Hochschule für angewandte Wissenschaften München
Fakultät für Informatik und Mathematik

Messaging mit Java (JMS)

Seminararbeit

Veranstaltung	Aktuelle Technologien zur Entwicklung verteilter Java-Anwendungen
Thema	T11 Messaging mit Java (JMS)
Verfasser	Marcel Binder (binder4@hm.edu)
Dozent	Michael Theis
Abgabetermin	19.Mai 2017

Inhaltsverzeichnis

1 Messaging als flexibelste Art der Integration

In keinem Unternehmen werden ausschließlich Anwendungen eines Hersteller eingesetzt, die zudem alle in der jeweils aktuellsten Technologie implementiert sind. Stattdessen existiert ein sich ständig veränderndes Netzwerk verschiedener Anwendungen, in dem jede eine bestimmte Aufgabe erfüllt. Meist wachsen und verändern sich diese Netzwerke mit der Zeit. Alte Anwendungen werden durch neue ganz oder teilweise abgelöst, Monolithen in Micro Services aufgeteilt oder nicht mehr benötigte Anwendungen entfernt. Dies führt zu einer Vielzahl eingesetzter Technologien und unterschiedlichen Ansätzen mit anderen Anwendungen zu kommunizieren. Die Notwendigkeit zur Kommunikation jedoch, wird immer bestehen, denn „[i]nteresting applications rarely live in isolation“ [3, S. xxix]. [3, S. 1 f., 39-56]

Durch Dateiaustausch kann eine hohe Entkopplung erreicht werden, da die Exporte unabhängig von der verwendeten Technologie und konkreten Implementierung sind. Lesen und Schreiben von Dateien ist in jeder Programmiersprache und unter jedem Betriebssystem möglich. Allerdings sind viele Abstimmungen wie Dateinamen, -pfade, Encoding und Datenformat notwendig. Für häufige Kommunikation ist der Dateiaustausch ferner zu ineffizient und nur verzögert möglich, da keine Benachrichtigung über neue Exporte stattfindet. [3, S.43 ff.]

Bei geteilten Datenbanken arbeiten alle Anwendungen stets auf den aktuellsten Daten. In vielen Technologien lassen sich Datenbanken inklusive Transaktionskontrolle einsetzen. Die Erstellung eines einheitliches Datenbankschema für ein großes Unternehmen ist jedoch nahezu unmöglich. Selbst wenn es existieren würde, wäre es undurchschaubar und für jede Änderung eines Datensatzes müsste die Implementierung aller anderen Anwendungen bekannt sein, um Fehler zu vermeiden. Durch die hohe Abhängigkeit lassen sich Erweiterungen nur schwer realisieren und mit steigender Anzahl an Zugriffen wird die zentrale Datenbasis der Flaschenhals des gesamten Unternehmensnetzwerks. [3, S.47 ff.]

Über Remote Procedure Calls (RPCs) wird anstelle von Daten die Funktionalität anderer Anwendungen genutzt. Dadurch ergibt sich eine höhere Entkopplung und Kontrollmöglichkeit als bei den bisher genannten Methoden. Die Daten sind in jeder Anwendung gekapselt. Technologien wie RMI schaffen allerdings nur eine scheinbare Entkopplung, da sie auf die konkrete Technologie beschränkt sind. Für Web Services müssen die Schnittstellen der anderen Anwendungen bekannt sein, weshalb ebenfalls Abhängigkeiten entstehen. [3, S.50 ff.]

Messaging ist eine Methode, lose Kopplung, zeitlich unabhängige (asynchrone) und zuverlässige Kommunikation zwischen Anwendungen zu ermöglichen. Sie eignet sich insbesondere zum häufigen Austausch kleinerer Datenmengen. Im Rahmen dieser Arbeit werden die Grundprinzipien von Messaging und die Möglichkeit diese in JavaEE Anwendungen zu nutzen vorgestellt. [3, S.53 ff.]

2 Grundlagen der Java Message Service API

2.1 Messaging Provider, Clients, Produzenten und Konsumenten

Messaging unterschiedet sich von anderen Integrationstechniken dahingehend, dass neben den kommunizierenden Anwendungen eine weitere Software beteiligt ist, der Messaging Provider. Anstelle der direkten Kommunikation zwischen Anwendungen (wie es bei Remote Procedure Calls oder Web Services üblich ist), erfolgt die Kommunikation indirekt über den Messaging Provider. [3, S. xxx ff.]

Um in einer Java Anwendung Messaging zu nutzen, wird die Java Message Service (JMS) API verwendet. Messaging Provider, die eine Anbindung über JMS unterstützen, implementieren vorgegebene Schnittstellen. Diese werden von Komponenten innerhalb einer Anwendung, sogenannten JMS Clients, zum Senden und Empfangen von Nachrichten genutzt. Dadurch ermöglicht JMS, verschiedene Messaging Provider über standardisierte Schnittstellen zu verwenden. Ebenso können Messaging Provider Schnittstellen mehrerer Technologien implementieren, um verschiedenste Anwendungen zu vernetzen. [4, S. 3]

Die Kommunikation über Messaging basiert auf zwei Prinzipien. Clients arbeiten nach dem *Send and Forget* Prinzip, welches asynchrone Kommunikation ermöglicht. Der sendende Client (Sender) übergibt die Nachricht an den Provider und kann anschließend mit anderen Aufgaben fortfahren, ohne auf die tatsächliche Zustellung der Nachricht zu warten. Würde ausschließlich dieses Prinzip angewendet werden, entstünde ein erheblicher Nachteil gegenüber synchroner Kommunikation: Die erfolgreiche Zustellung der Nachricht ist nicht sichergestellt. Daher nutzen Messaging Provider ein weiteres Prinzip namens *Store and Forward*. Wird eine Nachricht an den Messaging Provider zur Zustellung übergeben, wird diese zunächst auf dem Absendersystem gespeichert. Erst nach erfolgreicher Übertragung an und Speicherung der Nachricht auf dem Zielsystem wird diese vom Absendersystem gelöscht. [3, S. xxxii f., 122 ff.]

2.2 Nachrichtenziele und Kommunikationsarten

Durch die indirekte Kommunikation über den Messaging Provider entsteht eine hohe Entkopplung der Anwendungen, da sich Sender und Empfänger der Nachricht nicht kennen (bei Remote Procedure Calls "kennt"der Sender beispielsweise das `@Remote`-Interface des Empfängers, bei Web Services dessen URIs). Bei Messaging werden Nachrichten an Ziele (Destinations) adressiert, welche mit einem toten Briefkasten vergleichbar sind. Sender und Empfänger kennen nur das gleiche Ziel in dem Nachrichten abgelegt werden, nicht jedoch einander. Sender nutzen Produzenten als Schnittstelle zum

JMS Provider, um Nachrichten an ein Ziel zu senden. Konsumenten werden verwendet, um Nachrichten aus einem Ziel zu empfangen. Ein Ziel kann sowohl von mehreren Produzenten als auch Konsumenten genutzt werden. In JMS werden zwei Arten von Zielen unterschieden: Queues und Topics. Jede Zielart impliziert eine bestimmte Art der Kommunikation, deren Unterschiede im folgenden näher betrachtet werden. [4, S. 4 f., 10]

2.2.1 Queues für Point-to-Point Kommunikation

Wird eine Nachricht an eine Warteschlange (`Queue`) gesendet, verweilt sie darin bis zu ihrem Abruf. Es ist sichergestellt, dass jede Nachricht von genau einem Konsumenten abgerufen wird. Die Kommunikation über eine Queue erfolgt also stets von einem Produzenten zu einem Konsumenten und wird daher als Point-to-Point Kommunikation bezeichnet. Auf Grund ihres Speicherverhaltens sind Queues robust gegen den Ausfall von Konsumenten. Auch Nachrichten, die zu einem Zeitpunkt gesendet werden, zu dem kein Konsument aktiv ist, gehen nicht verloren. [4, S. 5]

Zudem werden sie zur Lastverteilung eingesetzt. Sollen viele Nachrichten parallel verarbeitet werden, können mehrere Threads mit Konsumenten für die Warteschlange erstellt werden. Da jede Nachricht nur von einem Konsumenten abgerufen wird, verteilt sich die Last gleichmäßig auf alle Konsumenten.

Anders als bei Remote Procedure Calls, die sofort abgearbeitet werden müssen, können Nachrichten in der Queue aufgestaut und in einer vom Empfänger festgelegten Rate abgearbeitet werden. Dadurch lässt sich eine dynamische Regelung der Auslastung beim Empfänger erreichen. [3, S. xxxiv]

2.2.2 Topics für Publish-Subscribe Kommunikation

Soll eine Nachrichten von mehreren Konsumenten empfangen werden können, wird über ein `Topic` kommuniziert. Interessierte Empfänger erstellen ein Abonnement (Subscription) für das Topic. Eingehende Nachrichten werden dann an alle zu diesem Zeitpunkt aktiven Abonnements als Kopie zugestellt und anschließend aus dem Topic gelöscht. Über einen Konsumenten lassen sich Nachrichten für ein Abonnement abrufen. Die Kommunikation wird als Publish-Subscribe bezeichnet und erfolgt von einem Produzenten zu beliebig vielen (auch keinem) Konsumenten. [4, S. 5 f.]

Anders als bei Queues ist es nicht möglich, eine Nachricht über ein Abonnement zu konsumieren, welches erst nach dem Senden erstellt wurde. Über JMS können Abonnements nur in Verbindung mit einem Konsumenten erstellt werden. Darüber hinaus sind Abonnements standardmäßig nicht dauerhaft (non durable) und bleiben über den Zeitraum hinweg bestehen, in dem auch der zugehörige Konsument existiert. [4, S. 5 f., 12 f.]

Bei Erstellung eines Konsumenten kann man jedoch festgelegen, dass ein dauerhaftes Abonnement (durable subscription) erstellt werden soll. Wird

der Konsument gelöscht, bleibt das Abonnement bestehen und sammelt Nachrichten an. Ein später neu erstellter Konsument kann über einen Namen das existierende Abonnement weiternutzen. [4, S. 13 ff.]

Durch die Vergabe eines Namens für ein Abonnement ist es ferner möglich, dass sich Konsumenten an einem bestehenden Abonnement beteiligen. Damit wird eine Lastverteilung auf mehrere Konsumenten erreicht. Jede an das Topic gesendete Nachricht wird also an alle zu diesem Zeitpunkt aktiven Abonnements und für jedes Abonnement jeweils an genau einen Konsumenten zugestellt. Dies wird in der JMS API als geteiltes Abonnement (shared subscription) bezeichnet. Auch die Kombination aus geteiltem und dauerhaftem Abonnement (shared durable subscription) ist möglich. [4, S. 13 ff.]

2.3 Nachrichten

Nachrichten bilden die atomare Übertragungseinheit. Anwendungen, die über JMS Daten senden oder empfangen, müssen diese in Nachrichten verpacken und an den JMS Provider übergeben. Bei den in Abschnitt 2.1 beschriebenen Prinzipien spielt die atomare Eigenschaft eine wichtige Rolle. Beim Senden von Nachrichten nach dem Send and Forget Prinzip wartet der Produzent, bis die Nachricht vom JMS Provider erfolgreich entgegengenommen wurde. Die Löschung aus dem Ziel erfolgt erst, wenn der Konsument die komplette Nachricht empfangen hat. Die Übergabe erfolgt also ganz oder gar nicht. Befindet sich das Topic auf einem anderen System, werden Nachrichten durch den JMS Provider über das Netzwerk übertragen. Durch das Anwenden des Store and Forward Prinzips wird auch hier sichergestellt, dass die Nachricht entweder ganz oder gar nicht erhalten wird. [3, S. xxxii f., 122 ff.]

2.3.1 Aufbau

JMS Nachrichten bestehen - ähnlich wie HTTP Nachrichten - aus drei Teilen: Header, Properties und Body. Diese Aufteilung ist an die interne Repräsentation von Nachrichten in den meisten JMS Providern angelehnt. [4, S. 16]

Der Header enthält Metadaten über die Nachricht, welche sowohl vom JMS Provider als auch von Anwendungen zur Identifikation, Routing und Zustellung verwendet werden. Die Metadaten bestehen aus Schlüssel-Wert-Paaren, wobei nur in JMS Spezifikation festgelegte Schlüssel zulässig sind. So enthält (`JMSDestination`) beispielsweise das Ziel und (`JMSMessageID`) eine eindeutige ID der Nachricht. Durch die Standardisierung der Header wird eine höchstmögliche Abstraktion von konkreten JMS Providern erreicht. Ein Großteil der Header wird beim Senden oder Zustellen der Nachricht automatisch gesetzt und kann von Anwendungen nicht überschrieben werden. Es besteht weiterhin die Möglichkeit leere Nachrichten zu senden, die ausschließlich aus Header und Properties bestehen. [4, S. 16 ff.]

Neben dem Header können anwendungs- und Provider-spezifische Metadaten in den sogenannten Properties festgelegt werden, welche ebenfalls aus Schlüssel-Wert Paaren bestehen. Die Schlüssel sind frei wählbar. Properties werden insbesondere zur Filterung von Nachrichten (siehe Abschnitt 3.7) verwendet. Des Weiteren werten JMS Provider bestimmte, Provider-spezifische Schlüssel aus, worüber zusätzlich Optionen konfiguriert werden können. Für Provider-unabhängige Lösungen soll auf diese Konfigurationsmöglichkeit weitestgehend verzichtet und stattdessen die Konfigurationsdatei des Providers angepasst werden. [4, S. 17]

Die Nutzdaten werden im Body der Nachricht übertragen. Kommunizierende Anwendungen müssen eine abgestimmte Syntax und Semantik für den Inhalt des Bodies verwenden. [4, S. 1, 16]

2.3.2 Intention

Mit dem Senden einer Nachricht wird in der Regel eine bestimmte Absicht/Intention verfolgt. Hohpe und Woolf unterteilen in [3] Nachrichten in drei Arten, die im Folgenden beschrieben werden.

2.3.2.1 Document Messages

Document Messages enthalten lediglich Daten. Der Absender weiß nicht, welche Aktionen das Senden der Daten auslöst. So kann eine Online Shop Anwendung eine Document Message für jede neue Bestellung in einer Queue ablegen. Durch den Einsatz dieser Nachrichtenform entsteht eine hohe Entkopplung, da der Absender keine Kenntnis über nachfolgende Verarbeitungsschritte haben muss. [3, S. 143, 147ff.]

2.3.2.2 Event Messages

Auch Event Messages legen nicht fest, welche Aktion ausgeführt werden soll. Sie dienen lediglich der Benachrichtigung anderer Anwendungen über aufgetretene Ereignisse. Meist wird diese Art der Nachricht an Topics gesendet, sodass sich mehrere interessierte Anwendungen für die Benachrichtigung registrieren können. Dies entspricht einem Observer Pattern, bei welchem die Benachrichtigung der Abonnenten asynchron erfolgt. Verwendet man Event Messages, die nur einen Bezeichner für das aufgetretene Ereignis beinhalten oder gar leer sind, spricht man vom Pull-Modell. Empfänger können bei Bedarf zusätzliche Informationen, wie etwa die Daten der Bestellung z.B. über eine weitere Nachricht abrufen. Im Gegenteil dazu werden beim Push-Modell die relevanten Informationen bereits in der Event Message mitgesendet, womit man eine Kombination aus Event- und Document Message erhält. [3, S. 143, 151ff.]

2.3.2.3 Command Messages

Command Messages werden genutzt, um Funktionalität anderer Anwendungen zu nutzen. In der Nachricht sind die auszuführende Aktion und die dazu nötigen Parameter enthalten. Um sicherzustellen, dass Command Messages genau einmal verarbeitet werden, erfolgt die Kommunikation in der Regel über eine Queue. Über Command Messages lassen sich Remote Procedure Call mittels Messaging durchführen, wodurch eine deutlich höhere Kopplung entsteht, als durch die Nutzung von Document- oder Event Messages. Allerdings lässt sich in vielen Fällen auf den Einsatz von Command Messages nicht verzichten. Häufig werden sie beispielsweise genutzt, um Daten von anderen Anwendungen anzufordern. Die Anfrage wird als Command Message, die Antwort als Document Message gesendet. Um dennoch eine möglichst hohe Entkopplung zu erreichen, sollen Command Messages wie Anfragen an ein bestimmtes Ziel formuliert werden. So können beispielsweise alle Anwendungen an das Topic Systemzustand angeschlossen sein. Um den Zustand abzufragen, wird eine Command Message an dieses Topic gesendet. Alle Systeme senden daraufhin den Systemstatus an das gleiche Topic zurück. In Abschnitt 3.8 wird vorgestellt, wie Anfragen und Antworten mit Messaging umgesetzt werden. [3, S. 143, 145f.]

3 Einsatz von JMS in Java EE Applikationen

Die aktuelle Version JMS 2.0 ist Bestandteil von Java EE 7. Abbildung 1 zeigt Bestandteile auf, die von einer in einen Applikationsserver deployten Anwendung benutzt werden, um Messaging über JMS zu betreiben. Auf die einzelnen Bestandteile und deren Zusammenwirken wird in den folgenden Abschnitten eingegangen.

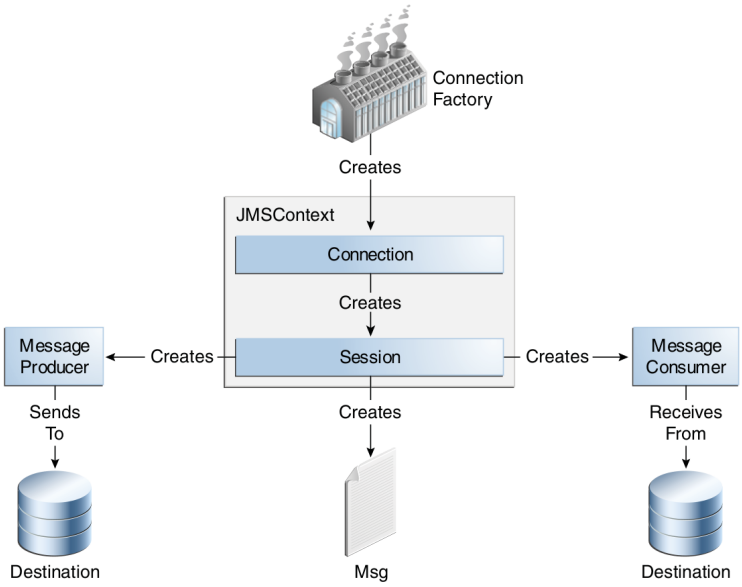

Abbildung 1: Bestandteile der JMS API
[aus 4, S.7 Abb. 45-5]

3.1 Administrative Objekte

Wie in Abbildung 1 dargestellt, existieren im Applikationsserver Ressourcen der Typen `ConnectionFactory`*, `Queue` und `Topic`, die von deployten Anwendungen während der Laufzeit über einen JNDI Lookup eingebunden werden. Die Konfiguration der Ressourcen findet im Applikationsserver statt und ist somit von der Anwendung entkoppelt, weshalb diese Ressourcen als administrative Objekte bezeichnet werden. [4, S. 6 ff.]

Eine `ConnectionFactory` wird verwendet, um eine Verbindung zum JMS Provider aufzubauen. `Queue` und `Topic` repräsentieren Ziele für Nachrichten. [4, S. 8 f.]

*Alle Klassen der JMS API befinden sich im Package `javax.jms`. Auf die Angabe des Packages wird im weiteren Verlauf der Arbeit verzichtet.

3.1.1 Konfiguration im Applikationsserver

Die Konfiguration der Ressourcen ist abhängig vom Applikationsserver und wird im Folgenden am Beispiel einer Payara 4.1 Domain erläutert. Eine Payara Domain lässt sich über das mitgelieferte Tool `asadmin` konfigurieren. Mit dem Subcommand `create-jms-resource` lassen sich administrative JMS Objekte als Ressourcen erstellen.

Folgender Befehl erstellt beispielsweise eine Queue mit JNDI Namen `jms/QueueA`. Über den Schalter `restype` wird dabei der Ressourcen-Typ in Form eines voll-qualifizierenden Klassennamens angegeben. [2, S. 9 ff.]

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

Die tatsächlichen, physikalischen Ziele (Queues und Topics) werden vom JMS Provider verwaltet und haben einen eindeutigen Bezeichner, welcher in diesem Fall `physical_queue_a` lautet. Existiert bei der ersten Verwendung der Ressource noch kein physikalisches Ziel mit diesem Namen, wird es erstellt. Legt man den physikalischen Name nicht explizit fest, wird er aus dem JNDI Namen abgeleitet. In Abschnitt 2.3.1 wurde beschrieben, dass kommunizierende Anwendungen das Ziel und Format der Nachrichten abstimmen müssen. Der physikalische Name und Typ wird abgestimmt. Es können also auch mehrere JMS Ressourcen mit unterschiedlichen JNDI Namen angelegt werden, die dasselbe physikalische Ziel verwenden. [2, S. 9 ff., 12]

Über den Subcommand `create-jmsdest` können physikalische Ziele unabhängig von Ziel Ressourcen angelegt werden. Darüber lassen sich beispielsweise die Kapazität und das Überlaufverhalten anpassen. Folgender Befehlt zeigt das separate Anlegen einer Queue mit Kapazität von 20 Nachrichten, wobei die älteste Nachricht im Falle eines Überlaufs gelöscht wird. [2, S. 12 f.]

```
asadmin create-jmsdest --desttype queue
   --property maxNumMsgs=20:limitBehavior=REMOVE_OLDEST
   physical_queue_a
```

Die Konfiguration von `Topics` erfolgt analog. Jede Payara Domain enthält bereits eine vorkonfigurierte Standard-`ConnectionFactory`, die für einfache Anwendungen ausreicht.

3.1.2 Einbindung via Resource Injection

JMS Ressourcen können innerhalb des Web- und EJB-Containers eingebunden werden. Da Messaging eine Integrationstechnik ist, sollten JMS Ressourcen nicht im Web-Container verwendet werden. Die Einbindung erfolgt über Resource Injection mit der `@Resource`-Annotation. Der Parameter `lookup` legt den JNDI Namen fest, der für den JNDI Lookup verwendet wird. Die

Konfiguration dieser JNDI Namen wird im vorherigen Abschnitt beschrieben. [4, S. 26 ff.]

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

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

@Resource
private ConnectionFactory connectionFactory;

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

Ist der Code unabhängig vom Typ des Ziels, kann anstatt `Queue` und `Topic` auch `Destination` verwendet werden. Für `ConnectionFactory` wird die Standard-ConnectionFactory eingebunden, wenn kein JNDI Name angegeben wird. [4, S. 26 ff.]

3.2 JMS Context

Anwendungen nutzen die `ConnectionFactory` in der Regel nur um einen `JMSContext` zu erstellen. Dazu ist jedoch ein try-with-resources-Block notwendig, um die Verbindung anschließend wieder ordnungsgemäß zu schließen. JMS stellt daher einen Producer bereit, sodass der JMSContext über CDI direkt eingebunden werden kann. Über die Qualifier-Annotation `@JMSConnectionFactory` lässt der JNDI Name einer ConnectionFactory angeben. Bei Weglassen der Annotation wird die Standard-ConnectionFactory verwendet. [4, S. 10, 28 f.]

```
@Inject
private JMSContext jmsContext;

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

Wie in Abbildung 1 zu sehen, besteht der `JMSContext` aus einer `Connection`, welche die logische Verbindung zum JMS Provider darstellt, und einer `Session`. Letztere bildet einen Transaktionskontext, der von einem einzigen Thread verwendet wird. Für jeden Aufruf einer `@Stateless` EJB wird eine neue Session verwendet. Besteht eine JTA Transaktion, wird diese an die Session gebunden. Transaktionen werden im Abschnitt 3.8 nochmals beschrieben. [4, S. 9 f.]

3.3 Nachrichten

3.3.1 Nachrichtentypen, Erstellung und Zugriff auf Body

JMS Nachrichten werden durch das Interface `Message` repräsentiert. Von `Message` abgeleitete Interfaces legen das Datenformat des Bodies fest. JMS bezeichnet dies als Nachrichtentypen. Die Implementierungen der Nachrichten Interfaces werden vom JMS Provider bereitgestellt, weshalb keine Instanziierung mit `new` möglich ist. Stattdessen werden Factory-Methoden des ebenfalls vom Provider implementierten `JMSContext` verwendet, um Nachrichten der verschiedenen Typen zu erstellen. Wie Instanzen des `JMSContexts` bezogen werden, ist im vorherigen Abschnitt beschrieben. [4, S. 16 ff.]

Tabelle 1 zeigt die unterschiedlichen Nachrichtentypen sowie die zugehörigen Factory-Methoden. Bei einigen Typen kann der Inhalt des bereits bei der Erstellung angegeben werden.

Nachrichtentyp	Factory-Methode in `JMSContext`
`Message`	`createMessage()`
`TextMessage`	`createTextMessage()` `createTextMessage(String text)`
`ObjectMessage`	`createObjectMessage()` `createObjectMessage(Serializable o)`
`MapMessage`	`createMapMessage()`
`StreamMessage`	`createStreamMessage()`
`BytesMessage`	`createBytesMessage()`

Tabelle 1: Factory-Methode für die verschiedenen Nachrichtentypen im `JMSContext`

3.3.1.1 `TextMessage`

Eine `TextMessage` enthält einen Java `String` im Body. Dieser Typ eignet sich vor allem für textbasierte Datenformate wie JSON und XML. Ein standardisiertes Datenformat lässt sich darüber hinaus technologie-unabhängig einsetzen. [4, S. 17] [1, S. 43 ff.]

3.3.1.2 `ObjectMessage`

Eine `ObjectMessage` enthält ein serialisiertes Java `Object` im Body. Anders als für JSON und XML werden Java Bordmittel zur Serialisierung genutzt. Es entsteht jedoch eine hohe Kopplung der Anwendungen, da zum Deserialisieren die gleichen Klassen benötigt werden. Ferner ist dieser Nachrichtentyp auf Java beschränkt. [4, S. 17] [1, S. 43 ff.]

3.3.1.3 MapMessage

Eine `MapMessage` enthält ein Mapping von Strings zu primitiven Java Datentypen oder `String`s. Dieser Nachrichtentyp eignet sich nicht zur Abbildung komplexer Objekte kann jedoch durchaus von anderen Technologien gelesen werden, sofern der Messaging Provider Maps aus anderen Technologien unterstützt. [4, S. 17] [1, S. 43 ff.]

3.3.1.4 StreamMessage

Eine `StreamMessage` enthält einen Strom von primitiven Java Datentypen oder Strings. Beim Schreiben und Lesen ist die Reihenfolge der Daten zu beachten. Auch dieser Typ kann von anderen Technologien gelesen werden. [4, S. 17] [1, S. 43 ff.]

3.3.1.5 BytesMessage

Eine `BytesMessage` enthält einen Strom von `bytes`. Dieser Nachrichtentyp sollte nur verwendet werden, um bestehende Nachrichtenformate nachzubilden. [4, S. 17] [1, S. 43 ff.]

3.3.1.6 Message

Eine `Message` enthält nur Header und Properties. Dieser Nachrichtentyp ist ideal für Event Messages, bei denen alle Informationen in Properties übermittelt werden. Empfänger erhalten Nachrichten stets im statischen Typ `Message`. Der JMS Provider stellt jedoch sicher, dass die ursprünglich verwendeten Nachrichtentypen erhalten bleiben. Zum Auslesen des Inhalts muss also zunächst auf den dynamischen Typ gecastet werden. Mit `instanceof` sollte vorher auf Kompatibilität mit dem gewünschten Nachrichtentyp geprüft werden. [4, S. 17 f.]

```
Message message = ...
if (message instanceof TextMessage) {
  TextMessage textMessage = (TextMessage) message;
  String json = textMessage.getText();
  ...
} else if (message instanceof ObjectMessage) {
  ObjectMessage objectMessage = (ObjectMessage) message;
  Order order = (Order) objectMessage.getObject();
  ...
}
```

Anstelle des Typecasts kann für alle Nachrichtentypen außer `StreamMessage` auch die `getBody`-Methode des `Message`-Interfaces benutzt werden, um auf den Inhalt der Nachricht zuzugreifen. Dieser wird der Typ des

Bodies übergeben (z.B. `String.class` für `TextMessage`). Bei `ObjectMessages` wird dadurch sogar einen weiteren Typecast gespart. [4, S. 18]

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

In Tabelle 2 ist beispielhaft für jeden Nachrichtentyp eine Schreib- sowie die zugehörige Leseoperation aufgeführt. Eine vollständige Liste findet sich in der Dokumentation des jeweiligen Nachrichtentyps.

Nachrichtentyp	Schreibzugriff	Lesezugriff
`TextMessage`	`setText(String)`	`getText()`
`ObjectMessage`	`setObject(Serializable)`	`getObject()`
`MapMessage`	`setInt(String, int)`	`getInt(String)`
`StreamMessage`	`writeInt(int)`	`readInt()`
`BytesMessage`	`writeUTF(String)`	`readUTF()`

Tabelle 2: Beispiele von Schreibe- und Leseoperationen für verschiedene Nachrichtentypen

3.3.2 Header und Properties

Die Header und Properties einer Nachricht können über entsprechende Getter- und Setter-Methoden des `Message`-Interface abgerufen und gesetzt werden. Beispielsweise lässt sich die vom Provider vergebene ID über `getJMSMessageID()` abfragen. [4, S. 16 f.]

```
message.getJMSMessageID();
```

Für Properties existieren Getter und Setter für alle primitive Java Datentypen und Strings. Die Methode `setObjectProperty` ist nur für Wrapper-Typen von primitiven Datentypen (z.B. `Integer`) und Strings zulässig.

```
message.setStringProperty("event", "ORDER_SHIPPED");
message.setLongProperty("orderId", 1001L);
message.setObjectProperty("tracingId", 3241);
```

3.4 Senden von Nachrichten

3.4.1 Produzenten

Das Senden einer Nachricht erfolgt über einen `JMSProducer`. Um eine Instanz zu erstellen, bietet der `JMSContext` eine weitere Factory-Methode `createProducer()` an. [4, S. 10]

Der `JSMProducer` ist nach dem Builder-Pattern aufgebaut: Jeder Methode gibt die Producer Instanz wieder zurück, wodurch eine Verkettung der Aufrufe möglich ist. Zu den wichtigsten Methoden zählen:

- `setProperty(String, int*)`
  setzt eine Property für alle über diesen Producer versendeten Nachrichten.

- `clearProperties()`
  löscht alle bisher auf der Producer Instanz gesetzten Properties.

- `send(Destination, Message)`
  sendet eine vorher erstellte Nachricht an ein Ziel. Beim Senden von Nachrichten ist die Art des Ziels (Queue oder Topic) nicht von Bedeutung. [4, S. 10]

- `send(Destination, String)`
  `send(Destination, Serializable)`
  `send(Destination, Map<String, Object>)`
  `send(Destination, byte[])`
  zum direkten Senden einer `Text`-, `Object`-, `Map`- bzw. `BytesMessage`. Nachrichten müssen also nicht separat über `JMSContext` erzeugt werden, wodurch leichter lesbarer Code entsteht. [4, S. 17 f.]

Listing 1† zeigt eine EJB, die einen Producer nutzt, um eine Document Message mit dem Versandetikett sowie eine Event Message zu senden. Für den Producer ist die Art des Ziels unerheblich und aus dem Codeausschnitt nicht erkennbar. Das Etikett wird im JSON Format als `TextMessage` übertragen, für das Event wird eine leere `Message` erstellt und alle relevanten Informationen als Properties gesetzt. Auch der Empfänger ist hieraus nicht ersichtlich.

*Überladungen für alle primitive Datentypen sowie `String`s vorhanden. Alternativ kann für Wrapper primitiver Typen (z.B. `Integer`) `setProperty(String, Object)` verwendet werden.

†Viele Methoden der JMS API können eine checked `JMSException` werfen. Um Codebeispiele möglichst übersichtlich zu halten, wird auf die Angabe dieser Exception verzichtet.

Listing 1: Einsatz eines `JMSProducers` in einer EJB

```
@Stateless
class CustomerNotifier {
  @Inject
  private JMSContext jmsContext;
  @Resource(lookup = "jms/ShippingDocuments")
  private Destination shippingDocuments;
  @Resource(lookup = "jms/OrderEvents")
  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);
  }
}
```

Der Thread des Aufrufers blockiert beim Aufruf von `send` bis die Nachricht erfolgreich an den JMS Provider übertragen wurde. Anschließend kehrt der Aufruf zurück. Die Zustellung erfolgt durch den JMS Provider im Hintergrund.

`JMSProducer` sind leichtgewichtige Objekte, die für jede Nachricht neu erstellt werden können. [4, S. 10]

3.4.2 Gültigkeit und Priorität von Nachrichten

Über den `JMSProducer` können das Zustellverhalten für gesendete Nachrichten beeinflusst werden.

- `setTimeToLive(long)`
  Für Nachrichten, die Daten begrenzter zeitlicher Gültigkeit enthalten (beispielsweise Aktienkurse), kann eine Lebensdauer festgelegt werden. Als Parameter gibt man die Lebensdauer in Millisekunden relativ zum Zeitpunkt des `send`-Aufrufs an. Ist die Nachricht uneingeschränkt gültig, kann der Wert 0 gesetzt werden. Der absolute Ablaufzeitpunkt wird vom JMS Provider berechnet und im Header `JSMExpiration` als Millisekunden seit Beginn der Unix Epoche gespeichert. Nach Erreichen dieses Zeitpunkts wird die Nachricht vom JMS Provider gelöscht. Der Header kann nach dem `send`-Aufruf von der gesendeten Nachricht über `getJMSExpiration()` abgerufen werden. [4, S. 21 ff.] [3, S. 176 ff.]

- `setDeliveryDelay(long)`
  Für Nachrichten, die erst nach einer bestimmten Zeit Gültigkeit erhalten, kann die Zustellung verzögert werden. Als Parameter gibt man die Zustellverzögerung in Millisekunden relativ zum Zeitpunkt des `send`-Aufrufs an. Ist die Nachricht ab sofort gültig, kann der Wert 0 gesetzt werden. Der absolute Zeitpunkt, ab dem die Nachricht frühestens von Konsumenten empfangen werden kann, wird vom JMS Provider berechnet und im Header `JMSDeliveryTime` als Millisekunden seit Beginn der Unix Epoche gespeichert. Dieser Header kann nach dem `send`-Aufruf von der gesendeten Nachricht über `getJMSDeliveryTime()` abgerufen werden. [4, S. 21 ff.]

- `setPriority(int)`
  Ferner kann die Priorität einer Nachricht geändert werden. Laut JMS Spezifikation gelten Prioritäten zwischen 0 und 4 als normal, die Werte 5-9 als höhere Priorität. [4, S. 21 ff.]

In Listing 2 wird für die erste Nachricht eine höhere Priorität und zeitlich begrenzte Gültigkeit von einer Minute festgelegt, die zweite Nachricht hingegen wird erst 30 Sekunden nach dem Senden gültig.

Listing 2: Änderung der Gültigkeit und Priorität von Nachrichten über den `JMSProducer`

```
jmsContext.createProducer()
  .setTimeToLive(TimeUnit.MINUTES.toMillis(1L))
  .setPriority(5)
  .send(destination, message1)
  .setTimeToLive(0L)
  .setDeliveryDelay(TimeUnit.SECONDS.toMillis(30L))
  .send(destination, message2);

LOGGER.debug("M1 valid until {}", new Date(message1.
    getJMSExpiration()));
LOGGER.debug("M2 valid from  {}", new Date(message2.
    getJMSDeliveryTime()));
```

Läuft eine Nachricht ab, wird sie bei entsprechender Konfiguration des JMS Providers in eine sog. Dead Message Queue (DMQ) verschoben. Auch Nachrichten, die aus anderen Gründen unzustellbar sind, da Konsumenten z.B. mehrmals den Empfang mit einer Exception abbrechen landen ebenfalls in der DMQ. Von dort können Nachrichten von einer Diagnose-Anwendung abgefragt und geloggt werden, um Fehler zu protokollieren. [4, S. 22] [3, S.176 ff.]

3.5 Synchrones Empfangen von Nachrichten

Das Gegenstück zum `JSMProducer`, der `JMSConsumer`, wird zum synchronen Empfangen von Nachrichten verwendet und ebenfalls über Factory-Methoden des `JMSContexts` erstellt. Nach dem Senden von Nachrichten muss der Consumer geschlossen werden. Da das `AutoCloseable`-Interface implementiert ist, kann man einen try-with-resources-Block verwenden. Das `JMSConsumer`-Interface bietet mehrere Methoden zum Abrufen von Nachrichten. Wie in Abschnitt 3.3.1.6 bereits erwähnt, liefern diese Methoden die empfangene Nachricht stets im statischen Typ `Message` zurück. [4, S. 12 ff.]

- `receive()`
  `receive(long timeout)`
  Blockiert den aufrufenden Thread bis eine neue Nachricht verfügbar oder ein optionales Timeout abgelaufen ist. Ein Timeout von 0 bedeutet dabei kein Timeout. Falls bis Ablauf des Timeouts keine Nachricht verfügbar war, liefert der Aufruf `null` zurück. [4, S. 12 ff.]

- `receiveNoWait()`
  Kehrt sofort mit einer neuen Nachricht zurück, oder `null` falls keine Nachricht verfügbar war.

Ist nur der Body einer neuen Nachricht von Interesse, so können die `receiveBody`- bzw. `receiveBodyNoWait`-Methoden verwendet werden. Sie funktionieren analog zu den `receive`-Methoden und erwarten als weiteren Parameter den Typ des Bodies (für `TextMessages` also `String.class`).

3.5.1 Konsumenten für Queues

Als Beispiel für die folgenden Erklärungen soll ein Online Shop System dienen, welches zunächst aus zwei Anwendungen besteht: Dem Shop, in dem Bestellungen aufgegeben werden, und der Versand-Anwendung, die für neue Bestellungen Versandetiketten bei Paketdienstleistern anfordert.

Die Anwendungen sind über eine Queue integriert, an die der Shop neue Bestellungen sendet. Beim Klicken auf eine Oberfläche oder periodisch über einen zeitgesteuerten Job ruft die Versand-Anwendung neue Bestellungen aus dieser Queue ab.

Zum Abrufen einer neuen Bestellung wird die EJB aus Listing 3 verwendet, in welcher zunächst über die `createConsumer`-Methode des `JSMContexts` ein neuer `JMSConsumer` erzeugt wird. Der Consumer ist an ein Ziel gebunden, welches beim Erstellen festgelegt wird. In diesem Fall die Queue für neue Bestellungen. Anschließend wird die `receiveBody`-Methode verwendet, um nur den Body einer neuen Nachricht abzufragen. Ein Timeout von 2s verhindert, dass der aufrufende Thread unnötig lange blockiert ist. Im Fall eines Timeouts liefert die Methode ein leeres `Optional` zurück. Wird die

zugehörige Geschäftslogik zeitgesteuert ausgeführt, könnte einfach auf den nächsten Aufruf gewartet werden.

Listing 3: Synchroner Nachrichtenempfang aus einer `Queue`

```
@Stateless
public class OrderReceiver {
  @Inject
  private JMSContext jmsContext;
  @Resource(lookup = "jms/NewOrders")
  private Destination newOrders;

  public Optional<Order> getNewOrder() {
    try (JMSConsumer consumer = jmsContext
      .createConsumer(newOrders)) {
      String json = consumer.receiveBody(String.class, 2000
         L);
      return json == null
        ? Optional.empty()
        : Optional.of(toOrder(json));
    }
  }
}
```

Nachrichten die zwischen zwei Aufrufen von `getNewOrder` in der Queue eintreffen, werden aufgestaut. Dadurch ist das Online Shop System ausfallsicher. Fällt der Shop aus, treffen keine weiteren Bestellungen ein. Die Versand-Anwendung arbeitet dennoch weiter und verarbeitet beispielsweise Nachrichten einer Lager-Anwendung, die mitteilt, dass Bestellungen versandfertig sind. Fällt die Versand-Anwendung aus, können trotzdem Bestellungen getätigt werden. Angestaute Nachrichten arbeitet die Versand-Anwendung ab, sobald sie wieder verfügbar ist.

Lastverteilung wird erreicht, wenn der zeitgesteuerte Job in der Versand-Anwendung mehrere Worker-Threads verwendet. Jede Nachricht wird nur vom Konsumenten eines Threads abgerufen. In eine Cluster könnten mehrere Instanzen der Versand-Anwendung betrieben werden. Damit ließe sich zudem die Ausfallsicherheit weiter erhöhen.

Bei Verwendung mehrerer Instanzen und zentral betriebenen JMS Provider, erfolgt der Nachrichtenabruf über Netzwerk. Dabei fiel auf, dass `receiveNoWait` bzw. `receiveBodyNoWait` nicht wie erwartet Nachrichten die sich bereits in der Queue befinden abruft und nur im Falle einer leeren Queue nicht wartet. Stattdessen kehrt der Aufruf auch bei gefüllter Queue sofort mit `null` zurück. Auf Grund des Netzwerks muss also ein Timeout festgelegt werden. In dem beschriebenen Fall handelte es sich um ein Netzwerk zwischen Docker Containern und ein Timeout von 100 ms reichte aus.

3.5.2 Konsumenten für Topics

3.5.2.1 Nicht-dauerhafte Abonnements

Das Beispiel des Online Shops wird nun um eine Rechnungs-Anwendung ergänzt, die für neue Bestellungen Rechnungen erstellt. Die Rechnungserstellung kann nebenläufig zur Anforderung des Versandetiketts erfolgen. Würde die Rechnungs-Anwendung ebenfalls Nachrichten aus der Queue neuer Bestellungen abrufen, fände eine Lastverteilung zwischen Versand- und Rechnungs-Anwendung statt. Das heißt Bestellungen würden entweder in Rechnung gestellt oder zum Versand vorbereitet werden.

Sowohl Versand- als auch Rechnungs-Anwendung sind an den gleichen Nachrichten interessiert, weshalb die Queue durch ein Topic ersetzt wird, welches von beiden Anwendungen abonniert wird. Dazu ist nur der Typ der JMS Ressource in der Konfiguration des Applikationsservers zu ändern (siehe Abschnitt 3.1.1).

Beide Anwendungen verwenden einen zeitgesteuerten Job zum Ausführen der Geschäftslogik, welche die EJB aus 3 nutzt um eine neue Bestellung aus dem Topic zu empfangen. Abschnitt 2.2.2 beschreibt, dass an ein Topic gesendete Nachrichten nur an Abonnements zugestellt werden, die zum Zeitpunkt des Sendens existieren. Standardmäßig erstellt die `createConsumer`-Methode für Topics ein nicht dauerhaftes Abonnement mit einem Konsumenten (non-shared non-durable Subscription). Das Abonnement wird also mit dem Erstellen des Konsumenten angelegt und beim Verlassen des try-with-resources-Block gelöscht. Daher gehen Nachrichten, die zwischen Aufrufen der `getNextOrder`-Methode gesendet werden, verloren. Für das Online Shop Beispiel wäre dies nicht praktikabel. [4, S. 12 ff.]

3.5.2.2 Dauerhafte Abonnements mit einem Konsumenten

Um einen Konsumenten mit dauerhaften Abonnement zu erstellen, wird die `createDurableConsumer`-Methode verwendet, welche neben dem Ziel als `Topic` einen Abonnement-Namen erwartet. Für Versand- und Rechnungs-Anwendung werden unterschiedliche Namen vergeben. Listing 4 zeigt die EJB in der Rechnungs-Anwendung. [4, S. 13 ff.]

Listing 4: Synchroner Nachrichtenempfang aus einem `Topic` über ein dauerhaftes Abonnement

```
@Stateless
public class OrderReceiver {
  @Inject
  private JMSContext jmsContext;
  @Resource(lookup = "jms/NewOrders")
  private Topic newOrders;

  public Optional<Order> getNewOrder() {
```

```
    try (JMSConsumer consumer = jmsContext
      .createDurableConsumer(newOrders, "createBill")) {
      ...
    }
  }
}
```

Nun wird eine Kopie jeder Nachricht an die Abonnements der Versand- und Rechnungs-Anwendung zugestellt. Das Abonnement bleibt über den try-with-resources-Block hinweg und wird über den Namen wiedergefunden. Es bleibt bestehen bis es über die `unsubscribe`-Methode des `JMSContexts` gelöscht wird. Abonnements können jedoch nur zusammen mit einem Konsumenten erstellt werden. [4, S. 12 ff.]

3.5.2.3 Nicht-dauerhafte Abonnements mit mehreren Konsumenten

Mittels `createDurableConsumer` erstellte Abonnements, können nur von einem Konsumenten gleichzeitig verwendet werden. Mehrere Worker-Threads oder Instanzen der Versand-Anwendung müssten also unterschiedliche Abonnement-Namen verwenden. Damit lässt sich jedoch keine Lastverteilung erreichen, da nun Kopien der Nachrichten an alle Threads zugestellt werden, mit der Folge, dass für eine Bestellung mehrere Versandetiketten angefordert werden. [4, S. 12 ff.]

Mit `createSharedConsumer` lassen sich hingegen Abonnements unter Konsumenten teilen. Existiert das Abonnement mit dem angegebenen Namen noch nicht, wird es erstellt. Andere Konsumenten können über die Angabe des gleichen Abonnement-Namens daran teilnehmen. Die Nachrichten, welche das Abonnement als Kopie erhält, können dann von einem Konsumenten abgerufen werden. Threads innerhalb einer Anwendung oder mehrere Instanzen dieser nutzen den gleichen Abonnement-Namen um Lastverteilung zu realisieren [4, S. 13 ff.].

Listing 5: Synchroner Nachrichtenempfang aus einem `Topic` über ein geteiltes Abonnement

```
@Stateless
public class OrderReceiver {
  @Inject
  private JMSContext jmsContext;
  @Resource(lookup = "jms/NewOrders")
  private Topic newOrders;

  public Optional<Order> getNewOrder() {
    try (JMSConsumer consumer = jmsContext
      .createSharedConsumer(newOrders, "createBill")) {
      ...
```

```
    }
  }
}
```

Das Abonnement bleibt bestehen, solange mindestens ein Konsument existiert. In Listing 5 also bis alle Threads den try-with-resources-Block verlassen. Selbstverständlich lässt sich das Abonnement zu einem späterem Zeitpunkt wieder anlegen. Nachrichten, die inzwischen gesendet werden, gehen jedoch verloren. [4, S. 13]

3.5.2.4 Dauerhafte Abonnements mit mehreren Konsumenten

Auch die Kombination der geteilten und dauerhaften Abonnements ist möglich, indem Konsumenten über `createSharedDurableConsumer` erstellt werden. Damit lassen sich Lastverteilung und Ausfallsicherheit vereinen, weshalb diese Art der Abonnements für viele Anwendungsfälle am geeignetsten ist. [4, S. 13 ff.]

Listing 6: Synchroner Nachrichtenempfang aus einem `Topic` über ein geteiltes und dauerhaftes Abonnement

```
@Stateless
public class OrderReceiver {
  @Inject
  private JMSContext jmsContext;
  @Resource(lookup = "jms/NewOrders")
  private Topic newOrders;

  public Optional<Order> getNewOrder() {
    try (JMSConsumer consumer = jmsContext
      .createSharedDurableConsumer(newOrders, "createBill")
        ) {
      ...
    }
  }
}
```

Dennoch sind dauerhafte Abonnements bei Topics nicht so robust wie Queues, da sie sich nicht im Vorfeld über den Applikationsserver anlegen lassen. Damit ein Abonnement existiert, muss einmal ein Konsument erstellt werden.

3.6 Asynchrones Empfangen von Nachrichten

Die Nachrichtenabfrage über `receive`- bzw. `receiveBody`, wie sie im vorherigen Abschnitt beschrieben ist, erfolgt synchron. Der aufrufende Thread steuert den Zeitpunkt, zu dem die Abfrage erfolgt und blockiert die Ausführung bis neue Nachrichten verfügbar sind oder ein Timeout eintritt. Dadurch

kann die Anwendung steuern, mit welcher Rate Nachrichten abgerufen werden und somit ihre Auslastung regeln. Wird der Online Shop beispielsweise überwiegend von Nutzern einer Zeitzone verwendet, wäre die Auslastung nachts deutlich geringer und erreicht Spitzen am Feierabend. Allerdings müsste zur Regelung der Abfragerate das Intervall des zeitgesteuerten Jobs zur Laufzeit anpassbar sein und eine Messung der Auslastung sowie des Nachrichtenaufkommens stattfinden. Gegebenenfalls müsste dieser Mechanismus über mehrere Instanzen hinweg regeln und steuern. [3, S. xxxiv]

Über sogenannte Message Driven Beans (MDBs) lassen sich Nachrichten asynchron empfangen. MDBs sind Enterprise Java Beans, welche mit `@MessageDriven` annotiert sind und das Interface `MessageListener` implementieren. Listing 7 zeigt eine MDB, die beispielsweise in der Rechnungs-Anwendung eingesetzt werden könnte. [4, S. 29 f.]

Listing 7: Asynchroner Nachrichtenempfang aus einem `Topic` über eine Message Driven Bean (MDB)

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(
    propertyName = "destinationLookup",
    propertyValue = "jms/NewOrders"
  ),
  @ActivationConfigProperty(
    propertyName = "destinationType",
    propertyValue = "javax.jms.Topic"
  )
})
public class NewOrderListener implements MessageListener {
  @Inject
  private BillBoundary billBoundary;

  @Override
  public void onMessage(Message message) {
    try {
      String json = message.getBody(String.class);
      Order order = fromJson(json);
      billBoundary.createBillForOrder(order);
    } catch (JMSException e) {
      LOGGER.warn(e);
    }
  }
}
```

Die Konfiguration der MDBs erfolgt über weitere `@ActivationConfigProperty`-Annotationen. `destinationLookup` legt den JNDI-Namen, `destinationType` den Typ des Ziels fest. Für neue Nachrichten wird die `onMessage`-Methode aus dem `MessageListener`-Interface aufgerufen. In obigen Beispiel wird die Bestellung aus der eintreffenden Nachricht ausgelesen

und an die Geschäftslogik übergeben. [4, S. 30 ff.]

Wird ein Topic als Ziel angegeben, werden für MDBs standardmäßig geteilte Abonnements erstellt, da die Anzahl der Instanzen vom EJB-Container verwaltet wird. Der Name des Abonnements wird vom Container vergeben und kann nicht geändert werden. In obigem Beispiel wird ein nicht-dauerhaftes Abonnement erstellt. Somit gehen Nachrichten, die während eines Anwendungsausfalls gesendet werden, verloren. Um ein dauerhaftes Abonnement zu erstellen werden folgende Annotationen ergänzt. Die Festlegung eines Abonnement-Name ist hierzu erforderlich. [4, S. 30 ff.]

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

Für Nachrichten, die nun während eines Ausfalls an das Ziel gesendet werden, wird `onMessage` aufgerufen nachdem die Anwendung erneut gestartet wird.

3.7 Filterung von Nachrichten

Beim Nachrichtenempfang können Nachricht auf Grundlage ihrer Properties und Header gefiltert werden. Verschiedene Anwendungen eines Online Shop Systems könnten beispielsweise Event Messages über den Fortschritt einer Bestellung gesammelt an ein Ziel senden. Die Shop-Anwendung ist allerdings nur an den Events *Rechnung erstellt* und *Bestellung versandt* interessiert, um dem Nutzer entsprechende Hinweise anzuzeigen. Es könnte also festgelegt werden, dass alle an dieses Ziel gesendeten Nachrichten eine Property `event` haben. Für synchronen Nachrichtenempfang wird der Filter beim Erstellen des Konsumenten angegeben und verwendet den Syntax bedingter Ausdrücken aus dem SQL 92-Standard, welche üblicherweise im WHERE-Teil einer SQL-Abfrage verwendet werden. [4, S. 12]

```
jmsContext.createConsumer(eventTopic,
  "event='BILL_AVAILABLE' OR event='ORDER_SHIPPED'");
```

Für die anderen `createConsumer`-Methoden existiert ebenfalls eine Überladung, bei der ein Filter angegeben werden kann. Bei MDBs werden Filter über eine `@ActivationConfigProperty`-Annotation konfiguriert: [4, S. 30 f.]

```
MessageDriven(activationConfig = {
  ...
  ActivationConfigProperty(
    propertyName = "messageSelector",
    propertyValue = "event='BILL_AVAILABLE' " +
      "OR event='ORDER_SHIPPED'"
  )
})
```

Bei Queues werden Nachrichten, die nicht den Filter-Kriterien entsprechen nicht aus der Queue entfernt. Bei Topics hingegen gilt der Filter auf ein Abonnement. Unpassende Nachrichten werden dem Abonnement also nicht hinzugefügt und stehen somit auch anderen Konsumenten nicht zur Verfügung. [4, S. 12]

3.8 Anfrage-Antwort und Transaktionen

Obwohl Messaging zur indirekten Kommunikation verwendet wird, werden in vielen Fällen Nachrichten zum Anfragen von Daten einer anderen Anwendung oder zur Auslösung einer Funktionalität genutzt. Als Ziel einer Anfrage-Nachricht wären hierfür sowohl Queues als auch Topics denkbar, je nach dem, ob die Anfrage von einer oder mehreren Anwendungen beantwortet werden soll. Beispielsweise könnte eine Monitoring-Anwendung zur Ermittlung des Systemstatus eine Anfrage an ein Topic senden. Alle Anwendungen des Systemverbunds sind verpflichtet bei einer entsprechenden Anfrage ihren Systemstatus zu melden. [3, S. 154 ff.]

Dazu muss einerseits bekannt sein, wohin die Anwendungen Antworten senden sollen. Andererseits muss eine Möglichkeit in Monitoring-Anwendung bestehen, Anfrage und Antwort zuzuordnen. JMS ermöglicht es, temporäre Ziele mittels der Methoden `createTemporaryQueue()`/`-Topic()` des `JMSContexts` zu erstellen. In der EJB aus Listing 8 wird ein temporäres Ziel definiert, an welches Anfragen gesendet werden sollen. [3, S. 154 ff.] [4, S. 23 f.]

Da dieses Ziel den anderen Anwendungen nicht bekannt ist, wird es über `setJMSReplyTo` in der Nachricht mitgesendet. Anschließend wird über einen Konsumenten auf Antworten gewartet und die Statusmeldungen gesammelt. Wird 30 Sekunden lang keine Nachricht empfangen, wird das temporäre Ziel mit `delete()` gelöscht. Nach 30 Sekunden wird auch die ursprüngliche Anfrage vom JMS Provider verworfen, da sie nun irrelevant ist. [4, S. 23 f.] [3, S. 159 ff.]

Listing 8: Umsetzung eines Anfrage-Antwort-Mechanismus über ein temporäres Ziel

```
@Stateless
public class SystemStatusRequestor {
  private static final long TIMEOUT = TimeUnit.SECONDS.
      toMillis(30L);
  @Inject
  private JMSContext jmsContext;
  @Resource(lookup = "jms/SystemStatus")
  private Destination requestDestination;

  @Transactional(NOT_SUPPORTED)
  public List<SystemStatus> getOverallStatus() {
    TemporaryQueue replyDestination =
        jmsContext.createTemporaryQueue();

    Message request = jmsContext.createMessage();
    jmsContext.createProducer()
      .setJMSReplyTo(replyDestination)
      .setTimeToLive(TIMEOUT)
      .send(requestDestination, request);

    List<SystemStatus> overallStatus = new LinkedList<>();
    try (JMSConsumer consumer = jmsContext.createConsumer(
        replyDestination)) {
      SystemStatus systemStatus;
      do {
        systemStatus = consumer.receiveBody(
          SystemStatus.class, TIMEOUT);
        if (systemStatus != null) {
          overallStatus.add(systemStatus);
        }
      } while (systemStatus != null);
    }
    replyDestination.delete();
    return overallStatus;
  }
}
```

Anwendungen setzen beispielsweise eine MDB wie in Listing 9 ein, um auf die Anfragen zu reagieren. Sie fragen den Systemstatus über die Geschäftslogik der Anwendung ab und senden ihn an das temporäre Ziel zurück, welches über die `getJMSReplyTo`-Methode der empfangenen Nachricht ausgelesen werden kann. Der JMS Provider liefert hierüber eine passende `Destination`-Instanz zurück. Da in diesem Beispiel nur eine Anfrage gestellt wurde, ist eine eindeutige Zuordnung der Antworten stets möglich. Der Name der beantwortenden Anwendung würde im SystemStatus Objekt enthalten sein. Dennoch wird hier noch die `JMSMessageID` der Anfrage im Header `CorrelationID`

der Antwort zurückgesandt, über welche eine Zuordnung von Anfrage und Antwort in der Monitoring-Anwendung möglich wäre. [4, S. 23 f.] [3, S. 163 ff.]

Listing 9: Senden von Antworten an ein über `JMSReplyTo` festgelegtes Ziel in einer MDB

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(
    propertyName = "destinationLookup",
    propertyValue = "jms/SystemStatus"
  )
  ...
})
public class SystemStatusListener implements
   MessageListener {
  @Inject
  private DiagnosticsBoundary diagnosticsBoundary;
  @Inject
  private JMSContext jmsContext;

  @Override
  public void onMessage(Message statusRequest) {
    SystemStatus systemStatus = diagnosticsBoundary
      .getSystemStatus();
    String requestId = statusRequest.getJMSMessageID();
    Destination replyDestination = statusRequest
       .getJMSReplyTo();
    jmsContext.createProducer()
      .setJMSCorrelationID(requestId)
      .send(replyDestination, systemStatus);
  }
}
```

Ein großer Vorteil von JMS ist die nahtlose Integration in den Transaktionsmechanismus (JTA) von JavaEE. Sendet eine Methode unter Transaktionskontrolle Nachrichten über JMS, werden diese erst beim Committen der Transaktion tatsächlich gesendet (i.d.R. beim Verlassen der Methode, welche die Transaktion geöffnet hat). Wird während der Ausführung durch einen Datenbankzugriff ein Rollback der Transaktion ausgelöst, wird auch das Senden der Nachrichten zurück gerollt. [4, S. 32 f.]

Ähnliches gilt für das Empfangen von Nachrichten innerhalb einer Transaktion. Erst durch das Committen werden die Nachrichten akzeptiert. Ein Rollback bewirkt in diesem Fall, dass Nachrichten wieder in dem Ziel verfügbar sind. Bei MDBs werden mehrere Zustellversuche durchgeführt bevor die Nachricht gelöscht oder in die Dead Message Queue verschoben wird. [4, S. 32 f.]

Würde die Methode `getOverallStatus` in Listing 8 unter einer Transaktion ablaufen, würde die Anfrage-Nachricht also erst nach dem Verlassen der

aufrufenden Geschäftsmethode gesendet werden. Dies hätte zur Folge, dass Anwendungen die Anfrage niemals erhalten, und die Methode nach Ablauf des Timeouts stets eine leere Liste zurückliefert. Die Monitoring-Anwendung unterstützt für Statusermittlung daher keine Transaktionen (`NOT_SUPPORTED`). [4, S. 24]

Besser wäre deshalb auf den Einsatz temporärer Ziele zu verzichten und stattdessen ein festgelegtes Ziel zu nutzen. Antworten könnten sogar an das Topic zurückgesendet werden. In der Monitoring-Anwendung würde dann anstelle des synchronen Abrufs eine MDB eingesetzt werden, um die Antworten einzusammeln.

4 Erzeugen einer realitätsnahen Umgebung für Demoanwendungen

Für die Demoanwendungen sollte eine reale Situation nachgebildet werden, in der JMS zur Kommunikation zwischen verschiedenen Maschinen eingesetzt wird. Dazu sollen Anwendungen in mehrere Payara-Domains deployt werden und über Messaging kommunizieren können. Ziel ist es dabei, möglichst wenige Änderungen an der Domain vorzunehmen. Jeweils eine Payara Instanz mit einer Domain wird dazu innerhalb eines Docker Contains betrieben.

Jeder Java EE Applikationsserver mit Full Profil liefert einen JMS Provider mit aus. Bei Payara handelt es sich um OpenMQ. Standardmäßig läuft dieser im Modus `EMBEDDED` innerhalb des Payara-Prozesses. Damit die Anwendungen kommunizieren können, müssen sie auch dieselben physikalischen Ziele nutzen. [4, S. 4] [2, S. 2]

Daher werden zwei Docker-Images erstellt. Die Domain des Master-Images betreibt den eingebetteten JMS Provider. Von diesem Image wird ein Container gestartet, dem eine feste IP-Adresse zugewiesen wird. Für das Slave-Image wird die Domain so konfiguriert, dass der JMS Provider des Master-Containers verwendet wird.

```
asadmin create-jms-host --mqhost 172.18.0.2 --mqport 7676
--mquser admin --mqpassword ****
master_jms_server
asadmin set configs.config.server-config.jms-service.type=
    REMOTE
asadmin set configs.config.server-config.jms-service.
    default-jms-host=master_jms_server
```

Der erste Befehl erstellt einen neuen Host und definiert über welche IP-Adresse und Port der Messaging Provider erreichbar ist. Diese Host Definition wird anschließend als Standard Host für JMS festgelegt. Der zweite Befehl ändert den JMS-Betriebsmodus auf `REMOTE` und gibt somit an, dass der Applikationsserver nicht seinen eingebetteten JMS Provider starten soll, sondern stattdessen eine Verbindung zu einem JMS Provider auf einer anderen Maschine aufbauen soll. Die Domain muss anschließend einmal neu gestartet werden. [2, S. 3, 5 f.]

Zur Verbesserung der Ausfallsicherheit würde für reale Anwendungen der JMS Provider extern betrieben werden (z.B. separater Docker-Container) und ebenfalls aus mehreren Instanzen bestehen. Die Demoanwendung ist davon vollständig entkoppelt. Lediglich die Konfiguration der Domain müsste angepasst werden.

5 Fazit und Ausblick

Message Driven Beans bieten klare Vorteile gegenüber der synchronen Nachrichtenabfrage, weshalb sie in JavaEE Anwendungen bevorzugt eingesetzt werden sollen. Allerdings lassen sie sich als Integrationskomponenten in der klassischen Schichtenarchitektur nicht optimal verordnen. Die Integrationsschicht besitzt keine Abhängigkeiten zur Businessschicht, weshalb MDBs in letzterer implementiert werden müssen, um Geschäftslogik aufzurufen. Allerdings sollte auch dann eine klare Trennung zur eigentlichen Geschäftslogik bestehen, indem MDBs beispielsweise nur Boundaries injecten dürfen. In eine hexagonale Architektur fügen sich MDBs hingegen sehr gut ein. Sie können einen eigenen Adapter bilden, der Geschäftslogik aus dem Kern aufruft.

Die asynchrone Kommunikation zwischen Anwendungen ist durchaus komplexer als beispielsweise Webservice-Aufrufe. Viele Teile der Geschäftslogik werden über Benutzeroberflächen ausgelöst und können durch Messaging schneller ablaufen, da nicht auf Antworten gewartet wird. Allerdings ist der Zeitpunkt zu dem Ergebnisse eintreffen nicht bekannt, sodass eine Kommunikation von Server zu Client erforderlich wäre, um die Benutzeroberflächen zu aktualisieren. Auch Anpassungen in Geschäftsprozessen sind notwendig, um eine asynchrone Abarbeitung zu ermöglichen.

Jedoch lässt sich Messaging mit JEE Standardmittel und wenig Mehraufwand zu WebServices einsetzen. Durch die losere Kopplung integrierter Anwendungen kann auch die Entwicklung beschleunigt werden, da vieles bereits von JMS übernommen wird (wie z.B. Wiederholtes Senden bei Übertragungsfehlern). Durch die Unabhängigkeit zwischen mehreren Instanzen der Anwendung eignet sich Messaging besonders im Bereich von Micro Services.

Viele spannende Punkte, wie die ausfallsichere Konfiguration eines Messaging Providers wurden in dieser Arbeit nicht behandelt. Ich freue mich darauf JMS oder eine andere Messaging Technologie in einem größeren Projekt eingesetzt zu sehen.

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, Release 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).

Abbildungsverzeichnis

Eidesstattliche Erklärung zur selbstständigen Verfassung

Hiermit erkläre ich an Eides Statt, dass ich die vorliegende Arbeit selbstständig und nur unter Zuhilfenahme der ausgewiesenen Hilfsmittel angefertigt habe. Sämtliche Stellen der Arbeit, die im Wortlaut oder dem Sinn nach anderen gedruckten oder im Internet verfügbaren Werken entnommen sind, habe ich durch genaue Quellenangaben kenntlich gemacht.

Taufkirchen, den 17.Mai 2017 Marcel Binder