

# JAVA MESSAGE SERVICE (JMS)

Veerle Ongenae

# Overzicht

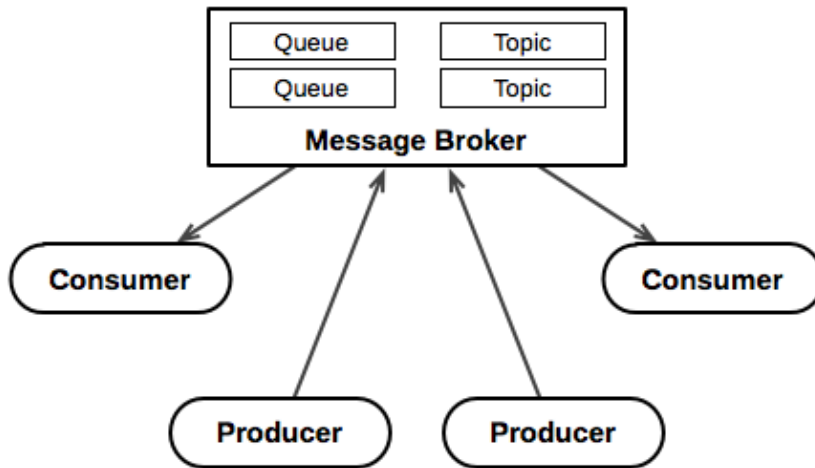
2

- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - Listeners
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

# Messaging

3



<http://torquebox.org/builds/html-docs/messaging.html>

Industrieel Ingenieur Informatica, UGent

Messaging is a method of communication between software components or applications. A messaging system is a **peer-to-peer** facility: A messaging client can send messages to, and receive messages from, any other client. Each client connects to a messaging agent that provides facilities for creating, sending, receiving, and reading messages.

Messaging enables distributed communication that is **loosely coupled**. A component sends a message to a destination, and the recipient can retrieve the message from the destination. What makes the communication loosely coupled is that the destination is all that the sender and receiver have in common. The sender and the receiver do **not** have to be **available at the same time** in order to communicate. In fact, the sender does **not** need to **know** anything about the receiver; nor does the receiver need to know anything about the sender. The sender and the receiver need to know only which **message format** and which **destination** to use.

In this respect, messaging differs from tightly coupled technologies, such as Remote Method Invocation (RMI), which require an application to know a remote application's methods.

Messaging also differs from electronic mail (email), which is a method of communication between people or between software applications and people. Messaging is used for communication between software applications or software components.

- Java Message Service
  - Berichten
  - Asynchroon
  - Betrouwbaar

The Java Message Service is a Java API that allows applications to create, send, receive, and read messages. The JMS API defines a common set of interfaces and associated semantics that allow programs written in the Java programming language to communicate with other messaging implementations.

The JMS API minimizes the set of concepts a programmer must learn in order to use messaging products but provides enough features to support sophisticated messaging applications. It also strives to maximize the portability of JMS applications across JMS providers.

JMS enables communication that is not only loosely coupled but also

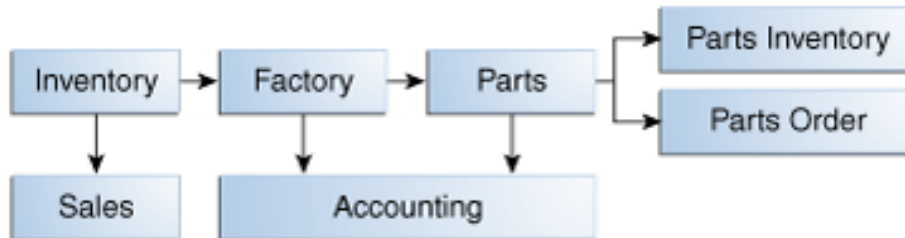
**Asynchronous:** A receiving client does not have to receive messages at the same time the sending client sends them. The sending client can send them and go on to other tasks; the receiving client can receive them much later.

**Reliable:** A messaging provider that implements the JMS API can ensure that a message is delivered once and only once. Lower levels of reliability are available for applications that can afford to miss messages or to receive duplicate messages.

The current version of the JMS specification is Version 2.0.

# Gebruik JMS

5



<https://docs.oracle.com/javaee/7/tutorial/jms-concepts001.htm#BNCDR>

Industrieel Ingenieur Informatica, UGent

## When Can You Use the JMS API?

An enterprise application provider is likely to choose a messaging API over a tightly coupled API, such as a remote procedure call (RPC), under the following circumstances.

- The provider wants the components not to depend on information about other components' interfaces, so components can be **easily replaced**.
- The provider wants the application to run whether or **not all components are up and running simultaneously**.
- The application business model allows a component to send information to another and to continue to operate **without receiving an immediate response**.

For example, components of an enterprise application for an automobile manufacturer can use the JMS API in situations like the following.

- The inventory component can send a message to the factory component when the inventory level for a product goes below a certain level so the factory can make more cars.
- The factory component can send a message to the parts components so the factory can assemble the parts it needs.
- The parts components in turn can send messages to their own inventory and order components to update their inventories and to order new parts from suppliers.
- Both the factory and the parts components can send messages to the accounting component to update budget numbers.

- The business can publish updated catalog items to its sales force.

Using messaging for these tasks allows the various components to interact with one another efficiently, without tying up network or other resources.

Manufacturing is only one example of how an enterprise can use the JMS API. Retail applications, financial services applications, health services applications, and many others can make use of messaging.

# JMS versus J2EE

6

- JMS API deel van J2EE
- Verschillende componenten kunnen berichten sturen en ontvangen
- Clients kunnen luisteren naar berichten
- Combineerbaar met JTA (Java Transaction API)

Industrieel Ingenieur Informatica, UGent

## How Does the JMS API Work with the Java EE Platform?

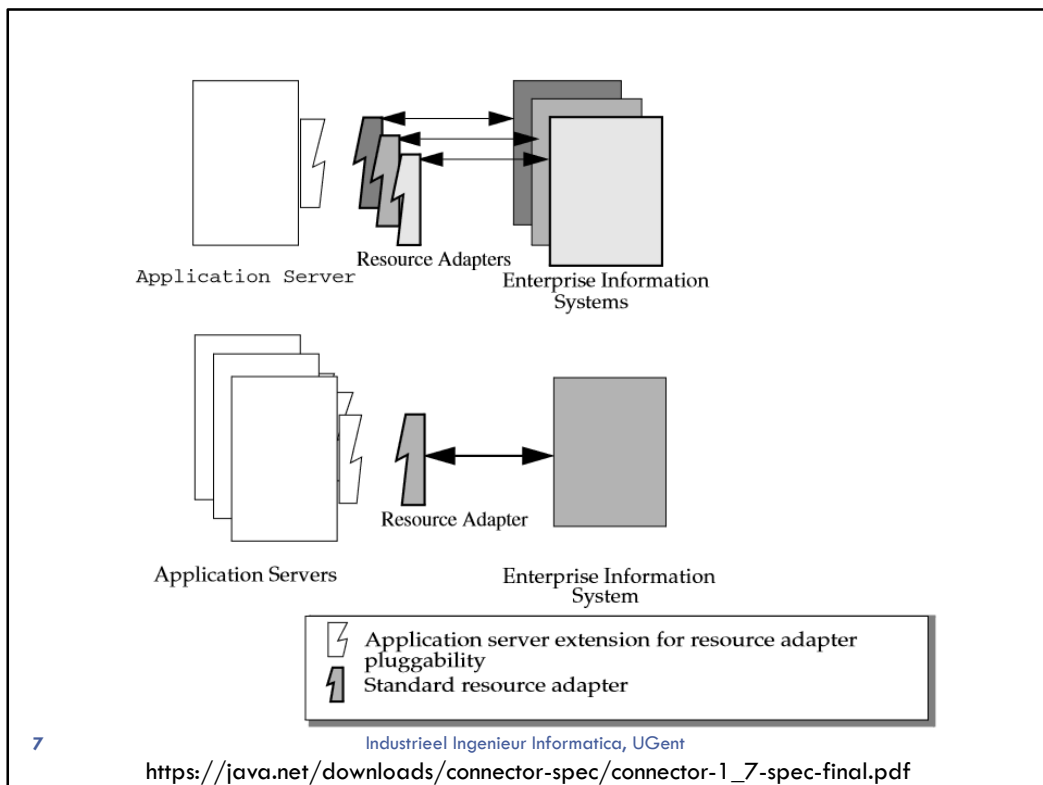
When the JMS API was first introduced, its most important purpose was to allow Java applications to access existing messaging-oriented middleware (MOM) systems. Since that time, many vendors have adopted and implemented the JMS API, so a JMS product can now provide a complete messaging capability for an enterprise. The JMS API is an integral part of the Java EE platform, and application developers can use messaging with Java EE components. JMS 2.0 is part of the Java EE 7 release. The JMS API in the Java EE platform has the following features.

- Application clients, Enterprise JavaBeans (EJB) components, and web components can send or synchronously receive a JMS message. Application clients can in addition set a message listener that allows JMS messages to be delivered to it asynchronously by being notified when a message is available.
- Message-driven beans, which are a kind of enterprise bean, enable the asynchronous consumption of messages in the EJB container. An application server typically pools message-driven beans to implement concurrent processing of messages.
- Message send and receive operations can participate in Java Transaction API (JTA) transactions, which allow JMS operations and database accesses to take place within a single transaction.

The JMS API enhances the other parts of the Java EE platform by simplifying enterprise development, allowing loosely coupled, reliable, asynchronous interactions among Java EE components and legacy systems capable of messaging. A

developer can easily add new behavior to a Java EE application that has existing business events by adding a new message-driven bean to operate on specific business events. The Java EE platform, moreover, enhances the JMS API by providing support for JTA transactions and allowing for the concurrent consumption of messages.

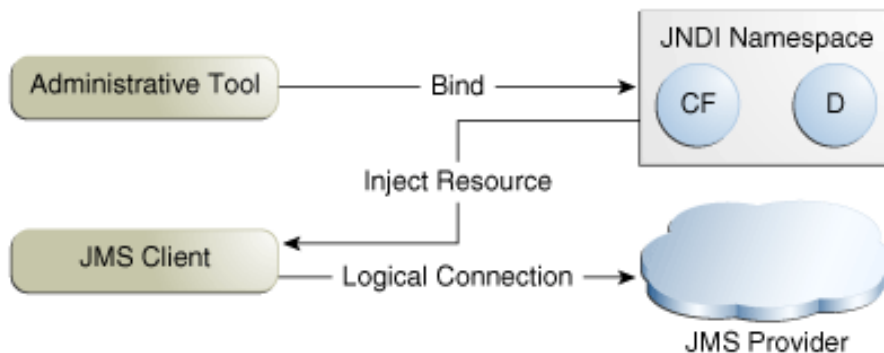




The JMS provider can be integrated with the application server using the Java EE Connector architecture. You access the JMS provider through a resource adapter. This capability allows vendors to create JMS providers that can be plugged in to multiple application servers, and it allows application servers to support multiple JMS providers.

# JMS API Architecture

8



<https://docs.oracle.com/javaee/7/tutorial/jms-concepts002.htm>

Industrieel Ingenieur Informatica, UGent

## JMS API Architecture

A JMS application is composed of the following parts.

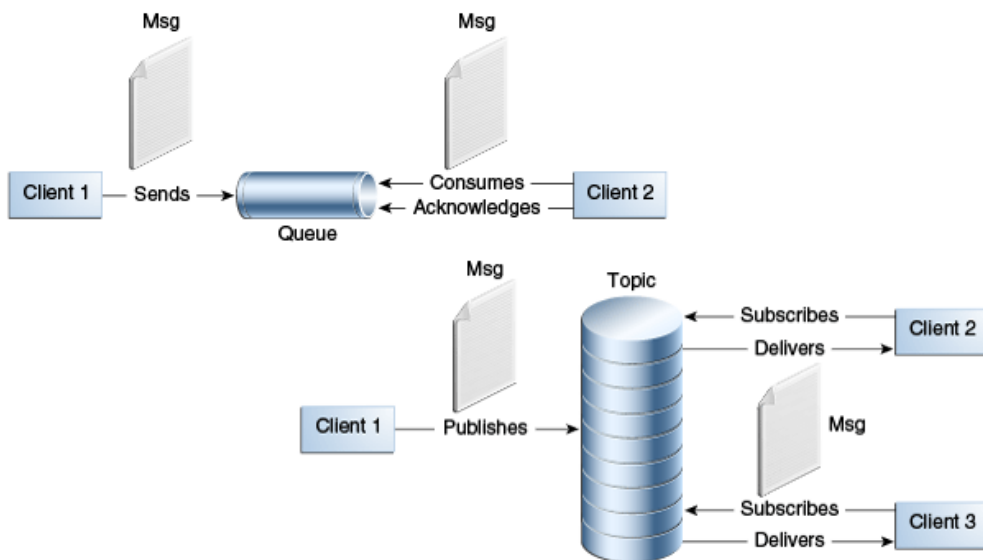
- A **JMS provider** is a messaging system that implements the JMS interfaces and provides administrative and control features. An implementation of the Java EE platform that supports the full profile includes a JMS provider.
- **JMS clients** are the programs or components, written in the Java programming language, that produce and consume messages. Any Java EE application component can act as a JMS client.
- Java SE applications can also act as JMS clients; the Message Queue Developer's Guide for Java Clients in the GlassFish Server documentation (<https://glassfish.java.net/docs/>) explains how to make this work.
- **Messages** are the objects that communicate information between JMS clients.
- **Administered objects** are JMS objects configured for the use of clients. The two kinds of JMS administered objects are **destinations** and **connection factories**, described in [JMS Administered Objects](#). An administrator can create objects that are available to all applications that use a particular installation of GlassFish Server; alternatively, a developer can use annotations to create objects that are specific to a particular application.

The figure illustrates the way these parts interact. Administrative tools or annotations allow you to bind destinations and connection factories into a JNDI namespace. A JMS client can then use resource injection to access the administered objects in the namespace and then establish a logical connection to the same

objects through the JMS provider.

# Messaging Styles

9



<https://docs.oracle.com/javaee/7/tutorial/jms-concepts002.htm>

## Messaging Styles

Before the JMS API existed, most messaging products supported either the **point-to-point** or the **publish/subscribe** style of messaging. The JMS specification defines compliance for each style. A JMS provider must implement both styles, and the JMS API provides interfaces that are specific to each. The following subsections describe these messaging styles.

The JMS API, however, makes it unnecessary to use only one of the two styles. It allows you to use the same code to send and receive messages using either the PTP or the pub/sub style. The destinations you use remain specific to one style, and the behavior of the application will depend in part on whether you are using a queue or a topic. However, **the code itself can be common to both styles**, making your applications flexible and reusable.

### Point-to-Point Messaging Style

A point-to-point (PTP) product or application is built on the concept of message queues, senders, and receivers. Each message is addressed to a specific queue, and receiving clients extract messages from the queues established to hold their messages. Queues retain all messages sent to them until the messages are consumed or expire.

PTP messaging, illustrated in Figure 1, has the following characteristics.

- Each message has only one consumer.
- The receiver can fetch the message whether or not it was running when the client sent the message.

Use PTP messaging when every message you send must be processed successfully by one consumer.

### **Publish/Subscribe Messaging Style**

In a publish/subscribe (pub/sub) product or application, clients address messages to a topic, which functions somewhat like a bulletin board. Publishers and subscribers can dynamically publish or subscribe to the topic. The system takes care of distributing the messages arriving from a topic's multiple publishers to its multiple subscribers. Topics retain messages only as long as it takes to distribute them to subscribers.

With pub/sub messaging, it is important to distinguish between the consumer that subscribes to a topic (the subscriber) and the subscription that is created. The consumer is a JMS object within an application, while the subscription is an entity within the JMS provider. Normally, a topic can have many consumers, but a subscription has only one subscriber. It is possible, however, to create shared subscriptions.

Pub/sub messaging has the following characteristics.

- Each message can have multiple consumers.
- A client that subscribes to a topic can consume only messages sent after the client has created a subscription, and the consumer must continue to be active in order for it to consume messages.
- The JMS API relaxes this requirement to some extent by allowing applications to create durable subscriptions, which receive messages sent while the consumers are not active. Durable subscriptions provide the flexibility and reliability of queues but still allow clients to send messages to many recipients.

Use pub/sub messaging when each message can be processed by any number of consumers (or none). [Figure 2](#) illustrates pub/sub messaging.

### **Message Consumption**

Messaging products are inherently asynchronous: There is no fundamental timing dependency between the production and the consumption of a message. However, the JMS specification uses this term in a more precise sense. Messages can be consumed in either of two ways.

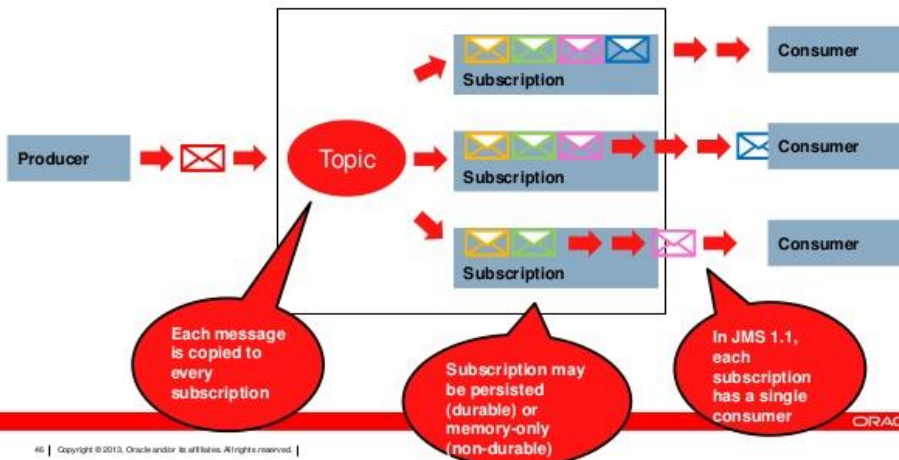
- **Synchronously:** A consumer explicitly fetches the message from the destination by calling the receive method. The receive method can block until a message arrives or can time out if a message does not arrive within a specified time limit.
- **Asynchronously:** An application client or a Java SE client can register a message listener with a consumer. A message listener is similar to an event listener. Whenever a message arrives at the destination, the JMS provider delivers the message by calling the listener's onMessage method, which acts on the contents of the message. In a Java EE application, a message-driven bean serves as a message listener (it too has an onMessage method), but a client does not need to register it with a consumer.

# Topics

10



## How topics work in JMS 1.1



46 | Copyright © 2013, Oracle and/or its affiliates. All rights reserved. |

<http://www.slideshare.net/C2B2/new-jms-featuresinglassfish4>

# Overzicht

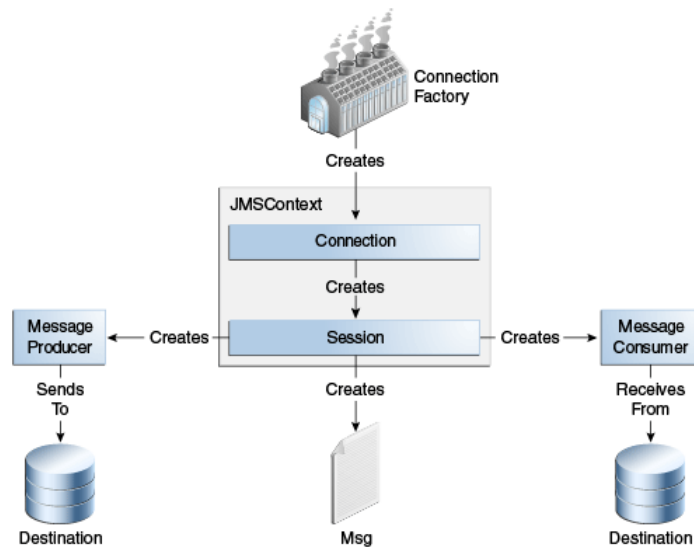
11

- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - Listeners
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

# The JMS API Programming Model

12



Industrieel Ingenieur Informatica, UGent

<https://docs.oracle.com/javaee/7/tutorial/jms-concepts003.htm>

## The JMS API Programming Model

The basic building blocks of a JMS application are

- Administered objects: connection factories and destinations
- Connections
- Sessions
- JMSContext objects, which combine a connection and a session in one object
- Message producers
- Message consumers
- Messages

The figure shows how all these objects fit together in a JMS client application.

JMS also provides queue browsers, objects that allow an application to browse messages on a queue.





## JMS Administered Objects

Two parts of a JMS application, **destinations** and **connection factories**, are commonly maintained administratively rather than programmatically. The technology underlying these objects is likely to be very different from one implementation of the JMS API to another. Therefore, the management of these objects belongs with other administrative tasks that vary from provider to provider.

JMS clients access administered objects through interfaces that are portable, so a client application can run with little or no change on more than one implementation of the JMS API. Ordinarily, an administrator configures administered objects in a JNDI namespace, and JMS clients then access them by using resource injection.

With GlassFish Server, you can use the Administration Console to create JMS administered objects in the form of connector resources. You can also specify the resources in a file named `glassfish-resources.xml` that you can bundle with an application.

The Java EE platform specification allows a developer to create administered objects using annotations or deployment descriptor elements. Objects created in this way are specific to the application for which they are created. Definitions in a deployment descriptor override those specified by annotations.

## Starting the JMS Provider

When you use GlassFish Server, your JMS provider is GlassFish Server.

### **Creating JMS Administered Objects**

This example uses the following JMS administered objects:

- A connection factory
- Two destination resources: a topic and a queue

# JMS Administered Objects

14

```
import javax.annotation.Resource;
import javax.jms.ConnectionFactory;
import javax.jms.Topic;
import javax.jms.Queue;

public class Producer {
    @Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
    private static ConnectionFactory connectionFactory;
    @Resource(lookup = "jms/MyQueue")
    private static Queue queue;
    @Resource(lookup = "jms/MyTopic")
    private static Topic topic;
```

## JMS Connection Factories

A connection factory is the object a client uses to create a connection to a provider. A connection factory encapsulates a set of connection configuration parameters that has been defined by an administrator. Each connection factory is an instance of the ConnectionFactory, QueueConnectionFactory, or TopicConnectionFactory interface.

At the beginning of a JMS client program, you usually inject a connection factory resource into a ConnectionFactory object. A Java EE server must provide a JMS connection factory with the logical JNDI name java:comp/DefaultJMSConnectionFactory. The actual JNDI name will be implementation-specific.

For example, the following code fragment looks up the default JMS connection factory and assigns it to a ConnectionFactory object:

```
@Resource(lookup = "java:comp/DefaultJMSConnectionFactory")
private static ConnectionFactory connectionFactory;
```

## JMS Destinations

A destination is the object a client uses to specify the target of messages it produces and the source of messages it consumes. In the PTP messaging style, destinations are called queues. In the pub/sub messaging style, destinations are called topics. A JMS application can use multiple queues or topics (or both).

To create a destination using GlassFish Server, you create a JMS destination resource that specifies a JNDI name for the destination.

In the GlassFish Server implementation of JMS, each destination resource refers to a physical destination. You can create a physical destination explicitly, but if you do not, the Application Server creates it when it is needed and deletes it when you delete the destination resource.

In addition to injecting a connection factory resource into a client program, you usually inject a destination resource. Unlike connection factories, destinations are specific to either the PTP or pub/sub messaging style. To create an application that allows you to use the same code for both topics and queues, you assign the destination to a Destination object.

The following code specifies two resources, a queue and a topic. The resource names are mapped to destination resources created in the JNDI namespace:

```
@Resource(lookup = "jms/MyQueue")
private static Queue queue;
@Resource(lookup = "jms/MyTopic")
private static Topic topic;
```

In a Java EE application, JMS administered objects are normally placed in the jms naming subcontext.

With the common interfaces, you can mix or match connection factories and destinations. That is, in addition to using the ConnectionFactory interface, you can inject a QueueConnectionFactory resource and use it with a Topic, and you can inject a TopicConnectionFactory resource and use it with a Queue. The behavior of the application will depend on the kind of destination you use and not on the kind of connection factory you use.

### **Resource Creation**

A resource is a program object that provides connections to such systems as database servers and messaging systems. Java EE components can access a wide variety of resources, including databases, mail sessions, Java Message Service objects, and URLs. The Java EE 7 platform provides mechanisms that allow you to access all these resources in a similar manner.

### **Resources and JNDI Naming**

In a distributed application, components need to access other components and resources, such as databases. For example, a servlet might invoke remote methods on an enterprise bean that retrieves information from a database. In the Java EE platform, the Java Naming and Directory Interface (JNDI) naming service enables components to locate other components and resources.

A resource is a program object that provides connections to systems, such as database servers and messaging systems. (A Java Database Connectivity resource is

sometimes referred to as a data source.) Each resource object is identified by a unique, people-friendly name, called the JNDI name. For example, the JNDI name of the preconfigured JDBC resource for the Java DB database that is shipped with GlassFish Server is `java:comp/DefaultDataSource`.

An administrator creates resources in a JNDI namespace. In GlassFish Server, you can use either the Administration Console or the `asadmin` command to create resources. Applications then use annotations to inject the resources. If an application uses resource injection, GlassFish Server invokes the JNDI API, and the application is not required to do so. However, it is also possible for an application to locate resources by making direct calls to the JNDI API.

A resource object and its JNDI name are bound together by the naming and directory service. To create a new resource, a new name/object binding is entered into the JNDI namespace. You inject resources by using the `@Resource` annotation in an application.

You can use a deployment descriptor to override the resource mapping that you specify in an annotation. Using a deployment descriptor allows you to change an application by repackaging it rather than by both recompiling the source files and repackaging. However, for most applications a deployment descriptor is not necessary.

### Creating Resources Administratively

Before you deploy or run many applications, you may need to create resources for them. An application can include a `glassfish-resources.xml` file that can be used to define resources for that application and others. You can then use the `asadmin` command, specifying as the argument a file named `glassfish-resources.xml`, to create the resources administratively, as shown.

```
asadmin add-resources glassfish-resources.xml
```

The `glassfish-resources.xml` file can be created in any project using NetBeans IDE or by hand.

You could also use the `asadmin create-jms-resource` command to create the resources for this example. When you are done using the resources, you would use `asadmin` command to display their names, and the `asadmin delete-jms-resource` command to remove them, regardless of the way you created the resources.

### Resource Injection

Resource injection enables you to inject any resource available in the JNDI namespace into any container-managed object, such as a servlet, an enterprise bean, or a managed bean. For example, you can use resource injection to inject data sources, connectors, or custom resources available in the JNDI namespace.

The type you use for the reference to the injected instance is usually an interface,

which decouples your code from the implementation of the resource.

For example, the following code injects a data source object that provides connections to the default Java DB database shipped with GlassFish Server:

```
public class MyServlet extends HttpServlet {
    @Resource(name="java:comp/DefaultDataSource")
    private javax.sql.DataSource dsc;
    ...
}
```

In addition to field-based injection as in the preceding example, you can inject resources using method-based injection:

```
public class MyServlet extends HttpServlet {
    private javax.sql.DataSource dsc;
    ...
    @Resource(name="java:comp/DefaultDataSource")
    public void setDsc(javax.sql.DataSource ds) {
        dsc = ds;
    }
}
```

To use method-based injection, the setter method must follow the JavaBeans conventions for property names: The method name must begin with set, have a void return type, and have only one parameter.

The `@Resource` annotation is in the `javax.annotation` package and is defined in JSR 250 (Common Annotations for the Java Platform). Resource injection resolves by name, so it is not typesafe: the type of the resource object is not known at compile time, so you can get runtime errors if the types of the object and its reference do not match.

# Overzicht

15

- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - Listeners
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

# JMSContext

16

```
try (JMSContext context = connectionFactory.createContext()); {  
    ... // berichten maken, browsen, sturen of ontvangen  
} catch (JMSRuntimeException e) {  
    ...  
}
```

## Connections

A connection encapsulates a virtual connection with a JMS provider. For example, a connection could represent an open TCP/IP socket between a client and a provider service daemon. You use a connection to create one or more sessions.

Note:

In the Java EE platform, the ability to create multiple sessions from a single connection is limited to application clients. In web and enterprise bean components, a connection can create no more than one session.

You normally create a connection by creating a JMSContext object.

## Sessions

A session is a single-threaded context for producing and consuming messages. You normally create a session (as well as a connection) by creating a JMSContext object.

You use sessions to create message producers, message consumers, messages, queue browsers, and temporary destinations.

Sessions serialize the execution of message listeners.

A session provides a transactional context with which to group a set of sends and receives into an atomic unit of work.

## JMSContext Objects

A JMSContext object combines a connection and a session in a single object. That is,



it provides both an active connection to a JMS provider and a single-threaded context for sending and receiving messages.

You use the JMSContext to create the following objects:

- Message producers
- Message consumers
- Messages
- Queue browsers

You can create a JMSContext in a try-with-resources block.

To create a JMSContext, call the createContext method on the connection factory:

```
JMSContext context = connectionFactory.createContext();
```

When called with no arguments from an application client or a Java SE client, or from the Java EE web or EJB container when there is no active JTA transaction in progress, the createContext method creates a non-transacted session with an acknowledgment mode of JMSContext.AUTO\_ACKNOWLEDGE.

When called with no arguments from the web or EJB container when there is an active JTA transaction in progress, the createContext method creates a transacted session.

From an application client or a Java SE client, you can also call the createContext method with the argument JMSContext.SESSION\_TRANSACTED to create a transacted session:

```
JMSContext context =  
connectionFactory.createContext(JMSContext.SESSION_TRANSACTED);
```

The session uses local transactions.

Alternatively, you can specify a non-default acknowledgment mode.

When you use a JMSContext, message delivery normally begins as soon as you create a consumer.

If you create a JMSContext in a try-with-resources block, you do not need to close it explicitly. It will be closed when the try block comes to an end. Make sure that your application completes all its JMS activity within the try-with-resources block. If you do not use a try-with-resources block, you must call the close method on the JMSContext to close the connection when the application has finished its work.

# JMS Message Producers

17

```
try (JMSContext context = connectionFactory.createContext();) {
    int count = 0;
    for (int i = 0; i < NUM_MSGS; i++) {
        String message = "This is message " + (i + 1)
            + " from producer";
        context.createProducer().send(dest, message);
        count += 1;
    }
    //Send a non-text control message indicating end of messages.
    context.createProducer().send(dest, context.createMessage());
} catch (JMSRuntimeException e) {
    ...
}
```

## JMS Message Producers

A message producer is an object that is created by a JMSContext or a session and used for sending messages to a destination. A message producer created by a JMSContext implements the JMSProducer interface. You could create it this way:

```
try (JMSContext context = connectionFactory.createContext();) {
    JMSProducer producer = context.createProducer(); ...
}
```

However, a JMSProducer is a lightweight object that does not consume significant resources. For this reason, you do not need to save the JMSProducer in a variable; you can create a new one each time you send a message. You send messages to a specific destination by using the send method. For example:

```
context.createProducer().send(dest, message);
```

You can create the message in a variable before sending it, as shown here, or you can create it within the send call.

# Producer

18

```
C:\Users\vongenae\Documents\Gedistribueerde toepassingen\voorbeelden\jms\Producer>appclient -client dist\producer.jar queue 3
okt 26, 2015 2:13:18 PM org.hibernate.validator.internal.util.Version <clinit>
INFO: HV000001: Hibernate Validator 5.0.0.Final
okt 26, 2015 2:13:18 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter: Version: 5.1 (Build 9
-b) Compile: July 29 2014 1229
okt 26, 2015 2:13:18 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter starting: broker is REMO
TE, connection mode is TCP
okt 26, 2015 2:13:18 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter Started:REMOTE
Destination type is queue
Sending message: This is message 1 from producer
Sending message: This is message 2 from producer
Sending message: This is message 3 from producer
Text messages sent: 3
```

Industrieel Ingenieur Informatica, UGent

This section shows how to create, package, and run simple JMS clients that are packaged as application clients. The clients demonstrate the basic tasks a JMS application must perform:

- Creating a JMSContext
- Creating message producers and consumers
- Sending and receiving messages

Each example uses two clients: one that sends messages and one that receives them. You can run the clients in two terminal windows or use Netbeans projects type “Enterprise Application Client”.

## Sending Messages

This section describes how to use a client to send messages. The Producer.java client will send messages in all of these examples.

The general steps this example performs are as follows.

- **Inject resources** for the administered objects used by the example.
- Accept and verify command-line arguments. You can use this example to send any number of messages to either a queue or a topic, so you specify the destination type and the number of messages on the command line when you run the program.
- **Create a JMSContext**, then **send** the specified number of **text messages** in the form of strings.
- **Send a final message** of type Message to indicate that the consumer should

expect no more messages.

- Catch any exceptions.

# Overzicht

19

- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - Listeners
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

# JMS Message Consumers

20

```
try (JMSContext context = connectionFactory.createContext()); {
    JMSConsumer consumer = context.createConsumer(dest);
    while (true) {
        Message m = consumer.receive(1000);
        if (m != null) {
            if (m instanceof TextMessage) {
                System.out.println("Reading message: " +
                    m.getBody(String.class));
            } else {break;}
        }
    }
} catch (JMSRuntimeException e) {
    ...
}
```

## JMS Message Consumers

A message consumer is an object that is created by a JMSContext or a session and used for receiving messages sent to a destination.

A message consumer created by a JMSContext implements the JMSConsumer interface. The simplest way to create a message consumer is to use the JMSContext.createConsumer method:

```
try (JMSContext context = connectionFactory.createContext()); {
    JMSConsumer consumer = context.createConsumer(dest);
    ...
}
```

A message consumer allows a JMS client to register interest in a destination with a JMS provider. The JMS provider manages the delivery of messages from a destination to the registered consumers of the destination.

When you use a JMSContext to create a message consumer, message delivery begins as soon as you have created the consumer. You can disable this behavior by calling setAutoStart(false) when you create the JMSContext and then calling the start method explicitly to start message delivery. If you want to stop message delivery temporarily without closing the connection, you can call the stop method; to restart message delivery, call start.

You use the receive method to consume a message synchronously. You can use this method at any time after you create the consumer.

If you specify no arguments or an argument of 0, the method blocks indefinitely until a message arrives:

```
Message m = consumer.receive();  
Message m = consumer.receive(0);
```

For a simple client, this may not matter. But if it is possible that a message might not be available, use a synchronous receive with a timeout:

Call the receive method with a timeout argument greater than 0. One second is a recommended timeout value:

```
Message m = consumer.receive(1000); // time out after a second
```

To enable asynchronous message delivery from an application client or a Java SE client, you use a message listener, as described in the next section.

You can use the `JMSTContext.createDurableConsumer` method to create a durable topic subscription. This method is valid only if you are using a topic. For topics, you can also create shared consumers.

# Consumer: voorbeeld

21

```
C:\Users\vongenae\Documents\Gedistribueerde toepassingen\voorbeelden\jms\SynchConsumer>appclient -client dist\SynchConsumer.jar queue
okt 26, 2015 2:17:08 PM org.hibernate.validator.internal.util.Version <clinit>
INFO: HV000001: Hibernate Validator 5.0.0.Final
okt 26, 2015 2:17:09 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter: Version: 5.1 (Build 9
-b) Compile: July 29 2014 1229
okt 26, 2015 2:17:09 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter starting: broker is REMO
TE, connection mode is TCP
okt 26, 2015 2:17:09 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter Started:REMOTE
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Messages received: 3
```

Industrieel Ingenieur Informatica, UGent

## Receiving Messages Synchronously

This section describes the receiving client, which uses the receive method to consume messages synchronously.

### The SynchConsumer.java Client

The receiving client, SynchConsumer.java, performs the following steps.

- **Injects resources** for a connection factory, queue, and topic.
- Assigns either **the queue or the topic** to a destination object, based on the specified destination type.
- Within a try-with-resources block, **creates a JMSContext**.
- **Creates a JMSConsumer**, starting message delivery:  
consumer = context.createConsumer(dest);
- **Receives the messages** sent to the destination until the end-of-message-stream control message is received:

```
int count = 0;
while (true) {
    Message m = consumer.receive(1000);
    if (m != null) {
        if (m instanceof TextMessage) {
            System.out.println("Reading message: " + m.getBody(String.class));
            count += 1;
        } else { break; }
    }
}
```



```
}
```

```
System.out.println("Messages received: " + count);
```

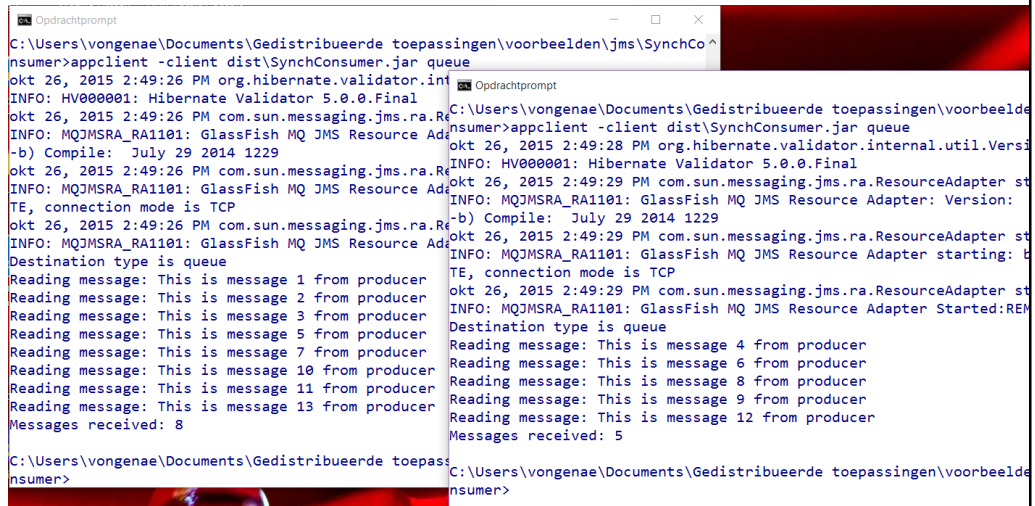
Because the control message is not a `TextMessage`, the receiving client terminates the while loop and stops receiving messages after the control message arrives.

- **Catches and handles any exceptions.** The end of the try-with-resources block automatically causes the `JMSContext` to be closed.

The `SynchConsumer` client uses an indefinite while loop to receive messages, calling `receive` with a timeout argument.

# Queue met 2 consumers

22



```
C:\Users\vongenae\Documents\Gedistribueerde toepassingen\voorbeelden\jms\SynchCo...
nsumer>appclient -client dist\SynchConsumer.jar queue
okt 26, 2015 2:49:26 PM org.hibernate.validator.int
INFO: HV000001: Hibernate Validator 5.0.0.Final
okt 26, 2015 2:49:26 PM com.sun.messaging.jms.ra.R
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Ad
-b) Compile: July 29 2014 1229
okt 26, 2015 2:49:26 PM com.sun.messaging.jms.ra.R
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Ad
TE, connection mode is TCP
okt 26, 2015 2:49:26 PM com.sun.messaging.jms.ra.R
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Ad
Destination type is queue
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Reading message: This is message 5 from producer
Reading message: This is message 7 from producer
Reading message: This is message 10 from producer
Reading message: This is message 11 from producer
Reading message: This is message 13 from producer
Messages received: 8

C:\Users\vongenae\Documents\Gedistribueerde toepas
nsumer>
```

```
C:\Users\vongenae\Documents\Gedistribueerde toepassingen\voorbeelde
nsumer>appclient -client dist\SynchConsumer.jar queue
okt 26, 2015 2:49:28 PM org.hibernate.validator.internal.util.Versi
INFO: HV000001: Hibernate Validator 5.0.0.Final
okt 26, 2015 2:49:29 PM com.sun.messaging.jms.ra.ResourceAdapter st
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter: Version:
-b) Compile: July 29 2014 1229
okt 26, 2015 2:49:29 PM com.sun.messaging.jms.ra.ResourceAdapter st
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter starting: b
TE, connection mode is TCP
okt 26, 2015 2:49:29 PM com.sun.messaging.jms.ra.ResourceAdapter st
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter Started:REM
Destination type is queue
Reading message: This is message 4 from producer
Reading message: This is message 6 from producer
Reading message: This is message 8 from producer
Reading message: This is message 9 from producer
Reading message: This is message 12 from producer
Messages received: 5

C:\Users\vongenae\Documents\Gedistribueerde toepas
nsumer>
```

Industrieel Ingenieur Informatica, UGent

# Overzicht

23

- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - **Listeners**
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

## Asynchronous consumers

24

```
try (JMSContext context = connectionFactory.createContext();) {  
  
    JMSConsumer consumer = context.createConsumer(dest);  
    TextListener listener = new TextListener();  
    consumer.setMessageListener(listener);  
  
} catch (JMSRuntimeException e) {  
    ...  
}
```

### Using a Message Listener for Asynchronous Message Delivery

This section describes the receiving clients in an example that uses a message listener for asynchronous message delivery. This section then explains how to compile and run the clients using GlassFish Server.

#### Note:

In the Java EE platform, message listeners can be used only in application clients, as in this example. To allow asynchronous message delivery in a web or enterprise bean application, you use a message-driven bean.

# JMS Message Listeners

25

```
public class TextListener implements MessageListener {
    @Override
    public void onMessage(Message m) {
        try {
            if (m instanceof TextMessage) {
                ...
            } else {
                ...
            }
        } catch (JMSException | JMSRuntimeException e) {
            ...
        }
    }
}
```

## JMS Message Listeners

A message listener is an object that acts as an asynchronous event handler for messages. This object implements the `MessageListener` interface, which contains one method, `onMessage`. In the `onMessage` method, you define the actions to be taken when a message arrives.

From an application client or a Java SE client, you register the message listener with a specific message consumer by using the `setMessageListener` method. For example, if you define a class named `Listener` that implements the `MessageListener` interface, you can register the message listener as follows:

```
Listener myListener = new Listener();
consumer.setMessageListener(myListener);
```

When message delivery begins, the JMS provider automatically calls the message listener's `onMessage` method whenever a message is delivered. The `onMessage` method takes one argument of type `Message`, which your implementation of the method can cast to another message subtype as needed.

In the Java EE web or EJB container, you use message-driven beans for asynchronous message delivery. A message-driven bean also implements the `MessageListener` interface and contains an `onMessage` method.

Your `onMessage` method should handle all exceptions. Throwing a `RuntimeException` is considered a programming error.

# Consumer: voorbeeld

26

```
C:\Users\vongenae\Documents\Gedistribueerde toepassingen\voorbeelden\jms\AsynchC
onsumer>appclient -client dist\AsynchConsumer.jar topic
okt 26, 2015 2:29:37 PM org.hibernate.validator.internal.util.Version <clinit>
INFO: HV000001: Hibernate Validator 5.0.0.Final
okt 26, 2015 2:29:37 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter: Version: 5.1 (Build 9
-b) Compile: July 29 2014 1229
okt 26, 2015 2:29:37 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter starting: broker is REMO
TE, connection mode is TCP
okt 26, 2015 2:29:37 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter Started:REMOTE
Destination type is topic
To end program, enter Q or q, then <return>
```

Industrieel Ingenieur Informatica, UGent

```
To end program, enter Q or q, then <return>
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Reading message: This is message 4 from producer
Reading message: This is message 5 from producer
Reading message: This is message 6 from producer
Reading message: This is message 7 from producer
Reading message: This is message 8 from producer
Reading message: This is message 9 from producer
Reading message: This is message 10 from producer
Reading message: This is message 11 from producer
Reading message: This is message 12 from producer
Reading message: This is message 13 from producer
Reading message: This is message 14 from producer
Reading message: This is message 15 from producer
Reading message: This is message 16 from producer
Reading message: This is message 17 from producer
Message is not a TextMessage
Reading message: This is message 1 from producer
Reading message: This is message 2 from producer
Reading message: This is message 3 from producer
Reading message: This is message 4 from producer
Reading message: This is message 5 from producer
Reading message: This is message 6 from producer
```

# Message Properties en selectors

28

```
Message msg = ...
```

```
// Setting message properties  
msg.setStringProperty("title", "Thriller");  
msg.setIntProperty("releaseYear", 1982);
```

```
JMSContext ctx = ...  
MessageConsumer consumer  
    = ctx.createConsumer(queue, "releaseYear < 1980");
```

## JMS Message Selectors

If your messaging application needs to filter the messages it receives, you can use a JMS message selector, which allows a message consumer for a destination to specify the messages that interest it. Message selectors assign the work of filtering messages to the JMS provider rather than to the application.

A message selector is a String that contains an expression. The syntax of the expression is based on a subset of the SQL92 conditional expression syntax. The message selector in the example selects any message that has a NewsType property that is set to the value 'Sports' or 'Opinion':

NewsType = 'Sports' OR NewsType = 'Opinion'

The createConsumer and createDurableConsumer methods, as well as the methods for creating shared consumers, allow you to specify a message selector as an argument when you create a message consumer.

The message consumer then receives only messages whose headers and properties match the selector. A message selector cannot select messages on the basis of the content of the message body.



# Overzicht

29

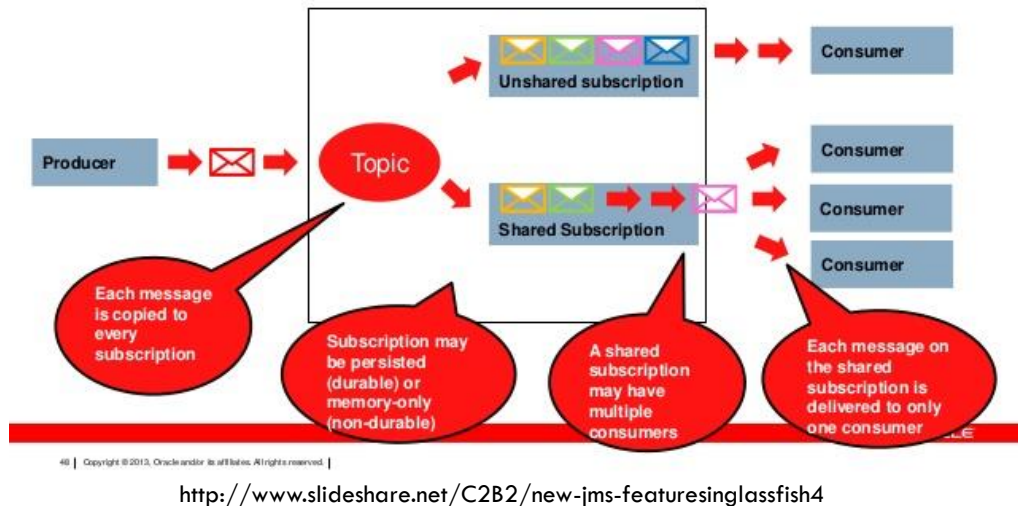
- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - Listeners
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

# Consuming Messages from Topics

30

## Shared subscriptions in JMS 2.0



### Consuming Messages from Topics

The semantics of consuming messages from topics are more complex than the semantics of consuming messages from queues.

An application consumes messages from a topic by creating a subscription on that topic and creating a consumer on that subscription. Subscriptions may be **durable** or **nondurable**, and they may be **shared** or **unshared**.

A subscription may be thought of as an entity within the JMS provider itself, whereas a consumer is a JMS object within the application.

A subscription will receive a copy of every message that is sent to the topic after the subscription is created, unless a message selector is specified. If a message selector is specified, only those messages whose properties match the message selector will be added to the subscription.

Unshared subscriptions are restricted to a single consumer. In this case, all the messages in the subscription are delivered to that consumer. Shared subscriptions allow multiple consumers. In this case, each message in the subscription is delivered to only one consumer. JMS does not define how messages are distributed between multiple consumers on the same subscription.

Subscriptions may be **durable** or **nondurable**.

A nondurable subscription exists only as long as there is an active consumer on the subscription. This means that any messages sent to the topic will be added to the subscription only while a consumer exists and is not closed.

A nondurable subscription may be either unshared or shared.

An unshared nondurable subscription does not have a name and may have only a single consumer object associated with it. It is created automatically when the consumer object is created. It is not persisted and is deleted automatically when the consumer object is closed.

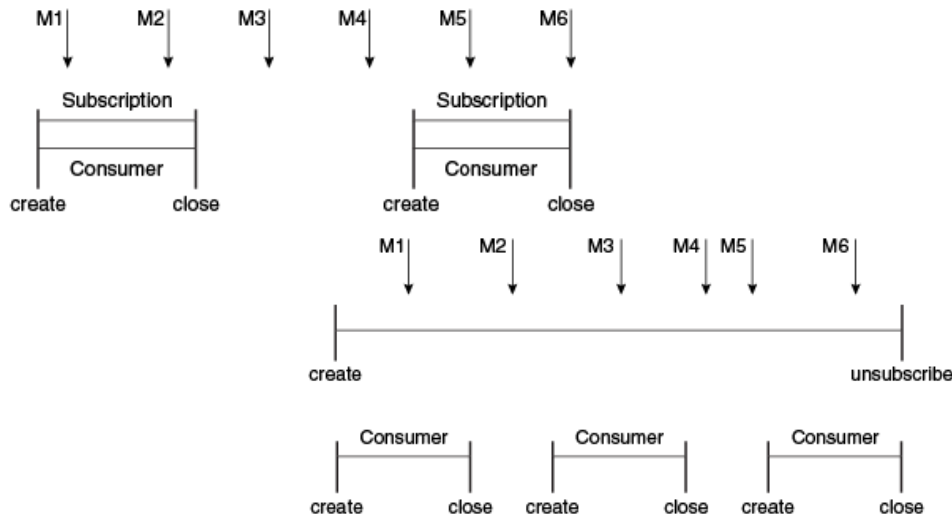
The `JMSContext.createConsumer` method creates a consumer on an unshared nondurable subscription if a topic is specified as the destination.

A shared nondurable subscription is identified by name and an optional client identifier, and may have several consumer objects consuming messages from it. It is created automatically when the first consumer object is created. It is not persisted and is deleted automatically when the last consumer object is closed.

At the cost of higher overhead, a subscription may be durable. A durable subscription is persisted and continues to accumulate messages until explicitly deleted, even if there are no consumer objects consuming messages from it.

# Durable Subscriptions

31



## Creating Durable Subscriptions

To ensure that a pub/sub application receives all sent messages, use durable subscriptions for the consumers on the topic.

Like a nondurable subscription, a durable subscription may be either unshared or shared.

An unshared durable subscription is identified by name and client identifier (which must be set) and may have only a single consumer object associated with it.

A shared durable subscription is identified by name and an optional client identifier, and may have several consumer objects consuming messages from it.

A durable subscription that exists but that does not currently have a non-closed consumer object associated with it is described as being inactive.

You can use the `JMSContext.createDurableConsumer` method to create a consumer on an unshared durable subscription. An unshared durable subscription can have only one active consumer at a time.

A consumer identifies the durable subscription from which it consumes messages by specifying a unique identity that is retained by the JMS provider. Subsequent consumer objects that have the same identity resume the subscription in the state in

which it was left by the preceding consumer. If a durable subscription has no active consumer, the JMS provider retains the subscription's messages until they are received by the subscription or until they expire.

You establish the unique identity of an unshared durable subscription by setting the following:

- A client ID for the connection
- A topic and a subscription name for the subscription

You can set the client ID administratively for a client-specific connection factory using either the command line or the Administration Console. (In an application client or a Java SE client, you can instead call `JMSContext.setClientID()`.)

After using this connection factory to create the `JMSContext`, you call the `createDurableConsumer` method with two arguments: the topic and a string that specifies the name of the subscription:

```
String subName = "MySub";
JMSConsumer consumer = context.createDurableConsumer(myTopic,
subName);
```

The subscription becomes active after you create the consumer. Later, you might close the consumer:

```
consumer.close();
```

The JMS provider stores the messages sent to the topic, as it would store messages sent to a queue. If the program or another application calls `createDurableConsumer` using the same connection factory and its client ID, the same topic, and the same subscription name, then the subscription is reactivated and the JMS provider delivers the messages that were sent while the subscription was inactive.

To delete a durable subscription, first close the consumer, then call the `unsubscribe` method with the subscription name as the argument:

```
consumer.close();
context.unsubscribe(subName);
```

The `unsubscribe` method deletes the state the provider maintains for the subscription.

Figure 1 and Figure 2 show the difference between a nondurable and a durable subscription. With an ordinary, nondurable subscription, the consumer and the subscription begin and end at the same point and are, in effect, identical. When the consumer is closed, the subscription also ends. Here, `create` stands for a call to `JMSContext.createConsumer` with a `Topic` argument, and `close` stands for a call to `JMSConsumer.close`. Any messages sent to the topic between the time of the first close and the time of the second create are not added to either subscription. In Figure 1, the consumers receive messages M1, M2, M5, and M6, but they do not

receive messages M3 and M4.

With a durable subscription, the consumer can be closed and re-created, but the subscription continues to exist and to hold messages until the application calls the unsubscribe method. In Figure 2, create stands for a call to `JMSContext.createDurableConsumer`, close stands for a call to `JMSConsumer.close`, and unsubscribe stands for a call to `JMSContext.unsubscribe`. Messages sent after the first consumer is closed are received when the second consumer is created (on the same durable subscription), so even though messages M2, M4, and M5 arrive while there is no consumer, they are not lost.

A shared durable subscription allows you to use multiple consumers to receive messages from a durable subscription. If you use a shared durable subscription, the connection factory you use does not need to have a client identifier. To create a shared durable subscription, call the `JMSContext.createSharedDurableConsumer` method, specifying the topic and subscription name:

```
JMSConsumer consumer = context.createSharedDurableConsumer(topic,
    "MakeltLast");
```

## Durable Subscriptions - aanmelden

32

```
try (  
    JMSContext context = durableConnectionFactory.createContext()  
{  
    consumer = context.createDurableConsumer(topic, "MakeItLast");  
    listener = new TextListener();  
    consumer.setMessageListener(listener);  
    ...  
} catch (JMSRuntimeException e) {...}
```

## Durable Subscriptions - uitschrijven

33

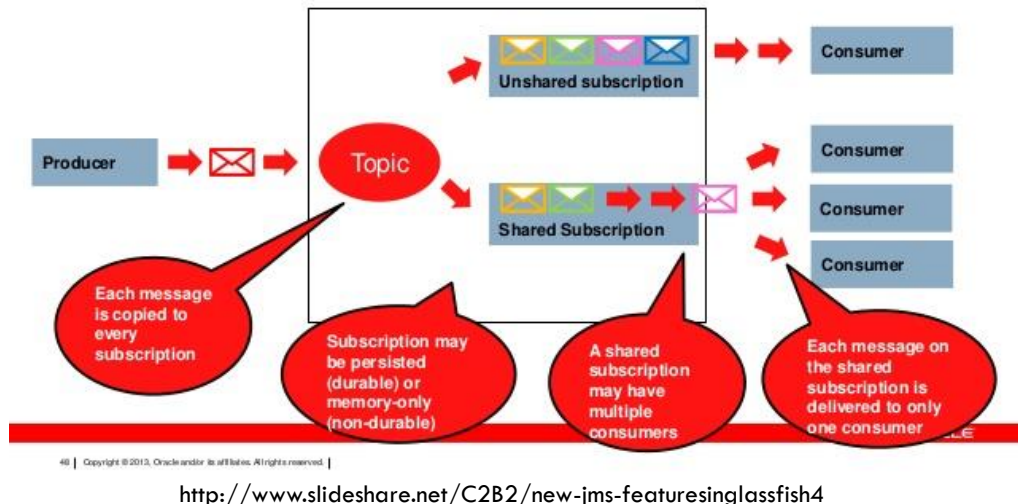
```
try (  
    JMSContext context = durableConnectionFactory.createContext()  
{  
  
    context.unsubscribe("MakeItLast");  
  
} catch (JMSRuntimeException e) {...}
```



# Shared Subscriptions

34

## Shared subscriptions in JMS 2.0



### Creating Shared Subscriptions

A topic subscription created by the `createConsumer` or `createDurableConsumer` method can have only one consumer (although a topic can have many). Multiple clients consuming from the same topic have, by definition, multiple subscriptions to the topic, and all the clients receive all the messages sent to the topic (unless they filter them with message selectors).

It is, however, possible to create a nondurable shared subscription to a topic by using the `createSharedConsumer` method and specifying not only a destination but a subscription name:

```
consumer = context.createSharedConsumer(topicName, "SubName");
```

With a shared subscription, messages will be distributed among multiple clients that use the same topic and subscription name. Each message sent to the topic will be added to every subscription (subject to any message selectors), but each message added to a subscription will be delivered to only one of the consumers on that subscription, so it will be received by only one of the clients. A shared subscription can be useful if you want to share the message load among several consumers on the subscription rather than having just one consumer on the subscription receive each message. This feature can improve the scalability of Java EE application client applications and Java SE applications. (Message-driven beans share the work of processing messages from a topic among multiple threads.)

You can also create shared durable subscriptions by using the `JMSContext.createSharedDurableConsumer` method.

## Shared Subscriptions

35

```
try (JMSContext context = connectionFactory.createContext()) {
    consumer = context.createSharedConsumer(topic, "SubName");

    listener = new TextListener();
    consumer.setMessageListener(listener);

    ...
} catch (JMSRuntimeException e) { ... }
```

## Shared Durable Subscriptions

36

```
try (JMSContext context = connectionFactory.createContext()) {
    consumer
        = context.createSharedDurableConsumer(topic, "MakeItLast");

    listener = new TextListener();
    consumer.setMessageListener(listener);
    ...
} catch (JMSRuntimeException e) { ... }
```

# Overzicht

37

- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - Listeners
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

| Header Field     | Description  | Set By                                |
|------------------|--|---------------------------------------|
| JMSDestination   | Destination to which the message is being sent   | JMS provider <code>send</code> method |
| JMSDeliveryMode  | Delivery mode specified when the message was sent (see <a href="#">Specifying Message Persistence</a> )                                      | JMS provider <code>send</code> method |
| JMSDeliveryTime  | The time the message was sent plus the delivery delay specified when the message was sent (see <a href="#">Specifying a Delivery Delay</a> ) | JMS provider <code>send</code> method |
| JMSExpiration    | Expiration time of the message (see <a href="#">Allowing Messages to Expire</a> )  | JMS provider <code>send</code> method |
| JMSPriority      | The priority of the message (see <a href="#">Setting Message Priority Levels</a> )   | JMS provider <code>send</code> method |
| JMSMessageID     | Value that uniquely identifies each message sent by a provider   | JMS provider <code>send</code> method |
| JMSTimestamp     | The time the message was handed off to a provider to be sent   | JMS provider <code>send</code> method |
| JMSCorrelationID | Value that links one message to another; commonly the <code>JMSMessageID</code> value is used  | Client application                    |
| JMSReplyTo       | Destination where replies to the message should be sent  | Client application                    |
| JMSType          | Type identifier supplied by client application   | Client application                    |
| JMSRedelivered   | Whether the message is being redelivered   | JMS provider prior to delivery        |

### JMS Messages

The ultimate purpose of a JMS application is to produce and consume messages that can then be used by other software applications. JMS messages have a basic format that is simple but highly flexible, allowing you to create messages that match formats used by non-JMS applications on heterogeneous platforms.

A JMS message can have three parts: a header, properties, and a body. Only the header is required. The following sections describe these parts.

For complete documentation of message headers, properties, and bodies, see the documentation of the `Message` interface in the API documentation.

### Message Headers

A JMS message header contains a number of predefined fields that contain values used by both clients and providers to identify and route messages. The table lists and describes the JMS message header fields and indicates how their values are set. For example, every message has a unique identifier, which is represented in the header field `JMSMessageID`. The value of another header field, `JMSDestination`, represents the queue or the topic to which the message is sent. Other fields include a timestamp and a priority level.

Each header field has associated setter and getter methods, which are documented in the description of the `Message` interface. Some header fields are intended to be set by a client, but many are set automatically by the `send` method, which overrides any client-set values.

# Message Body

39

| Message Type  | Body Contains   |
|---------------|---|
| TextMessage   | A <code>java.lang.String</code> object (for example, the contents of an XML file).  |
| MapMessage    | A set of name-value pairs, with names as <code>String</code> objects and values as primitive types in the Java programming language. The entries can be accessed sequentially by enumerator or randomly by name. The order of the entries is undefined. |
| BytesMessage  | A stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.  |
| StreamMessage | A stream of primitive values in the Java programming language, filled and read sequentially.  |
| ObjectMessage | A <code>Serializable</code> object in the Java programming language.  |
| Message       | Nothing. Composed of header fields and properties only. This message type is useful when a message body is not required.  |

## Message Bodies

The JMS API defines six different types of messages. Each message type corresponds to a different message body. These message types allow you to send and receive data in many different forms. [The table](#) describes these message types.

The JMS API provides methods for creating messages of each type and for filling in their contents. For example, to create and send a `TextMessage`, you might use the following statements:

```
TextMessage message = context.createTextMessage();
message.setText(msg_text); // msg_text is a String
context.createProducer().send(message);
```

At the consuming end, a message arrives as a generic `Message` object. You can then cast the object to the appropriate message type and use more specific methods to access the body and extract the message contents (and its headers and properties if needed). For example, you might use the stream-oriented read methods of `BytesMessage`. You must always cast to the appropriate message type to retrieve the body of a `StreamMessage`.

Instead of casting the message to a message type, you can call the `getBody` method on the `Message`, specifying the type of the message as an argument. For example, you can retrieve a `TextMessage` as a `String`. The following code fragment uses the `getBody` method:

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    String message = m.getBody(String.class);
    System.out.println("Reading message: " + message);
} else { // Handle error or process another message type }
```

The JMS API provides shortcuts for creating and receiving a `TextMessage`, `BytesMessage`, `MapMessage`, or `ObjectMessage`. For example, you do not have to wrap a string in a `TextMessage`; instead, you can send and receive the string directly. For example, you can send a string as follows:

```
String message = "This is a message";
context.createProducer().send(dest, message);
```

You can receive the message by using the `receiveBody` method:

```
String message = receiver.receiveBody(String.class);
```

You can use the `receiveBody` method to receive any type of message except `StreamMessage` and `Message`, as long as the body of the message can be assigned to a particular type.

An empty `Message` can be useful if you want to send a message that is simply a signal to the application.

For example:

```
context.createProducer().send(dest, context.createMessage());
```

The consumer code can then interpret a non-text message as a signal that all the messages sent have now been received.



## Berichten aanmaken

40

```
TextMessage message = context.createTextMessage();  
message.setText(msg_text);    // msg_text is a String  
context.createProducer().send(dest, message);
```

```
String message = "This is a message";  
context.createProducer().send(dest, message);
```

```
context.createProducer().send(dest,  
    context.createMessage());
```

Industrieel Ingenieur Informatica, UGent

## Berichten ontvangen

41

```
Message m = consumer.receive();
if (m instanceof TextMessage) {
    String message = m.getBody(String.class);
    System.out.println("Reading message: " + message);
} else {
    // Handle error or process another message type
}
```

```
String message = consumer.receiveBody(String.class);
```

# Overzicht

42

- Wat is JMS?
- JMS API
  - ▣ Configuratie
  - ▣ Producers
  - ▣ Consumers
    - Listeners
    - Subscriptions
  - ▣ Messages
  - ▣ Browsers

Industrieel Ingenieur Informatica, UGent

# JMS Message Browsers

43

```
try (JMSContext context = connectionFactory.createContext();) {
    QueueBrowser browser = context.createBrowser(queue);
    Enumeration msgs = browser.getEnumeration();

    if (!msgs.hasMoreElements()) {
        System.out.println("No messages in queue");
    } else {
        while (msgs.hasMoreElements()) {
            Message tempMsg = (Message) msgs.nextElement();
            System.out.println("\nMessage: " + tempMsg);
        }
    }
} catch (JMSRuntimeException e) {    ...    }
```

## JMS Queue Browsers

Messages sent to a queue remain in the queue until the message consumer for that queue consumes them. The JMS API provides a QueueBrowser object that allows you to browse the messages in the queue and display the header values for each message. To create a QueueBrowser object, use the JMSContext.createBrowser method. For example:

```
QueueBrowser browser = context.createBrowser(queue);
```

The createBrowser method allows you to specify a message selector as a second argument when you create a QueueBrowser.

The JMS API provides no mechanism for browsing a topic. Messages usually disappear from a topic as soon as they appear: If there are no message consumers to consume them, the JMS provider removes them. Although durable subscriptions allow messages to remain on a topic while the message consumer is not active, JMS does not define any facility for examining them.

```
C:\Users\vongenae\Documents\Gedistribueerde toepassingen\voorbeelden\jms\Message
Browser>appclient -client dist\MessageBrowser.jar
okt 26, 2015 2:37:23 PM org.hibernate.validator.internal.util.Version <clinit>
INFO: HV000001: Hibernate Validator 5.0.0.Final
okt 26, 2015 2:37:23 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter: Version: 5.1 (Build 9
-b) Compile: July 29 2014 1229
okt 26, 2015 2:37:23 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter starting: broker is REMO
TE, connection mode is TCP
okt 26, 2015 2:37:23 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter Started:REMOTE
No messages in queue
```

Industrieel Ingenieur Informatica, UGent

This section describes an example that creates a QueueBrowser object to examine messages on a queue.

```
C:\Users\vongenae\Documents\Gedistribueerde toepassingen\voorbeelden\jms\Message
Browser>appclient -client dist\MessageBrowser.jar
okt 26, 2015 2:38:28 PM org.hibernate.validator.internal.util.Version <clinit>
INFO: HV000001: Hibernate Validator 5.0.0.Final
okt 26, 2015 2:38:28 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter: Version: 5.1 (Build 9
-b) Compile: July 29 2014 1229
okt 26, 2015 2:38:28 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter starting: broker is REMO
TE, connection mode is TCP
okt 26, 2015 2:38:28 PM com.sun.messaging.jms.ra.ResourceAdapter start
INFO: MQJMSRA_RA1101: GlassFish MQ JMS Resource Adapter Started:REMOTE

Message:
Text: This is message 1 from producer
Class: com.sun.messaging.jmq.jmsclient.TextMessageImpl
getJMSMessageID(): ID:9-192.168.56.1(b2:3:ea:b1:f2:e5)-7388-1445866678820
getJMSTimestamp(): 1445866678820
getJMSCorrelationID(): null
JMSReplyTo: null
JMSDestination: PhysicalQueue
getJMSDeliveryMode(): PERSISTENT
getJMSRedelivered(): false
getJMSType(): null
getJMSExpiration(): 0
```