----------------------------------------------------------
  // return the instance
  // -----------------------------------------------------------
  Result := Factory;
edn;
```

This code example is useful for most simple client applications. However, because the local factory variable is declared as IConnectionFactory, advanced configuration properties in the class TBTConnectionFactory such as ClientID and SendTimeout are not accessible.

To access them, declare the local factory with the class type as shown in the next example:
function TExample.CreateConfiguredFactory: IConnectionFactory;
var
    Factory: TBTConnectionFactory;
begin
    // ------------------------------------------------------------
    // create and assign to local variable
    // ------------------------------------------------------------
    Factory := TBTConnectionFactory.Create;
    // ------------------------------------------------------------
    // additional configuration
    // -------------------------------------------------------------
    Factory.BrokerURL := 'broker.example.com';
    Factory.UserName := 'guest';
    Factory.Password := 'guest';
    Factory.ClientID := 'myclientId';
    Factory.SendTimeOut := 10000;
    Factory.ConnectTimeOut := 10000; // Indy only
    // -------------------------------------------------------------
    // return the configured factory
    // -------------------------------------------------------------
    Result := Factory;
end;

Warning: if the method signature is changed to return the class TBTConnectionFactory
instead, a memory leak will occur.

function TExample.Run;
var
    F: IConnectionFactory;
    C: IConnection;
begin
    // ------------------------------------------------------------
    // get a factory and use it to create a connection object
    // ------------------------------------------------------------
    F := CreateConfiguredFactory;
    C := F.CreateConnection;
    // ------------------------------------------------------------
    // start and use the connection
    // ------------------------------------------------------------
    C.Start;
    ...
    // ------------------------------------------------------------
    // close the connection
    // ------------------------------------------------------------
    C.Close;
end;
Connection URL parameters

Heart-beating Support
STOMP 1.1 introduced heart-beating, its configuration is covered in the chapter Stomp 1.2

Failover Support
The Failover transport layers reconnect logic on top of the Stomp transport.

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)
```

Failover Transport Options

<table>
<thead>
<tr>
<th>Option Name</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>initialReconnectDelay</td>
<td>10</td>
<td>How long to wait before the first reconnect attempt (in ms)</td>
</tr>
<tr>
<td>maxReconnectDelay</td>
<td>30000</td>
<td>The maximum amount of time we ever wait between reconnect attempts (in ms)</td>
</tr>
<tr>
<td>backOffMultiplier</td>
<td>2.0</td>
<td>The exponent used in the exponential backoff attempts</td>
</tr>
<tr>
<td>maxReconnectAttempts</td>
<td>-1</td>
<td>-1 is default and means retry forever, 0 means don’t retry (only try connection once but no retry) If set to &gt; 0, then this is the maximum number of reconnect attempts before an error is sent back to the client</td>
</tr>
<tr>
<td>randomize</td>
<td>true</td>
<td>use a random algorithm to choose the the URI to use for reconnect from the list provided</td>
</tr>
</tbody>
</table>

Table 2: Failover Transport Options

Example URI:

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

### Code example

```go
localhost:61613)?maxReconnectAttempts=3&randomize=false') do
  try
    Conn := Factory.CreateConnection;
    Conn.Start;
    ...
    Conn.Stop;
  finally
    Conn.Close;
  end;
```

---

**Receipt Support**

The STOMP standard supports receipt messages since version 1.0:

"Any client frame other than CONNECT may specify a receipt header with an arbitrary value. This will cause the server to acknowledge receipt of the frame with a RECEIPT frame which contains the value of this header as the value of the receipt-id header in the RECEIPT packet."

With Habari STOMP Client for RabbitMQ, client applications may configure receipt headers for the frame types listed below.

After the STOMP frame has been sent to the broker, the client library waits for the RECEIPT frame for a defined time, which may be configured per frame type. If the broker does not send a receipt within the time-out interval, the client library will raise an exception. If the client receives a receipt with the wrong receipt-id header, it will raise an exception.

### Receipt Support Parameters

<table>
<thead>
<tr>
<th>STOMP frame</th>
<th>Parameter</th>
<th>Example URL</th>
</tr>
</thead>
<tbody>
<tr>
<td>SUBSCRIBE</td>
<td>subscribe.receipt</td>
<td>stomp://localhost?subscribe.receipt=true</td>
</tr>
<tr>
<td>UNSUBSCRIBE</td>
<td>subscribe.receipt</td>
<td>stomp://localhost?unsubscribe.receipt=true</td>
</tr>
<tr>
<td>SEND</td>
<td>send.receipt</td>
<td>stomp://localhost?send.receipt=true</td>
</tr>
</tbody>
</table>

---

5 https://stomp.github.io/stomp-specification-1.0.html
6 https://stomp.github.io/stomp-specification-1.1.html#Header_receipt
7 https://stomp.github.io/stomp-specification-1.2.html#Header_receipt
DISCONNECT | disconnect.receipt | stomp://localhost?disconnect.receipt=true

**SUBSCRIBE Receipt**
To request server receipts for SUBSCRIBE frames, use the optional connection URL parameter, subscribe.receipt.

**Code example**
```
```

If the broker does not send a receipt within the time-out interval, the client library will raise an exception.

**UNSUBSCRIBE Receipt**
To request server receipts for UNSUBSCRIBE frames, use the optional connection URL parameter, unsubscribe.receipt.

**Code example**
```
```

If the broker does not send a receipt within the time-out interval, the client library will raise an exception.

**SEND Receipt**
To request server receipts for SEND frames, use the optional connection URL parameter, send.receipt.

**Code example**
```
```

If the broker does not send a receipt within the time-out interval, the client library will raise an exception.
Note: for additional reliability, the client can use transactional send (see section "Transacted Sessions").

DISCONNECT Receipt

To request server receipts for DISCONNECT frames, use the optional connection URL parameter, disconnect.receipt.

Code example

```java
```

Without this parameter, the client will disconnect the socket connection immediately after sending the DISCONNECT frame to the broker.

With disconnect.receipt=true, the client will send the DISCONNECT frame and then wait for the broker receipt frame. If the broker does not answer, the client library will raise an exception. The client application should treat its messages as undelivered.

Note: for additional reliability, the client can use transactional send (see section "Transacted Sessions"), and message receipts (see section "SEND Receipt").
Connections and Sessions

Connections use Stomp 1.2 by default

Connections use Stomp 1.2 by default since:

- Habari Client for Apache ActiveMQ 5.1
- Habari Client for Apache Artemis 5.1
- Habari Client for RabbitMQ 5.1

With OpenMQ, the library still uses Stomp 1.0. The default protocol version is defined in the BTBrokerConsts unit. The Stomp version may be specified by a connection URL parameter.

Step-by-Step Example

Overview

This example will send a single message to a destination queue (ExampleQueue).

Add required units

Three units are required for this example:

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

The SysUtils unit is necessary for the exception handling.

Code example

```pascal
program SendOneMessage;
{$APPTYPE CONSOLE}
uses
  BTCommAdapterIndy,
  BTConnectionFactory,
  BTJMSInterfaces,
  SysUtils;
```
Creating a new Connection

New connections are created by calling the CreateConnection method of a connection factory.

Code example

```pascal
var
  Factory: IConnectionFactory;
  Connection: IConnection;
  ...
begin
  Factory := TBTConnectionFactory.Create('user', 'password', 'stomp://localhost');
  Connection := Factory.CreateConnection;
  ...
```

- For connection factory creation and configuration options please see chapter "Creation and configuration".
- Since IConnection is an interface type, the connection instance will be destroyed automatically if there are no more references to it in the program.

Connection URL Parameters

Connection URL parameters are documented in chapter "Connection URL parameters" and in chapter "Stomp 1.2".

Creating a Session

To create the communication session,

- declare a variable of type ISession
- use the helper method CreateSession of the connection, and specify the acknowledgment 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.

Code example

```pascal
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.

**Code example**

```go
Destination := Session.CreateQueue('ExampleQueue');
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.

**Code example**

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

**Note:** Close will be called automatically if the connection is destroyed. But because unclosed connections use resources, Close should be called when the connection is no longer needed. When logging is enabled, the connection class will also log a message when a connection is destroyed without calling Close.

---

**Session types overview**

The table below shows the supported parameter combinations for the Connection.CreateSession method and their effect on the session transaction and acknowledgment features.

<table>
<thead>
<tr>
<th>Parameters</th>
<th>Client MUST acknowledge message receipt</th>
<th>Transaction support for</th>
<th>STOMP Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>CreateSession(False, amAutoAcknowledge)</td>
<td>No</td>
<td>-</td>
<td>1.0</td>
</tr>
<tr>
<td>CreateSession(False, amClientAcknowledge)</td>
<td>Yes (cumulative effect)</td>
<td>-</td>
<td>1.0</td>
</tr>
</tbody>
</table>

---

8 [https://stomp.github.io/stomp-specification-1.2.html#SUBSCRIBE_ack_Header](https://stomp.github.io/stomp-specification-1.2.html#SUBSCRIBE_ack_Header)
Connections and Sessions

<table>
<thead>
<tr>
<th>Method</th>
<th>Supported</th>
<th>Newer</th>
<th>Earlier</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CreateSession(False, amClientIndividual)</code></td>
<td>Yes</td>
<td>-</td>
<td>-</td>
<td>1.2</td>
</tr>
<tr>
<td><code>CreateSession(True, amAutoAcknowledge)</code></td>
<td>No</td>
<td>✓</td>
<td>-</td>
<td>1.0</td>
</tr>
<tr>
<td><code>CreateSession(True, amClientAcknowledge)</code></td>
<td>Yes (cumulative effect)</td>
<td>✓</td>
<td>✓ badge 1.0</td>
<td></td>
</tr>
<tr>
<td><code>CreateSession(True, amClientIndividual)</code></td>
<td>Yes</td>
<td>✓</td>
<td>✓ badge 1.0</td>
<td>1.2</td>
</tr>
<tr>
<td><code>CreateSession(True, amTransactional)</code></td>
<td>No</td>
<td>✓</td>
<td>-</td>
<td>1.0</td>
</tr>
</tbody>
</table>

Table 3: Session creation parameters

 badge 1.0 – not supported by ActiveMQ Artemis

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 into an atomic unit of work.

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.

Create a transacted session

To create a transacted session, set the parameter of `CreateSession` to `amTransactional` as shown in the code example

```csharp
Code example

Session := Connection.CreateSession(amTransactional);
```

or (using the older API version)

```csharp
Code example

Session := Connection.CreateSession(True, amTransactional);
```

This code will automatically start a new transaction for this session.
Send messages
Now send messages using the transacted session.

Code example

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

Committing a transaction
If your client code has successfully sent its messages, the transaction must be committed to make the messages visible on the destination.

Code example

```plaintext
// send messages ...
finally
  // commit all messages
  Session.Commit;
end;
```

Note: committing a transaction automatically starts a new transaction

Rolling back a transaction
If your client code runs wants to undo the sending of its messages, the transaction may be rolled back, and the messages will not become visible on the destination.

Code example

```plaintext
// send messages ...
except
  ... // error!
  Session.Rollback;
  ...
end;
```

Note: rolling back a transaction automatically starts a new transaction. A transacted session will be rolled back automatically if the connection is closed.
Transacted message acknowledgement

Some library versions (see table "Communication Adapters" on page 11) support transactions also for the acknowledgement of received messages.

When a transaction is rolled back or the connection is closed without a commit, messages which have been acknowledged after the transaction start will return to unacknowledged state.

```csharp
// receive in a transacted session
Session := Connection.CreateSession(True, amClientAcknowledge);
Queue := Session.CreateQueue(GetQueueName);
Consumer := Session.CreateConsumer(Queue);
Msg := Consumer.Receive(1000);

// process the message
...

// acknowledge the message
Msg.Acknowledge;

... 

// in case of errors, roll back all acknowledgements
Session.Rollback;
```

This is an experimental feature. It requires the STOMP 1.2 communication protocol.
Destinations

Introduction

The API supports two models:

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.

---

Destinations

Code example

```csharp
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.

Code example

```csharp
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).
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.

Code 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.

Persistent messages
The delivery mode for outgoing messages may be set to persistent in one of two ways. From the docs for TBTMessageProducer: "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."

Setting the default delivery mode
Code example
```
Destination := Session.CreateQueue('foo');
Producer := Session.CreateProducer(Destination);
Producer.DeliveryMode := dmPersistent;
Producer.Send(Session.CreateTextMessage('Test message'));
```

Setting the delivery mode for an individual message
Code example
```
Destination := Session.CreateQueue('foo');
Producer := Session.CreateProducer(Destination);
Producer.Send(Session.CreateTextMessage('Test message'), dmPersistent,
 BTBrokerConsts.DEFAULT_PRIORITY, 0);
```
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.

Code example

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

Message Selector

A message consumer can be created with a message selector\(^\text{10}\). A message selector allows the client to restrict the messages delivered to the message consumer to those that match the selector.

Synchronous Receive

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

Code example

```pascal
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;
```

Receive and ReceiveNoWait

There are three different methods for synchronous receive:

- **Receive**
  - The Receive method with no arguments will block (wait until a message is available).

- **Receive(TimeOut)**
  - The Receive method with a timeout parameter will wait for the given time in milliseconds. If no message arrived, it will return nil.

\(^{10}\)The RabbitMQ message broker does not support message selectors.
**ReceiveNoWait**  The ReceiveNoWait method will return immediately. If no message arrived, it will return nil.
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 message broker 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.\footnote{11}

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 durable subscriber MessageConsumer is created with a unique clientId and durable subscriber name.

Only one thread can be actively consuming from a given logical topic subscriber.

\footnote{11} \url{http://download.oracle.com/javaee/5/api/javax/jms/TopicSession.html}
Temporary Queues

Introduction

“Temporary destinations (temporary queues or temporary topics) are proposed as a lightweight alternative in a scalable system architecture that could be used as unique destinations for replies. Such destinations have a scope limited to the connection that created it, and are removed on the server side as soon as the connection is closed.” (“Designing Messaging Applications with Temporary Queues”, by Thakur Thribhuvan 12)

Library Support

Temporary destinations are supported by

- ActiveMQ
- OpenMQ
- RabbitMQ

Resource Management

The session should be closed as soon as processing is completed so that TemporaryQueues will be deleted on the server side.

Message Options

Standard Properties

The Apache ActiveMQ message broker supports some JMS standard properties in the STOMP adapter. These properties are based on the JMS specification of the Message interface.\(^\text{13}\)

Habari Client libraries for other message brokers may support a subset of these standard properties.

Note: If your application makes use of these properties, your application depends on a broker-specific feature which is not guaranteed to be available in the STOMP adapter of other message brokers

Properties for outgoing messages

<table>
<thead>
<tr>
<th>JMSCorrelationID</th>
<th>The correlation ID for the message.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSExpiration</td>
<td>The message's expiration value.</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>Whether or not the message is persistent.(^\text{14})</td>
</tr>
<tr>
<td>JMSPriority(^\text{15})</td>
<td>The message priority level.</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>The Destination object to which a reply to this message should be sent.</td>
</tr>
</tbody>
</table>

Properties for incoming messages

<table>
<thead>
<tr>
<th>JMSCorrelationID</th>
<th>The correlation ID for the message.</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSExpiration</td>
<td>The message's expiration value.</td>
</tr>
<tr>
<td>JMSDeliveryMode</td>
<td>Whether or not the message is persistent.</td>
</tr>
<tr>
<td>JMSPriority</td>
<td>The message priority level.</td>
</tr>
</tbody>
</table>

\(^\text{13}\)http://download.oracle.com/javaee/5/api/javax/jms/Message.html
\(^\text{14}\)For sending persistent messages please see documentation for IMessageProducer
\(^\text{15}\)Clients set the JMSPriority not directly, but either on the producer or as a parameter in the Send method
<table>
<thead>
<tr>
<th>Property Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>JMSTimestamp</td>
<td>The timestamp the broker added to the message.</td>
</tr>
<tr>
<td>JMSMessageId</td>
<td>The message ID which is set by the provider.</td>
</tr>
<tr>
<td>JMSReplyTo</td>
<td>The Destination object to which a reply to this message should be sent.</td>
</tr>
</tbody>
</table>

**Reserved property names**

Some headers names are defined by the Stomp specifications, and by broker-specific extensions of the Stomp protocol. These reserved Stomp header names can not be used as names for user defined properties.

Note: The client library will raise an Exception if the application tries to send a message with a reserved property name.

**Examples**

- login
- passcode
- transaction
- session
- message
- destination
- id
- ack
- selector
- type
- content-length
- content-type
- correlation-id
- expires
- persistent
- priority
- reply-to
- message-id
- timestamp
- client-id
- redelivered
Prefix for custom headers
A common practice to avoid name collisions is using a prefix for your own properties (example: `x-type` instead of `type`).

Selectors
Selectors are a way of attaching a filter to a subscription to perform content based routing. For more documentation on the detail of selectors see the reference on `javax.jmx.Message`\(^{16}\).

Supported message brokers
Message selectors are supported by
- Habari Client for ActiveMQ
- Habari Client for Artemis
- Habari Client for OpenMQ

Code example

```java
Consumer := Session.CreateConsumer(Destination, 'type='car' and color='blue'');
```

All supported brokers allow supports string type properties and operations in selectors. ActiveMQ also allows integer properties and operations in selectors (see special note\(^{17}\)).

\(^{16}\) [http://docs.oracle.com/javaee/5/api/javax/jms/Message.html](http://docs.oracle.com/javaee/5/api/javax/jms/Message.html)

\(^{17}\) [http://activemq.apache.org/selectors.html](http://activemq.apache.org/selectors.html)
Map Messages

Introduction
A map message is used to exchange a set of name-value pairs. The names are strings, the values are also strings (but may be textual representations of other data types).

Usage Example
Create a map message and add map entries:

```pascal
MapMessage := Session.CreateMapMessage;
MapMessage.SetString('key', 'value');
MapMessage.SetInt('key_int', 4096);
MapMessage.SetBoolean('key_b', True);
```

Read a map message from a consumer and access its entries:

```pascal
MapMessage := Consumer.Receive(1000) as IMapMessage;
StringValue := MapMessage.GetString('key');
IntegerVale := MapMessage.GetInt('key_int');
BoolValue   := MapMessage.GetBoolean('key_b');
```

Enumerate map entries:

```pascal
MapKeys := MapMessage.GetMapNames;
for I := 0 to Length(MapKeys) - 1 do
begin
  MapKey := MapKeys[I];
  MapValue := MapMessage.GetString(MapKey);
  ... // process map entry
end;
```

Map Message Transformer
To send and receive map messages, the application needs to convert incoming and outgoing map messages from and to the STOMP message body.

The IMessageTransformer interface must be implemented for map message and object message transformation. This interface defines two methods, ConsumerTransform and ProducerTransform.
Interface

```pascal
function ConsumerTransform(const Session: ISession;
    const Consumer: IMessageConsumer; const AMessage: IMessage): IMessage;

function ProducerTransform(const Session: ISession;
    const Producer: IMessageProducer; const AMessage: IMessage): IMessage;
```

Implementation guide for map messages:

1. create a class which implements the IMessageTransformer interface
   - for ConsumerTransform, the incoming map message is passed as the AMessage parameter, the method must read its body to reconstruct the map properties, and return the map message as function result
   - for ProducerTransform, the outgoing map message is passed as the AMessage parameter, the method must write its body to store a representation of the map, and return the map message as function result

2. create an instance of this class and register it as the message transformer on the IConnection instance
   - Note: only one map message transformer may be active for one connection

Code example

```pascal
Connection := Factory.CreateConnection;
try
    MyMapTransformer := TMyMapMessageTransformer.Create;
    // use the helper method in unit BTConnection:
    SetMapMessageTransformer(Connection, MyMapTransformer, 'my-map-message');
    Connection.Start;

    // send / receive messages
finally
    Connection.Close;
end;
```

Transformation Identifier

To detect that an incoming message is a map message, it needs to carry a special header property. Without this transformation identifier, the message will still be delivered but its actual type will be undefined – it may arrive as a ITextMessage or IbytesMessage.

By default, the library will set this header property to the transformation identifier passed to the SetTransformer method.

You may explicitly set the header property on the created message:
Habari STOMP Client for RabbitMQ 7.0

**Code example**

```pascal
MapMessage := Session.CreateMapMessage;
... // add map entries

// add the transformation identifiert
MapMessage.SetStringProperty(SH_TRANSFORMATION, 'my-map-message');
Producer.Send(MapMessage);
```

**Example ProducerTransform implementation with TStrings**

This implementation uses a TStrings to collect the map entries. The outgoing message contains the TStrings as body.

Notes:

- the method uses a method of a helper interface, IContentProvider.SetContent, to write the body content
- the method returns nil if the passes message is no map message

**Code example**

```pascal
function TMyMapMessageTransformer.ProducerTransform(const Session: ISession;
 const Producer: IMessageProducer; const AMessage: IMessage): IMessage;
var
 TmpMapMsg: IMapMessage;
 Keys: PMStrings;
 I: Integer;
 MapKey: string;
 MapValue: string;
 MapStrings: TStrings;
begin
 Result := nil;

 if Supports(AMessage, IMapMessage, TmpMapMsg) then
 begin
 MapStrings := TStringList.Create;
 try
 Keys := TmpMapMsg.GetMapNames;

 for I := 0 to Length(Keys) - 1 do
 begin
 MapKey := Keys[I];
 MapValue := TmpMapMsg.GetString(MapKey);
 MapStrings.Values[MapKey] := MapValue;
 end;

 (AMessage as IContentProvider).SetContent(Utf8Encode(MapStrings.Text));
 Result := AMessage;
 finally
 MapStrings.Free;
 end;
 end;
end;
```
See unit MapMessageTransformerTests for integration / unit tests.
Object Messages

"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."

Introduction

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.

Object Message Transformer

To send and receive object messages, the application needs to convert incoming and outgoing object messages from and to the STOMP message body.

The IMessageTransformer interface must be implemented for map message and and object message transformation.

This interface defines two methods, ConsumerTransform and ProducerTransform.

Code example

```plaintext
function ConsumerTransform(const Session: ISession;
                           const Consumer: IMessageConsumer; const AMessage: IMessage): IMessage;

function ProducerTransform(const Session: ISession;
                           const Producer: IMessageProducer; const AMessage: IMessage): IMessage;
```

Implementation guide for map messages:

3. create a class which implements the IMessageTransformer interface
   - for ConsumerTransform, the incoming object message is passed as the AMessage parameter; the method must read its body to reconstruct the object, and return the object message as function result
   - for ProducerTransform, the outgoing object message is passed as the AMessage parameter, the method must write its body to store a representation of the object, and return the object message as function result

18 https://www.oracle.com/technical-resources/articles/java/serializationapi.html
4. create an instance of this class and register it as the message transformer on the IConnection instance
  - Note: only one object message transformer may be active for one connection

See unit **ObjectMessageTransformerTests** for integration / unit tests.
Simplified API

New interface types
The new API\(^\text{19}\) is based on three new interfaces which reduce the amount of client code:

- IMQContext
- IMQProducer
- IMQConsumer

IMQContext interface
A IMQContext object encapsulates both the IConnection and the ISession object of the classic API. The connection factory interface contains new methods to create IMQContext objects:

Code example

```pascal
function CreateContext: IMQContext; overload;
function CreateContext(const AcknowledgeMode: TAcknowledgementMode): IMQContext; overload;
function CreateContext(const Username, Password: string): IMQContext; overload;
function CreateContext(const Username, Password: string; const AcknowledgeMode: TAcknowledgementMode): IMQContext; overload;
```

The IMQContext provides methods to create messages, producer and consumer objects, destinations (queues, topics, temporary queues, temporary topics, durable subscribers and so forth), and for transaction control (commit, rollback).

IMQProducer interface
A IMQProducer object provides methods to produce and send messages to the broker. As a shortcut, a method allows to send text or bytes messages without creating ITextMessage or IBytesMessage object by providing the text or bytes as a parameter.

Code example

```pascal
function Send(const Destination: IDestination;
```

\(^{19}\) Since version 6.0
IMQConsumer interface

An IMQConsumer object provides methods to consume messages from the broker.

The following example is taken from the unit tests. It uses the new API to create and send a text message to a broker queue destination, and then receives the message from this queue.

Source code example

```pascal
procedure TNewApiTests.TestSendMessage;
var
  Context: IMQContext;
  Destination: IQueue;
  Producer: IMQProducer;
  Consumer: IMQConsumer;
  TextMessage: ITextMessage;
begin
  Context := Factory.CreateContext;
  Destination := Context.CreateQueue(GetQueueName);
  Producer := Context.CreateProducer;
  Producer.Send(Destination, 'Hello World');
  Consumer := Context.CreateConsumer(Destination);
  TextMessage := Consumer.Receive(2500) as ITextMessage;
  CheckEquals('Hello World', TextMessage.Text);
  Context.Close;
end;
```
Stomp 1.2

Connection configuration
A connection string can use additional URL parameters to configure Stomp version 1.1 and 1.2.

All Parameters are case sensitive.
Parameters can be omitted to use the default value.

<table>
<thead>
<tr>
<th>Switch</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
</table>
| connect.accept-version  
20                                   | Supported Stomp versions in ascending order                                  | Broker specific, see below   |
| connect.host  
21                                   | The name of a virtual host that the client wishes to connect to. It is recommended clients set this to the host name that the socket was established against, or to any name of their choosing. If this header does not match a known virtual host, servers supporting virtual hosting MAY select a default virtual host or reject the connection. | Server URI                   |
| connect.heart-beat  
22                                   | Heart beat (outgoing, incoming)                                              | 0,0                          |

Default Stomp version (broker-specific)  

If the connection URL does not contain the connect.accept-version parameter, the client library will add an accept-version header to the CONNECT frame with the value defined in the SH_DEFAULT_STOMP_VERSION constant in the BTBrokerConsts unit.

<table>
<thead>
<tr>
<th>Default Stomp version</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveMQ</td>
</tr>
<tr>
<td>1.2</td>
</tr>
</tbody>
</table>

20 [http://stomp.github.com//stomp-specification-1.2.html#protocol_negotiation](http://stomp.github.com//stomp-specification-1.2.html#protocol_negotiation)
21 [http://stomp.github.com//stomp-specification-1.2.html#CONNECT_or_STOMP_Frame](http://stomp.github.com//stomp-specification-1.2.html#CONNECT_or_STOMP_Frame)
22 [http://stomp.github.com//stomp-specification-1.2.html#Heart-beating](http://stomp.github.com//stomp-specification-1.2.html#Heart-beating)
23 Since version 5.1 (2017.06)
Connection Factory Code Example:

```pascal
Factory := TBTConnectionFactory.Create(
    'stomp://localhost:61613?connect.accept-version=1.2&connect.heart-beat=1000,0');
```

This example creates a connection factory with these connection settings:

- **host**: localhost
- **port**: 61613
- **accept-version**: 1.2
- **heart-beat**: 1000,0

  - virtual host is localhost
  - the client requests Stomp 1.2 protocol
  - client heart beat interval is 1000 milliseconds, no server heart beat signals

### Specification

For details see the Stomp specification pages:

- [http://stomp.github.com//stomp-specification-1.2.html](http://stomp.github.com//stomp-specification-1.2.html)

### Sending heart-beat signals

A client can use the `SendHeartbeat` method of the connection object to send a heart-beat byte (newline 0x0A).

SendHeartbeat is a method of the IHeartbeat interface, which is declared in the BTSessionIntf unit. A cast of the IConnection object is required to access this method.

```pascal
(Connection as IHeartbeat).SendHeartbeat;
```
Notes:

- the client application code is responsible for sending a heartbeat message within the maximum interval which was specified in the connect parameter – the Habari Client library does not send heart-beats automatically
- client messages which are sent after the heart-beat interval expires may be lost

---

Checking for incoming heartbeats

The Habari client library stores a time-stamp of the last incoming data. If the time which elapsed since this time-stamp is greater than two times the heart-beat interval, calling CheckHeartbeat will raise an exception of type EBTStompServerHeartbeatMissing.

Code example

```csharp
(Connection as IHeartbeat).CheckHeartbeat;
```

Notes:

- the method raises an exception if the connection does not use server-side heart-beating
- the method only checks the time elapsed since the last heart-beat, it does not try to read any data from the connection

---

Reading server-side heartbeats

If the client never needs to consume any messages, but still needs to check for server-side heartbeats, it can use the ReceiveHeartbeat method of the connection object.

This method takes one argument, TimeOut.
The function returns True if it found at least one heart-beat signal on the connection.

Calling ReceiveHeartbeat is only useful for applications which never call Receive, to check if the server is still healthy, and to consume the pending heart-beat signals from the connection.

If the client reads messages (using Consumer.Receive), calling ReceiveHeartbeat is not required.
## Example Applications

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>common</td>
<td>Shared units</td>
</tr>
<tr>
<td>common-consumertool</td>
<td>Receive messages from broker</td>
</tr>
<tr>
<td>common-consumertool-fpc</td>
<td>Free Pascal version</td>
</tr>
<tr>
<td>common-producertool</td>
<td>Send messages to broker</td>
</tr>
<tr>
<td>common-producertool-fpc</td>
<td>Free Pascal version</td>
</tr>
<tr>
<td>common-producertool-ssl</td>
<td>Send messages to broker with SSL connection</td>
</tr>
<tr>
<td>common-tests</td>
<td>DUnit tests(Delphi 2009)</td>
</tr>
<tr>
<td>common-tests-fpc</td>
<td>FPCUnit tests</td>
</tr>
<tr>
<td>delphichat</td>
<td>Simple chat client (Delphi 2009)</td>
</tr>
<tr>
<td>heartbeat-server</td>
<td>Uses server-side heart-beating to check the connection / server health²⁴</td>
</tr>
<tr>
<td>loadbalancing</td>
<td>File transfer from LoadServer to LoadClient application</td>
</tr>
<tr>
<td>performance</td>
<td>Multi-threaded performance test application (Delphi 2009)</td>
</tr>
<tr>
<td>reconnect</td>
<td>Send messages and reconnect on connection failure</td>
</tr>
<tr>
<td>rpc</td>
<td>Use temporary queues to implement request/response style communication (not supported on all message brokers²⁵)</td>
</tr>
<tr>
<td>textmessage</td>
<td>Simple text message example</td>
</tr>
<tr>
<td>throughput</td>
<td>Produces and consumes messages continuously</td>
</tr>
<tr>
<td>throughput-fpc</td>
<td>Free Pascal version</td>
</tr>
<tr>
<td>transactions</td>
<td>Transaction example</td>
</tr>
</tbody>
</table>

²⁴ Requires STOMP 1.2; not supported by OpenMQ  
²⁵ Not available with ActiveMQ Artemis message broker
### Directory

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tutorial1</td>
<td>Tutorial one</td>
</tr>
<tr>
<td>tutorial2</td>
<td>Tutorial two</td>
</tr>
</tbody>
</table>

*Table 4: Example Applications (in alphabetic order)*

---

## Shared units for demo projects

The directory `demo/common` contains shared units:

- connection configuration form
- command line parameter support class
- LoggingHelper example unit (see “Logging with SLF4P” on page 60)

---

![Connection configuration dialog example](image)

*Illustration 2: Connection configuration dialog example*
**ConsumerTool**

The ConsumerTool demo may be used to receive messages from a queue or topic. This example application is configurable by command line parameters, all are optional.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AckMode</td>
<td>CLIENT_ACKNOWLEDGE</td>
<td>Acknowledgment mode, possible values are: CLIENT_ACKNOWLEDGE, AUTO_ACKNOWLEDGE or SESSION_TRANSACTED</td>
</tr>
<tr>
<td>ClientId</td>
<td></td>
<td>Client Id for durable subscriber</td>
</tr>
<tr>
<td>ConsumerName</td>
<td>Habari</td>
<td>name of the message consumer - for durable subscriber</td>
</tr>
<tr>
<td>Durable</td>
<td>false</td>
<td>true: use a durable subscriber</td>
</tr>
<tr>
<td>MaximumMessages</td>
<td>10</td>
<td>expected number of messages</td>
</tr>
<tr>
<td>Password</td>
<td></td>
<td>Password</td>
</tr>
<tr>
<td>PauseBeforeShutDown</td>
<td>false</td>
<td>true: wait for key press</td>
</tr>
<tr>
<td>ReceiveTimeOut</td>
<td>0</td>
<td>consume messages while they continue to be delivered within the given time out</td>
</tr>
<tr>
<td>SleepTime</td>
<td>0</td>
<td>time to sleep after receive</td>
</tr>
<tr>
<td>Subject</td>
<td>TOOL.DEFAULT</td>
<td>queue or topic name</td>
</tr>
<tr>
<td>Topic</td>
<td>false</td>
<td>true: topic false: queue</td>
</tr>
<tr>
<td>Transacted</td>
<td>false</td>
<td>true: transacted session</td>
</tr>
<tr>
<td>URL</td>
<td>localhost</td>
<td>server url</td>
</tr>
<tr>
<td>User</td>
<td></td>
<td>user name</td>
</tr>
<tr>
<td>Verbose</td>
<td>true</td>
<td>verbose output</td>
</tr>
</tbody>
</table>

*Table 5: ConsumerTool Command Line Options*

**Illustration 3: ConsumerTool demo application**
Examples

Receive 1000 messages from local broker

```
ConsumerTool --MaximumMessages=1000
```

Receive 10 messages from local broker and wait for any key

```
ConsumerTool --PauseBeforeShutDown
```

Use a transacted session to receive 10,000 messages from local broker

```
ConsumerTool --MaximumMessages=10000 --Transacted --AckMode=SESSION_TRANSACTED
```
**ProducerTool**

The ProducerTool demo can be used to send messages to the broker. It is configurable by command line parameters, all are optional.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>MessageCount</td>
<td>10</td>
<td>Number of messages</td>
</tr>
<tr>
<td>MessageSize</td>
<td>255</td>
<td>Length of a message in bytes</td>
</tr>
<tr>
<td>Persistent</td>
<td>false</td>
<td>Delivery mode 'persistent'</td>
</tr>
<tr>
<td>SleepTime</td>
<td>0</td>
<td>Pause between messages in milliseconds</td>
</tr>
<tr>
<td>Subject</td>
<td>TOOL.DEFAULT</td>
<td>Destination name</td>
</tr>
<tr>
<td>TimeToLive</td>
<td>0</td>
<td>Message expiration time</td>
</tr>
<tr>
<td>Topic</td>
<td>false</td>
<td>Destination is a topic</td>
</tr>
<tr>
<td>Transacted</td>
<td>false</td>
<td>Use a transaction</td>
</tr>
<tr>
<td>URL</td>
<td>localhost</td>
<td>Message broker URL</td>
</tr>
<tr>
<td>Verbose</td>
<td>true</td>
<td>Verbose output</td>
</tr>
<tr>
<td>User</td>
<td></td>
<td>User name</td>
</tr>
<tr>
<td>Password</td>
<td></td>
<td>Password</td>
</tr>
</tbody>
</table>

Table 6: ProducerTool Command Line Options

Illustration 4: ProducerTool demo application

**Examples**

Send 10,000 messages to the queue TOOL.DEFAULT on the local broker

```
ProducerTool --MessageCount 10000
```
Send 10 messages to the topic ExampleTopic on the local broker

```
ProducerTool --Topic --Subject=ExampleTopic
```
Performance test

The performance test application provides a GUI for multi-threaded sending and receiving of messages.

- A broker configuration dialog can be invoked by clicking the URL field
- The communication library (Indy or Synapse) can be selected
- Number and length of messages and thread number can be adjusted using the sliders

For every thread a message queue with the name ExampleQueue.<n> will be used.

Habari STOMP Client for RabbitMQ 5.1 includes an enhanced performance test application, which optionally collects message rates of multiple test runs and displays the sample median. Shown above is an example for a client configuration:
• 21 test runs (triggered by a shift-click on the test button)
• 2000 messages per thread
• 210 bytes payload
• two producer threads, two consumer threads

To start the long-running tests, shift-click on the run button. Taking all test samples takes around ten seconds.
Throughput test

This example application is configurable by command line parameters, all are optional.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Default Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Password</td>
<td>(broker-specific)</td>
<td>Password</td>
</tr>
<tr>
<td>Subject</td>
<td>ExampleTopic</td>
<td>Topic name</td>
</tr>
<tr>
<td>URL</td>
<td>(broker-specific)</td>
<td>Connection URL</td>
</tr>
<tr>
<td>User</td>
<td>(broker-specific)</td>
<td>User name</td>
</tr>
</tbody>
</table>

Table 7: Throughput Test Tool Command Line Options

Examples

Use remote broker 'mybroker' and specify user and password

```
tptest --url=stomp://mybroker --user=test1 --password=secret
```

Illustration 6: Throughput test tool output
# Unit Tests

## Introduction

Habari Client libraries include DUnit and FPCUnit tests. They require the classic DUnit framework (included in Delphi 2009) or FPCUnit (included in Lazarus 2.0.12).

The test projects are installed in the common-tests and common-tests-fpc folders.

## Test project configuration

### Logging

To switch on SLF4P logging, add the conditional symbol HABARI_LOGGING (see chapter ‘Logging with SLF4P’) and rebuild the project. Set the DEFAULT_LOG_LEVEL constant in unit TestHelper to a valid SLF4P level.

### Optional units

To switch on tests for optional units (object message exchange), add the conditional symbol TEST_OPTIONAL_UNITS and rebuild the project.

## Test units

The common-tests folder contains these units

<table>
<thead>
<tr>
<th>Test setup and test case base classes</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>TestHelper</td>
<td>Main test set-up and utility unit, contains no tests</td>
</tr>
<tr>
<td>HabariTestCase</td>
<td>Test case base classes used for most tests</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Unit tests</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ApiTests</td>
<td>Tests Habari Client core API methods – part 1</td>
</tr>
<tr>
<td>BasicTests</td>
<td>Tests Habari Client core API methods – part 2</td>
</tr>
<tr>
<td>Test Case</td>
<td>Description</td>
</tr>
<tr>
<td>-------------------------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>BrokerExtensionsTests</td>
<td>Tests broker-specific features and extensions of the STOMP protocol</td>
</tr>
<tr>
<td>HabariExtensionsTests</td>
<td>Tests non-standard features provided by the Habari Client library</td>
</tr>
<tr>
<td>HabariTypesTests</td>
<td>Tests internal data types</td>
</tr>
<tr>
<td>ObjectExchangeTests</td>
<td>Tests object message exchange (for Delphi DUnit only)</td>
</tr>
<tr>
<td>Stomp12Tests</td>
<td>Tests features introduced with version 1.2 of the STOMP standard</td>
</tr>
<tr>
<td>StubServerTests</td>
<td>Tests using a simple local Stomp server</td>
</tr>
</tbody>
</table>

Free Pascal specific test units are in the folder common-tests-fpc

**Test execution**

**Requirements**

The test projects require a message broker running on the local system, which accepts STOMP connections on the default port, with the default user credentials. User name and password for the default user are defined in unit BTBrokerConsts.

**Test destinations**

Most tests create a test-specific destination (queue or a topic) to reduce the risk of side effects.

The name of the destination is the combination of the test class name and the unit test name.

Note: the unit tests will not clean up or remove these destination objects after usage.

**STOMP 1.2**

Since Habari STOMP Client for RabbitMQ 5.0, the unit test use STOMP 1.2 for connections.

26 only added to the test suite if TEST_OPTIONAL_UNITS is defined
Logging with SLF4P

Introduction
Habari Client libraries include the free open source logging framework SL4FP as an optional dependency.
SLF4P is available at https://github.com/michaelJustin/slf4p

IDE and project configuration
In order to compile with SLF4P support,
1. include the path to the slf4p library in the project search or in the global library path
2. add the conditional symbol HABARI_LOGGING to the project options

Delphi
• choose Project | Options... | Delphi Compiler > Conditional defines
• add HABARI_LOGGING

Lazarus
• choose Project | Project Options ... | Compiler Options > Other
• add -dHABARI_LOGGING in the Custom options field

LoggingHelper unit
A simple LoggingHelper unit is located in the demo\common\ directory and can be copied to a project to add slf4p support with little extra coding.

Code example

uses
  LoggingHelper,
...
begin
  // set up logging
Logging with SLF4P

The LoggingHelper unit may be adjusted to your configuration needs. Here is an example which uses the SimpleLogger implementation (included in SLF4P).

```
unit LoggingHelper;

interface

uses
{$IFDEF HABARI_LOGGING}
djLogOverSimpleLogger, SimpleLogger
{$ENDIF HABARI_LOGGING};

const
    DEFAULT_LOG_LEVEL = 'info';

procedure ConfigureLogging(const LogLevel: string = DEFAULT_LOG_LEVEL);

implementation

procedure ConfigureLogging(const LogLevel: string);
begin
{$IFDEF HABARI_LOGGING}
    SimpleLogger.Configure('defaultLogLevel', LogLevel);
    SimpleLogger.Configure('showDateTime', 'true');
{$ENDIF HABARI_LOGGING}
end;
end.
```
Conditional Symbols

Caution

All conditional symbols enable experimental or optional features, which are not covered by the free basic support plan. Feedback (suggestions for improvements, feature requests, and bug reports) are always welcome.

Conditional symbols for release builds

**HABARI_ALLOW_UNKNOWN_URL_PARAMS**

Disables strict connection URL parameter checking.

If this symbol is defined, connection URLs may contain arbitrary parameters. By default, the library only accepts well-known connection parameters and raises an exception for unknown parameters.

**Broker versions:** all broker versions.

**HABARI_LOGGING**

Enables logging support. Requires the open source SLF4P logging facade.

**Broker versions:** all broker versions.

See also: Logging with SLF4P

**HABARI_SSL_SUPPORT**

Enables SSL support. Support for SSL connections is an advanced / optional feature, technical support is not included in the basic support plan.

The directory source/optional contains example implementations of Indy and Synapse adapter classes with OpenSSL support. Please note that these are basic implementations and not supported in the free basic support plan.

**Broker versions:** all broker versions.

See also: SSL/TLS Support
HABARI_USE_INTERCEPT
Enables detailed logging of Stomp message frames
This uses the Indy interceptor implementation in unit IdInterceptSimLog.
All communication data will be logged to a file. A new file will be created for every new STOMP connection. The file is located in a folder below the current working directory.
If this symbol is defined in a release build, a compiler warning will be emitted:

```
HABARI_USE_INTERCEPT should not be used for release builds
```

Broker versions: all broker versions.
Indy communication adapter only
Note: this feature requires permissions
- create a directory in the current directory if it does not exist
- create files

HABARI_USE_INTERCEPT_STDOUT
Enables detailed logging of Stomp message frames to the console (Windows only)
This uses the interceptor implementation in unit BTInterceptSimLog.
All communication data will be logged to stadoit (console).
If this symbol is defined in a release build, a compiler warning will be emitted:

```
HABARI_USE_INTERCEPT_STDOUT should not be used for release builds
```

Broker versions: all broker versions.
Indy communication adapter only
Note: this feature is only supported on the Windows platform.
Example output:

```
Send:Bytes:71
CONNECT
login:user
passcode:password
accept-version:1.2
client-id:in

Recv:Bytes:74
CONNECTED
server:ActiveMQ/5.16.3
heart-beat:0,0
session:in
version:1.2

Send:Bytes:115
```
Conditional symbols for unit test projects

**HABARI_TEST_OPTIONAL_UNITS**
Enables tests for experimental / optional units.

**HABARI_TEST_SYNAPSE**
Enables Synapse communication adapter in DUnit/FPCUnit tests, default is Indy.

**Supported for:** all versions.
SSL/TLS Support

SSL communication adapter classes

Habari STOMP Client for RabbitMQ includes two experimental adapter classes for usage with OpenSSL, one for Indy (Internet Direct) and one for Synapse. The units for these classes are in the source\optional folder.

<table>
<thead>
<tr>
<th>Adapter Class</th>
<th>Unit</th>
</tr>
</thead>
<tbody>
<tr>
<td>TBTCMMCommAdapterIndySSL</td>
<td>BTCommAdapterIndySSL</td>
</tr>
<tr>
<td>TBTCMMCommAdapterSynapseSSL</td>
<td>BTCommAdapterSynapseSSL</td>
</tr>
</tbody>
</table>

Table 8: Communication Adapters with SSL Support

Mixed Use

It is possible to use SSL and non-SSLL connections in the same project:

- connections with the “stomp://” scheme will remain unencrypted
- connections with the “stomp+ssl://” scheme will use SSL

SSL configuration

The TBTCMMCommAdapterIndySSL class includes very basic configuration of the Indy SSL handler. Your server or your specific security requirements may require additional configuration.

Indy SSL Demo

A demo application is included in common-producertool-ssl.

Code example

```
program ProducerToolIndySSL;
  {$APPTYPE CONSOLE}
  uses
  // the Habari Client adapter class for Indy + SSL
```
Habari STOMP Client for RabbitMQ 7.0

```pascal
BTCommAdapterIndySSL,
  // required to set the default adapter
BTAdapterRegistry,
  // the common demo unit for the producer tool
ProducerToolUnit in '..\common-produertool\ProducerToolUnit.pas',
  // configuration support unit
CommandLineSupport in '..\common\CommandLineSupport.pas',
  SysUtils;

begin
  BTAdapterRegistry.SetDefaultAdapter(TBTCommAdapterIndySSL);

  with TProducerTool.Create do
    try
      Run;
      try
        Run;
      except
        on E:Exception do WriteLn(E.Message);
      end
    finally
      Free;
    end;
  end;

  ReadLn;
end.
```

**Notes**

- the `TBTCCommAdapterIndySSL` class must be registered using `BTAdapterRegistry.SetDefaultAdapter(TBTCommAdapterIndySSL)`
- the project must be compiled with `HABARI_SSL_SUPPORT`
- the connection URL must be in the form `"stomp+ssl://server.com:sslport"`
- the OpenSSL libraries must be in the application search path

**Example output**

```
Habari Client for RabbitMQ 5.1.0 (c) 2008-2017 Michael Justin
Connecting to URL: stomp+ssl://localhost:61614
Publishing a Message with size 255 to queue: ExampleQueue
Using persistent messages
Sleeping between publish 0 ms
313 INFO habari.BTTCCommAdapterIndySSL - Verifying SSL certificate
313 INFO habari.BTTCCommAdapterIndySSL - Issuer: /C=GB/ST=Greater Manchester/L=Salford/O=COMODO CA Limited/CN=COMODO RSA Domain Validation Secure Server CA
313 INFO habari.BTTCCommAdapterIndySSL - Not After: 09.04.2018 01:59:59
313 INFO habari.BTTCCommAdapterIndySSL - Verifying SSL certificate
313 INFO habari.BTTCCommAdapterIndySSL - Issuer: /C=GB/ST=Greater Manchester/L=Salford/O=COMODO CA Limited/CN=COMODO RSA Domain Validation Secure Server CA
313 INFO habari.BTTCCommAdapterIndySSL - Not After: 09.04.2018 01:59:59
329 INFO habari.BTTCStompClient - Connected with RabbitMQ/3.6.10 using STOMP 1.2
Sending message: Message: 0 sent at: 28.06.2017 10:26:43 ...
Sending message: Message: 1 sent at: 28.06.2017 10:26:43 ...
Sending message: Message: 2 sent at: 28.06.2017 10:26:43 ...
Sending message: Message: 3 sent at: 28.06.2017 10:26:43 ...
Sending message: Message: 4 sent at: 28.06.2017 10:26:43 ...
Sending message: Message: 5 sent at: 28.06.2017 10:26:43 ...
Sending message: Message: 6 sent at: 28.06.2017 10:26:43 ...
Sending message: Message: 7 sent at: 28.06.2017 10:26:43 ...
```
Support

Support for SSL/TLS connections and the example adapter classes is not included in the basic support package of Habari STOMP Client for RabbitMQ.
Useful Units

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

**Code 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
  // release the file stream
  FS.Free;
end;
```

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

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

function TimeStampToJavaDate(const TimeStamp: TDateTime): Int64;
```
**Library Limitations**

**MessageConsumer**

**How do I implement synchronous receive from multiple destinations?**

The library does not support synchronous receive from more than one destination over a single connection.

To receive messages synchronously (using Receive and ReceiveNoWait) from two or more destinations, create one connection per destination.

Background: all pending messages in a connection are serialized in one TCP stream, so reading only the messages which come from one of the destinations would require 'skipping' all messages for other destinations.

**Message properties**

**Only string data type supported by Stomp**

The STOMP protocol uses string type key/value lists for the representation of message properties. Regardless of the method used to set message properties, 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 a Unix time stamp value, which is the internal representation in Java. But still, these values can not be used with date type expressions.

**Broker-specific exceptions**

Apache ActiveMQ 5.6 introduced support for numeric expressions in JMS selectors.

**Multi threading**

A session supports transactions and it is difficult to implement transactions that are multi-threaded; a session should not be used concurrently by multiple threads.
Free Pascal specific restrictions

- the library has only been tested on the Windows platform
- the included unit test project uses FPCUnit for Free Pascal / Lazarus instead of DUnit
- the complimentary code for map and object messages do not support Free Pascal
- the library source code uses the Delphi mode switch {$MODE DELPHI}$
- other limitations or restrictions may apply

Broker-specific limitations

Transacted Sessions

Transactional acknowledging
The STOMP implementations of Artemis and OpenMQ message broker do not support transactional acknowledging of incoming messages.

Other broker specific limitations
For broker-specific notes, please read chapter Broker-specific notes.
Frequently Asked Questions

Technical questions

Why am I getting 'undeclared identifier IndyTextEncoding_UTF8'?

Short answer
Your Indy version is too old.

Long answer
The library requires a current Indy 10.6.2 version.

Solution
Please download a newer Indy version.

Why am I getting 'Undeclared identifier: 'TimeSeparator''?

Short answer
Your Synapse version does not support your version of Delphi

Long answer

Solution
Please use Indy instead of Synapse or use a compatible version of Synapse.

Why am I getting 'Found no matching consumer' errors?

Short answer
The client closed a consumer while there still were pending messages on the wire for it, and then tried to receive the pending messages with a new consumer.
Long answer

If the client subscribes to a destination, it creates a unique subscription identifier and passes it to the broker. Messages which the broker sends to the client always include this subscription identifier in their header properties. The client verifies that the subscription id in the incoming message has the same id as the consumer.

If the client closes the consumer before all messages waiting on the wire have been consumed, and creates a new subscription (which has a new unique id), the remaining messages which are waiting on the wire, will have a subscription id which does not match the id of the new subscription. The client will raise an exception if no matching consumer can be found.

Solution

Do not create another consumer on the same connection while there are still pending messages for the first consumer. To discard all pending messages which are still waiting on the wire, the client can simply close the close the connection and create a new consumer on a new connection.

Example

Here is a small code example which causes this error:

```pascal
procedure TErrorHandlingTests.TestReceiveMessageForOtherSubscription;
var
  Factory: IConnectionFactory;
  Conn: IConnection;
  Session: ISession;
  Destination: IDestination;
  Producer: IMessageProducer;
  Consumer: IMessageConsumer;
  Msg: IMessage;
begin
  Factory := TBTConnectionFactory.Create;
  Conn := Factory.CreateConnection;
  Conn.Start;
  Session := Conn.CreateSession(amAutoAcknowledge);
  Destination := Session.CreateQueue(GetQueueName);
  Consumer := Session.CreateConsumer(Destination);
  Producer := Session.CreateProducer(Destination);
  Msg := Session.CreateMessage;
  Producer.Send(Msg);
  Consumer.Close;
  Consumer := Session.CreateConsumer(Destination);
  Consumer.Receive(1000);
end;
```

In line 20 and 21, the consumer is closed and a new consumer created for the same destination.

The Receive in line 22 will detect that the incoming message does not have a matching consumer id and raise an EIllegalStateException.

27 This code example is included in the library unit test project.
Does the library support non-Unicode Delphi versions?

Short answer
No, the library does not support non-Unicode Delphi versions.

Long answer
The library makes uses of language features which have been added in Delphi 2009 / Free
Pascal 3.2.0. Support for non-Unicode Delphi ended in April 2017.

How can the client application detect network connection loss?

Short answer
Use Stomp heart-beating

Long answer
By enabling heart-beating, the client can request server-side sending of heart beat bytes.
Even if the client only wants to consume messages and never send messages, the server
should continuously send heart-beat bytes within the negotiated time.
To detect if the server has sent a heart-beat, the client calls the method ReceiveHeartbeat.
For more details, please check the paragraph “Reading server-side heartbeats” on page 48.
Online Resources

Third-party libraries

**Indy**
Indy is an open source client/server communications library that supports TCP/UDP/RAW sockets, as well as over 100 higher level protocols including SMTP, POP3, IMAP, NNTP, HTTP, FTP, and many more. Indy is written in Delphi but is available for C++Builder, Delphi, FreePascal, .NET, and Kylix.

Project home  https://www.indyproject.org/
GitHub  https://github.com/IndySockets

**SLF4P**
SLF4P is a simple logging facade for Object Pascal, developed with Dephi 2009 and Lazarus 2.0. Tested with DUnit and FPCUnit.

Project home  https://github.com/michaelJustin/slf4p

**JsonDataObjects**
JsonDataObjects is a JSON parser for Delphi 2009 and newer

GitHub  https://github.com/ahausladen/JsonDataObjects

**Synapse**

Project home  http://synapse.ararat.cz
Subversion  http://svn.code.sf.net/p/synalist/code/trunk/
Specifications

**Stomp** – Simple (or Streaming) Text Oriented Messaging Protocol

- **Stomp home**: [https://stomp.github.io/index.html](https://stomp.github.io/index.html)
- **Stomp 1.2**: [https://stomp.github.io/stomp-specification-1.2.html](https://stomp.github.io/stomp-specification-1.2.html)
- **Stomp 1.1**: [https://stomp.github.io/stomp-specification-1.1.html](https://stomp.github.io/stomp-specification-1.1.html)
- **Stomp 1.0**: [https://stomp.github.io/stomp-specification-1.0.html](https://stomp.github.io/stomp-specification-1.0.html)

Broker-specific Stomp documentation

- **ActiveMQ**: [https://activemq.apache.org/stomp.html](https://activemq.apache.org/stomp.html)
- **Artemis**: [https://activemq.apache.org/components/artemis/documentation/latest/stomp.html](https://activemq.apache.org/components/artemis/documentation/latest/stomp.html)
- **RabbitMQ**: [https://www.rabbitmq.com/stomp.html](https://www.rabbitmq.com/stomp.html)

Online articles

<table>
<thead>
<tr>
<th>Title</th>
<th>Broker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Firebird Database Events and Message-oriented Middleware</td>
<td>All</td>
</tr>
<tr>
<td>Discover ActiveMQ brokers with Delphi XE4 and Indy 10.6</td>
<td>ActiveMQ</td>
</tr>
<tr>
<td>Official RabbitMQ Management REST API Documentation</td>
<td>RabbitMQ</td>
</tr>
<tr>
<td>How to use the RabbitMQ Web-Stomp Plugin</td>
<td>RabbitMQ</td>
</tr>
<tr>
<td>RPC with Delphi client and Java server using RabbitMQ</td>
<td>RabbitMQ</td>
</tr>
</tbody>
</table>

33 [https://mikejustin.wordpress.com/2013/05/21/rpc-with-delphi-client-and-java-server-using-rabbitmq/](https://mikejustin.wordpress.com/2013/05/21/rpc-with-delphi-client-and-java-server-using-rabbitmq/)
## Online Videos

<table>
<thead>
<tr>
<th>Title</th>
<th>Broker</th>
</tr>
</thead>
<tbody>
<tr>
<td>Introduction to Messaging With Apache ActiveMQ[^34]</td>
<td>ActiveMQ</td>
</tr>
<tr>
<td>GlassFish Message Queue – High Availability Clusters[^35]</td>
<td>OpenMQ</td>
</tr>
</tbody>
</table>

[^34]: [http://vimeo.com/12654513](http://vimeo.com/12654513)
[^35]: [http://www.youtube.com/watch?v=RHUJBsy3udU](http://www.youtube.com/watch?v=RHUJBsy3udU)
Support

Bug reports and support inquiries
Please send bug reports and support inquiries to Habarisoft and specify your message broker type and version.
To allow fast processing of your inquiry, please provide a detailed problem description, including configuration and environment, or code examples which help to reproduce the problem.

Advanced support
Advanced and experimental features such as (for example) SSL, third party libraries, Free Pascal, Linux, non-Unicode Delphi versions and message broker configuration are not covered by the basic support scheme.
**Broker-specific notes**

**Minimum supported broker version**

The minimum supported broker version for Habari STOMP Client for RabbitMQ 7.0 is RabbitMQ 3.8.0.\(^{36}\)

The client library does **not** check the broker version, however client code may check the server version string.\(^{37}\)

**Online resources**

The web page [https://www.rabbitmq.com/stomp.html](https://www.rabbitmq.com/stomp.html) documents details of the STOMP implementation in RabbitMQ, including broker-specific extensions.

*Note* If you use broker-specific extensions, be aware that moving to a different broker and a different version of Habari Client library later will require more than a simple recompilation of source code.

**Message type detection**

The library determines the type (binary or text) of incoming messages based on the **content-type** header:

- If the header starts with `text/`, the message will be treated as a text message.
- Otherwise, it will be treated as a binary message (`IBytesMessage`).

Other STOMP clients (for example node.js) may be not aware of this RabbitMQ specific rule. If they send a text message without setting content-type to `text/plain`, Habari STOMP Client for RabbitMQ will misinterpret them as binary messages.

To fix this, adjust the producer client code to include the content-type header with value `text/plain`.

\(^{36}\)[https://www.rabbitmq.com/changelog.html](https://www.rabbitmq.com/changelog.html)

\(^{37}\)see `IConnectionInfo.StompServerName`
**Prefetch count**

The RabbitMQ STOMP documentation explains that

“The prefetch count for all subscriptions is set to unlimited by default. This can be controlled by setting the `prefetch-count` header on SUBSCRIBE frames to the desired integer count.”

With Habari STOMP Client for RabbitMQ, the prefetch-count header can be set using a parameter on the destination name for a message consumer:

```go
// create a queue with a prefetch count of 3 Queue := Session.CreateQueue('ExampleQueue?prefetch-count=3');

// create a consumer for this queue Consumer := Session.CreateConsumer(Queue);
Msg := Consumer.Receive(1000);
```

**Destination types**

The RabbitMQ STOMP documentation describes five destination types:

- `/exchange` - SEND to arbitrary routing keys and SUBSCRIBE to arbitrary binding patterns;
- `/queue` - SEND and SUBSCRIBE to queues managed by the STOMP gateway;
- `/amq/queue` - SEND and SUBSCRIBE to queues created outside the STOMP gateway;
- `/topic` - SEND and SUBSCRIBE to transient and durable topics;
- `/temp-queue/` - create temporary queues (in reply-to headers only).

Habari STOMP Client for RabbitMQ supports all these types: for the special RabbitMQ destination names with “/amq/queue” or “/exchange”, the prefixes can be used in the `Session.CreateTopic` / `Session.CreateQueue` methods.

```go
// effective destination address: /queue/myqueue Queue := Session.CreateQueue('myqueue');

// effective destination address: /topic/mytopic Topic := Session.CreateTopic('mytopic');

// effective destination address: /exchange/myexchange/key Queue := Session.CreateQueue('exchange/myexchange/key');
// or Topic := Session.CreateTopic('exchange/myexchange/key');

// effective destination address: /amq/queue/rpc_queue Queue := Session.CreateQueue('/amq/queue/rpc_queue');
```
Header properties

Habari STOMP Client for RabbitMQ does not process these RabbitMQ specific\(^{38}\) STOMP headers:

- `amqp-message-id` the AMQP message-id property
- `content-encoding` the content-encoding property

Auto-delete queues

The RabbitMQ STOMP plug-in supports advanced queue features, which can be defined in the management interface but also from clients when the queue is created. No matter how these features have been declared, RabbitMQ requires that the client specifies the same feature settings anytime when this queue is used.

If a queue has been created with the auto-delete flag set, the queue is deleted when all consumers have finished using it.\(^{39}\)

Creation of an auto-delete queue

If the queue does not exist yet, it may be created dynamically by subscribing

```python
ClientCallbackQueue := Session.CreateQueue('Callback?auto-delete=true');
Consumer := Session.CreateConsumer(ClientCallbackQueue);
```

The admin interface will show that the `auto-delete` feature is enabled.

Sending a message to the auto-delete queue

Sending a message to this queue requires to specify that the auto-delete feature is enabled:

```python
Msg := Session.CreateTextMessage;
Msg.SetStringProperty('auto-delete', 'true');
Producer.Send(Msg);
```

\(^{38}\) https://www.rabbitmq.com/stomp.html
**Queues with x-max-priority**

The RabbitMQ STOMP plug-in supports advanced queue features, which can be defined in the management interface but also from clients when the queue is created. No matter how these features have been declared, RabbitMQ requires that the client specifies the same feature settings anytime when this queue is used.

**Creation of the queue**

If the queue does not exist yet, it may be created dynamically by subscribing

```go
PriorityQueue := Session.CreateQueue('Priority?x-max-priority=20');
Consumer := Session.CreateConsumer(PriorityQueue);
```

The admin interface will show that the **maximum priority** is 20.

**Sending a message to the queue**

Sending a message to this queue requires to specify that the maximum priority is 20:

```go
Msg := Session.CreateTextMessage;
Msg.SetIntProperty('x-max-priority', 20);
Producer.Send(Msg);
```

**Hint: check the broker log**

If your STOMP client code works with special destination features and does not work as expected, always check the RabbitMQ broker log file. On Windows, you may find it in %APPDATA%\RabbitMQ\log. On Unix, it is located in ${install_prefix}/var/log/rabbitmq (File Locations documentation).

**Quorum queues**

Quorum queues can be read, if the type is specified in the subscription STOMP frame. To do this, append the x-queue-type parameter to the queue name:

```go
// create a destination
Queue := Session.CreateQueue('my-queue?x-queue-type=quorum');
```
Send a value to a quorum queue

```c
// create a destination (but do not specify the queue type)
Queue := Session.CreateQueue('my-queue');

// send the quorum queue value
Producer := Session.CreateProducer(Queue);
Producer.Send(Session.CreateTextmessage('42'));
```

Temporary queues

RabbitMQ does not support message acknowledge with temporary queues.

To notify about this limitation, Habari STOMP Client for RabbitMQ raises an exception when `Msg.Acknowledge` is called on a temporary destination (queue or topic).

Special character encoding in STOMP headers

If a client sends a STOMP header to the RabbitMQ message broker which contains a colon character, the broker will escape it according to the STOMP 1.1 specification as `\c`.

However this happens independent of the STOMP version – even if the library uses STOMP 1.0 for the connection.

The Habari Client for RabbitMQ will not translate this non-standard escape sequence back to the colon character.

As a workaround, client applications should prefer STOMP 1.2 as this will activate correct escape sequence conversion.
Durable subscriptions with RabbitMQ

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 broker 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.⁴⁰

In RabbitMQ, the combination of the topic name and the durable subscriber name uniquely identifies the durable topic subscription.⁴¹

AMQP Semantics

For SUBSCRIBE frames, a shared queue is created for each distinct subscription ID x destination pair, and bound to the amq.topic exchange with routing key <name>. A subscription is created against the queue.

After you restart your program and re-subscribe, the broker will know which messages you need that were published while you were away.

Note: if the same combination of topic name and durable subscriber name is used by more than one client, the broker behavior is undefined – messages can be distributed between clients, or one client will receive all messages and other clients never see a message.

Creation

The ISession interface contains the CreateDurableSubscriber method which creates a durable subscriber to the specified topic.

A durable subscriber MessageConsumer is created with a unique durable subscriber name.

Code example

```java
// create a durable subscription

Topic := Session.CreateTopic('ExampleTopic');

Consumer := Session.CreateDurableSubscriber(Topic, 'my-subscription-id');
```

40 https://download.oracle.com/javaee/5/api/javax/jms/TopicSession.html
41 https://www.rabbitmq.com/stomp.html
Deletion

To delete a durable subscriber, RabbitMQ requires that the client first subscribes and then unsubscribes.

Code example

```go
// first subscribe, then unsubscribe

Topic := Session.CreateTopic('ExampleTopic');
Session.CreateDurableSubscriber(Topic, 'my-subscription-id');
Session.Unsubscribe(Topic, 'my-subscription-id');
```

Test tool example

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

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

and receive them from a client:

```bash
ConsumerTool --MaximumMessages=1000 --Topic --Subject=test-durable --Durable --
ConsumerName=12345 --Verbose
```
Connection troubleshooting

Performance demo

Socket error 10060 (Connection timed out)
If the specified host is unreachable, a „Connection timed out“ error will occur.

Socket error 10061 (Connection refused)
If the broker service is not running on the specified host and port, a „Connection refused“ error will occur:

The default port for STOMP on RabbitMQ is 61613. The port can be specified in the Broker UR:
Socket error 10054 (Connection reset by peer)

If the broker service is running on the specified host and port, but the port does not accept STOMP client connections, a "Connection reset by peer" error will occur:

In this case the broker log contains a 'bad_header' <="CONNECT\n"> error message. The IP address of the client and the port number used by the client are included in the log messages, so they can be identified.

Note that in the example below, client and host are on localhost (127.0.0.1).

```
... [info] <0.1233.0> accepting AMQP connection <0.1233.0> (127.0.0.1:56184 -> 127.0.0.1:5672)
... [error] <0.1233.0> closing AMQP connection <0.1233.0> (127.0.0.1:56184 -> 127.0.0.1:5672):
... [error] <0.1233.0> (bad_header,<="CONNECT\n">)
```

References

See also:

- https://www.rabbitmq.com/troubleshooting-networking.html
- https://www.rabbitmq.com/stomp.html
- https://www.rabbitmq.com/networking.html
Socket error 10060..............................84
Socket error 10061..............................84
SSL.............................................76
Stomp 1.2.......................................45
Stomp+ssl.....................................64
Subscribe.receipt.............................19
Support.........................................76
Synapse........................................8, 10
Synchronous receive..........................68
TBTCCommAdapterIndySSL...................64
TCP.............................................68
Test destinations.............................58

The DUnit test suite requires the Delphi
2009 version of DUnit for compilation........8
The FPCUnit test suite requires Lazarus.......8
Throughput test................................66
Topic..........................................28
TopicSubscriber................................32, 82
Transacted Sessions............................24, 69
Transactions.....................................68
Unit Tests.......................................57
Virtual host....................................45
.receipt.......................................19f.

Table Index
Table 1: Communication Adapters..........................10
Table 2: Failover Transport Options............................17
Table 3: Session creation parameters............................24
Table 4: Example Applications (in alphabetic order)..........49
Table 5: ConsumerTool Command Line Options...............50
Table 6: ProducerTool Command Line Options................52
Table 7: Throughput Test Tool Command Line Options ..........56
Table 8: Communication Adapters with SSL Support.............64

Illustration Index
Illustration 1: Programming Model.............................11
Illustration 2: Connection configuration dialog example.........49
Illustration 3: ConsumerTool demo application..................50
Illustration 4: ProducerTool demo application....................52
Illustration 5: Performance Test Application......................54
Illustration 6: Throughput test tool output.......................56