JMS – JAVA MESSAGE SERVICES

Entwicklung von Webanwendungen SS 07

Marc Seeger [ms155]
Stephan Helten [sh094]

HOCHSCHULE DER MEDIEN

Agenda

- **Teil 1: Marc Seeger [ms155]**
  - Einführung: Was ist Messaging
  - Message Oriented Middleware [MOM]
  - Bestandteile einer JMS-Applikation
  - Was JMS nicht erledigt
  - Messages

---------- *Kurze Pause zum Popcorn holen* ----------

- **Teil 2: Stephan Helten [sh094]**
  - Die **JMS-API**
  - Der Aufbau eines **synchronen** JMS-Programms
  - Der Aufbau eines **asynchronen** JMS-Programms
  - Vom Sourcecode zum laufenden Programm
  - Anschauliches Praxisbeispiel in NetBeans

TEIL 1

Marc Seeger [ms155]

MESSAGING?

Messaging?

MASSAGEN? DIRK?

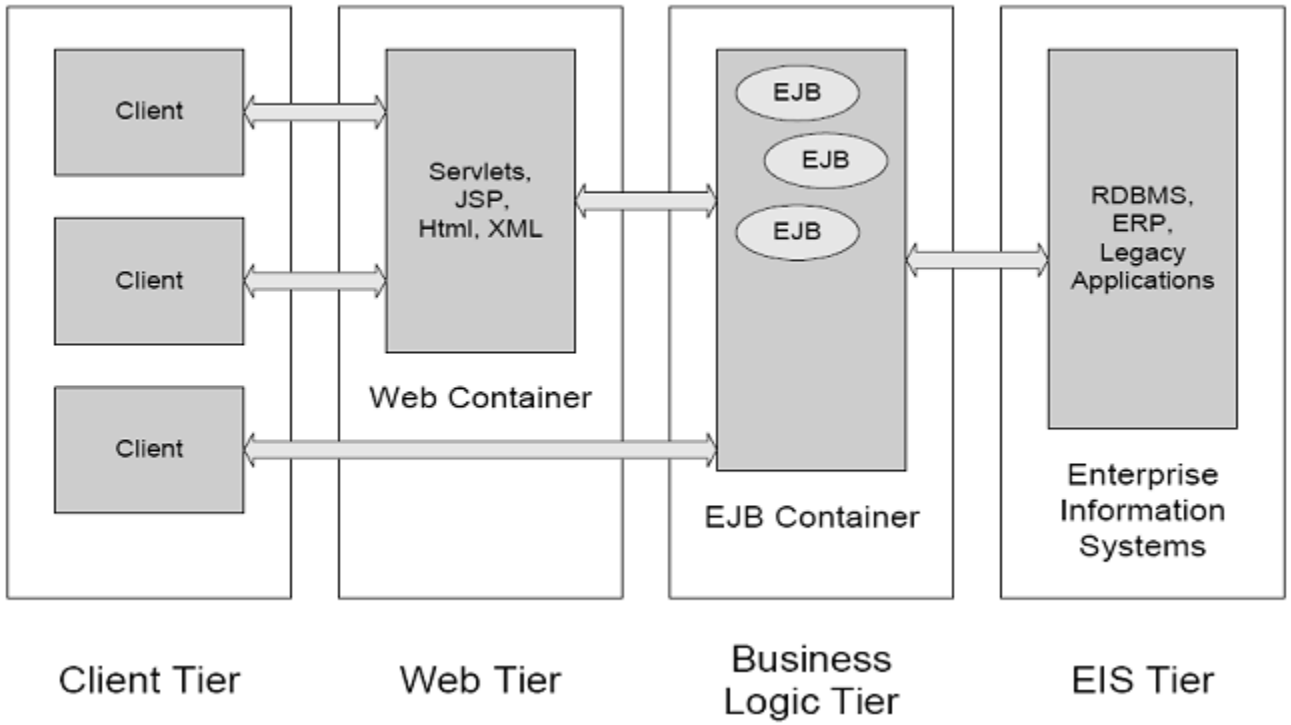
Messaging?

Messaging?


Serveradresse: messi.mi.hdm-stuttgart.de
Benutzer & Passwort = HdM Account
Stephan Helten
[19:47] Donnerstag, 7. Juni 2007
[19:47] Marc: VM auf meinem TFT
[19:47] Marc: mac os auf meinem LCD vom notebook
[19:48] Stephan Helten: roxxort hart!
[19:48] Marc: ziemlich
[20:02] Marc: Bitte recht freundlich, das ist ein Screenshot für die Präsentation
[20:05] Stephan Helten: Oh, hallo Marc Seeger! Sie wissen doch, dass ich stets freundlich bin!
[23:31] Marc Seeger: EIGNETLICH sollte das automatisch geschehen
[23:31] Andreas Lichtenberger: versteh schon
[20:03] Donnerstag, 7. Juni 2007
[20:03] Marc Seeger: Bitte recht freundlich, das wird ein Screenshot für die JMS Präsentation
[20:03] Andreas Lichtenberger: lach
Kontakte
Marc
Anwesend
MIB4
Andreas Lichtenberger
Sonstiges
HdM
Ligosworld
Sebastian Roth
Messi Team
Dirk Wendling
I've been away since 17:54.
General
Bots
Prof-Info Bot
Kontaktdaten und mehr
Woerterbuch
Ok, let's go, I am ready f...
News-Fritze
Notiz Bot
Das virtuelle Post-it
Schreihals
Seen Bot
Bereit Anfragen zu bea...
Whois Bot
whoisbot is waiting for ...
CCWN

Messaging?

"lose gekoppelte, verteilte Kommunikation"

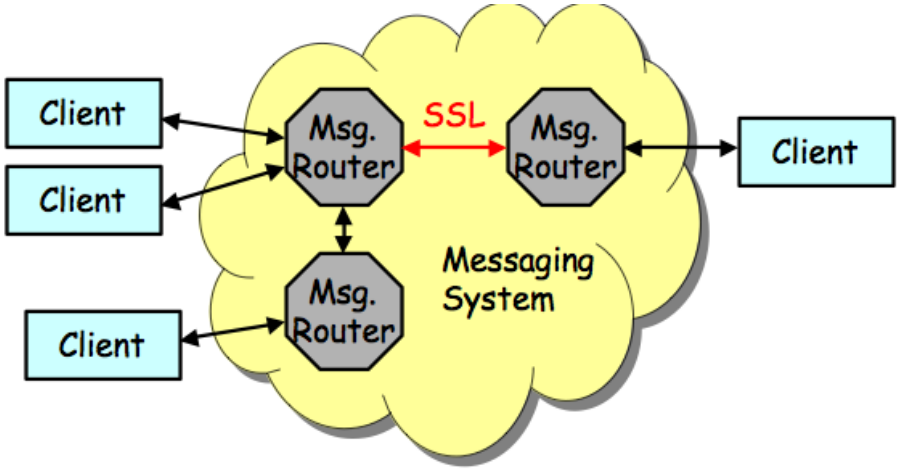


MESSAGE ORIENTED MIDDLEWARE

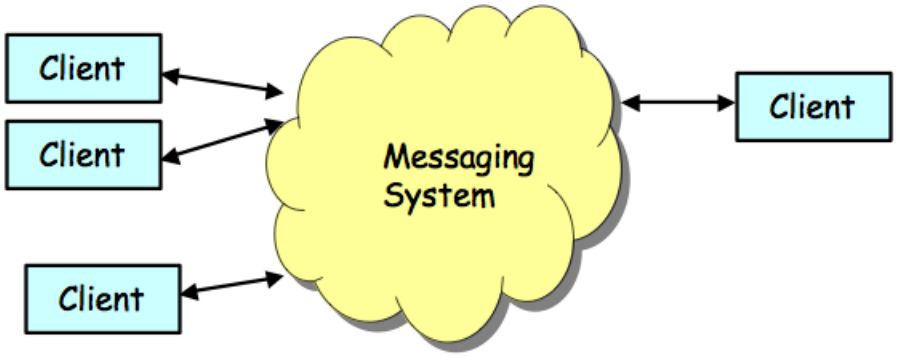
Every DAD needs a MOM

DAD = Distributed Application Development
MOM = Message Oriented Middleware

Ohne MOM


Client
Client
Client
Msg. Router
SSL
Msg. Router
Msg. Router
Messaging System
Client

Mit MOM



Was macht MOM genau?

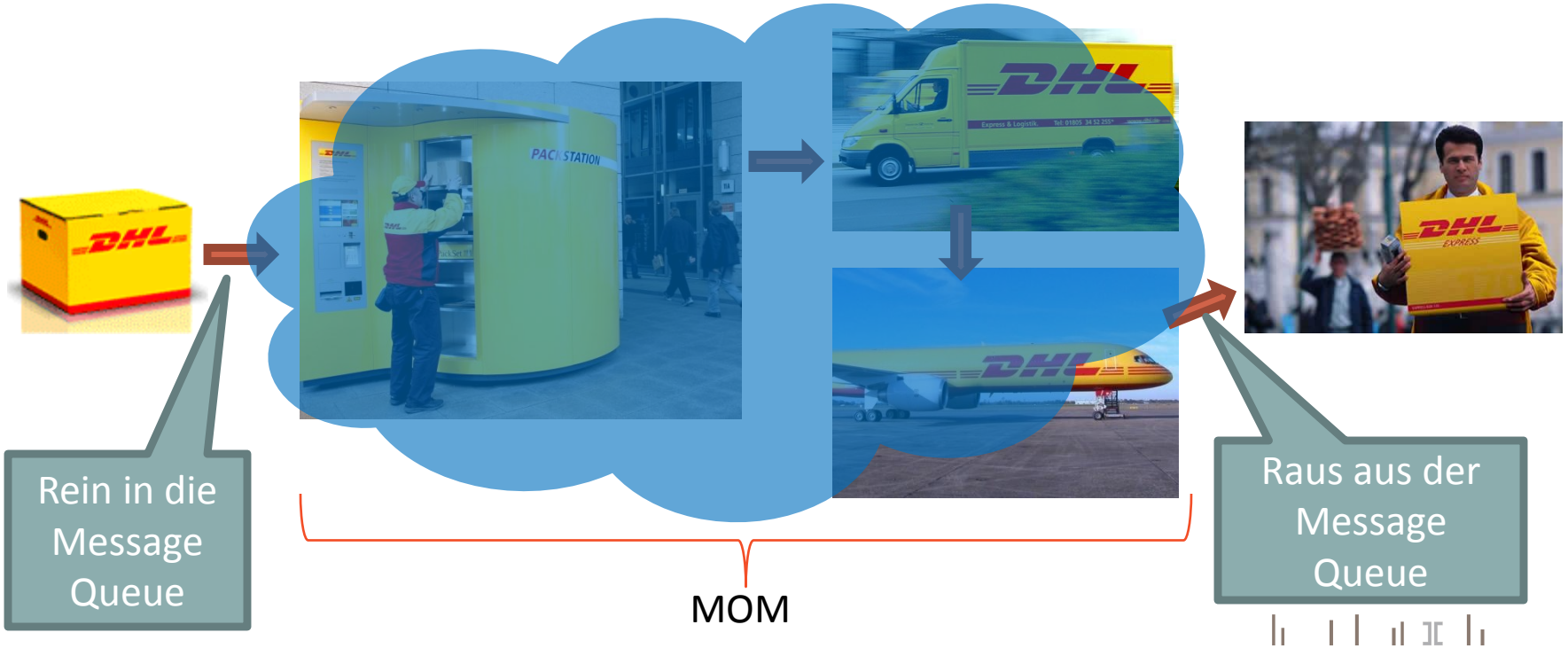
- Indirekte, verlässliche, asynchrone Kommunikation über Message-Queues

→Kommunikation ohne direkte, logische Verbindung → entkoppelt

→Client und Server können zu unterschiedlichen Zeiten im Netz aktiv sein

→Kommunikationsteilnehmer unterhalten sich durch "hineinlegen" und "herausnehmen" von Nachrichten in das/aus dem Netz

→Nachricht wird "once and only once" zugestellt



Ist indirekte, asynchrone Kommunikation wirklich so toll?

- Zeitliche Entkopplung zwischen Beauftragung und Bearbeitung
- Lose Kopplung erleichtert Austausch einzelner Systeme
- Flexibel konfigurierbare Kommunikation und Diensteverteilung
- Plattform- und programmiersprachenunabhängige Anbindung an Fremdsysteme z.B. Hosts
- Batchbetrieb ist möglich
- Verteilte Anwendungen
- Verteilte Transaktionen
- Wartungsfreundlichkeit
- Definierbare Sicherheitsmechanismen
- Hohe Ausfallsicherheit erreichbar
- Hohe Skalierbarkeit

Was hat das mit JMS zu tun?

- MOMs existieren schon seit den 70er Jahren
- JMS wurde 1999 von Sun als "MOM API" spezifiziert
- viele MOM Hersteller bieten nun JMS Schnittstellen ("JMS Provider") an
- Beispiel für JMS Provider:
  - SunONE Message Queue (SUN)
  - MQ JMS (IBM) (MQSeries kann als JMS Provider konfiguriert werden.)
  - Websphere MQ (IBM)
  - WebLogic JMS (BEA)
  - JBoss Messaging
  - OpenJMS

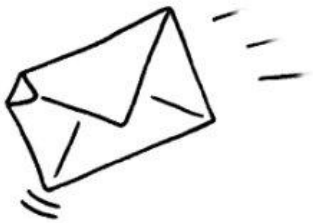
AUS WAS BESTEHT SO EINE JMS APPLIKATION?

Aus was besteht so eine JMS Applikation?

JMS Clients

Non-JMS Clients

JMS-Provider

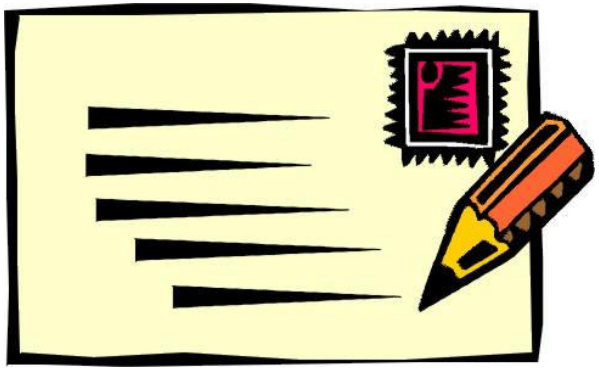


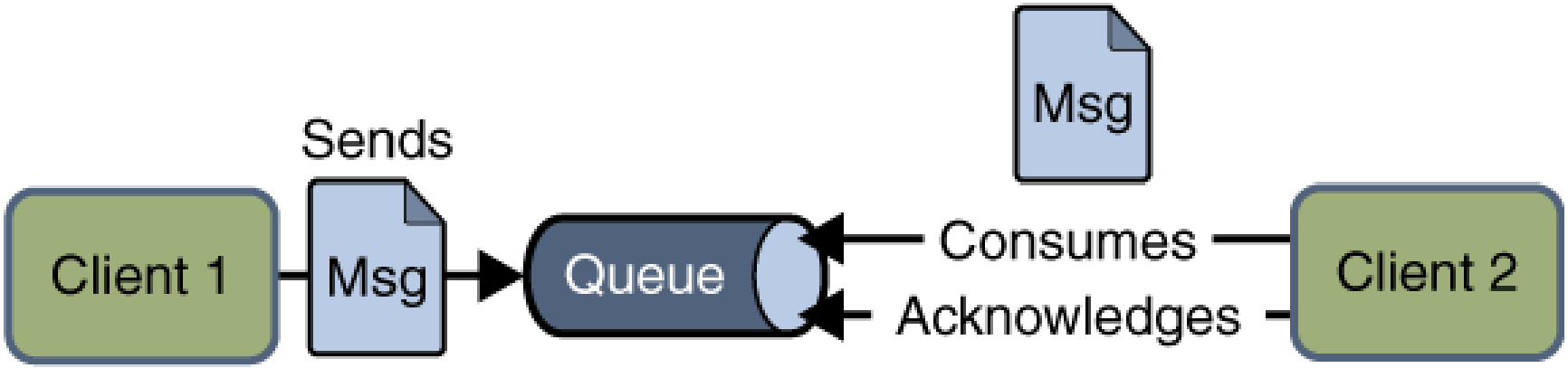
Message

Administered Objects

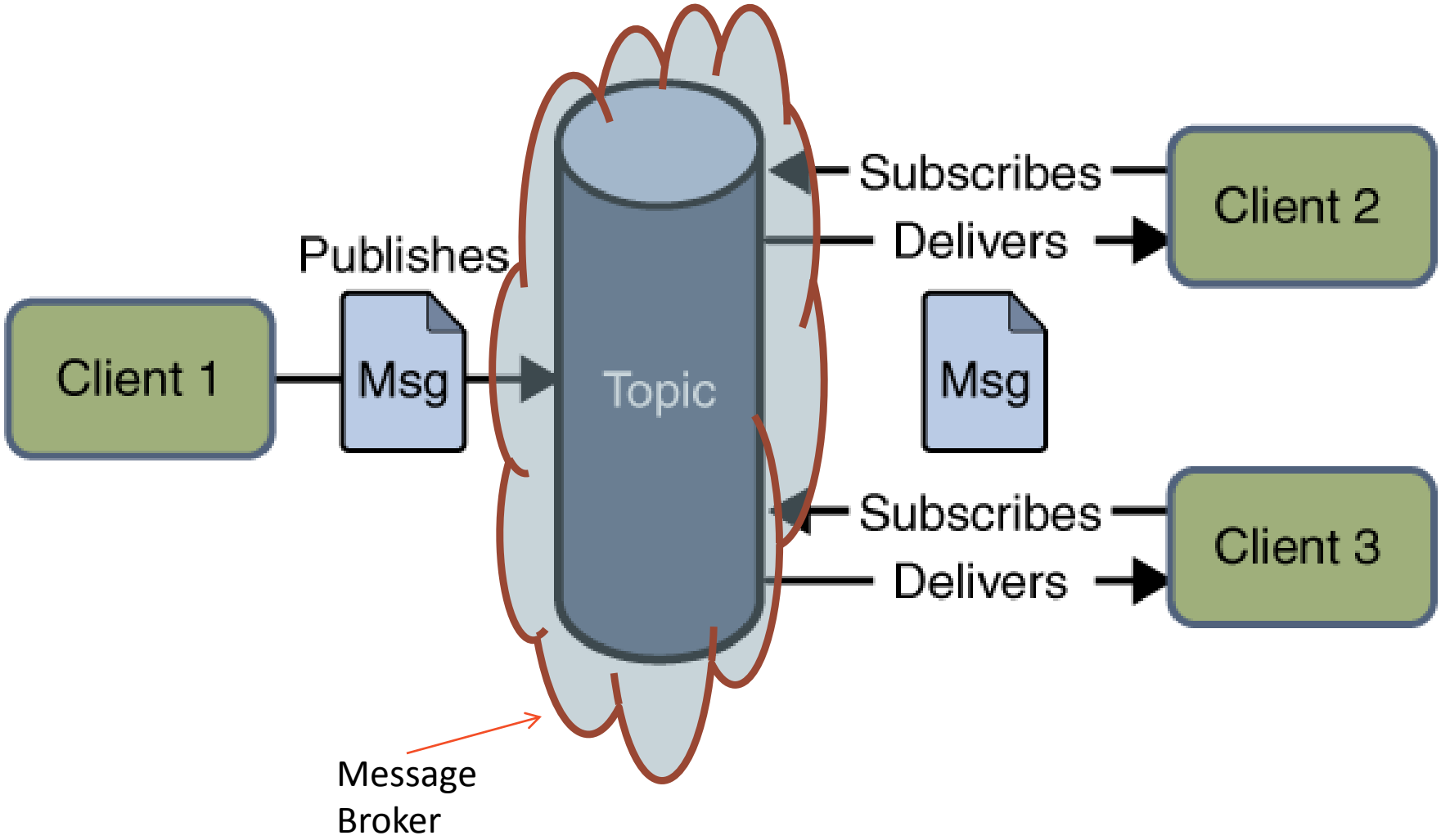
QUEUE UND TOPIC

Die Queue ("Point to Point")





Das Topic ("Publish Subscribe" aka "PubSub")



Das Topic ("Publish Subscribe" aka "PubSub")

Subscriber

Message!

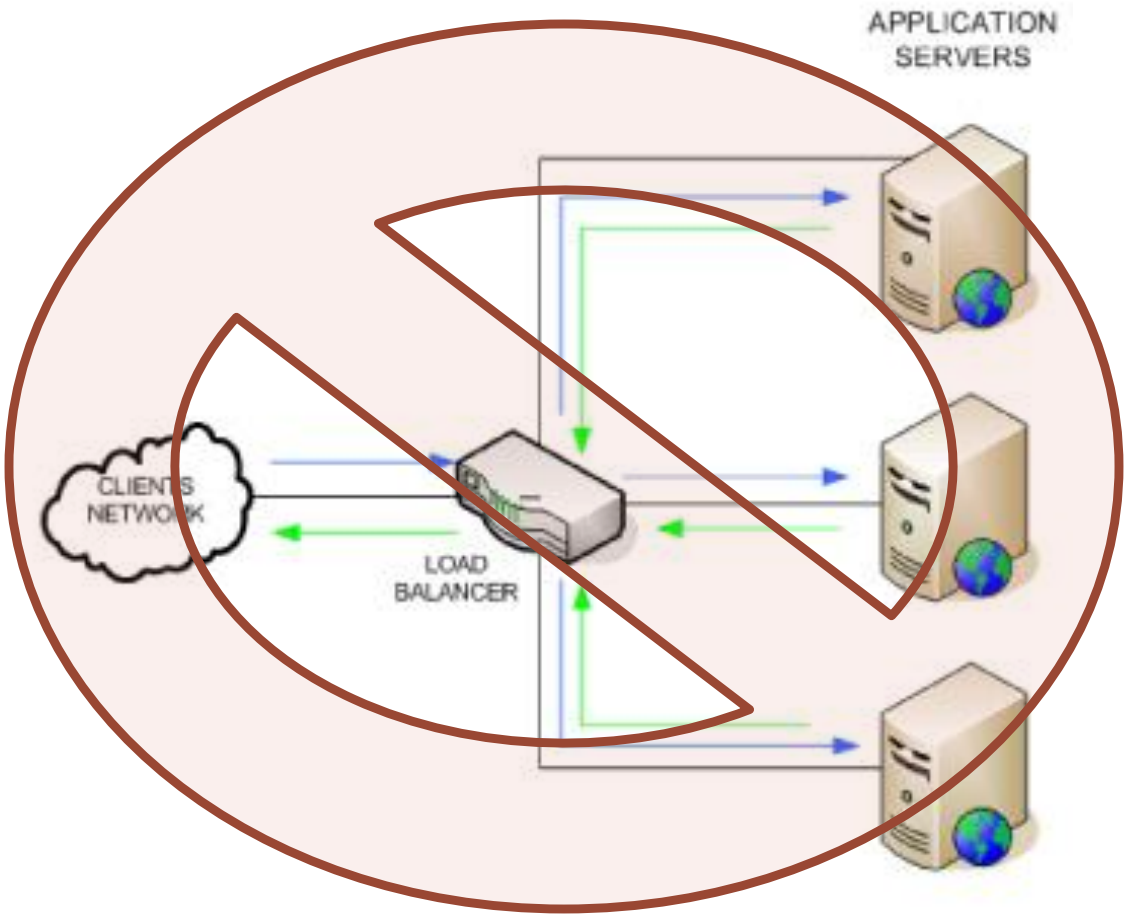
Hörsaal

"Publish"

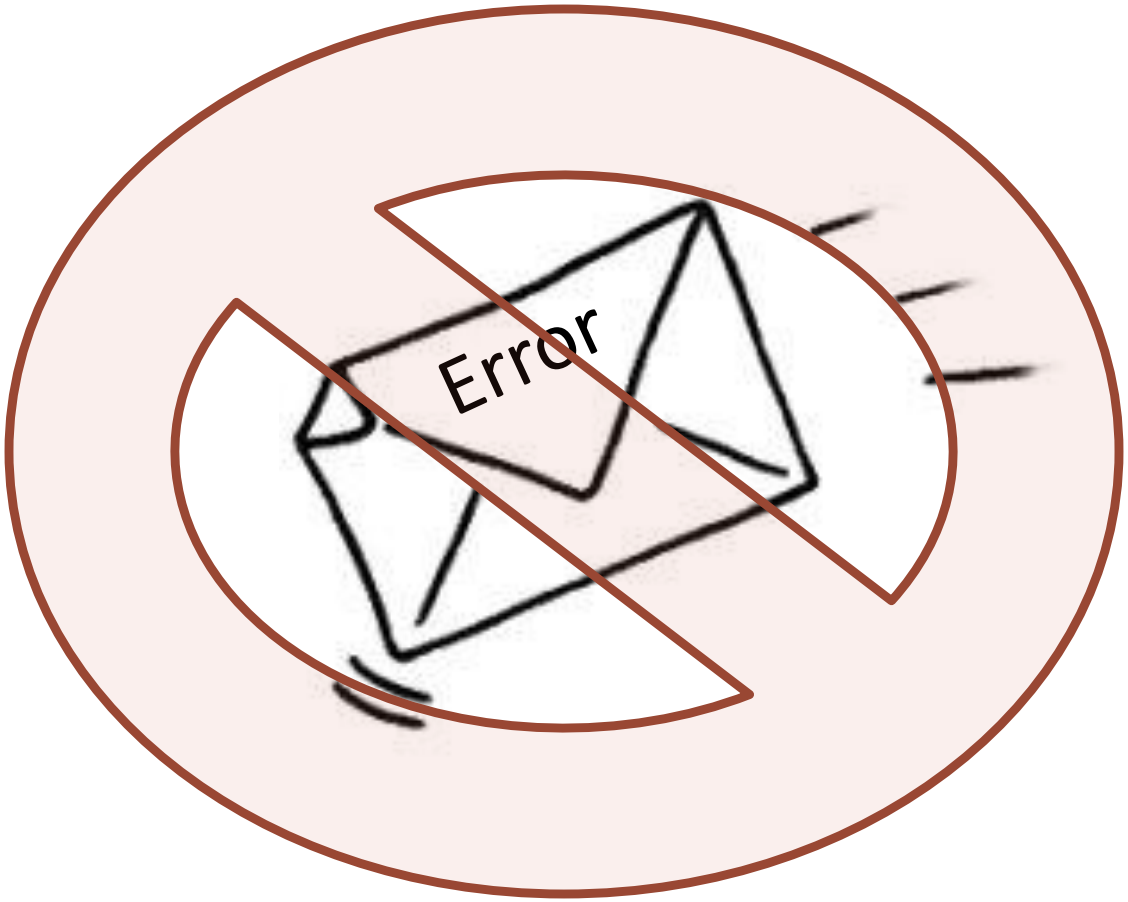
Keine Subscriber

WAS JMS NICHT ERLEDIGT

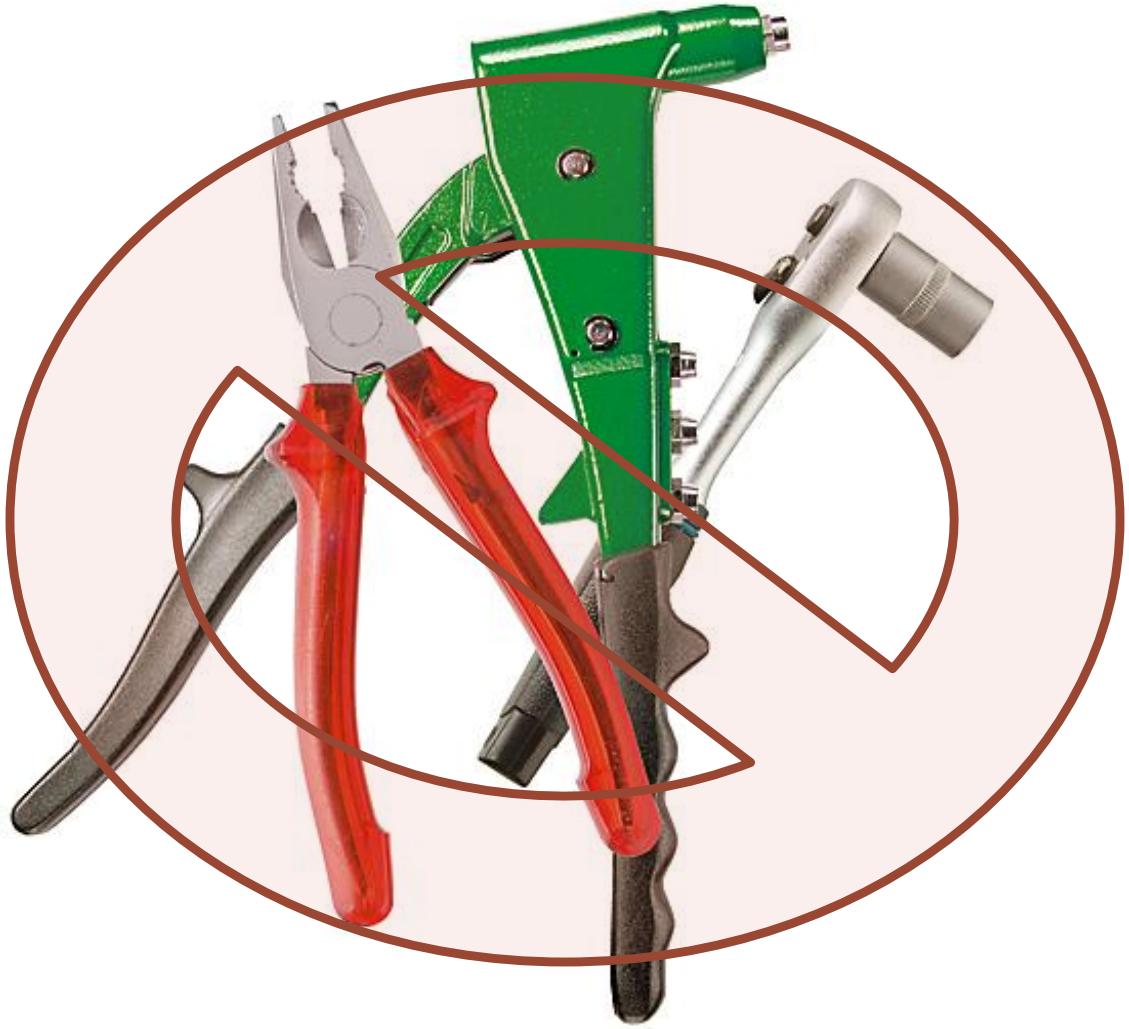
Load Balancing / Ausfallsicherheit



Fehlerbenachrichtigung



Administration

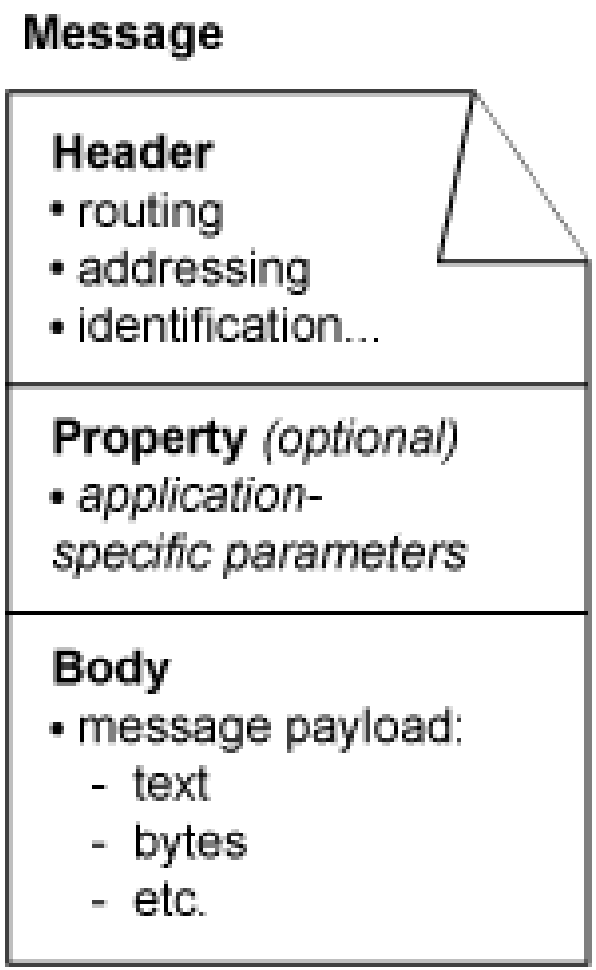


Security

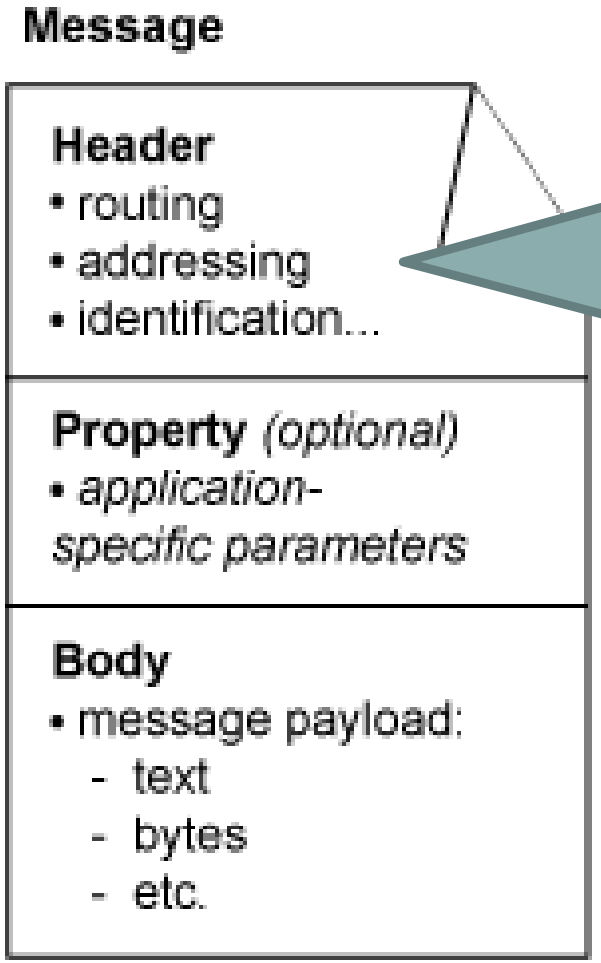
Message Type Repository

MESSAGES

Aufbau


Message
Header
• routing
• addressing
• identification...
Property (optional)
• application-specific parameters
Body
• message payload:
- text
- bytes
- etc.

Header



Header Fields	Set By
JMSDestination	Send Method
JMSDeliveryMode	Send Method *
JMSExpiration	Send Method *
JMSPriority	Send Method *
JMSMessageID	Send Method
JMSTimestamp	Send Method
JMSCorrelationID	Client
JMSReplyTo	Client
JMSType	Client
JMSRedelivered	Provider

* Administrator kann "Override" setzen

Property

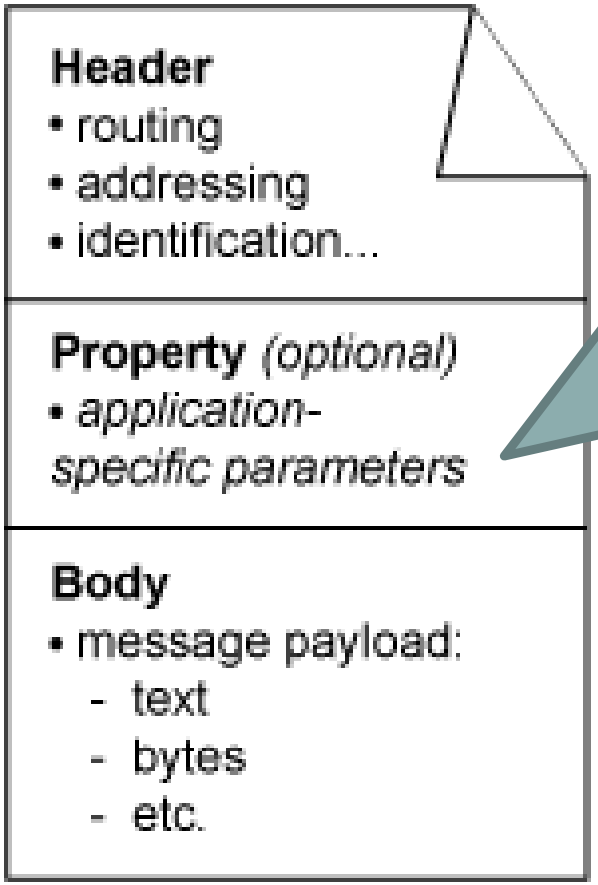


Table 3-3 JMS Defined Properties

Name	Type	Set By	Use
JMSXUserID	String	Provider on Send	The identity of the user sending the message
JMSXAppID	String	Provider on Send	The identity of the application sending the message
JMSXDeliveryCount	int	Provider on Receive	The number of message delivery attempts; the first is 1, the second 2,...
JMSXGroupID	String	Client	The identity of the message group this message is part of
JMSXGroupSeq	int	Client	The sequence number of this message within the group; the first message is 1, the second 2,...
JMSXProducerTXID	String	Provider on Send	The transaction identifier of the transaction within which this message was produced
JMSXConsumerTXID	String	Provider on Receive	The transaction identifier of the transaction within which this message was consumed
JMSXRcvTimestamp	long	Provider on Receive	The time JMS delivered the message to the consumer
JMSXState	int	Provider	Assume there exists a message warehouse that contains a separate copy of each message sent to each consumer and that these copies exist from the time the original message was sent. Each copy's state is one of: 1(waiting), 2(ready), 3(expired) or 4(retained). Since state is of no interest to producers and consumers, it is not provided to either. It is only relevant to messages looked up in a warehouse, and JMS provides no API for this.

Property

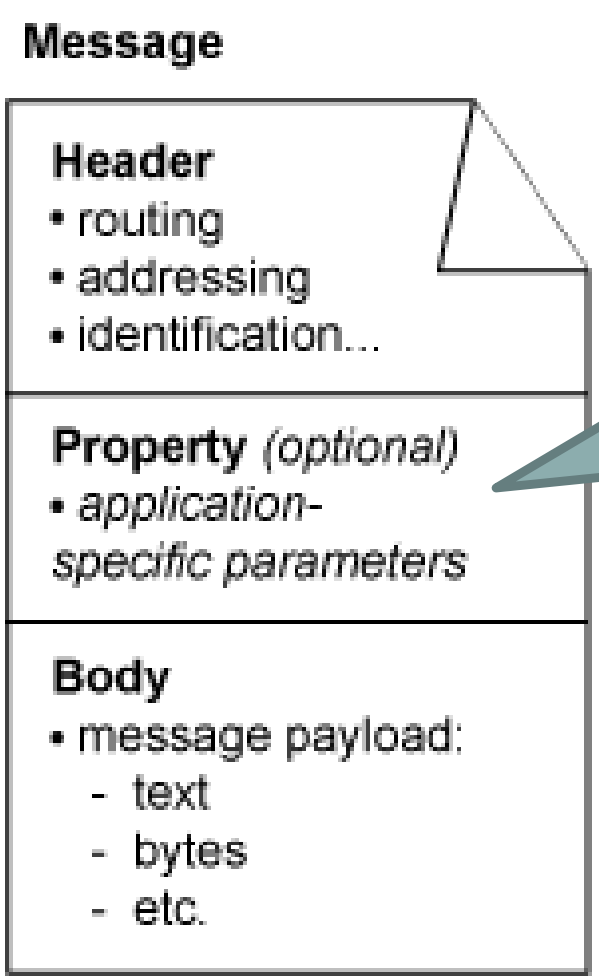
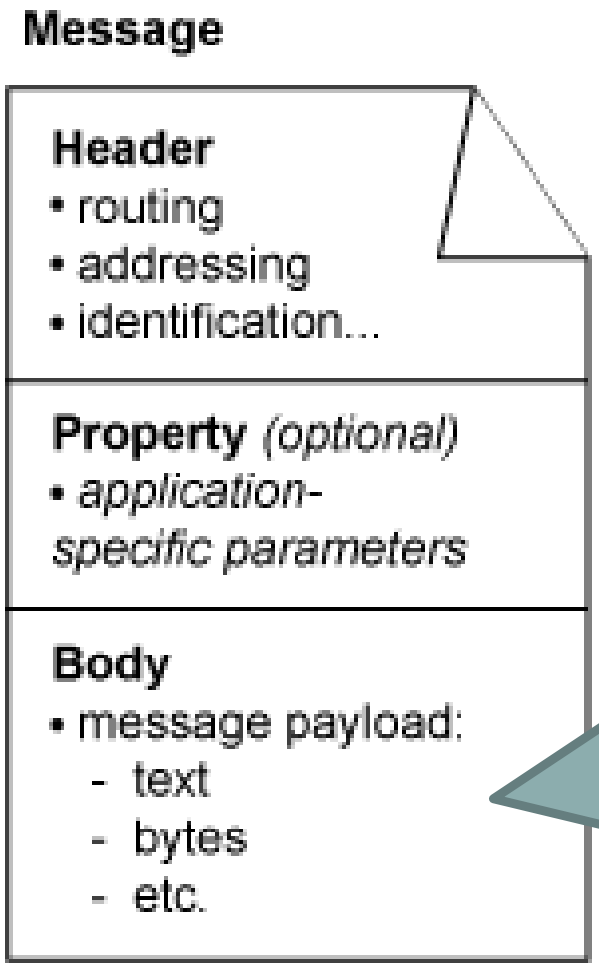


Table 3-2 Property Value Conversion

	boolean	byte	short	int	long	float	double	String
boolean	X							X
byte		X	X	X	X			X
short			X	X	X			X
int				X	X			X
long					X			X
float						X	X	X
double							X	X
String	X	X	X	X	X	X	X	X

Body



StreamMessage	a message whose body contains a stream of Java primitive values. It is filled and read sequentially.
MapMessage	a message whose body contains a set of name-value pairs where names are Strings and values are Java primitive types.
TextMessage	a message whose body contains a java.lang.String.
ObjectMessage	a message that contains a Serializable Java object.
BytesMessage	a message that contains a stream of uninterpreted bytes. This message type is for literally encoding a body to match an existing message format.

Message Selector

TEIL 2

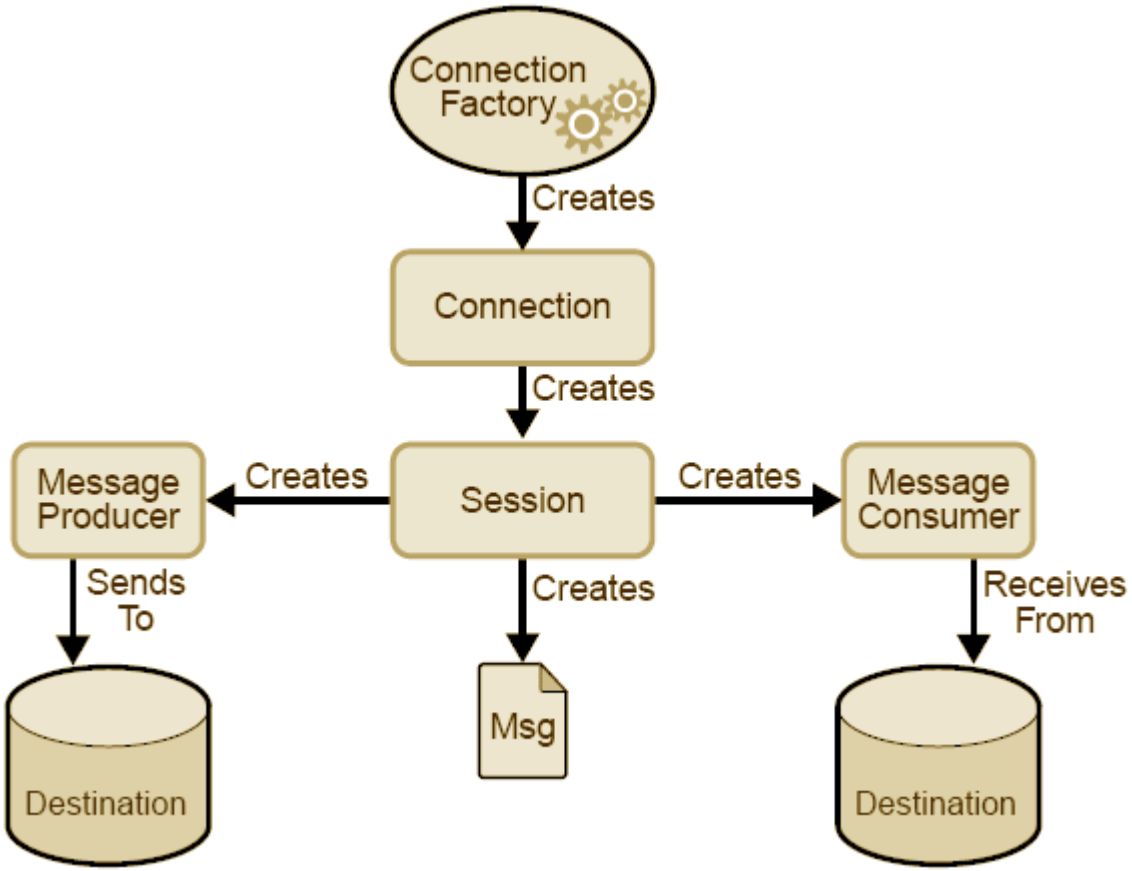
Stephan Helten [sh094]

DIE JMS-API

Überblick: Die zentralen Interfaces der JMS-API

- **Connection Factory**
- **Connection**
- **Session**
- **Message Producer**
- **Message Consumer**
- **Destination**
  - **Queue**
  - **Topic**
- **Message**

Das Programmier-Model der JMS-API zusammengefasst



Connection Factories

- Wird vom Entwickler benutzt um eine **Connection** zum Provider aufzubauen
- Die Konfigurationsparameter werden nicht vom Entwickler sondern von einem Administrator definiert

Connections

- Eine Connection kann nur über eine Connection Factory erstellt werden
- Eine Connection **repräsentiert die Verbindung** zum JMS-Provider

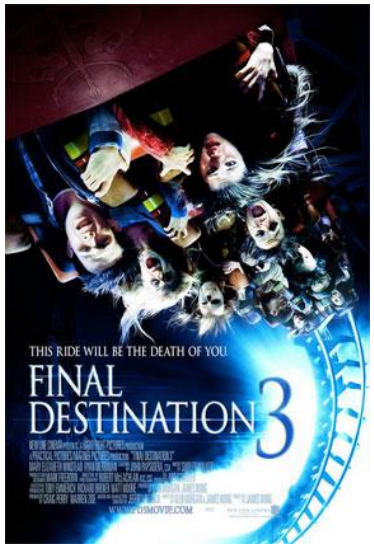
- Bevor das Programm beendet wird muss die **Connection wieder geschlossen** werden, sonst kann es passieren, dass der JMS-Provider keine neuen Connections mehr freigibt: *Connection.close()*

- ***Bevor Nachrichten empfangen werden können, muss die Methode start() aufgerufen werden***

- Mit *stop()* kann das Empfangen von Nachrichten ausgeschaltet werden
- Ein erneuter Aufruf von *start()* schaltet das Empfangen von Nachrichten erneut ein

Destinations

- Destination ist die Superklasse von
  - Queue (Point to Point)
  - Topic (Publish / Subscribe)



- Eine Destination wird dazu benötigt um das **Ziel** und die **Quelle** von Nachrichten festzulegen

- Destination wird genutzt, wenn das Programm sowohl für Queues als auch Topics laufen soll, ohne dass man neu compiliert
  - Welche spezifische Destination genutzt wird hängt dann beispielsweise von Aufrufeparametern oder Properties-Files ab

Sessions

- Sessions werden von Connection-Objekten erstellt
- Sessions stellen einen Kontext dar, in dem Messages gesendet und empfangen werden
- Eine Session kann u.a. folgende Bestandteile erstellen:
  - MessageProducer
  - MesageConsumer
  - Message

MessageProducers

- Ein Message Producer kann Messages **in** eine **Destination schreiben**
- Message Producer muss man sich **über ein Session-Objekt** erstellen lassen, dem man **eine Destination** übergibt

Vorgehensweisen beim Empfangen von Messages

Das Messaging ist zwar eigentlich immer asynchron, da im Grunde keine Zeitabhängigkeiten zwischen dem Senden und Empfangen der Nachrichten besteht, aber bei der Programmierung wird zwischen **zwei Möglichkeiten des Empfangens** von Nachrichten unterschieden:

1. **Synchron:**
   - Explizites Abholen der Nachrichten bei Destination durch Aufruf der *receive()*-Methode des Consumer-Objekts
   - Programmablauf wird geblockt
   - TimeOut möglich

2. **Asynchron:**
   - Beim Consumer wird ein *MessageListener* angemeldet
   - Der JMS-Provider liefert die Message durch den Aufruf der *onMessage()*-Methode des Listeners
   - Der Programmablauf wird nicht geblockt

MessageConsumers

- Ein Message Consumer kann Nachrichten **von** einer **Destination lesen**
- Message Consumer muss man sich **über ein Session-Objekt** erstellen lassen, dem man eine Destination übergibt
- Um aus einem Message Consumer Nachrichten zu lesen, kann die Methode *receive(timeout)* benutzt werden
- Man kann die Methode *close()* benutzen um den Message Consumer in einen inaktiven Zustand zu versetzen

MessageListeners

- Ein Message Listener kann ebenfalls **von** einer **Destination lesen**
- Ein Message Listener benötigt einen Consumer, bei dem er angemeldet werden muss
- Ein Message Listener arbeitet wie ein EventListener
  - Sobald eine Nachricht eintrifft, ruft der JMS-Provider eine über das Interface bekannte Methode auf, der ein Message-Objekt übergeben wird
- Da die Session die MessageListener-Aufrufe verwaltet, können keine concurrency –Probleme auftreten

Exceptions

- **JMSException**
  - IllegalStateException
  - InvalidClientIDException
  - InvalidDestinationException
  - InvalidSelectorException
  - JMSSecurityException
  - MessageEOFException
  - MessageFormatException
  - MessageNotReadableException
  - MessageNotWriteableException
  - ResourceAllocationException
  - TransactionInProgressException
  - TransactionRolledBackException

DER AUFBAU EINES SYNCHRONEN JMS-PROGRAMMS

1. Administered Objects in den eigenen Code injizieren

```
@Resource(mappedName = "jms/ConnectionFactory")
private static ConnectionFactory connectionFactory;

@Resource(mappedName = "jms/Queue")
private static Queue queue;

@Resource(mappedName = "jms/Topic")
private static Topic topic;

//Topic oder Queue als Destination verwenden
private static Destination dest = topic;
//private static Destination dest = queue;
```

2. Eine Connection und eine Session holen

```
try {
connection = connectionFactory.createConnection();
Session session = connection.createSession(false,
                        Session.AUTO_ACKNOWLEDGE);
} catch (JMSException e) {
    System.err.println("Exception occurred: " +
                        e.toString());
} finally {
    if (connection != null) {
        try {
            connection.close();
        } catch (JMSException e) {
        }
    }
}
```

3a. Für sendendes Programm: Message Producer erstellen

```
MessageProducer producer = session.createProducer(dest);
TextMessage message = session.createTextMessage();

message.setText("Hallo Dirk, Hallo Michi. Wann gibt es
                 wieder köstliche Leberkäswecken?");
producer.send(message);
```

3b. Für empfangendes Programm: Message Consumer erstellen

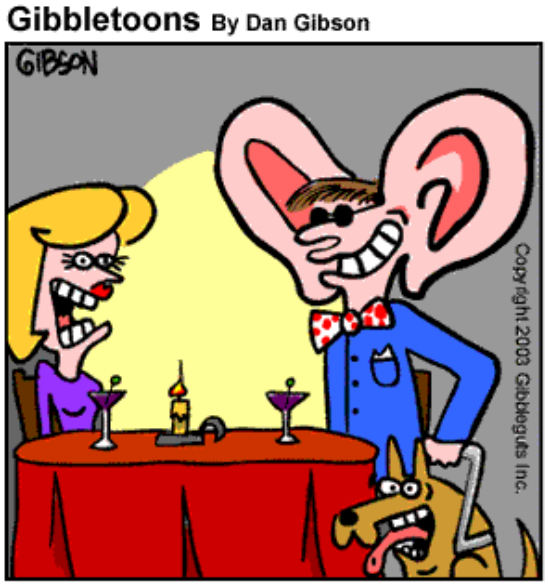
```
MessageConsumer consumer = session.createConsumer(dest);
TextMessage m = (TextMessage)consumer.receive(1);

System.out.println(m.getText());
```

DER AUFBAU EINES ASYNCHRONEN JMS-PROGRAMMS

1. MessageListener erstellen

Ein Message-Listener hat folgenden Aufbau:

```
MessageListener listener = new MessageListener() {
        public void onMessage(Message message)
        {
                ...
        }
}
```



Oh Tom...you're such a good listener.

2. MessageListener bei Consumer registrieren

Der MessageListener muss beim Consumer bekannt sein:

```
consumer.setMessageListener(listener);
```

3. Connection starten

Nicht vergessen:

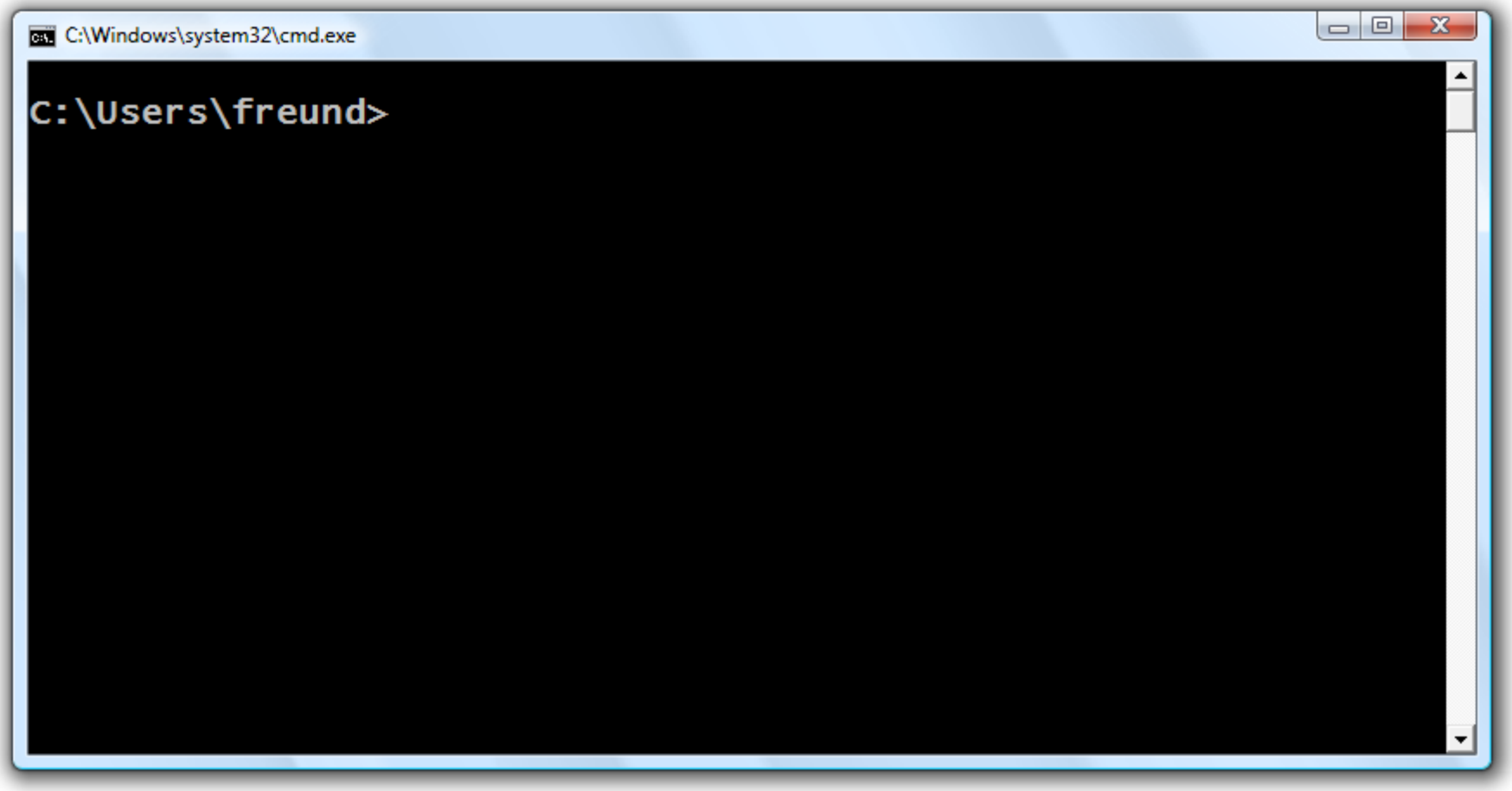
Damit der JMS-Provider auch wirklich die Nachrichten an die *onMessage()*-Methode weiterleitet , muss die *start()*-Methode der Connection aufgerufen werden, die wiederum implizit bewirkt, dass die Auslierferung gestartet wird!

Wird die *start()*-Methode aufgerufen bevor der Listener angemeldet ist können Nachrichten verlorengehen!

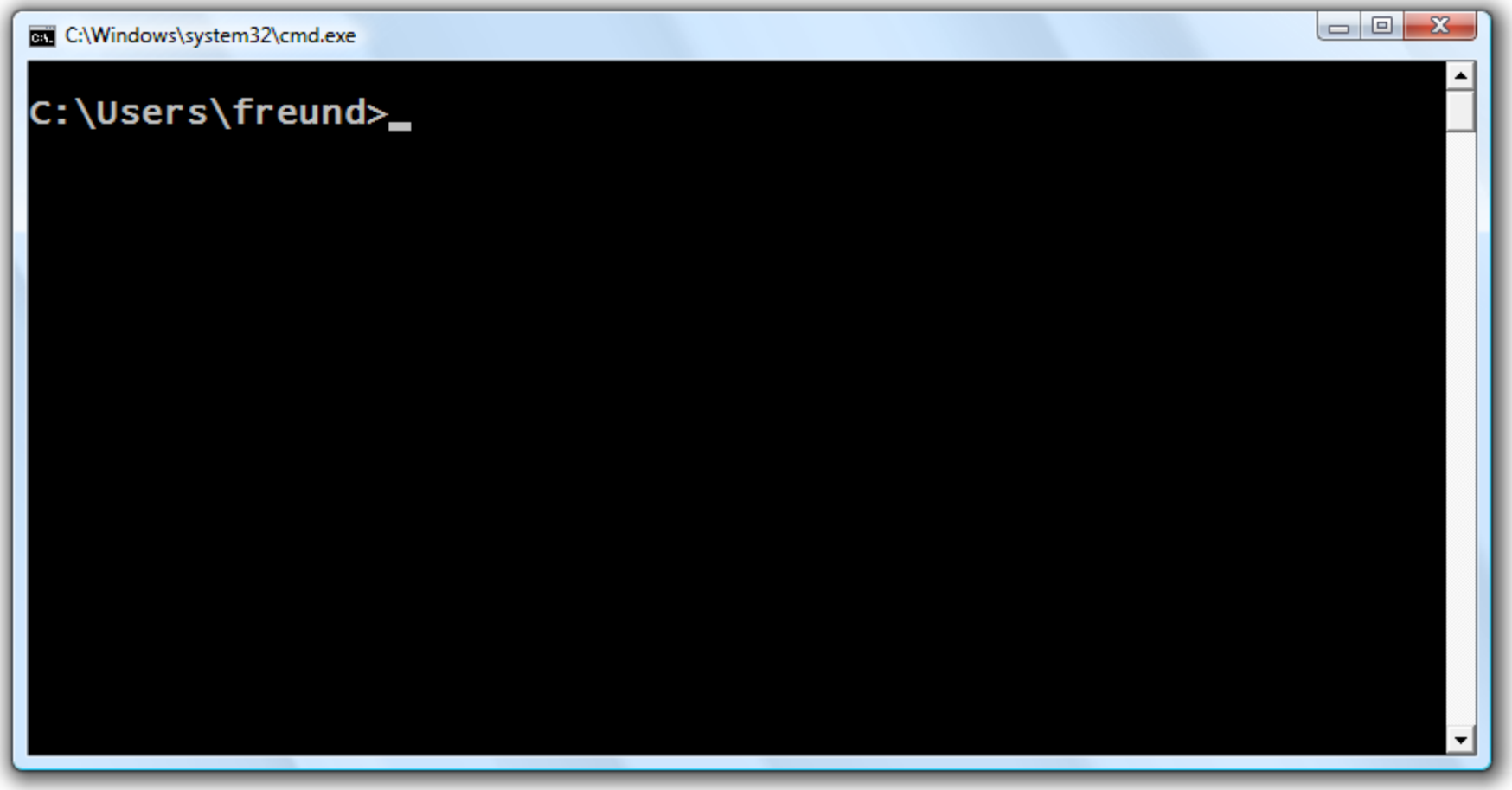
```
connection.start();
```

VOM SOURCECODE ZUM LAUFENDEN PROGRAMM

Schritt 1: Starten des Sun Java System Application Servers



Schritt 2: JMS Administered Objects erstellen



Schritt 3: Java-Code compilieren

1. Java-Code in **ByteCode** compilieren
2. Class-Dateien in Java-Archiv packen: **JAR-Datei**
3. **Manifest-Datei** (MANIFEST.MF), die einen Verweis auf die Main-Klasse enthält, muss dem JAR hinzugefügt werden

Hinweis:

*Dieser Vorgang kann durch ein **ANT-Build-Script** beschrieben werden und NetBeans führt dieses automatisch für den Programmierer aus.*

Schritt 4: Deploy durch Application Server

Auf der Basis der generierten JAR-Datei erstellt der Application Server beim **deploy-Vorgang** den finalen ByteCode und speichert das Ergebnis in seine Verzeichnisstruktur.

Dabei werden die **Administered Objects**, die als Annotations angegeben wurden, in Form von JMS-Provider spezifischem Bytecode, in den vorhandenen Bytecode injiziert.

Erst nach diesem Vorgang ist das Programm lauffähig .

Hinweis:

Dieser Vorgang wird ebenfalls automatisch von NetBeans veranlasst und ist ein Grund dafür, dass ein JMS-Programmstart länger dauert als das Starten von gewöhnlichen Java-Programmen.

VORFÜHRUNG EINES BEISPIELPROGRAMMES

QUELLEN

Quellen

Java EE 5 Tutorial von Sun:

http://java.sun.com/javaee/5/docs/tutorial/doc

JMS 1.1 SDK:

http://java.sun.com/products/jms/docs.html

Wir danken allen aufmerksamen Zuhörern!