Messaging mit Java (JMS)

Präsentation der Seminararbeit im FWP
Aktuelle Technologien verteilter Java-Anwendungen

Hochschule für angewandte Wissenschaften München
Fakultät für Informatik und Mathematik

„*Interesting applications rarely live in isolation.*“
Hohpe G., Woolf B.
Enterprise Integration Patterns, S. xxix

Gliederung

- indirekte Kommunikation über Zusatzsoftware (Messaging Provider) → hohe Entkopplung der Anwendungen
- Zugriff von Java mittels Java Message Service (JMS) API
- spezifizierte Schnittstellen
  - implementiert durch Messaging Provider → technologie-unabhängig
  - genutzt von Java Anwendung → provider-unabhängig
- Client: Send and Forget Prinzip → asynchrone Übertragung
- Messaging Provider: Store and Forward Prinzip → garantierte Übertragung

Nachrichtenziele und Kommunikationsarten

- Adressierung der Nachrichten an Nachrichtenziele (anstatt an konkreten Empfänger)
- Produzenten/Konsumenten zum Senden/Abrufen von Nachrichten an/aus einem Ziel
- pro Ziel beliebig viele Produzenten und Konsumenten möglich
- zwei Zielarten $\rightarrow$ unterschiedliche Kommunikationsarten

Point-to-Point Kommunikation mit Queues

- Einreihung neuer Nachrichten in Warteschlange (Queue)
- Aufstauen der Nachrichten bis Abruf
  → Abrufrate (Auslastung des Empfängers) wählbar
  → garantierte Abarbeitung
- Entnahme jeder Nachricht durch einen Konsumenten möglich
  → Lastverteilung
  → Kommunikation von einem Produzenten zu genau einem Konsumenten (Point-to-Point)
- Empfang in Abwesenheit gesendeter Nachrichten möglich
  → robust gegenüber Ausfall von Anwendungen
  → zeitliche Entkopplung sendender und empfangender Anwendungen

Publish-Subscribe Kommunikation mit Topics

- Sammlung von Nachrichten eines bestimmten Themas (Topic)
- Veröffentlichung von Nachrichten (Publishing)
- Erstellung von Abonnements (Subscription) durch interessierte Empfänger
- Abonnement vergleichbar mit Queue
- Zustellung neuer Nachrichten als Kopie an Abonnements
- keine Speicherung im Topic
- keine Zustellung an nach dem Senden erstellte Abonnements
- Abruf von Nachrichten über Konsumenten aus Subscription
  $\rightarrow$ Kommunikation von einem Produzenten zu beliebig vielen (auch keinem) Konsumenten (Publish-Subscribe)

Nachrichten

- atomare Übertragungseinheit
  - Übergabe an/Empfang vom Messaging Provider
  - Übertragung an Ziel
- Verpackung von Nutzdaten

Aufbau

Header

Properties

Body

Aufbau

Header

- Metadaten für Identifikation, Routing und Zustellung
- Auswertung durch Messaging Provider und Anwendungen
- Schlüssel-Wert-Paare (nur festgelegte Schlüssel zulässig)
- großteils automatisches Setzen beim Senden und Zustellen
  $\rightarrow$ nicht überschreibbar
- *z.B.* `JMSMessageId`, `JMSDestination`

Properties

Body

Aufbau

Header

Properties

- anwendungs- und Provider-spezifische Metadaten
- Schlüssel-Wert-Paare (beliebige Schlüssel)
- Auswertung insb. zur Filterung von Nachrichten
- Provider-spezifische Properties
  $\rightarrow$ Konfiguration von Zusatzoptionen
- *z.B.* `CustomerId`, `ExpressShipping`

Body

Aufbau

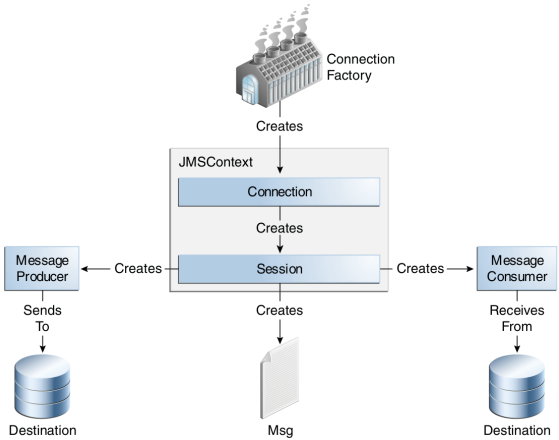
Header

Properties

Body

- Nutzdaten
- optional, leerer Body möglich
- *z.B. JSON oder XML*

Bestandteile der JMS API



[aus *The JEE Tutorial*, S. 7 Abb. 45-5]

Administrative Objekte

- `ConnectionFactory`
  Aufbau von Verbindungen zum JMS Provider
- `Queue`, `Topic`, `Destination`
  Repräsentation von Nachrichtenzielen
- administrative Konfiguration als Ressourcen im Applikationsserver
- Einbindung von deployten Anwendungen zur Laufzeit

Konfiguration im Applikationsserver

am Beispiel von Payara 4.1

```
asadmin create-jms-resource --restype javax.jms.Queue
  --property Name=physical_queue_a
  jms/QueueA
```

Einbindung via Ressource Injection

- Einbindung von JMS Ressourcen innerhalb des Web- und EJB-Containers
  JMS = Integrationstechnik → nur in EJB-Container!

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

@Resource(lookup = "jms/TopicA")
private Topic topicA;

@Resource(lookup = "jms/TopicA")
private Destination topicA;
```

JMS Context

- Kapselung von Verbindung und Transaktionskontext, selbst **kein administratives Objekt**
- Ausgangspunkt für das Erstellen von Objekten der JMS API
- Bereitstellung eines CDI-Producers für `JMSContext` durch JMS → CDI `@Inject` möglich
- Festlegen der `ConnectionFactory` über Qualifier-Annotation `@JMSConnectionFactory`

```
@Inject
private JMSContext jmsContext;

@Inject
@JMSConnectionFactory("jms/MyConnectionFactory")
private JMSContext myJmsContext;
```

Nachrichtentypen, Erstellung und Zugriff auf Body

- Interface `Message`
- verschiedene Datenformats des Bodies (Nachrichtentypen)
  → abgeleitete Interfaces, z.B.:
  - `TextMessage`
  - `StreamMessage`
- Erstellung über Factory-Methoden des `JMSContexts`
  - `createMessage()`
  - `createTextMessage(String)`
  - `createStreamMessage()`

TextMessage

```
Message toTextMessage(Order order) {
  String json = toJson(order);
  return jmsContext.createTextMessage(json);
}

Order toOrder(TextMessage message) {
  String json = message.getText();
  return fromJson(json);
}
```

Message

- statischer Typ für alle empfangenen Nachrichten
  → Typecast auf dynamischen Typ

```
Order toOrder(Message message) {
  if (message instanceof TextMessage) {
    return toOrder((TextMessage) message);
  } else if (message instanceof ObjectMessage) {
    return toOrder((ObjectMessage) message);
  }
  ...
}
```

Message

- Kurzschreibweisen zum Entpacken des Bodies mit `getBody(Class<T>)`-Methode

```
Order toOrder(Message message) {
  if (message instanceof TextMessage) {
    String json = message.getBody(String.class);
    return fromJson(json);
  } else if (message instanceof ObjectMessage) {
    return message.getBody(Order.class);
  }
  ...
}
```

Produzenten

- Interface `JMSProducer` zum Senden von Nachrichten
- Erstellung über Factory-Methode `createProducer()` des `JMSContexts`
- Builder-Pattern → Verkettung von Aufrufen

Builder-Pattern der Produzenten

Producer konfigurieren

- `setProperty(String, int)`
- `clearProperties()`

Nachricht senden

- `send(Destination, Message)`
- `send(Destination, String)`
- `send(Destination, Serializable)`
- `send(Destination, Map<String, Object>)`
- `send(Destination, byte[])`

Blockierung des aufrufenden Threads bis erfolgter Übergabe der Nachricht an Messaging Provider
→ Übergabe synchron, Übertragung asynchron

```
@Stateless
class CustomerNotifier {
  @Inject private JMSContext jmsContext;
  @Resource(...) private Destination shippingDocuments;
  @Resource(...) private Destination orderEvents;

  public void orderShipped(long orderId, int trackingId, ShippingLabel
      shippingLabel) {
    Message event = jmsContext.createMessage();
    jmsContext.createProducer()
      .setProperty("document", "SHIPPING_LABEL")
      .send(shippingDocuments, toJson(shippingLabel))
      .clearProperties()
      .setProperty("event", "ORDER_SHIPPED")
      .setProperty("orderId", orderId)
      .setProperty("trackingId", trackingId)
      .send(orderEvents, event);
  }
}
```

Beispielanwendung: Online Shop

- Shop Anwendung
  - Aufgeben/Generieren von Bestellungen
  - Ablage in Queue
- Versand Anwendung
  - Empfangen aus Queue
  - Anforderung von Versandetiketten

Realitätsnahe Umgebung

- Docker Container mit je einer Payara Domain und Anwendung
- Master Container
  Betreiben des mitgelieferter JMS Providers OpenMQ im `EMBEDDED`-Modus
- Slave Container
  Verwenden des JMS Providers im `REMOTE`-Modus

Konsumenten

- Interface `JMSConsumer` zum synchronen Empfang von Nachrichten
- Erstellung über Factory-Methoden des `JMSContext`s
- Konsumenten implementieren `AutoCloseable`
  → `try`-with-resources-Block

Empfangen von Nachrichten

Methoden zum synchronen Nachrichtenabruf

- `Message receive()`
  `Message receive(long timeout)`
  Blockieren des aufrufenden Threads bis zur Verfügbarkeit einer neuen Nachricht
  optional: Timeout
- `Message receiveNoWait()`
  Sofortiges Zurückkehren des Aufrufs mit neuer Nachricht oder `null`

Kurzschreibweise zum Empfangen und Entpacken

- **T** receive**Body**(**Class<T>**)
- **T** receive**Body**(**Class<T>**, long timeout)
- **T** receive**Body**NoWait(**Class<T>**)

T = Typ des Bodies (z.B. `TextMessage` → `String.class`)

Konsumenten für Queues

`createConsumer(Destination)`

- Bildung an Ziel
- Lastverteilung durch parallele Konsumenten
  z.B. Anforderung von Versandetiketten für neue Bestellungen
- keine Ausführung verschiedener Verarbeitungsschritte möglich
  z.B. entweder Versandvorbereitung oder Rechnungserstellung für eine Bestellung

Konsumenten für Topic

Nicht-dauerhafte Abonnements

`createConsumer(Destination)`

- Ausführung verschiedener Verarbeitungsschritte möglich
  z.B. Versandvorbereitung und Rechnungserstellung
- Erstellen eines Abonnements nur über Konsument möglich
- Lebensdauer Abonnement = Lebensdauer Konsument (nicht dauerhaft)
  $\rightarrow$ keine Ausfallsicherheit *z.B. Verlust von Bestellungen*

Konsumenten für Topic

Dauerhafte Abonnements

`createDurableConsumer(Topic, String subscriptionName)`

- Vergabe eines eindeutigen Abonnementnamens
- Lebensdauer Abonnement > Lebensdauer Konsument
  Löschung mit `unsubscribe(String)`
- Wiederaufnahme des Abonnements über Abonnementname durch Nachfolgende Konsumenten möglich
  → Ausfallsicherheit (*z.B. bei Ausfall der Versand-Anwendung*
- keine gleichzeitige Verwendung eines Abonnements möglich
  → keine Lastverteilung

Konsumenten für Topic

Geteilte nicht-dauerhafte Abonnements

`createSharedConsumer(Topic, String subscriptionName)`

- gemeinsame Verwendung von Abonnements durch mehrere Konsumenten
  ()Teilnahme über gleichen Abonnementnamen)
  $\rightarrow$ Lastverteilung möglich
- keine Ausfallsicherheit

Konsumenten für Topic

Geteilte dauerhafte Abonnements

`createSharedDurableConsumer(Topic, String subscriptionName)`

- Kombination von geteiltem und dauerhaften Abonnement
- Lastverteilung
- Ausfallsicherheit
- **WICHTIG:** Erstmalige Erstellung des dauerhaftes Abonnements via Konsument nötig! $\leftrightarrow$ Queue

Message Driven Beans

- EJBs + `@MessageDriven` + `implements MessageListener`
- Steuerung der Instanz-Anzahl durch EJB-Container
  → für Topics immer geteilte Abonnements (Abonnementname von Container vergeben)
- Asynchroner Nachrichtenempfang neuer Nachrichten über Callback
  `onMessage(Message)`

Message Driven Beans

- Konfiguration über `@ActivationConfigProperty`-Annotations

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(
    propertyName = "destinationLookup",
    propertyValue = "jms/NewOrders"
  ),
  @ActivationConfigProperty(
    propertyName = "destinationType",
    propertyValue = "javax.jms.Topic"
  )
})
```

Message Driven Beans

- Abonnementname erforderlich für dauerhafte Abonnements

```
@MessageDriven(activationConfig = {
  ...
  @ActivationConfigProperty(
    propertyName = "subscriptionDurability",
    propertyValue = "Durable"
  ),
  @ActivationConfigProperty(
    propertyName = "subscriptionName",
    propertyValue = "createBillForNewOrder"
  )
})
```

Literatur

1 N. Deakin, M. Hapner, R. Burridge u. a. *JSR-343 Java Message Service (JMS) 2.0 (Specification").* Oracle America, Inc., 2013.
URL: http://download.oracle.com/otn-pub/jcp/jms-2_0-fr-eval-spec/JMS20.pdf (besucht am 07.05.2017).

2 *GlassFish Server Open Source Edition Administration Guide, Relea- se 4.0.* Oracle, 2013. Kap. 18 Administering the Java Message Service (JMS).
URL: https://glassfish.java.net/docs/4.0/administration-guide.pdf (besucht am 29.04.2017).

3 G. Hohpe und B. Woolf. *Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions.* Boston: Addison-Wesley, 2004.

4 E. Jendrock, R. Cervera-Navarro, I. Evans u. a. *Java Platform, Enterprise Edition The Java EE Tutorial, Release 7.* Oracle, 2014. Kap. 45 Java Message Service Concepts.
URL: https://docs.oracle.com/javaee/7/JEETT.pdf (besucht am 07.04.2017).

Alle verwendeten eingetragenen Waren- und Dienstleistungsmarken sind Eigentum ihrer jeweiligen Rechteinhaber.