



Voyager Messaging Developer's Guide

Version 1.0 for Voyager 8.0-RC3

Table of Contents

Introduction	5
Overview.....	5
Preface.....	5
Messaging Requirements	5
JMS	5
MSMQ	5
Contacting Technical Support	6
Voyager Messaging API Overview	6
Messaging Configuration.....	6
JMS	6
MSMQ	7
Default queue/topic configuration	7
Key interfaces	7
Queues versus Topics	8
Client controlled versus automatic sessions	8
Automatic Mode	8
Client-controlled	8
JMS Message converters.....	8
JavaMessageFormatter.....	9
JMSMessage	9
MsmqMessage	10
Message Listeners.....	10
DestinationUtil.....	10
Destination Decorators.....	10
Release and Close	10
Hidden Session Close	11
Appendices – Examples	11
Running the Examples	11
Basic.....	11
Source code location.....	11
Client control	11
Source code location.....	12
Converters.....	12
Source code location.....	12
Decorator.....	12
Source code location.....	12
Listeners.....	12
Source code location.....	12
MSMQ	12
Source code location.....	13
Queues.....	13
Source code location.....	13

Topics.....	13
Source code location.....	13

<This page intentionally left blank>

Introduction

Overview

The Voyager™ Enterprise Messaging API provides a Java or .NET agent or application client the ability to communicate with a JMS provider or an MSMQ service or both. This API can also be used to create a message bridge between destinations in different providers.

Preface

The purpose of this manual is to provide an introduction to the Voyager Enterprise Messaging API. This guide assumes knowledge of JMS and/or MSMQ as well as an understanding of fundamental messaging concepts and practices.

This preface covers the following topics:

- Voyager Messaging Requirements
- Contacting technical support

Messaging Requirements

Before attempting to use the enterprise messaging features of Voyager:

JMS

- A Java Development Kit (JDK) installed on your computer. VOYAGER requires JDK 1.4.2 or later. You can download the latest release of the JDK from www.javasoft.com at no charge.
- A working version of Voyager installed.
- A JMS 1.1 compliant provider installed.

MSMQ

- .NET framework 2.0
- JDK 1.4.2 or higher to run the Java and MSMQ examples
- A working version of Voyager installed.

Copyright © 2007 – 2010 Recursion Software, Inc.
All Rights Reserved

- MSMQ 2.0 or higher installed.

Contacting Technical Support

Recursion Software welcomes your problem reports, and appreciates all comments and suggestions for improving VOYAGER. Please send all feedback to the Recursion Software Technical Support department.

Technical support for VOYAGER is available via the web, email, and phone. You can contact Technical Support by sending email to psupport@recursionsw.com or by calling (972) 731-8800.

Voyager Messaging API Overview

The goals of the Enterprise Messaging API are:

- Simplify use of JMS and MSMQ
- Allow an agent or application client (either Java or .NET) to access JMS and/or MSMQ from all Voyager supported platforms.
 - A remote “pure” Java client needs only `ve-messaging-client.jar` and (optionally) the JMS API archive (`jms.jar`) in addition to the standard Voyager libraries to access both MSMQ and JMS, making this a low footprint addition.
- Separate configuration from messaging operations (`send/receive/purge/etc.`)
- Provide a common API for JMS and MSMQ, allowing the two to be bridged and one to be changed for the other with minimal code impacts.

Messaging Configuration

JMS

JMS providers are configured via JNDI. Please consult your provider’s documentation for the proper settings. For example, here are the settings for a socket-based connection to a JBoss Messaging server running on the local machine:

```
java.naming.factory.initial=org.jnp.interfaces.NamingContextFactory
java.naming.factory.url.pkgs=org.jnp.interfaces
java.naming.provider.url=localhost
```

The other important provider specific considerations are the JMS ConnectionFactory JNDI name (“ConnectionFactory” for JBoss) and the JMS Destination names. Both JNDI and administrative Destination names are supported by the Messaging API. Please refer to the API documentation for `JMSDestinationAddress` for more information. As an example, access to the test queue “A” on JBoss Messaging may be either via “`queue/A;queue=jndi`” (by JNDI name) or “`A;queue=native`” (by administrative name).

Copyright © 2007 – 2010 Recursion Software, Inc.
All Rights Reserved

The relevant configuration classes for JMS are `JMSConfiguration` and `JMSDestinationConfiguration`. Please refer to the API documentation for these classes for more information.

MSMQ

`MsmqConfiguration` and `MsmqDestinationConfiguration` are the main configuration classes for MSMQ. The address class for MSMQ queues is `MsmqAddress`. `MsmqAddress` accepts a URI format based on Microsoft Corporation's WFC addressing for MSMQ. For example, the address for a private queue on the local machine would be: `net.msmq://localhost/private/queueA`. Please refer to the API documentation for `MsmqConfiguration`, `MsmqDestinationConfiguration` and `MsmqAddress` for more information.

Default queue/topic configuration

`MsmqConfiguration` and `JMSConfiguration` both support the ability to define a default destination configuration. In this case, queue or topic destinations not defined explicitly in the destination factory configuration will take their values from the default destination configuration. A destination configuration object can also be created from another in the case of small differences in configuration. These two facilities can help minimize destination configuration code.

Key interfaces

`IDestinationProvider`, `IDestinationFactory`, `IDestinationSession`, and `IDestination` are the key interfaces to understand in using the Messaging API.

- **IDestinationProvider** – The parent interface of `IDestinationFactory` and `IDestinationSession`, provides access to an `IDestination` by address (e.g., `JMSDestinationAddress`) or a properly formatted string.
- **IDestinationFactory** – Used to obtain `IDestinations` or create `IDestinationSessions` and access the configuration. An `IDestinationFactory` is thread-safe.
- **IDestinationSession** – Used to obtain `IDestinations`. Allows client management of transactions and message acknowledgement for one or more `IDestinations`. In most cases, a client should only use one `IDestinationSession` from a given `IDestinationFactory` at a time in order to conserve provider resources. An `IDestinationSession` is *not* thread-safe.
- **IDestination** – Supports all key messaging operations: send, receive, request (send/receive pair), purge, and asynchronous receives via message listeners (see `IVEMessageListener`). An `IDestination` also provides access to its parent `IDestinationSession`. An `IDestination` is *not* thread-safe.

Queues versus Topics

Depending on whether the underlying destination is a queue (point-to-point) or topic (publish/subscribe), a different interface subtype of `IDestination` will be returned by an `IDestinationProvider`. These interfaces are `IQueueDestination` and `ITopicDestination`. `IQueueDestination` supports “peek” operations (i.e., the ability to check messages on a queue without removing them from the queue). `ITopicDestination` currently has no additional methods, but topics have additional configuration options available (for JMS) related to topic subscribers.

Client controlled versus automatic sessions

The Messaging API may be used either in “automatic” mode, where each messaging operation stands on its own (with transactions and acknowledgement handled either by the provider or API implementation), or with acknowledgement and transactions managed by the application client (via the `IDestinationSession` interface).

Automatic Mode

For JMS, the default automatic acknowledgement mode is client acknowledgement handled by the API implementation. This guarantees that the batch-receive operation, `receiveAll`, will not lose messages if a provider or connection error occurs (for MSMQ you would have to use a transactional queue). Other acknowledgement options for JMS are automatic acknowledgement and “duplicates ok” automatic acknowledgement by the provider.

If the session is transacted, then each messaging operation is run within its own transaction. Transactions are the only way to guard against a partial send when the batch-send operation, `sendAll`, is used (if not transacted, and an error occurs during the `sendAll` call, the resulting exception will report the index at which the error occurred).

Client-controlled

For JMS, the non-transacted client-controlled mode is equivalent to a client-acknowledged session. The client must handle message acknowledgement and recovery in this case (using the `acknowledge` and `recover` methods, respectively). Likewise, for a transacted session, the client must call `commit` or `rollback`.

JMS Message converters

The `send/receive/request` methods are not required to send or receive provider specific message types (for JMS, `javax.jms.Message` and its subtypes). Instead, converters are configured to translate between arbitrary objects and provider specific types. For JMS, a set of standard

converters is available in `JMSMessageConverters`. The default converters for JMS work in the following way:

Send:

- A `String` is converted to a `TextMessage`
- A `byte []` is converted to a `BytesMessage`
- A `Map` is converted to a `MapMessage` (and thus keys and values must conform to the limitations of `MapMessage`)
- If it is `Serializable`, and not of one of the above types, it is converted to an `ObjectMessage`
- If it is null, it is converted into a `Message`
- There is no default conversion to a `StreamMessage`

Receive:

- A `TextMessage` is converted to its `String` payload
- An `ObjectMessage` is converted to its `Serializable` payload
- A `BytesMessage` is converted to its `byte []` payload
- A `MapMessage` is converted into a `Map`
- A `Message` is converted into null
- A `StreamMessage` has no default conversion and will throw an exception

An example of sending and receiving a `StreamMessage` using a custom converter is provided.

JavaMessageFormatter

The `JavaMessageFormatter` is the default message formatter used for an MSMQ destination factory. It enables Java serializable objects to be stored in MSMQ. It is backed, by default, by a `BinaryMessageFormatter`. The `JavaMessageFormatter` uses that formatter to store .NET serializable types. Message formatters are configured per destination. If the `XmlMessageFormatter` is selected, you must also specify the type names to be serialized.

JMSMessage

The `JMSMessage` object is used to provide access to, and the ability to set, JMS header and custom properties without tying the client to the provider specific JMS Message types. For a message send, it can provide the delivery mode, priority, and time to live values and can override the configured converter to send a specific JMS Message type (other than a `StreamMessage`). A destination can be configured to return a `Message` (or subtype) as a `JMSMessage` via the `receiveAsJMSMessage` property.

MsmqMessage

Copyright © 2007 – 2010 Recursion Software, Inc.
All Rights Reserved

The `MsmqMessage` object is used to provide access to the properties of an MSMQ Message object when the `System.Messaging` namespace is not available (e.g., a “pure” Java environment). It is also used as a configuration object to set the default properties for a message send operation.

Message Listeners

Support for asynchronous message receipt is provided via message listeners. A message listener must implement the `IVEMessageListener` interface. The `IVEMessageListener` implementer will receive messages converted via the configured converter or message formatter, and optionally as a `JMSMessage`, `MsmqMessage`, or native message type. The `IVEMessageListener` should not throw any exceptions and should use the return value to indicate whether the message should be acknowledged or the transaction committed.

DestinationUtil

Voyager integration of the Messaging API is provided via the `DestinationUtil` utility class. This enables remote access to a JMS or MSMQ `DestinationFactory`. It can also “bridge” destinations between providers. A `DestinationFactory` may be used standalone without the utility, but it is highly recommended to use it for deployment flexibility (and is necessary to create a destination bridge). For more information about `DestinationUtil`, refer to the API documentation and the messaging examples.

Destination Decorators

A decorator class may be configured for a destination. This decorator could be used for logging/tracing, message validation, message conversion, etc. One example would be to enable a .NET type to be sent to and retrieved from a JMS queue without the agent or application client having to handle it explicitly. There are two main categories of destination decorators: “normal” and “late-bound”. Classes for normal decorators must be available to the Voyager instance hosting the destination factory, as they are applied on the server-side. If a normal decorator is `Serializable`, then the class must be available on the client as well. Late bound decorators are applied at the client-side and the class for the decorator has to be available to the client, but need not be available on the server. Please see the decorator examples for more information.

Release and Close

The `release` method is used to release the underlying provider resources for an `IDestination` or `IDestinationProvider`, but allow them to be reacquired on demand. For an `IDestinationSession`, this means that the child destination resources will be released as well, and for an `IDestinationFactory`, the child session resources will be released.

The `close` method releases all provider resources and prevents them from being reacquired by the `IDestination` or `IDestinationProvider`. This means that for an `IDestination` acquired

from an `IDestinationSession`, once closed, it will remain closed even if reacquired from that session.

Hidden Session Close

If an `IDestination` is acquired from an `IDestinationFactory`, and the `getSession` method of the `IDestination` is never called, then close on the `IDestination` will close its “hidden” `IDestinationSession`. Note that this is not symmetric with `releaseResources` (`releaseResources` on the `IDestinationSession` must be called for the session resources to be released).

Appendices – Examples

This appendix provides an overview of the messaging API examples.

Running the Examples

After you install Voyager, the source code for the examples is located in the `examples/java/se-cdc/java/examples/messaging` directory under your Voyager installation. For Java the `CLASSPATH` must include `ve-messaging.jar`, the `examples/java` directory and (in most cases) the JMS provider jar or jars. The `ve-messaging.jar` can be found in the `platform/jse/lib/messaging` directory under your Voyager installation. You also need the examples class files to successfully run the examples. Please edit the `MessagingConfig.java` and `MessagingConstants.java` files based on your provider. They are configured for JBoss messaging by default.

Basic

This example (contained in `BasicMessagingA` and `B`) illustrates bridged access to the Messaging API via `DestinationUtil` and simple send and receive operations on queues.

Source code location

The example can be found under the `examples/java/se-cdc/java/examples/messaging/basic` directory.

Client control

The two examples in this directory illustrate client controlled acknowledgement (`ClientAcknowledgedSession`) and transactions (`TransactedSession`).

Source code location

Copyright © 2007 – 2010 Recursion Software, Inc.
All Rights Reserved

The examples can be found under the `examples/java/se-cdc/java/examples/messaging/clientcontrol` directory.

Converters

The example in this directory (`ConverterExample`) illustrates custom converters.

Source code location

The example can be found under the `examples/java/se-cdc/java/examples/messaging/converters` directory.

Decorator

The examples in this directory (`Bridge1/2.java` and `SimpleDecoratorExample1/2.java`) illustrate the use of destination decorators.

Source code location

The examples can be found under the `examples/java/se-cdc/java/examples/messaging/decorator` directory.

Listeners

The examples under this directory (`ListenerExample` and `RequestExample`) illustrate message listeners. The `RequestExample` also illustrates the use of the request method.

Source code location

The examples can be found under the `examples/java/se-cdc/java/examples/messaging/listeners` directory.

MSMQ

The examples `BasicMsmqMessagingA/B.java`, `MsmqListeners.java`, and `MsmqTransactionsExample.java` illustrate the use of MSMQ from Java. The example `JMSMSMQBridgeExample` shows how to bridge destinations from different providers (in this case a JMS provider and MSMQ).

Source code location

The examples can be found under the `examples/java/se-cdc/java/examples/messaging/msmq` directory.

Copyright © 2007 – 2010 Recursion Software, Inc.
All Rights Reserved

Queues

This example (PeekExample) illustrates the use of the peek methods for the IQueueDestination subtype.

Source code location

The examples can be found under the `examples/java/se-cdc/java/examples/messaging/queues` directory.

Topics

This example illustrates a topic “broadcast” to a set of topic subscribers.

Source code location

The examples can be found under the `examples/java/se-cdc/java/examples/messaging/topics` directory.