Berner
Fachhochschule

Java Message Service

Stephan Fischli

Winter 2022

Goals

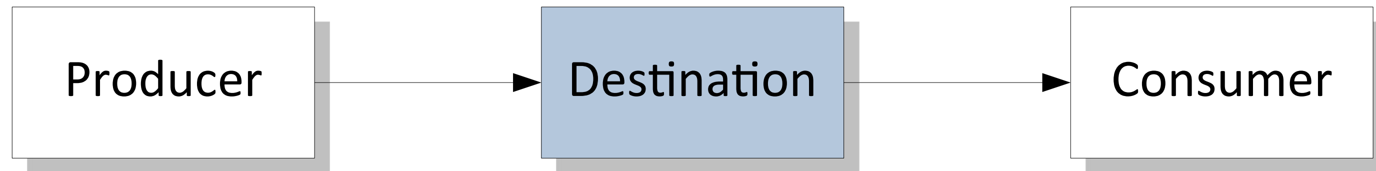
- Understand the messaging paradigm
- Know the different JMS messaging models
- Develop a Spring application using JMS

Contents

- Introduction
- Messages
- Programming Model
- Spring JMS

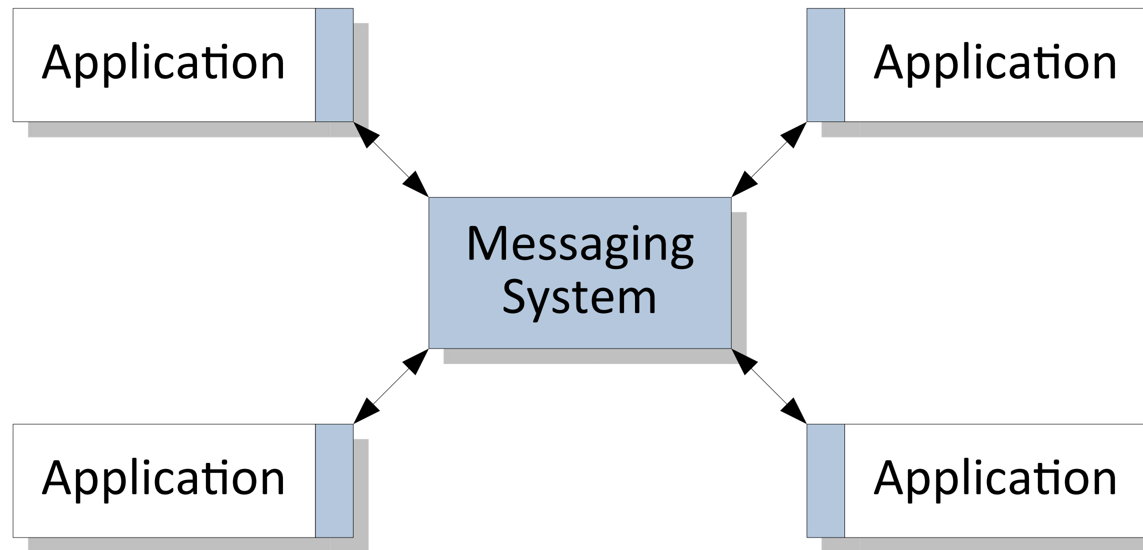
Introduction

Messaging versus RPC



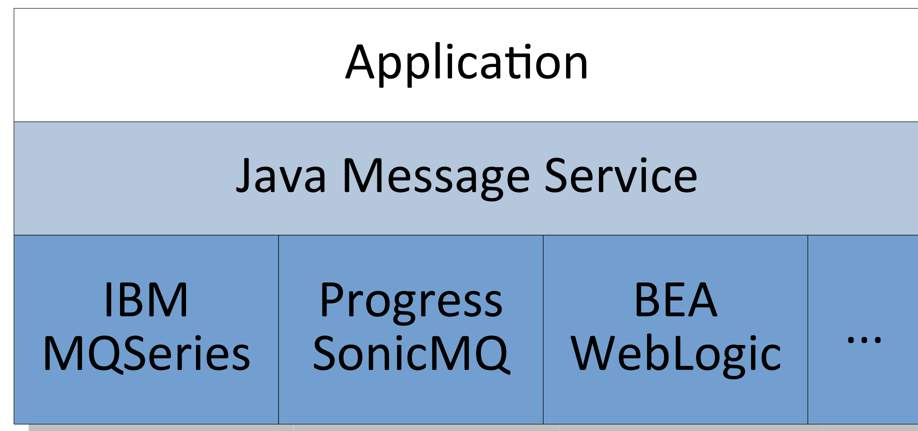
- When a remote method is invoked, the caller is blocked until the method completes
- The synchronized nature of RPC tightly couples the client to the server and creates highly interdependent systems
- Messaging applications exchange messages through virtual channels called destinations so that senders and receivers are not bound to each other
- Messages are delivered asynchronously, i.e. the sender is not required to wait for the message to be received by the recipient

Messaging Systems



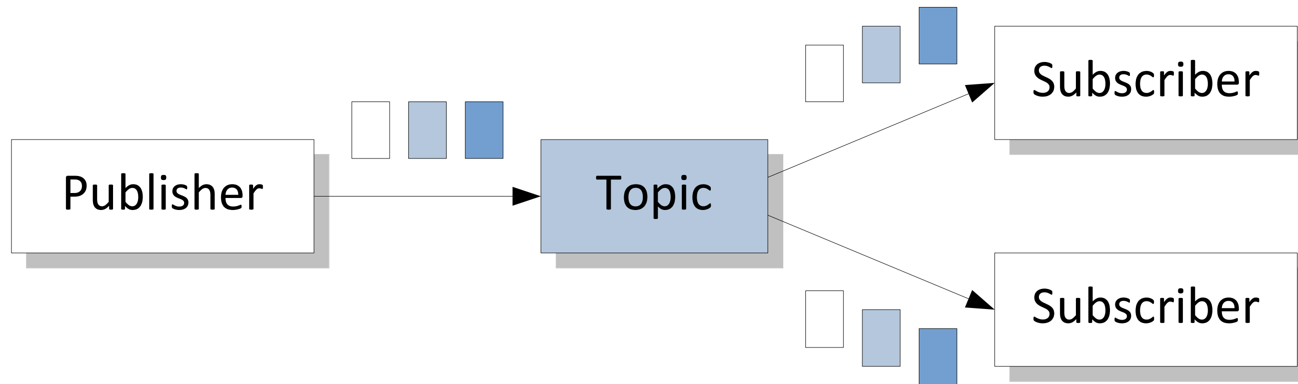
- Messaging systems allow two or more applications to exchange information in the form of messages
- A message is a self-contained package of business data and network routing headers
- Messaging systems provide fault tolerance, load balancing, scalability, and transactional support

Java Message Service



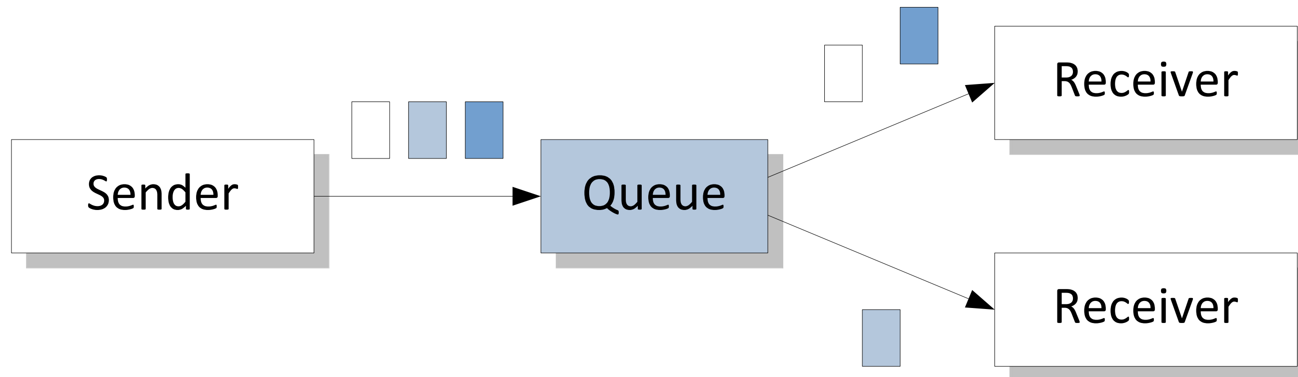
- Messaging systems use different message formats and network protocols (TCP/IP, HTTP, SSL, IP multicast), but the basic semantics are the same
- The Java Message Service (JMS) is a standardized API for sending and receiving messages that can be used with many different messaging systems

Publish-Subscribe Domain



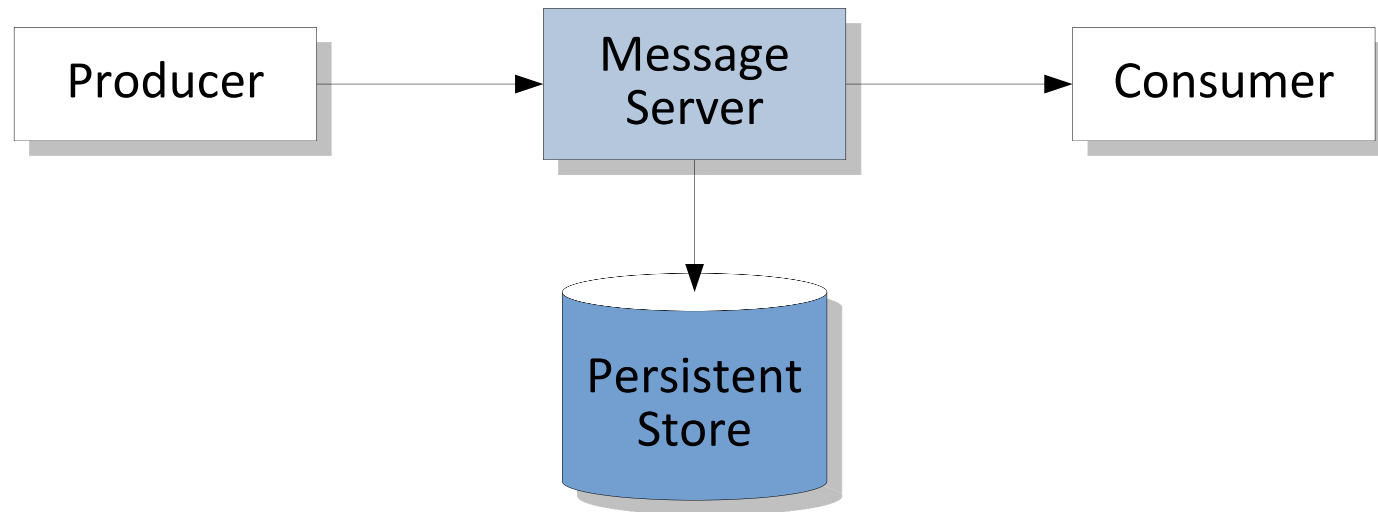
- In the publish-and-subscribe domain, a producer can send a message to many consumers through a destination called topic
- Consumers can subscribe to a topic and receive a copy of each message
- Messages are usually broadcast to consumers (push-based model)
- A topic retains messages only as long as it takes to deliver them to the current subscribers
- There is a timing dependency between publishers and subscribers

Point-to-Point Domain



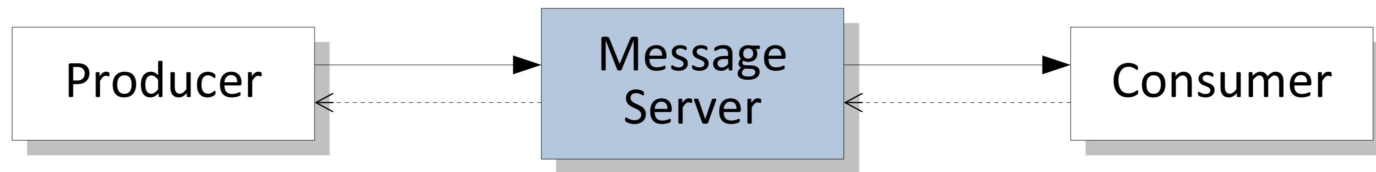
- In the point-to-point domain, a producer can send a message to one consumer through a destination called queue
- A given queue may have multiple receivers, but only one receiver may consume each message
- Messages are usually requested from the queue (pull-based model)
- A queue delivers messages in the order they were placed into it
- A queue retains messages until they are consumed or expire
- There is no timing dependency between senders and receivers

Guaranteed Delivery



- Messaging systems provide guaranteed delivery which ensures that the intended consumers will eventually receive a message even if a partial failure occurs
- Guaranteed delivery uses store-and-forward mechanism, i.e. the message server writes the incoming messages to a persistent store and then forwards them to the intended consumers
- If the message server crashes, it will deliver the persistent messages to consumers as soon as it starts up again

Message Acknowledgment



- Message acknowledgment is part of the protocol between the client runtime library of the JMS provider and the message server
- The server acknowledges the receipt of messages from producers, consumers acknowledge the receipt of messages from the server
- The acknowledgment protocol allows the JMS provider to manage the distribution of messages and guarantee their delivery
- If a consumer fails to acknowledge a message, the server considers the message undelivered and will attempt to redeliver it

Transactional Messaging

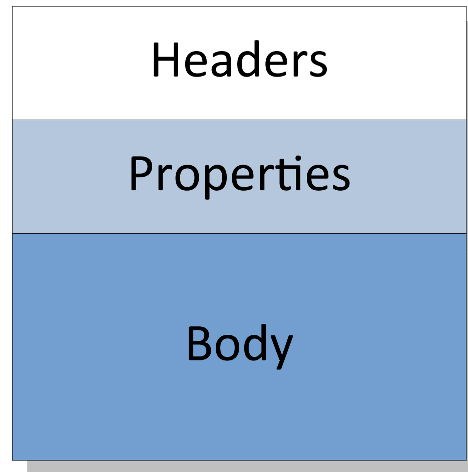
- A JMS client can group multiple send or receive operations into an atomic unit of work
- With transactional sends, messages delivered to the server are not forwarded to the consumers until the producer commits the transaction
- With transactional receives, messages delivered to the consumer are not deleted by the server until the consumer commits the transaction
- JMS supports distributed transactions across different transactional resources using the two-phase commit protocol (2PC)

References

- Mark Richards and Richard Monson-Haefel
  Java Message Service (2nd Edition)
  Creating Distributed Enterprise Applications
  O'Reilly & Associates, 2009
- Scott Grant (Editor)
  Professional JMS Programming
  Wrox Press Inc, 2001
- Mark Hapner et.al.
  Java Message Service API Tutorial and Reference
  Addison Wesley Professional, 2002

Messages

Anatomy of a JMS Message



- A JMS message carries application data and provides event notification
- A JMS message has three parts:
  - the message headers provide metadata and routing information
  - the message properties are defined by the JMS client
  - the message body carries the payload of the message
- When a message is delivered, the properties and the body of the message are made read-only

Headers

- Every JMS message has a set of standard headers
- For each header there is a corresponding set and get method
- Most JMS headers are automatically assigned, i.e. their values are set by the JMS provider depending on declarations made by the developer
- Other headers must be set explicitly on the message before it is delivered by the producer

Automatically Assigned Headers

JMSMessageID

- The JMSMessageID is a string value that uniquely identifies a message

JMSTimestamp

- The JMSTimestamp header is set automatically by the message producer when the message is sent

JMSDestination

- The JMSDestination header identifies the destination of a message with either a topic or a queue

JMSDeliveryMode

- A persistent message should be delivered once-and-only-once even if the JMS provider fails
- A non-persistent message is delivered at-most-once, which means that it can be lost if the JMS provider fails
- The delivery mode can be set on the message producer using the setJMSDeliveryMode() method (default is persistent)

Automatically Assigned Headers (cont.)

JMSRedelivered

- The JMSRedelivered header indicates that a message was redelivered to the consumer
- A message may be marked redelivered if a consumer failed to acknowledge previous delivery of the message

Developer-Assigned Headers

JMSType

- The JMSType header can be set by the message producer to identify the message structure and type of payload

JMSExpiration

- A message's expiration date prevents the message from being delivered to consumers after it has expired
- The expiration time can be set on the message producer using the setTimeToLive() method (by default a message doesn't expire)

JMSPriority

- The message server may use a message's priority to prioritize delivery to consumers
- Levels 0 to 4 are gradations of normal priority, levels 5 to 9 are gradations of expedited priority
- The priority can be set on the message producer using the setPriority() method (default is 4)

Developer-Assigned Headers (cont.)

JMSReplyTo

- The JMSReplyTo header contains a destination to which the consumer of the message should reply to

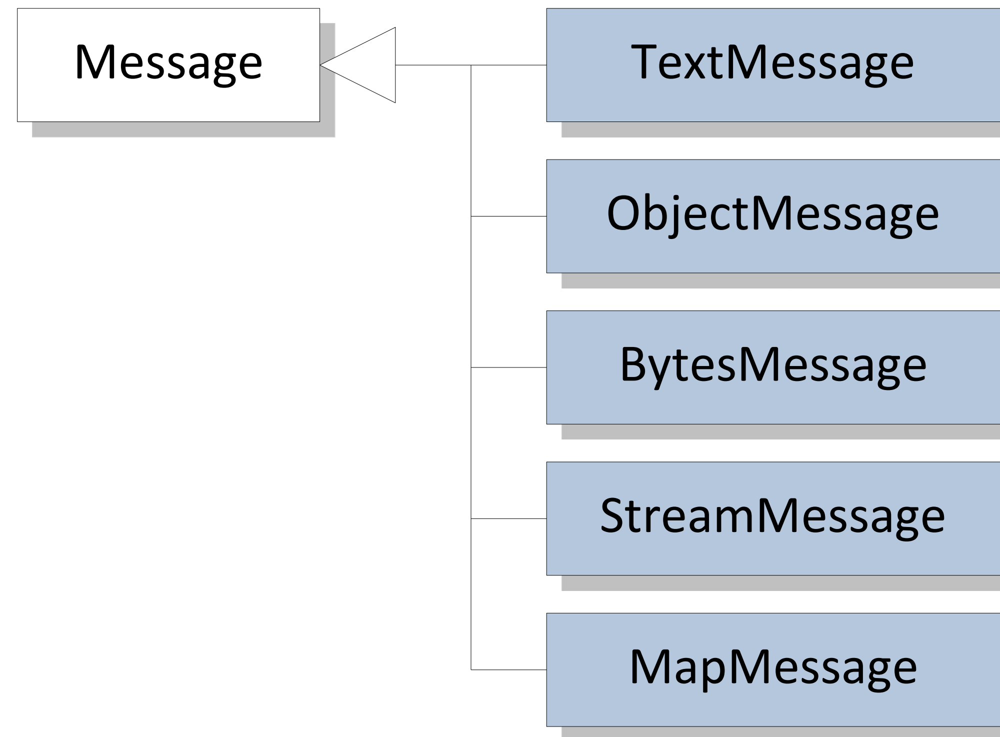
JMSCorrelationID

- The JMSCorrelationID header is used for associating the current message with some previous message, e.g. to tag a message as a reply to a previous message

Properties

- Properties are like additional headers that can be assigned to a message
- The value of a property can be a String, a primitive value or a wrapper object thereof
- There are three categories of properties:
    - Application-specific properties are defined and applied to a message by the application developer
    - JMS-defined properties act as optional JMS headers and are set by the JMS provider when the message is sent
    - Provider-specific properties are proprietary properties that are defined by the JMS provider

Message Types



- The message types represent the kind of payload a message can have
- Some types were included to support legacy payloads, other types were defined to facilitate emerging needs

Message Types (cont.)

Message

- The type Message serves as the base interface of the other message types
- It contains only JMS headers and properties and is used for event notification

TextMessage

- The type TextMessage carries a String as its payload
- It is useful for exchanging simple text messages and more complex character data like XML documents

ObjectMessage

- The type ObjectMessage carries a serializable Java object as its payload
- The producer and consumer must be Java programs, and the class definition of the object has to be available to both of them

Message Types (cont.)

BytesMessage

- The ByteMessage type carries an array of bytes as its payload
- It is useful for exchanging data in an application's native format or when the message payload is opaque to the JMS client

StreamMessage

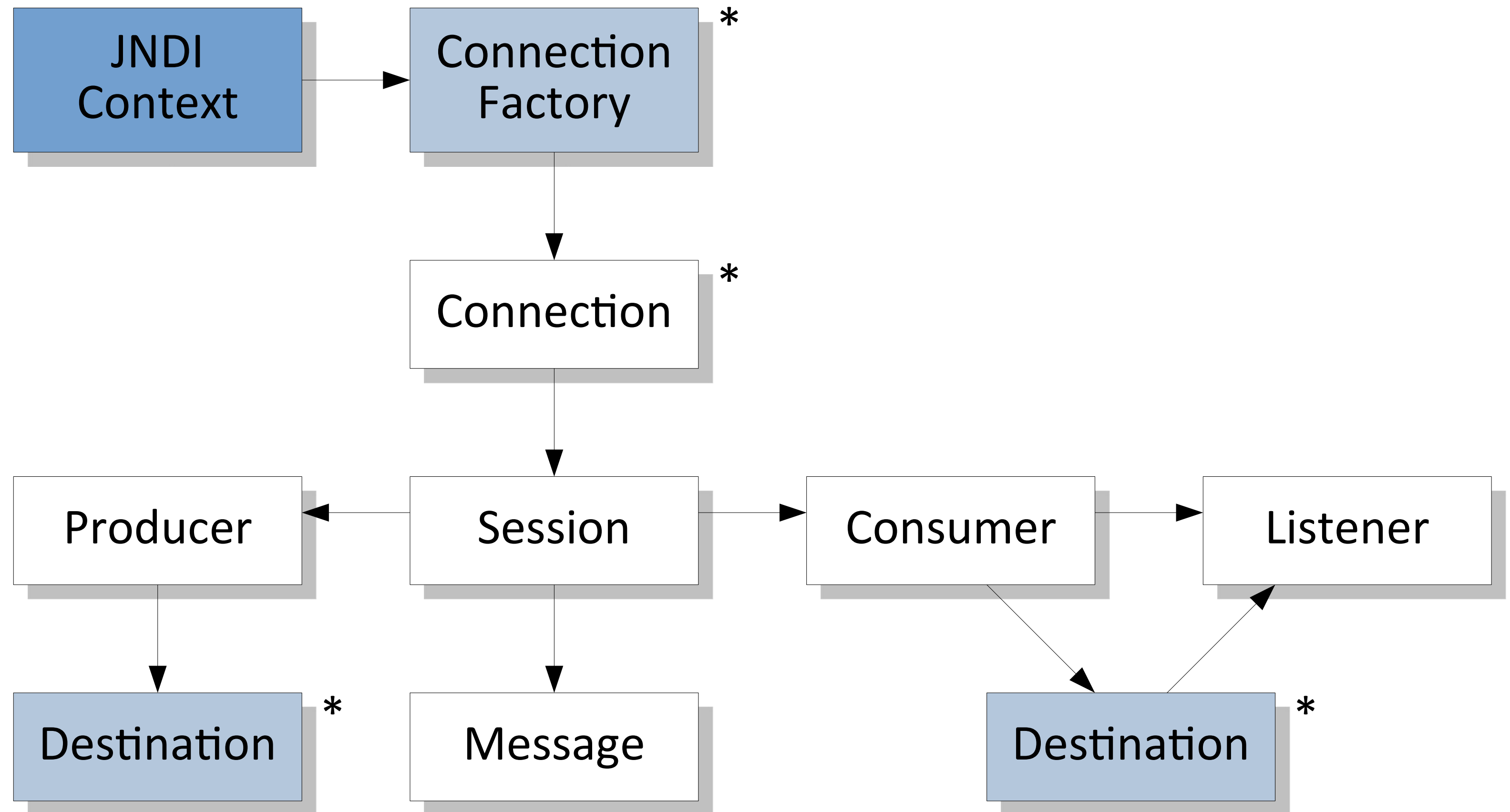
- The StreamMessage type carries a stream of primitive Java types as its payload
- It keeps track of the order and types of primitives written to the message

MapMessage

- The MapMessage type carries a set of name-value pairs as its payload
- It is useful for delivering keyed data that may change from one message to the next

Programming Model

Overview



* These objects are thread-safe

Administered Objects

- Connection factories and destinations are established and configured by the system administrator
- A JMS client can obtain access to connection factories and destinations by looking them up using JNDI
- JNDI is a standard Java extension that provides a uniform API for directory and naming services
- Creating a connection to a JNDI naming service requires an initial context be created with appropriate properties

Connection Factories

- A connection factory is the object a client uses to create a connection with a JMS provider
- A connection factory encapsulates a set of connection configuration parameters (server address, port, protocol etc.)
- A JMS client usually performs a JNDI lookup of the connection factory

Destinations

- A destination is the object a client uses as the target of messages it produces and the source of messages it consumes
- A JMS application may use multiple queues and/or topics
- A JMS client usually performs a JNDI lookup of the destination

Connections

- A connection encapsulates a virtual connection with a JMS provider (e.g. a TCP/IP socket) and is used to create one or more sessions
- When an application completes any connections need to be closed, otherwise resources may not to be released by the JMS provider
- Before an application can consume messages, the connection's start() method must be called
- To stop message delivery temporarily without closing the connection, the stop() method can be called

Sessions

- A session is used to create message producers, message consumers and messages
- A session may not be operated on by more than one thread at a time (single-threaded context)
- A session provides a transactional context with which to group sends and receives into an atomic unit of work
- A session defines the acknowledgment behavior of messages

Message Producers

- A message producer is an object which is used for sending messages to a destination
- With an unidentified producer, the destination of a message can be specified when the message is sent

Message Consumers

- A message consumer is an object which is used for receiving messages from a destination
- A message consumer allows a JMS client to register interest in a destination, and the JMS provider manages the delivery of messages to the registered consumers
- The receive() method is used to consume a message synchronously
- To consume messages asynchronously, a message listener is needed

Message Listeners

- A message listener is an object that acts as an asynchronous event handler for messages
- A message listener implements the onMessage() method which defines the actions to be taken when a message arrives
- The message listener is registered with a specific message consumer
- A message listener is not specific to a particular destination type, however it usually expects a specific message type and format
- The session used to create the message consumer serializes the execution of all message listeners

Message Selectors

- A message selector allows a JMS consumer to be selective about the messages it receives from a destination
- Message selectors use message properties and headers as criteria in conditional expressions (based on a subset of the SQL syntax for WHERE clauses)
- Message selectors are declared when the message consumer is created
- Messages that are not selected by a consumer are not delivered to that consumer but to other consumers

Queue Browsers

- A queue browser is a specialized object that allows to peek ahead at pending messages on a queue without consuming them
- Queue browsing can be useful for monitoring the contents of a queue from an administrative tool
- Messages obtained from a queue browser only provide a snapshot of the queue's content

Temporary Destinations

- A temporary destination is a destination that is dynamically created by a JMS client and only lives as long as the client lives
- A temporary destination is unavailable to other clients unless its identity is transferred in a JMSReplyTo message header
- While any client may send messages to a temporary destination, only the client that created the destination may receive messages from it
- The JMSReplyTo message header and temporary destinations can be used to create a synchronous request-reply conversation

Spring JMS

Spring JMS Support

- Spring Boot automatically creates a ConnectionFactory and the bean JmsTemplate which simplifies synchronous JMS messaging
- To receive messages asynchronously, any method of a managed bean can be annotated with @JmsListener
- To trigger the discovery of such methods, a configuration class must be annotated with @EnableJms

```
@SpringBootApplication
@EnableJms
public class Application {
   @Autowired
   private JmsTemplate jmsTemplate;
   @JmsListener(...)
   public void onMessage(Message message) { ... }
   ...
}
```

Spring JMS Support (cont.)

- By default, Spring Boot uses queues, but to use topics the property spring.jms.pub-sub-domain can be set
- When using ActiveMQ Artemis as a message broker, its URL can be configured via the property spring.artemis.broker-url

```
spring.jms.pub-sub-domain=true
spring.artemis.broker-url=tcp://localhost:61616
```

Sending Messages

- A message can be sent to a destination using the send() method of the Spring JMS template and by providing a message creator

```
jmsTemplate.send(destination, session -> {
    Message message = session.createTextMessage(text);
    ...
    return message;
});
```

- If the message body can be converted by a configured message converter, the message can be sent using the convertAndSend() method

```
jmsTemplate.convertAndSend(destination, text);
```

Receiving Messages

- A message can be received from a destination using the receive() method of the Spring JMS template
- The method blocks until the message becomes available or the configured timeout is exceeded

```
Message message = jmsTemplate.receive(destination);
String text = ((TextMessage) message).getText();
...
```

- If the message body can be converted by a configured message converter, a message can be received using the receiveAndConvert() method

```
String text = (String) jmsTemplate.receiveAndConvert(destination);
```

Asynchronously Receiving Messages

- A message can asynchronously be received from a destination by annotating a callback method with @JmsListener

```
@JmsListener(destination = "...")
public void onMessage(Message message) {
    String text = ((TextMessage) message).getText();
    ...
}
```

- If the message body can be converted by a configured message converter, the type of the body can be used as parameter type

```
@JmsListener(destination = "...")
public void onMessage(String text) {
    ...
}
```

- The annotation's concurrency element can be used to define the number of concurrent consumers (default is 1)

Selectively Receiving Messages

- Messages can selectively be received from a destination by providing a selector to the receive() method or by using the selector attribute of the @JmsListener annotation

```
Message message = jmsTemplate.receive(destination, selector);
String text = ((TextMessage) message).getText();
...

@JmsListener(destination = "...", selector = "...")
public void onMessage(Message message) {
    String text = ((TextMessage) message).getText();
    ...
}
```

Message Converter

- By default, the SimpleMessageConverter is able to handle messages of type TextMessage, BytesMessage, MapMessage and ObjectMessage
- A custom message converter can be provided by implementing the MessageConverter interface

```
@Component
public class CustomMessageConverter implements MessageConverter {
    public Object fromMessage(Message message)
        throws JMSException, MessageConversionException {
        ...
    }
    public Message toMessage(Object object, Session session)
        throws JMSException, MessageConversionException {
        ...
    }
}
```

JSON Message Converter

- Spring JMS provides the MappingJackson2MessageConverter that can convert messages to and from JSON
- The converter must be configured using a type ID property and ID to class mappings to allow conversion of different message types

```
@Configuration
public class JmsConfig {
    @Bean
    public MessageConverter jacksonJmsMessageConverter() {
        MappingJackson2MessageConverter converter =
            new MappingJackson2MessageConverter();
        converter.setTargetType(MessageType.TEXT);
        converter.setTypeIdPropertyName("typeId");
        converter.setTypeIdMappings(...);
        return converter;
    }
}
```

Sending Messages and Receiving Replies

- The method sendAndReceive() can be used to send a request message to a destination and to receive a reply message
- A temporary queue is created and set in the JMSReplyTo header of the request message

```
Message reply = jmsTemplate.sendAndReceive(destination,
    session -> session.createTextMessage(text)
);
```

Sending Reply Messages

- Replies to request messages are sent to the temporary queue provided in the request's JMSReplyTo header
- Optionally, the JMSCorrelationID header of the reply message can be set to the request's message identifier

```
jmsTemplate.send(request.getJMSReplyTo(),
    session -> {
        TextMessage reply = session.createTextMessage(text);
        reply.setJMSCorrelationID(request.getJMSMessageID());
        return reply;
    });
```