JavaOne™

ORACLE®

What's ^probably^ coming in JMS 2.1

Nigel Deakin
JMS Specification Lead

26 October 2015

nigel.deakin@oracle.com
@jms_spec

Agenda

1 JMS 2.0 recap

2 What's (probably) coming in JMS 2.1?

3 Improving JMS MDBs

4 CDI beans as message listeners

5 Other new features

JMS 2.0 Recap

Now available
in these Java EE 7 application servers

IBM WebSphere
Application Server
Version 8.5.5.6
(Liberty Profile)

ORACLE
FUSION MIDDLEWARE
WEBLOGIC SERVER

Oracle WebLogic Server 12.2.1

WildFly

WildFly 8.x

Glassfish Server
Open Source Edition 4.0

Cosminexus

Hitachi Application Server v10.0

TmaxSoft

TMAX JEUS 8

Java EE 7 Full Platform Compatible Implementations
http://www.oracle.com/technetwork/java/javaee/overview/compatibility-jsp-136984.html

JavaOne ORACLE

Sending a message in a Java EE 6 (JMS 1.1) application

```
@Resource(lookup = "jms/orderQueue")
private Queue orderQueue;

@Resource(lookup = "jms/myConnectionFactory")
private ConnectionFactory connectionFactory;

public void sendMessageEE6(String body) {
    Connection connection = null;
    try {
        connection = connectionFactory.createConnection();
        Session session = connection.createSession(true, Session.SESSION_TRANSACTED);
        MessageProducer messageProducer = session.createProducer(orderQueue);
        TextMessage textMessage = session.createTextMessage(body);
        messageProducer.send(textMessage);
    } catch (JMSException e) { // Handle exceptions
    } finally {
        try {
            connection.close();
        } catch (JMSException e) { // Handle exception in close()
        }
    }
}
```

Sending a message using the JMS 2.0 simplified API

```
@Resource(lookup = "jms/orderQueue")
private Queue orderQueue;

@Resource(lookup = "jms/myConnectionFactory")
private ConnectionFactory connectionFactory;

public void sendMessageEE7(String body) {
    try (JMSContext context = connectionFactory.createContext()){
        context.createProducer().send(orderQueue, body);
    } catch (JMSRuntimeException e) {
        // Handle exceptions
    }
}
```

Sending a message using the JMS 2.0 simplified API and JMSContext injection

```
@Resource(lookup = "jms/orderQueue")
private Queue orderQueue;

@Inject @JMSConnectionFactory("jms/myConnectionFactory")
JMSContext context;

public void sendMessageEE7WithInjection(String body) {
    try {
        context.createProducer().send(orderQueue, body);
    } catch (JMSRuntimeException e) {
        // Handle exceptions
    }
}
```

Unfinished business in JMS 2.0

- The JMS 2.0 simplified API simplifies the code you need to write to
  - Send a message
  - Receive a message synchronously (using `receive(timeout)`)
- No changes to the code you need to write to
  - Receive a message asynchronously in a Java EE application
  - You still need to create a MDB

Other new features in JMS 2.0

- Asynchronous send
- Multiple consumers on a topic subscription
- Delivery delay
- Delivery count
- Resource definitions (Java EE)
- Platform default JMS connection factory (Java EE)

What's (probably) coming in JMS 2.1?

JMS 2.1 (JSR 368) Status

Stage	Initial plan (Sep 2014)	Current plan (Updated Jun 2015)	Actual
JSR approval	Sep 2014		Sep 2014
Expert group formation	Q3 (Sep) 2014		Dec 2014
Early draft 1			Oct 2015
Early draft 2	Q1 (Mar) 2015	Q4 (Dec) 2015	
Public review	Q3 (Sep) 2015	Q1 (Mar) 2016	
Proposed final draft	Q4 (Dec) 2015	Q3 (Sep) 2016	
Final release	Q3 (Sep) 2016	H1 (Jun) 2017	

New!

Improving JMS MDBs for JMS 2.1

Competing unfinished business from JMS 2.0

What's wrong with JMS MDBs?

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationLookup",
        propertyValue = "java:global/requestQueue"),
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue") })
public class MyMDB implements MessageListener {

    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
            // ...
        }
    }
}
```

What's wrong with JMS MDBs?

Verbose, generic annotations

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationLookup",
        propertyValue = "java:global/requestQueue"),
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue") })
public class MyMDB implements MessageListener {

    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
            // ...
        }
    }
}
```

What's wrong with JMS MDBs?

Verbose, generic annotations

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationLookup",
        propertyValue = "java:global/requestQueue"),
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue") })
public class MyMDB implements MessageListener {

    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
            // ...
        }
    }
}
```

Use of key-value pairs means no compile-time checking of property names, and no type checking

What's wrong with JMS MDBs?

Verbose, generic annotations

Use of key-value pairs means no compile-time checking of property names, and no type checking

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationLookup",
        propertyValue = "java:global/requestQueue"),
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue") })
public class MyMDB implements MessageListener {

    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
            // ...
        }
    }
}
```

Must implement `javax.jms.MessageListener`

What's wrong with JMS MDBs?

```
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(
        propertyName = "destinationLookup",
        propertyValue = "java:global/requestQueue"),
    @ActivationConfigProperty(
        propertyName = "destinationType",
        propertyValue = "javax.jms.Queue") })
public class MyMDB implements MessageListener {

    public void onMessage(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
            // ...
        }
    }
}
```

Verbose, generic annotations

Use of key-value pairs means no compile-time checking of property names, and no type checking

Must implement `javax.jms.MessageListener`

Fixed MDB lifecycle

What's good about JMS MDBs?

- Declarative
  - No need to explitly create them
- Scalable
  - MDB can be a pool of bean instances, processing messages concurrently

Improving JMS MDBs for JMS 2.1

- Flexible JMS MDBs
- Allowing CDI managed beans (i.e. not just MDBs) to listen for JMS messages

Introducing "Flexible JMS MDBs"

- Configured using simpler, JMS-specific annotations
- Doesn't implement `javax.jms.MessageListener`
- User-defined callback methods
- More than one callback method (perhaps)
- Flexible method signatures
  - direct access to concrete message type, message body, messages headers, message properties
- These are still MDBs
  - MDB lifecycle, can be pooled

Flexible JMS MDBs - Queues

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(destinationLookup="java:global/requestQueue")
    public void myMessageCallback(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
        // ...
        }
    }
}
```

Flexible JMS MDBs - Queues

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(
        connectionFactoryLookup="java:global/connectionFactory",
        destinationLookup="java:global/requestQueue",
        messageSelector="JMSType = 'car' AND colour = 'pink'",
        acknowledge=Mode.AUTO_ACKNOWLEDGE)
    public void myMessageCallback(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
        // ...
        }
    }
}
```

Flexible JMS MDBs - Topics (non-durable subscriptions)

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSNonDurableTopicListener(destinationLookup="java:global/pricefeed")
    public void myMessageCallback(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
        // ...
        }
    }
}
```

Flexible JMS MDBs - Topics (non-durable subscriptions)

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSNonDurableTopicListener(destinationLookup=" java:global/pricefeed ")
        destinationLookup="java:global/priceFeed",
        messageSelector="JMSType = 'StockPrice' AND ticker = 'ORCL'",
        acknowledge=Mode.AUTO_ACKNOWLEDGE)
    public void myMessageCallback(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
        // ...
        }
    }
}
```

Flexible JMS MDBs - Topics (durable subscriptions)

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSDurableTopicListener(
        destinationLookup="java:global/priceFeed",
        clientId="myClientId",
        subscriptionName="mySubscriptionName")
    public void myMessageCallback(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
        // ...
        }
    }
}
```

Message subtype as a callback parameter

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(destinationLookup="java:global/requestQueue")
    public void myMessageCallback(TextMessage textMessage) {
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
        // ...
        }
    }
}
```

Message body as a callback parameter

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(destinationLookup="java:global/requestQueue")
    public void myMessageCallback(String messageText) {
        // process message text
    }
}
```

Message body extracted using existing JMS method on `Message`

```
<T> T getBody(Class<T> c) throws JMSException
```

Message headers and properties as callback parameters

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(destinationLookup="java:global/requestQueue")
    public void myMessageCallback(
        String messageText,
        @MessageHeader(Header.JMSCorrelationID) String correlationID,
        @MessageProperty("price") long price) {
            // process message text
    }
}
```

Multiple callback methods

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(destinationLookup="java:global/queue1")
    public void myMessageCallback1(String messageText) {
        // process message from queue1
    }

    @JMSQueueListener(destinationLookup="java:global/queue2")
    public void myMessageCallback2(String messageText) {
        // process message from queue2
    }
}
```

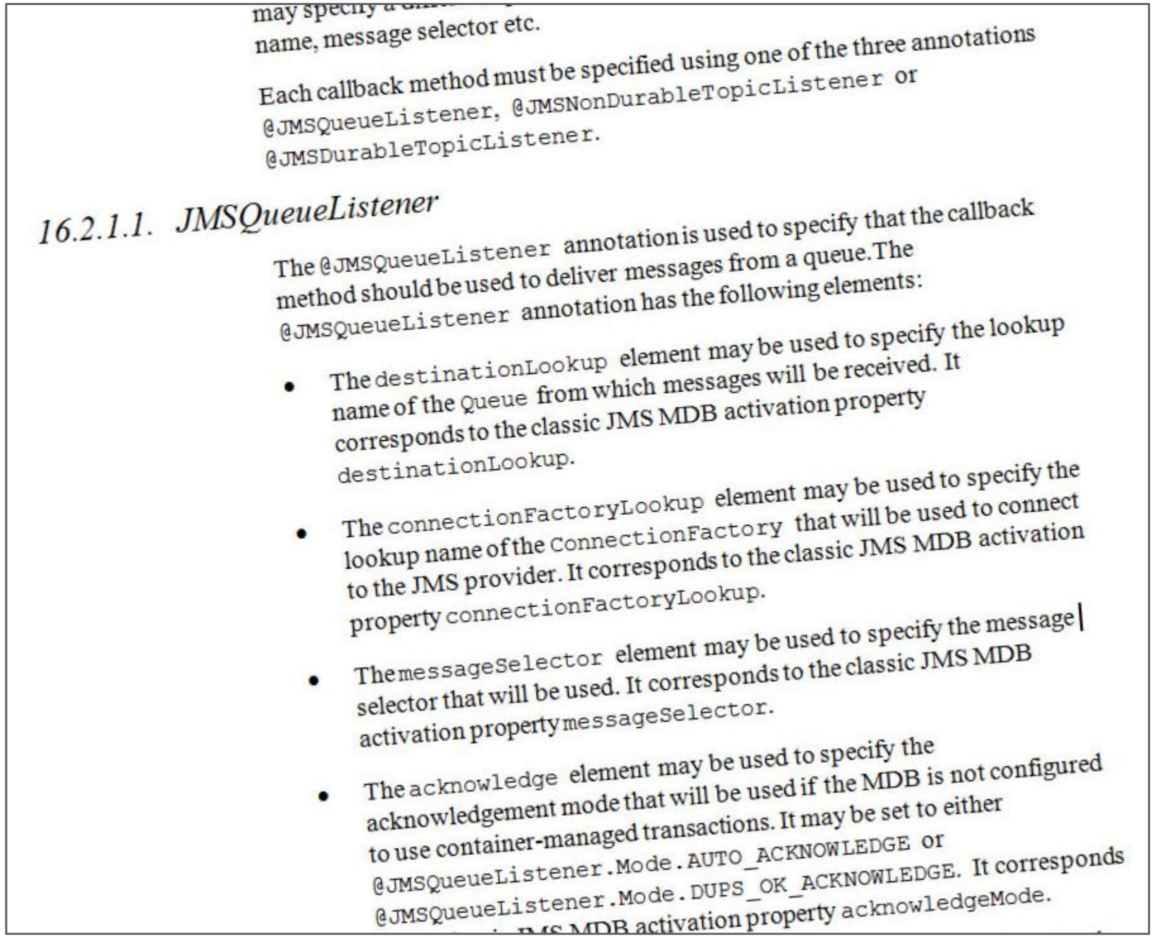
Specifying proprietary properties

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(destinationLookup="java:global/queue1")
    @JMSListenerProperty(name="reconnectAttempts", value="10")
    @JMSListenerProperty(name="reconnectInterval", value="30000")
    public void myMessageCallback1(String messageText) {
        // process message from queue1
    }

}
```

Some issues still to resolve

- Should we allow multiple callback methods on the same MDB?
- Should user-defined callback methods be allowed to throw checked exceptions?
- How should parameter conversion errors be handled?
  - if the incoming message has the wrong type for the specified parameter
  - or if a specified header or property has the wrong type for the specified parameter

JMS 2.1 Early Draft 1 now released



may specify a
name, message selector etc.

Each callback method must be specified using one of the three annotations `@JMSQueueListener`, `@JMSNonDurableTopicListener` or `@JMSDurableTopicListener`.

16.2.1.1. *JMSQueueListener*

The `@JMSQueueListener` annotation is used to specify that the callback method should be used to deliver messages from a queue. The `@JMSQueueListener` annotation has the following elements:

- The `destinationLookup` element may be used to specify the lookup name of the `Queue` from which messages will be received. It corresponds to the classic JMS MDB activation property `destinationLookup`.
- The `connectionFactoryLookup` element may be used to specify the lookup name of the `ConnectionFactory` that will be used to connect to the JMS provider. It corresponds to the classic JMS MDB activation property `connectionFactoryLookup`.
- The `messageSelector` element may be used to specify the message selector that will be used. It corresponds to the classic JMS MDB activation property `messageSelector`.
- The `acknowledge` element may be used to specify the acknowledgement mode that will be used if the MDB is not configured to use container-managed transactions. It may be set to either `@JMSQueueListener.Mode.AUTO_ACKNOWLEDGE` or `@JMSQueueListener.Mode.DUPS_OK_ACKNOWLEDGE`. It corresponds
JMS MDB activation property `acknowledgeMode`.

- Contains detailed proposals for flexible JMS MDBs
- Released specifically to encourage comments, especially on open issues
- Available now at
  - https://jcp.org/en/jsr/detail?id=368
  - http://jms-spec.java.net
- Please provide feedback!

CDI managed beans as JMS listeners

JMS listener beans – the basic idea

- Any CDI managed beans can listen for JMS messages
- Callback method(s) are defined in the same way as for "flexible JMS MDBs"
- When is the JMS consumer created?
- How many JMS consumers?
    - 1 per listener bean instance, or
    - 1 per listener bean class
- When is the listener bean created?
- Depends on the scope of the bean
- Perhaps copy how CDI events work

Firing (sending) an event in CDI

```
@Inject @SomeQualifier Event<MyObject> eventFirer;

void fireMyEvent(){

    MyObject myObj = ...
    eventFirer.fire(myObj);

}
```

Observing (listening for) an event in CDI

```
public class MyEventObserver {

  public void myObserverMethod(@Observes @SomeQualifier MyObject myObj){
      ...
  }
}
```

How a dependent-scoped observer bean works in CDI

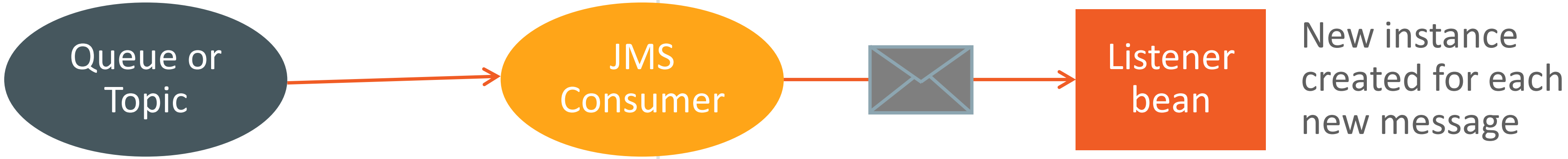
```
// @Dependent
public class MyEventObserver {

  public void myObserverMethod(
    @Observes @SomeQualifier
    MyObject myObj){
      ...
  }
}
```

- For every event that is fired
  - A new instance of the observer bean is created
  - The observer method is invoked
  - The observer bean is destroyed
- Any injected observer beans are ignored

How a dependent-scoped JMS listener bean might work

```
// @Dependent
public class MyDepScopedBean {

  @JMSQueueListener(
    destinationLookup="myQueue")
  public void callMe(Message message) {
    ...
  }


}
```

- For every message that arrives
  - A new instance of the listener bean is created
  - The callback method is invoked
  - The listener bean is destroyed
- Any injected listener beans are ignored



New instance created for each new message

Consumer created at application startup

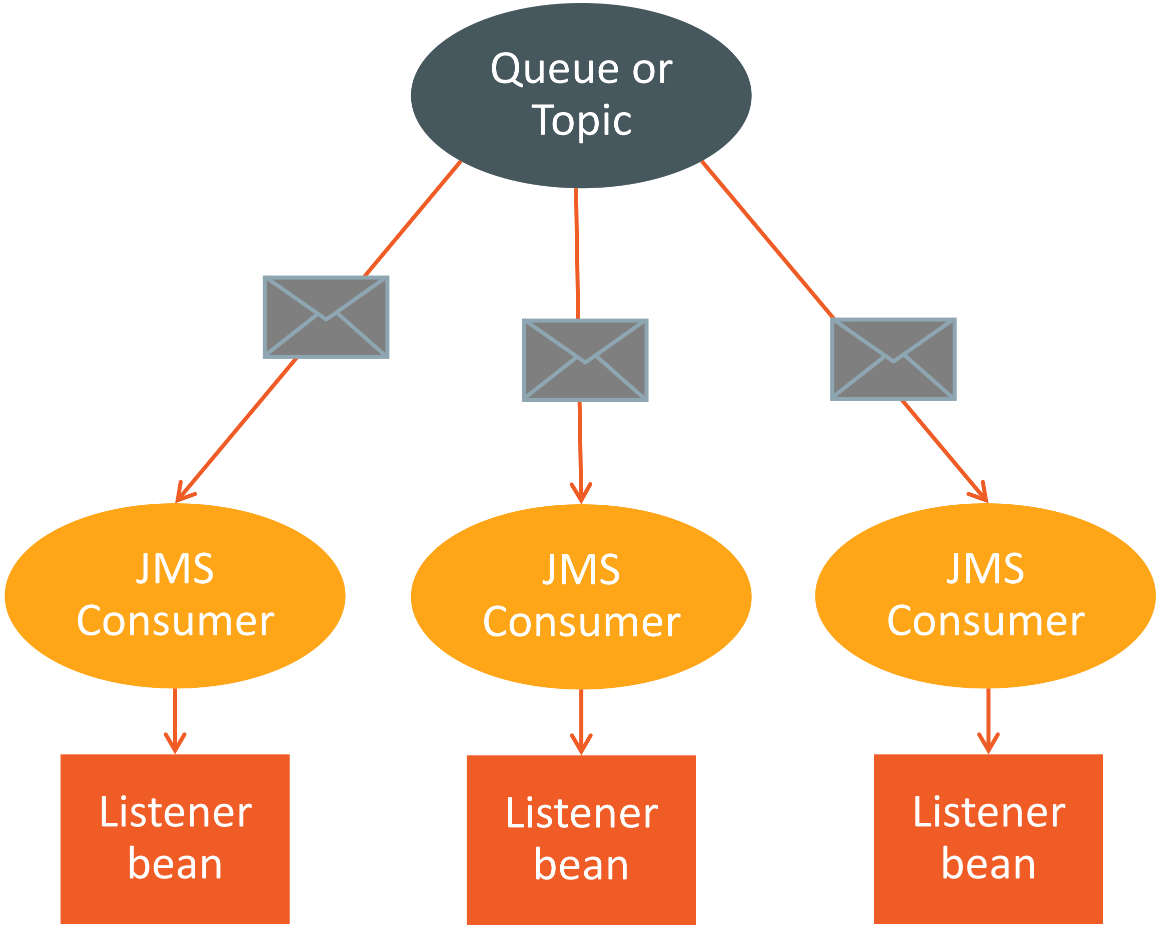
Managing concurrency with dependent-scoped JMS listener beans

```
// @Dependent
public class MyDepScopedBean {

  @MaxInstances(10)
  @JMSQueueListener(
    destinationLookup="myQueue")
  public void callMe(Message message) {
    ...
  }


}
```

- Each callback performed on a separate bean instance
- To increase throughput, allow messages to be delivered concurrently in multiple threads
- Define annotations to configure max number of instances
- To avoid creating and destroying beans, allow pooling of beans
- Hmm, sounds familar...

An alternative approach to dependent-scoped JMS listener beans

- Listener bean is injected
- Bean created when parent bean created
- Bean destroyed when parent bean destroyed
- One JMS consumer per bean instance
  - Created when bean is created
  - Closed when bean destroyed



Each instance and its consumer created when parent object created

An alternative approach to dependent-scoped JMS listener beans

Define the CDI bean

```
// @Dependent
public class MyDepScopedBean {

  @JMSQueueListener(
    destinationLookup="myQueue")
  public void callMe(Message message) {
    // increment count of messages
    ...
  }

  public int getNumMessages(){
    // return count of messages
    ...
  }
}
```

Inject it into a servlet

```
@WebServlet("/myjmsservlet1")

public class MyServlet extends HttpServlet {

  @Inject MyDepScopeListenerBean listener;
  // listener active for lifetime of servlet

  public void service(
    ServletRequest req, ServletResponse res)
    throws Exception {

   res.getWriter().println(
     "Number of messages received =
     "+ getNumMessages);
  }

}
```

Managing concurrency with the alternative approach to dependent-scoped JMS listener beans

```
// @Dependent
public class MyDepScopedBean {

  @MaxThreads(10)
  @JMSQueueListener(
    destinationLookup="myQueue")
  public void callMe(Message message) {
    // increment count of messages
    ...
  }

  public int getNumMessages(){
    // return count of messages
    ...
  }
}
```

- For a given JMS consumer, all message callbacks performed on the same bean instance
- To increase throughput, allow messages to be delivered concurrently in multiple threads to the same instance
- Define annotation to configure max number of threads
- Needs bean to be threadsafe

Two options for dependent-scoped JMS listener beans

Event-style approach

- Listener bean is not injected
- One JMS consumer per bean class
  - Created when application started
  - Closed when application shut down
- New bean for each message
  - Bean created when a message arrives
  - Bean destroyed after callback returns

Alternative approach

- Listener bean is injected and follows lifecycle of parent bean
- One JMS consumer per bean instance
  - Created when bean is created
  - Closed when bean destroyed
- Same bean for each message
  - Bean created when parent bean created
  - Bean destroyed when parent bean destroyed

How a "normal" scoped observer bean works in CDI

```
@RequestScoped
public class MyEventObserver {

  public void myObserverMethod(
    @Observes @SomeQualifier
    MyObject myObj){
      ...
  }
}
```

- Observer will only receive events fired within the SAME scope context
- For every event that is fired
  - The instance of the observer bean for this scope context is obtained
  - If doesn't exist then an instance is (by default) created
  - The observer method is invoked
- Observer beans can be injected and accessed directly

How normal scoped JMS listener bean might work

- In CDI, an event observer will only receive events that were fired from within the SAME scope context
- This doesn't make sense for JMS messages

@ApplicationScoped JMS listener beans

- When the application starts, create a consumer which delivers messages to the JMS listener bean
- For every message that is received
  - The instance of the JMS listener bean for is obtained
  - If doesn't exist then an instance is created
  - The callback method is invoked
- JMS Listener beans can be injected and accessed directly

@ApplicationScoped JMS listener beans

Define the CDI bean

```
@ApplicationScoped
public class MyAppScopedBean {

  @JMSQueueListener(
    destinationLookup="myQueue")
  public void callMe(Message message) {
    // increment count of messages
    ...
  }

  public int getNumMessages(){
    // return count of messages
    ...
  }
}
```

Inject it into your application

```
@Inject MyAppScopedBean myBean;
```

Call methods on the bean as needed

```
int count = myBean.getNumMessage();
```

"Application scope" means that all places it is injected will obtain the same single bean instance

@ApplicationScoped JMS listener beans: threading

```
@ApplicationScoped
public class MyAppScopedBean {

  @MaxThreads(10)
  @JMSQueueListener(
    destinationLookup="myQueue")
  public void callMe(Message message) {
    // increment count of messages
    ...
  }

  public int getNumMessages(){
    // return count of messages
    ...
  }
}
```

- Only one bean instance, so all message callbacks performed on the same bean instance
- To increase throughput, allow messages to be delivered concurrently in multiple threads to the same instance
- Define annotation to configure max number of threads
- Needs bean to be threadsafe

CDI beans as JMS message listeners: options

	Event-style approach (global consumer)	**Alternative approach (1 consumer per listener)**
Dependent-scoped	Feasible	Feasible

	Event-style approach
Application scoped	Feasible
Other normal scopes	?

API to configure async message listeners in Java EE

- `consumer.setMessageListener(MessageListener listener)`
- Currently not allowed by Java EE specification
- Some vendors do allow it anyway

Other improvements for JMS 2.1

Making use of Java SE 8 repeatable annotations

Java EE 7

```
@JMSConnectionFactoryDefinitions({
  @JMSConnectionFactoryDefinition(
    name="java:app/MyJMSCF1",
    interfaceName=
      "javax.jms.QueueConnectionFactory",
    resourceAdapter="myJMSRA"),
  @JMSConnectionFactoryDefinition(
    name="java:app/MyJMSCF2",
    interfaceName=
      "javax.jms.QueueConnectionFactory",
      resourceAdapter="myJMSRA")
})
```

Java EE 8

```
@JMSConnectionFactoryDefinition(
  name="java:app/MyJMSCF1",
  interfaceName=
    "javax.jms.QueueConnectionFactory",
  resourceAdapter="myJMSRA")

@JMSConnectionFactoryDefinition(
  name="java:app/MyJMSCF2",
  interfaceName=
    "javax.jms.QueueConnectionFactory",
  resourceAdapter="myJMSRA")
```

Making use of Java SE 8 repeatable annotations

Java EE 7

```
@JMSDestinationDefinitions({
  @JMSDestinationDefinition(
    name="java:app/MyJMSQueue",
    interfaceName="javax.jms.Queue",
    destinationName="myQueue1"),
  @JMSDestinationDefinition(
    name="java:app/MyJMSQueue",
    interfaceName="javax.jms.Queue",
    destinationName="myQueue2")
})
```

Java EE 8

```
@JMSDestinationDefinition(
  name="java:app/MyJMSQueue",
  interfaceName="javax.jms.Queue",
  destinationName="myQueue1")

@JMSDestinationDefinition(
  name="java:app/MyJMSQueue",
  interfaceName="javax.jms.Queue",
  destinationName="myQueue2")
```

Configuring message redelivery for MDBs (both new flexible MDBs and classic MDBs)

```
@MessageDriven
public class MyFlexibleMDB {

    @JMSQueueListener(destinationLookup="java:global/requestQueue",
        redeliveryInterval=1000,
        redeliveryLimit=10,
        deadMessageLookup="java:global/DMQ")
    public void myMessageCallback(Message message) {
        TextMessage textMessage = (TextMessage)message;
        try {
            String messageText = textMessage.getText();
            // process message text
            // ...
        } catch (JMSException e) {
            // exception handling
        // ...
        }
    }
}
```

Allowing messages to be delivered to MDBs in batches (taking advantage of flexible MDBs)

- Allow callback parameter to be an array of messages
- Configure with @Batch annotation

```
@MessageDriven
public class MyFlexibleMDB {

  @JMSQueueListener(
  destinationLookup="java:global/myQueue")
  public void myMessageCallback(
    @Batch(batchSize=10,batchTimeout=1000)
    Message[] messages) {
    ...
  }
}
```

- Enables multiple messages to be handled in same transaction
- `batchSize` : messages will be delivered in batches of up to batchSize messages
- `batchTimeOut`: Max time (in ms) app server may defer message delivery in order to assemble a batch of messages that is as large as possible but no larger than the batch size.

New and custom message acknowledgement modes

```
Session session = connection.createSession(int ackMode);
```

- Existing acknowledgement modes
  - AUTO_ACKNOWLEDGE, DUPS_OK_ACKNOWLEDGE, CLIENT_ACKNOWLEDGE
- New acknowledgement modes
  - NO_ACKNOWLEDGE: message deleted from queue/subscription when sent, no acknowledgement used, no redelivery on failure
  - INDIVIDUAL_ACKNOWLEDGE: `message.acknowledge()` acknowledges only that message, not previous messages received by the same session
- Custom acknowledgement modes
  - Allocate range of mode values for use by JMS vendors

API to create ConnectionFactory objects

- No standard API to create these objects in a Java SE application
  - Java EE applications can now use resource definition annotations
- Need a static method on a standard factory class (like JDBC `DriverManager`)

```
ConnectionFactory cf = javax.jms.ConnectionFactoryCreator.create(url, props);
```

- Standard implementation needs to be able to find out which JMS provider to use
- Perhaps use `java.util.ServiceLoader` to choose a JMS provider that supported the specified URL

API to create Queue and Topic objects

- Existing methods on `Session` and `JMSContext`:
  - `createQueue(String queueName)`
  - `createTopic (String topicName)`
- `queueName` and `topicName` are not portable

JMS in a Java EE application: adding clarifications and removing restrictions

- Defining the behavior of a JMS session that is created outside a JTA transaction but used to send or receive a message within a JTA transaction, and vice versa.
- Defining an API to allow a JMS connection factory, connection or session to opt-out of a JTA transaction
- Clarifying the existing restrictions on using client-acknowledgement and local transactions in a Java EE environment and removing these restrictions where possible
- Removing the restriction on calling `setMessageListener` in a Java EE application

Minor corrections to JMS 2.0 features

- Missing method `createXAJMSContext()` on `XAJMSContext` to allow multiple `XAJMSContexts` to share the same connection.
- API to allow application servers to implement `JMSContext` without needing an additional connection pool

Please get involved

- Download the JMS 2.1 early draft 1
- Join the JMS community mailing list
- Visit jms-spec.java.net for links to everything
- Follow (and reply to) @jms_spec
- Join the discussion at the JMS BOF tonight (9pm, here)

Safe Harbor Statement

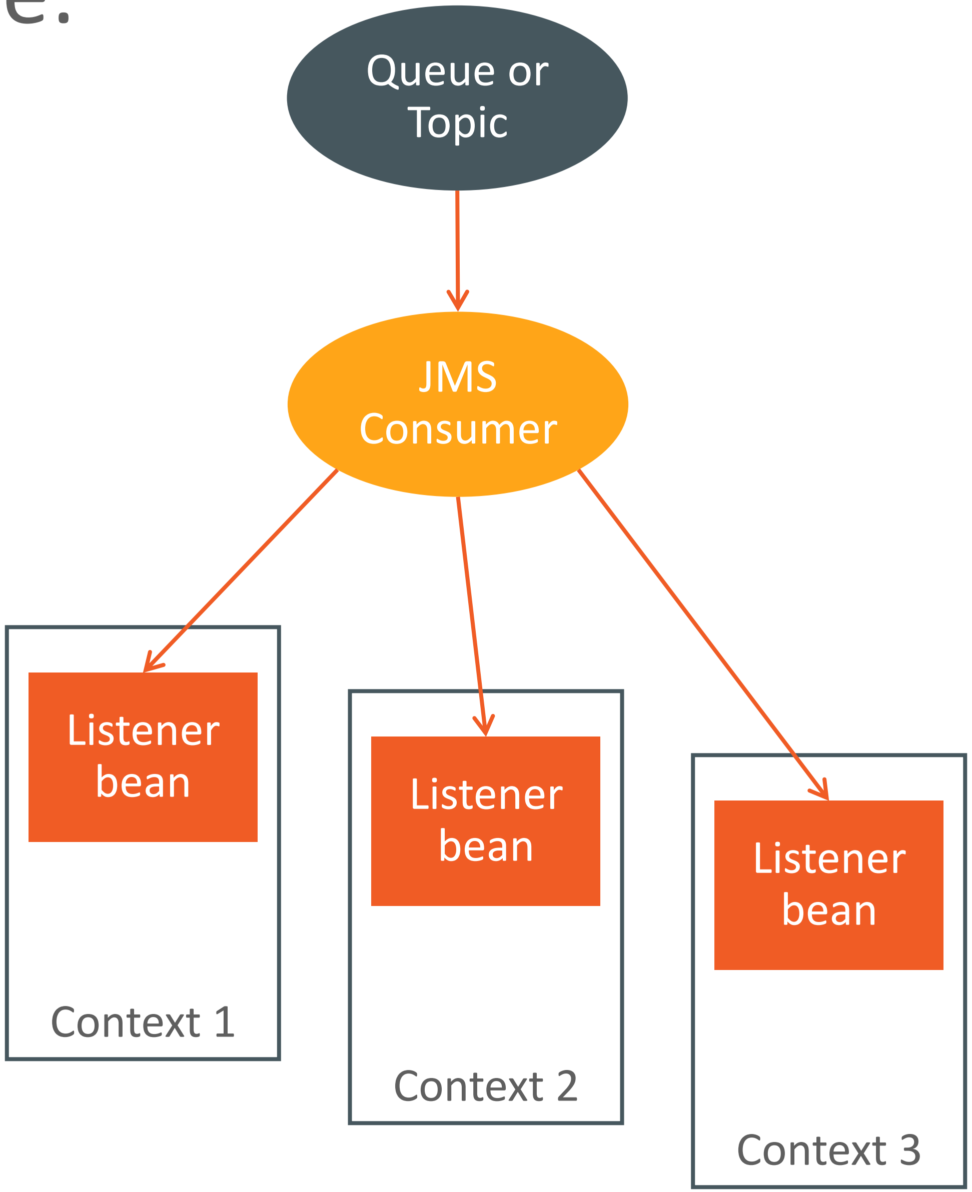
The preceding is intended to outline our general product direction. It is intended for information purposes only, and may not be incorporated into any contract. It is not a commitment to deliver any material, code, or functionality, and should not be relied upon in making purchasing decisions. The development, release, and timing of any features or functionality described for Oracle's products remains at the sole discretion of Oracle.

JavaOne™

ORACLE®

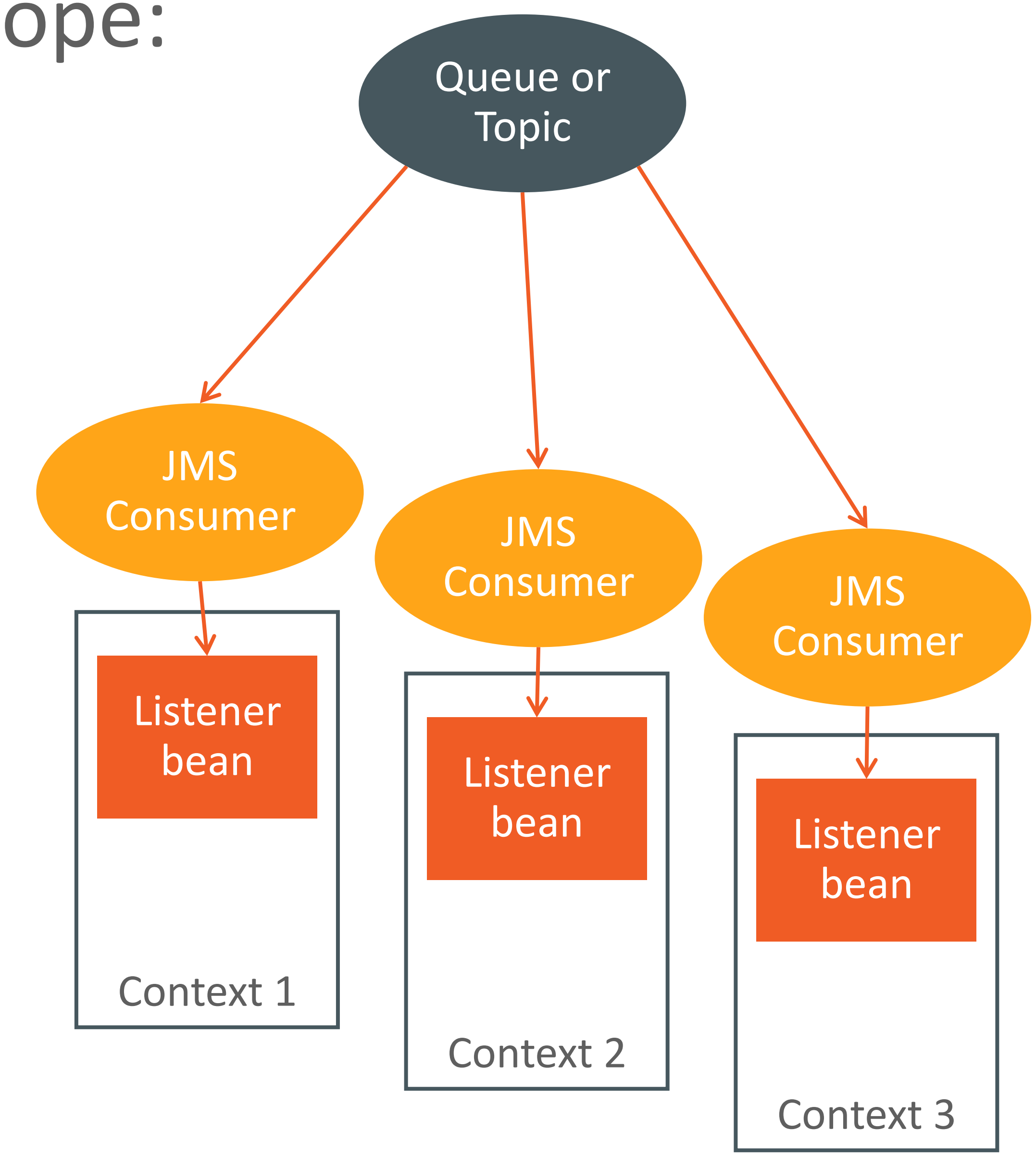
JMS listener beans with normal scope: Single consumer approach

- 1 consumer per listener bean class
  - Created at application startup
- When a message arrives from a topic
  - find each scope context,
  - obtain or create the listener bean for that context
  - deliver the message
- When a message arrives from a queue
  - randomly choose a single scope context
  - obtain or create the listener bean for that context
  - deliver the message



JMS listener beans with normal scope: 1 consumer per listener approach

- 1 consumer per listener bean instance
  - Created when listener bean is created
- Listener bean is created eagerly when scope context starts
  - otherwise would only be created lazily when the application calls a method on it
- Consumer delivers messages to the associated listener bean
- Listener bean destroyed when scope context ends


Queue or Topic
JMS Consumer
JMS Consumer
JMS Consumer
Listener bean
Listener bean
Listener bean
Context 1
Context 2
Context 3