Федеральное государственное автономное
образовательное учреждение
высшего образования
«СИБИРСКИЙ ФЕДЕРАЛЬНЫЙ УНИВЕРСИТЕТ»
Институт космических и информационных технологий
Кафедра «Информатика»

Конспект лекции №8
Обмен сообщениями в Spring.

Составил: Старший преподаватель кафедры «Информатика»
Черниговский Алексей Сергеевич

Красноярск 2020

Содержание

Предисловие

Ранее рассказывалось как пользоваться веб-службами для организации взаимодействий между приложениями. Этот механизм взаимодействий имеет синхронную природу, когда клиентское приложение напрямую взаимодействует с удаленной службой и ожидает, пока удаленная процедура не выполнится, прежде чем продолжить работу.

Синхронные взаимодействия имеют место быть, но это не единственный способ организации взаимодействий между приложениями. Асинхронный обмен сообщениями позволяет отправлять сообщения из одного приложения в другое, не ожидая ответа. Асинхронный способ взаимодействий имеет свои преимущества перед синхронным, как будет показано ниже.

Java Message Service (JMS) – стандартный API для организации асинхронного обмена сообщениями. В этой теме будет показано, насколько фреймворк Spring упрощает отправку и прием сообщений с помощью JMS. Помимо простых операций приема и передачи сообщений, здесь также будет рассказываться о простых Java-объектах (POJO).

1 Краткое введение в JMS

JMS – это механизм организации взаимодействий между приложениями, во многом напоминающий механизмы удаленных взаимодействий и интерфейсы REST, представленные в предыдущей главе. Но важным отличием JMS от других механизмов является способ передачи данных между системами.

Механизмы удаленных взаимодействий в большинстве своем имеют синхронную природу. Как показано на рисунке 1, когда клиент вызывает удаленный метод, он вынужден ждать завершения выполнения метода, прежде чем продолжить работу. Даже если удаленный метод ничего не возвращает клиенту, клиент все равно будет простаивать, пока обслуживание его запроса не завершится.

Механизм JMS, напротив, обеспечивает асинхронный режим взаимодействий между приложениями. Когда сообщения отправляются асинхронно, как показано на рисунке 2, клиенту не приходится ждать, пока служба обработает сообщение или доставит его адресату. Клиент просто отправляет свое сообщение и продолжает работу, надеясь, что служба рано ли поздно примет и обработает его.

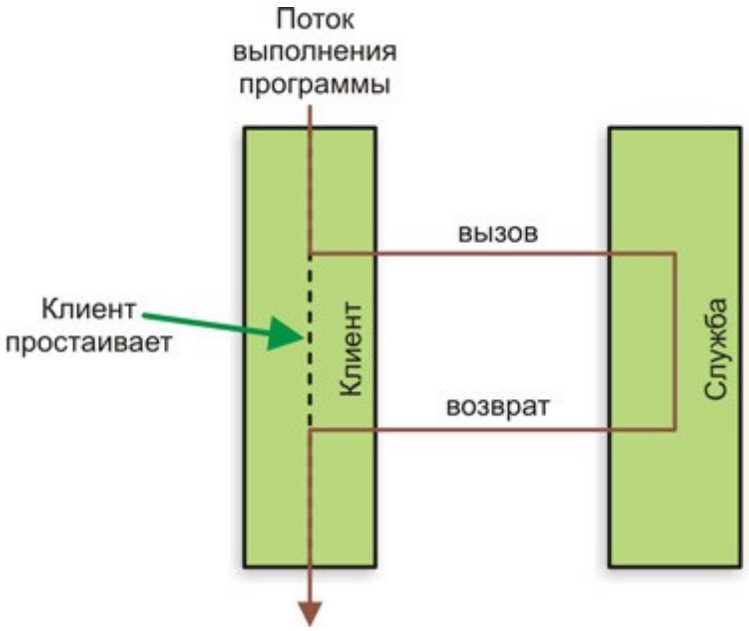

Рисунок 1 — При синхронных взаимодействиях клиент вынужден ждать завершения операции

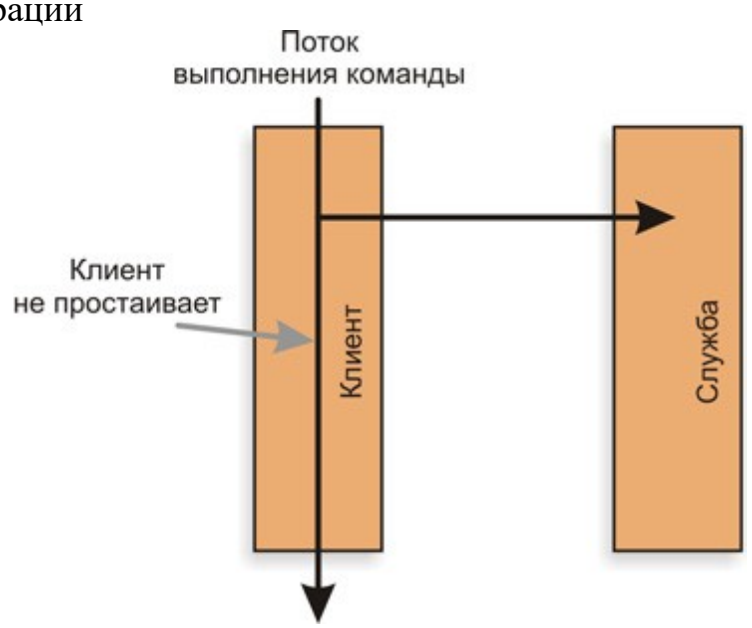

Рисунок 2 — При асинхронных взаимодействиях клиенту не приходится ждать ответа службы

Асинхронные взаимодействия посредством JMS имеют ряд преимуществ перед синхронными взаимодействиями. Эти преимущества будут рассматриваться чуть ниже. Но сначала посмотрим, как выполняется отправка сообщений с помощью JMS.

1.1 Архитектура JMS

Большинство из нас не раз пользовались услугами почты. Ежедневно миллионы людей передают письма, открытки и посылки в руки почтальонов, будучи уверенными, что они будут доставлены адресатам. Мир слишком велик,

чтобы все эти отправления можно было передавать из рук в руки лично, поэтому мы полагаемся на почтовую службу. Мы обращаемся на почту, оплачиваем услуги за пересылку, передаем почтовое отправление и ни на секунду не задумываемся, каким образом оно будет доставлено.

Преимуществом почтовой службы является отсутствие необходимости прямого участия в доставке. Было бы крайне неудобно лично заниматься доставкой, например, поздравительной открытки. В зависимости от расстояния, разделяющего вас, чтобы доставить открытку лично, пришлось бы отложить свои дела на несколько часов, а то и дней. К счастью, почта доставит эту открытку без вашего участия.

Аналогично действует и механизм JMS. Когда одно приложение отправляет информацию другому приложению посредством JMS, не требуется устанавливать прямую связь между двумя приложениями. Вместо этого приложение-отправитель передает сообщение в руки службе, которая гарантирует его доставку приложению-получателю.

В JMS имеются две основные действующие стороны: брокеры сообщений и приемники.

Когда приложение отправляет сообщение, оно передается в руки брокера сообщений. Брокер сообщений – это аналог почтового отделения в JMS. Брокер сообщений гарантирует доставку сообщения в указанный приемник, позволяя отправителю продолжать заниматься своими делами. При отправке письма по почте важно указать адрес на конверте, чтобы работники почтовой службы могли узнать, куда его доставить. В JMS также имеется понятие адресов, которые определяют приемники. Приемники – это своего рода почтовые ящики, куда будут помещаться сообщения.

Рисунок 3 — Очередь сообщений обеспечивает независимость отправителя сообщения от его получателя. Несмотря на то что очередь может использоваться несколькими получателями, каждое сообщение может быть вынуто из очереди только один раз

Но, в отличие от почтовых адресов, которые определяют имя человека или номер дома с названием улицы, адреса в JMS являются менее определенными. Адреса в JMS определяют лишь место, где получатель сможет получить сообщение, но не определяют, кто сможет получить его. То есть в JMS допускается посылать письма по адресу, например: «директору организации». В JMS существуют два типа приемников: очереди и темы . Каждый из них следует

определенной модели обмена сообщениями: либо модель «точка–точка» (очереди), либо модель «публикация–подписка» (темы).

1.1.1 Модель «точка-точка»

В модели «точка–точка» каждое сообщение имеет точно одного получателя и одного отправителя, как показано на рисунке 3. Когда брокер сообщений получает сообщение, он помещает его в очередь. Когда получатель обращается за следующим сообщением в очереди, это сообщение удаляется из очереди и передается получателю. Поскольку в момент доставки сообщение удаляется из очереди, его гарантированно получит только один получатель.

Несмотря на то что каждое сообщение, имеющееся в очереди, передается только одному получателю, это не означает, что очередью может пользоваться только один получатель. В действительности сообщения из очереди могут извлекаться несколькими получателями. Но каждый из них получит свое, уникальное сообщение.

Это напоминает ожидание в очереди в банке. В процессе ожидания можно заметить, что очередь обслуживается несколькими операторами банка. Как только один клиент будет обслужен, оператор вызовет следующего из очереди. Когда подойдет ваша очередь, вас вызовут, и вы будете обслужены одним из операторов. Другие операторы будут обслуживать других клиентов.

Еще одно наблюдение, которое можно сделать в банке, – когда клиент становится в очередь, он не знает наверняка, какой оператор будет его обслуживать. Можно прикинуть количество людей в очереди, соотнести это число с количеством операторов, учесть, кто из операторов работает быстрее других, и затем выдвинуть предположение о том, кто из операторов будет вас обслуживать. Но велика вероятность, что вы ошибетесь, и вас будет обслуживать другой оператор.

То же самое происходит и в JMS, если очередь обслуживается несколькими получателями, нет никакой возможности определить, который из них будет обрабатывать то или иное сообщение. Эта неопределенность – благо, потому что позволяет приложению увеличивать скорость обработки сообщений простым добавлением новых получателей.

1.1.2 Модель «публикация–подписка»

В модели «публикация–подписка» сообщения отправляются в тему. Подобно очередям, тему могут обслуживать несколько получателей. Но, в отличие от очередей, где каждое сообщение передается точно одному получателю, все подписчики на тему получат собственные копии сообщений, как показано на рисунке 4.

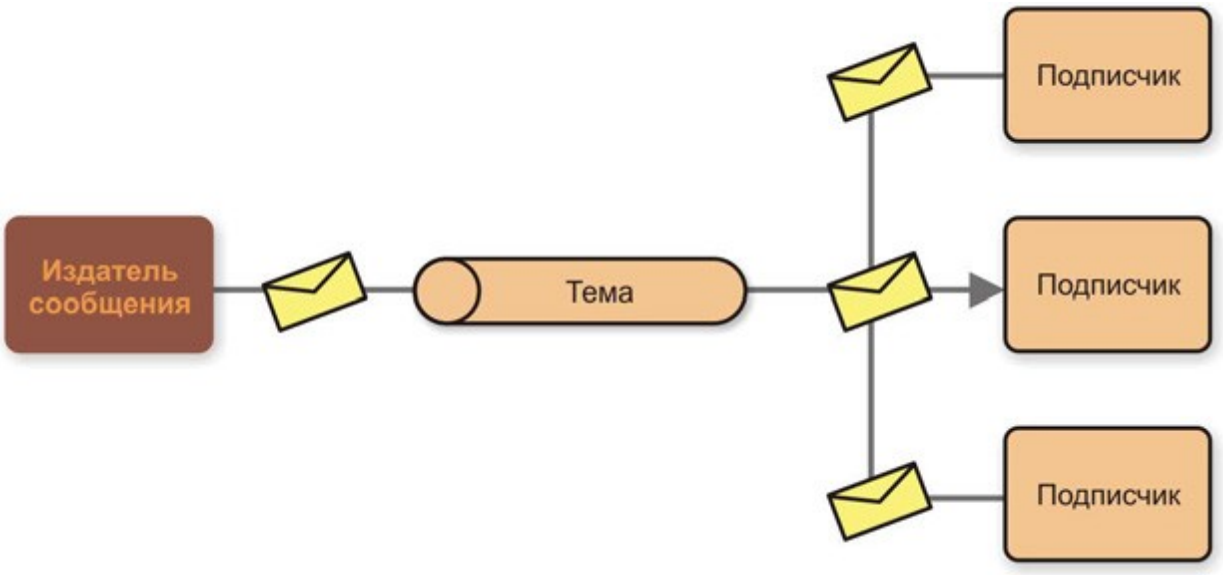

Рисунок 4 — Подобно очередям, темы обеспечивают независимость отправителей сообщений от их получателей. В отличие от очередей, сообщение в теме может быть доставлено нескольким получателям, подписавшимся на тему

Как следует из названия модели «публикация–подписка», она во многом напоминает модель издательства, выпускающего журнал, и подписчиков. Журнал (сообщение) публикуется и передается почтовой службе, которая затем доставляет копии журнала подписчикам.

Аналогия с журналом не совсем корректна в данном случае, потому что в JMS издатель не знает, кто будет играть роль подписчика. Единственное, что известно издателю, – его сообщения будут опубликованы в определенной теме, но не известно, кто подписан на эту тему. Из этого также следует, что издатель не знает, как будут обрабатываться его сообщения.

1.2 Преимущества JMS

Несмотря на всю свою простоту, синхронные взаимодействия накладывают некоторые ограничения на удаленные службы и их клиентов, наиболее значимые из которых перечислены ниже.

- Синхронные взаимодействия предполагают наличие этапов ожидания. Когда клиент вызывает метод удаленной службы, он вынужден ожидать завершения удаленного метода, прежде чем продолжить работу. Если клиент обращается к удаленной службе достаточно часто и/или служба имеет невысокое быстродействие, это может отрицательно сказаться на производительности клиентского приложения.
- Клиент тесно связан со службой через интерфейс службы. Если потребуется изменить интерфейс службы, придется изменить и все клиентские приложения. Клиент тесно связан с местонахождением службы. Клиента необходимо настраивать, чтобы сообщить ему сетевой адрес службы. Если

топология сети изменится, клиентов придется перенастраивать, чтобы указать им новый адрес службы.

- Клиент существенно зависит от доступности службы. Если служба окажется недоступной, клиент окажется парализованным.

Синхронные взаимодействия имеют право на существование, но перечисленные выше недостатки необходимо учитывать при выборе механизма взаимодействий для своего приложения. Если перечисленные ограничения не устраивают вас, подумайте о возможности использования асинхронных взаимодействий с помощью механизма JMS, свободного от этих проблем.

Отсутствие ожидания

Когда передача сообщений осуществляется с помощью JMS, клиент не должен ждать, пока оно будет обработано или хотя бы доставлено. Клиент отдает сообщение брокеру и продолжает свое выполнение, надеясь, что сообщение будет доставлено по указанному адресу. Поскольку клиенту не приходится ждать, он может заниматься решением других задач. Отсутствие простоя может существенно повысить производительность клиента.

Ориентированность на сообщения и независимость

В отличие от механизма RPC, суть которого состоит в организации вызовов методов, в основе механизма JMS лежат данные. То есть клиент не стеснен рамками сигнатуры метода. Он сможет поместить в очередь или в тему любое сообщение, и ему не требуется знать все тонкости его обработки.

Независимость от местоположения

Синхронные службы RPC обычно связаны с определенными сетевыми адресами. Как следствие, клиенты оказываются очень чувствительны к изменениям топологии сети. Если IP-адрес или порт службы изменится, потребуется изменить соответствующие на- стройки клиентов, иначе они не смогут получить доступ к службе.

Клиенты JMS, напротив, понятия не имеют, кто будет обрабатывать их сообщения или где находится служба. Клиент знает только очередь или тему для отправки сообщений. В результате клиент оказывается независимым от службы, при условии что он в состоянии извлекать сообщения из очереди или из темы.

Преимущество независимости от местоположения службы в модели «точка–точка» можно использовать для создания кластеров служб. Если клиенту не требуется знать местоположение службы и единственным требованием службы является доступность брокера сообщений, то нет никаких причин, почему нельзя было бы настроить несколько служб на извлечение сообщений из единой очереди. Если службы оказываются перегруженными и начинают запаздывать, достаточно просто добавить еще несколько экземпляров службы, обслуживающих ту же очередь.

Независимость от местоположения влечет за собой еще один интересный побочный эффект в модели «публикация–подписка». На одну тему может быть подписано несколько служб, извлекающих копии одного и того же сообщения. Но каждая служба обрабатывает сообщение по-своему. Например, допустим, что имеется множество служб, реализующих обработку информации при найме нового работника. Одна из служб может добавлять работника в систему расчета заработной платы, другая – в систему учета персонала, а третья – открывает работнику доступ к системам, которые ему потребуются для выполнения своих служебных обязанностей. Каждая служба действует независимо и использует те же исходные данные, что и другие службы, подписанные на получение сообщений из темы.

Гарантированная доставка

Чтобы клиент мог взаимодействовать с синхронной службой, служба должна быть готова к приему запросов на указанном IP-адресе с указанным номером порта. Если служба не будет запущена или будет недоступна по каким-то другим причинам, клиент не сможет продолжить выполнение.

Когда отправка сообщений производится посредством JMS, клиент может быть уверен, что его сообщения будут доставлены службе. Даже если служба окажется недоступна на момент отправки сообщения, оно будет храниться, пока служба снова не станет доступна.

Теперь, когда вы получили общее представление о JMS и асинхронной передаче сообщений, попробуем настроить брокер сообщений JMS, который будет использоваться в наших примерах. Вы можете использовать любой брокер сообщений JMS, но здесь будет использоваться наиболее популярный брокер ActiveMQ.

2. Настройка брокера сообщений в Spring

ActiveMQ – отличный брокер сообщений, распространяемый с открытыми исходными текстами, и замечательный выбор для организации асинхронного обмена сообщениями посредством JMS. На текущий момент это версия ActiveMQ 5.16.0. Прежде чем приступать к работе с ActiveMQ, необходимо получить дистрибутив, который можно загрузить по адресу: http://activemq.apache.org После загрузки дистрибутива распакуйте его в каталог на жестком диске. Чтобы получить возможность пользоваться ActiveMQ API в ваш Maven проект можно добавить зависимость org.apache.activemq.

2.1. Создание фабрики соединений

В этой главе будут рассматриваться различные способы отправки и приема сообщений с помощью фреймворка Spring. Во всех случаях, чтобы иметь возможность отправлять сообщения через брокер, нам потребуется использовать фабрику соединений JMS. Поскольку в этой главе предполагается использовать брокер ActiveMQ, необходимо настроить фабрику соединений

JMS так, чтобы она создавала соединения с ActiveMQ. В состав ActiveMQ входит собственная фабрика соединений ActiveMQConnectionFactory, которая настраивается в Spring, как показано ниже:

```
<bean id="connectionFactory"
      class="org.apache.activemq.spring.ActiveMQConnectionFactory">
<     property name="brokerURL" value="tcp://localhost:61616"/>
</bean>
```

Кроме того, поскольку известно, что будет использоваться брокер ActiveMQ, для объявления фабрики соединений можно использовать пространство имен, определяемое брокером ActiveMQ (доступно во всех версиях ActiveMQ, начиная с версии 4.1). Сначала необходимо подключить пространство имен amq в конфигурационном XML-файле Spring:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans       xmlns="http://www.springframework.org/schema/beans"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:jms="http://www.springframework.org/schema/jms"
xmlns:amq="http://activemq.apache.org/schema/core"
xsi:schemaLocation="http://activemq.apache.org/schema/core
      http://activemq.apache.org/schema/core/activemq-core-5.5.0.xsd
      http://www.springframework.org/schema/jms
      http://www.springframework.org/schema/jms/spring-jms-3.0.xsd
      http://www.springframework.org/schema/beans
      http://www.springframework.org/schema/beans/spring-beans-3.0.xsd">

</beans>
```

А затем с помощью элемента <amq:connectionFactory> объявить фабрику соединений:

```
<amq:connectionFactory id="connectionFactory"
             brokerURL="tcp://localhost:61616"/>
```

Обратите внимание, что элемент <amq:connectionFactory> является особенностью брокера ActiveMQ. При использовании других реализаций брокера сообщений конфигурационные пространства имен могут отсутствовать, и в этом случае необходимо объявлять фабрику соединений как обычный компонент.

Далее в этой главе мы часто будем возвращаться к определению компонента connectionFactory. Но сейчас достаточно будет знать, что атрибут brokerURL определяет местоположение брокера. В данном случае URL в атрибуте brokerURL сообщает, что брокер действует на локальном компьютере и принимает соединения на порту с номером 61616 (номер порта, используемый брокером ActiveMQ по умолчанию).

2.2. Объявление приемников ActiveMQ

Помимо фабрики соединений, необходимо настроить приемник, откуда можно будет извлекать сообщения. Приемник может быть либо очередью, либо темой, в зависимости от потребностей приложения. Независимо от типа

приемника, очередь или тема, компонент-приемник должен быть настроен как экземпляр класса, характерного для используемого брокера сообщений. Например, следующее объявление <bean> определяет очередь ActiveMQ:

```
<bean id="queue" class="org.apache.activemq.command.ActiveMQQueue">
        <constructor-arg value="student.queue"/>
 </bean>
```

Аналогично следующее объявление <bean> определяет тему ActiveMQ:

```
<bean id="topic" class="org.apache.activemq.command.ActiveMQTopic">
      <constructor-arg value="student.topic"/>
</bean>
```

В обоих случаях элемент <constructor-arg> определяет имя очереди или темы, которая будет использоваться брокером сообщений – student.topic в последнем примере. Как и в случае с фабрикой соединений, в пространстве имен ActiveMQ предоставляется альтернативный способ объявления очередей и тем. Для определения очереди можно использовать элемент

<amq:queue>: <amq:queue id="queue" physicalName="student.queue" />

А для определения темы – элемент <amq:topic>:

```
<amq:topic id="topic" physicalName="student.topic" />
```

В обоих случаях атрибут physicalName определяет имя канала обмена сообщениями. К настоящему моменту было показано, как объявлять основные компоненты для отправки и приема сообщений. Теперь можно приступать к непосредственной реализации операций. Для этого воспользуемся классом JmsTemplate, входящим в состав Spring и являющимся основой поддержки JMS в Spring. Но, чтобы оценить все преимущества использования класса JmsTemplate, сначала посмотрим, как реализовать обмен сообщениями без него.

3 Работа с шаблонами JMS в Spring

Механизм JMS дает разработчикам приложений на языке Java стандартный API для взаимодействия с брокерами сообщений, а также для отправки и приема сообщений. Кроме того, практически все существующие реализации брокеров сообщений поддерживают JMS. Поэтому нет причин изучать частные API обмена сообщениями для организации взаимодействий с брокерами.

Да, JMS предлагает универсальный интерфейс для работы с любыми брокерами, но за это удобство приходится платить. Отправить и принять сообщение с помощью JMS намного сложнее, чем просто шлепнуть печать на конверт. Как будет показано далее, JMS требует, чтобы вы еще и заправили почтовый грузовик (фигурально выражаясь).

3.1. Борьба с разбуханием JMS-кода

В теме «Работа с БД в Spring» было показано, насколько громоздким может быть код для работы с JDBC, необходимый для обработки установления соединений, выполнения запросов, получения результатов и обработки исключений. К сожалению, JMS API подвержен той же болезни, как можно заметить в листинге ниже:.

Отправка сообщения без привлечения Spring

```
ConnectionFactory cf =
                new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
Session session = null;
try {
        conn = cf.createConnection();
        session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination destination = new ActiveMQQueue("student.queue");
        MessageProducer producer = session.createProducer(destination);
        TextMessage message = session.createTextMessage();
        message.setText("Hello world!");
        producer.send(message); // Отправка сообщения
} catch (JMSException e) {
        // обработать исключение?
} finally {
        try {
                if (session != null) {
                        session.close();
                } if (conn != null) {
                        conn.close();
                }
        } catch (JMSException ex) {
        }
}
```

Ох уж этот шаблонный код! Как и в примере использования JDBC, потребовалось более 20 строк кода, только чтобы отправить простое сообщение «Hello world!». При этом к отправке сообщения прямое отношение имеют лишь несколько строк, остальные выполняют подготовительные операции. Не лучше выглядит и реализация приема сообщения, как показано в листинге ниже.

Прием сообщения без привлечения Spring

```
ConnectionFactory cf =
        new ActiveMQConnectionFactory("tcp://localhost:61616");
Connection conn = null;
```

```
Session session = null;
try {
        conn = cf.createConnection();
        conn.start();
        session = conn.createSession(false, Session.AUTO_ACKNOWLEDGE);
        Destination destination =
                new ActiveMQQueue("student.queue");
        MessageConsumer consumer = session.createConsumer(destination);
        Message message = consumer.receive();
        TextMessage textMessage = (TextMessage) message;
        System.out.println("GOT A MESSAGE: " + textMessage.getText());
        conn.start();
} catch (JMSException e) {
        // обработать исключение?
} finally {
        try {
                if (session != null) {
                        session.close();
                } if (conn != null) {
                        conn.close();
                }
         } catch (JMSException ex) {
        }
}
```

И снова, как и в предыдущем примере, здесь используется масса шаблонного кода. Если провести построчное сравнение, можно обнаружить, что эти два листинга практически идентичны. И если посмотреть на тысячи других примеров использования JMS, можно с тем же успехом заметить их поразительное сходство. Где-то вместо фабрики соединений используется JNDI, а где-то вместо очередей используются темы, но все они следуют одному и тому же шаблону.

Как следствие при работе с JMS вы постоянно будете обнаруживать, что пишете один и тот же код снова и снова. Хуже того, вы будете обнаруживать, что повторяете программный код, написанный другими разработчиками.

В «Работа с БД в Spring» уже было показано, как использование класса JdbcTemplate позволяет избавиться от шаблонного кода при работе с JDBC. Теперь пришло время посмотреть, как тот же эффект помогает получить класс JmsTemplate при работе с JMS.

3.2. Работа с шаблонами JMS

Класс JmsTemplate – это ответ фреймворка Spring на необходимость писать массу шаблонного кода для работы с JMS. Класс JmsTemplate берет на

себя все хлопоты по созданию соединений, открытию сеансов и приему/передаче сообщений. Он позволяет разработчику сосредоточиться на конструировании сообщений для передачи или обработке принимаемых сообщений.

Более того, JmsTemplate может обрабатывать все эти неудобные исключения JMSException . Если в процессе работы в классе JmsTemplate будет возбуждено исключение JMSException, оно будет перехвачено, и повторно будет возбуждено неконтролируемое исключение, являющееся одним из подклассов класса JmsException.

К чести JMS API, иерархия JMSException включает богатый набор подклассов, позволяющих определить причины, повлекшие исключительную ситуацию. Однако все подклассы JMSException являются контролируемыми исключениями и потому обязательно должны перехватываться. Класс JmsTemplate автоматически перехватывает все эти исключения и возбуждает соответствующие им неконтролируемые исключения, наследующие класс JmsException.

Внедрение шаблона JMS

Чтобы задействовать класс JmsTemplate, необходимо объявить соответствующий компонент в конфигурационном файле Spring, как показано ниже:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
      <property name="connectionFactory" ref="connectionFactory" />
 </bean>
```

Поскольку экземпляру JmsTemplate нужно знать, как приобрести соединение с брокером сообщений, в свойстве connectionFactory компонента следует указать ссылку на компонент, реализующий интерфейс JMS ConnectionFactory. В примере выше внедряется ссылка на компонент connectionFactory, объявленный выше.

Вот и все, что необходимо, чтобы задействовать класс JmsTemplate, – теперь все готово к работе. Начнем с отправки сообщений!

Отправка сообщений

Одной из особенностей, которую мы предполагаем ввести в наше приложение является возможность извещения (возможно, по электронной почте) других пользователей (допустим студентов, о задолженностях). Эту функцию можно было бы добавить непосредственно в том месте, где производится создание сообщения. Но фактическая отправка извещений в этом месте может занять продолжительное время и отрицательно сказаться на субъективном восприятии производительности приложения.

Чтобы не тратить времени на отправку извещений во время добавления сообщения, проще будет поместить это извещение в очередь и выполнить его

рассылку позднее, после отправки ответа пользователю. Время, необходимое на передачу извещения в очередь или тему, весьма незначительно, особенно если сравнить со временем, необходимым для рассылки извещений другим пользователям.

Для поддержки асинхронной рассылки извещений введем в приложение службу AlertService:

```
public interface AlertService {
	void sendAlert(Alert alert);
	Alert getAlert();
}
```

Как видите, AlertService – это интерфейс, определяющий единственный метод sendAlert(). Класс AlertServiceImpl – это реализация интерфейса AlertService, использующая компонент JmsTemplate для отправки объектов Spittle в очередь сообщений с целью их обработки в более позднее время.

```
import javax.jms.JMSException;
import javax.jms.Message;
import javax.jms.Session;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jms.core.JmsTemplate;
import org.springframework.jms.core.MessageCreator;

public class AlertServiceImpl implements AlertService {
public void sendAlert(final Alert alert) {
	jmsTemplate.send(                        // Отправка сообщения
			"student.alert.queue",  // Имя приемника
			new MessageCreator() {
				public Message createMessage(Session session)
						throws JMSException {
					return session.createObjectMessage(alert);
                                        // Создание
				}                       // сообщения
			}
			);
}

@Autowired
JmsTemplate jmsTemplate;
}
```

Первый параметр метода send() класса JmsTemplate – имя приемника JMS, куда отправляется сообщение. При вызове метода send() класс JmsTemplate приобретет JMS-соединение и сеанс и отправит сообщение от имени отправителя (рисунок 5).

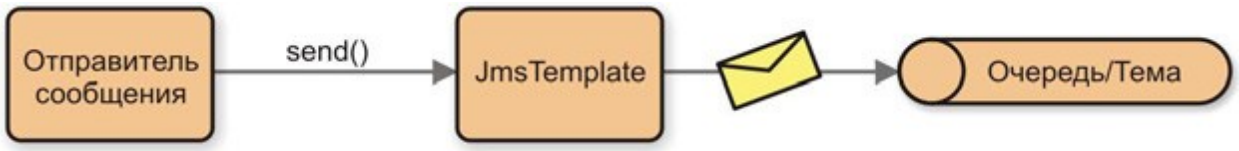

Рисунок 5 — Класс JmsTemplate берет на себя все сложности, связанные с отправкой сообщений

Что касается самого сообщения, оно конструируется с использованием реализации интерфейса MessageCreator , организованной в виде анонимного вложенного класса. В методе createMessage() реализации интерфейса MessageCreator программа просто предлагает сеансу создать новый объект сообщения и добавить в него объект Alert.

Вот и все! Обратите внимание, что метод sendAlert() занимается исключительно сборкой и отправкой сообщения. В нем отсутствует код, управляющий соединением или сеансом, – все это автоматически делает класс JmsTemplate. И в нем отсутствует код, перехватывающий исключения JMSException, – класс JmsTemplate перехватывает все исключения JMSException и возбуждает в ответ неконтролируемые исключения.

Настройка приемника по умолчанию

В примере выше явно был указан конкретный приемник, куда будут отправляться сообщения методом send(). Такую форму метода send() удобно использовать, когда требуется организовать выбор приемника программно. Но в реализации AlertServiceImpl все сообщения отправляются в один и тот же приемник, поэтому преимущества данной формы метода send() неочевидны.

Вместо явного определения приемника при отправке каждого сообщения в объявлении компонента JmsTemplate можно было бы определить приемник по умолчанию:

```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
      <property name="connectionFactory" ref="connectionFactory" />
      <property name="defaultDestinationName" value="student.alert.queue"/>
</bean>
```

Теперь вызов метода send() класса JmsTemplate можно немного упростить, убрав первый параметр:

```
jmsTemplate.send(
      new MessageCreator() {
            ...
      }
);
```

Эта форма метода send() принимает только реализацию интерфейса MessageCreator, а в качестве приемника используется приемник по умолчанию.

Извлечение сообщений

Выше было показано, как с помощью JmsTemplate отправлять сообщения. А как быть на принимающей стороне? Можно ли использовать класс JmsTemplate для приема сообщений?

Да, можно. Фактически прием сообщений с помощью класса JmsTemp late реализуется еще проще. Чтобы извлечь сообщение, достаточно вызвать метод receive() класса JmsTemplate, как показано в примере ниже.

Прием сообщения с помощью класса JmsTemplate

```
public Alert getAlert() {
	try {
		ObjectMessage receivedMessage =
			(ObjectMessage)jmsTemplate.receive(); // Прием сообщения
		return (Alert) receivedMessage.getObject();
		// Извлечение объекта
	} catch (JMSException jmsException) {
			// Возбудить преобразованное исключение
		throw JmsUtils.convertJmsAccessException(jmsException);
	}
}
```

Метод receive() класса JmsTemplate пытается получить сообщение, обратившись к брокеру сообщений. Если сообщение недоступно, метод receive() будет ждать, пока сообщение не станет доступно. Это взаимодействие изображено на рисунке 6.

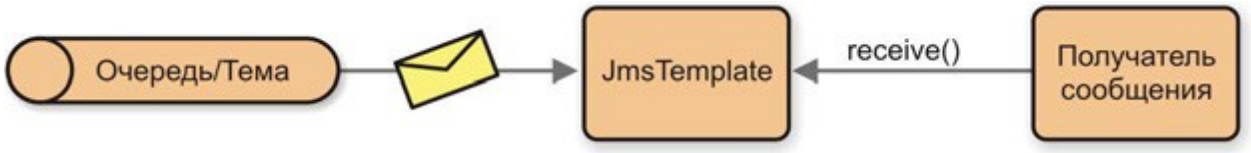

Рисунок 6 — Прием сообщений из темы или из очереди с помощью класса JmsTemplate осуществляется простым вызовом его метода receive(). Обо всем остальном позаботится сам класс JmsTemplate

Поскольку известно, что в нашем приложении отправляемые сообщения содержат объект, их можно привести к типу ObjectMessage при приеме. А затем вызвать метод getObject(), чтобы извлечь объект Alert из сообщения ObjectMessage и вернуть его.

Единственная неприятность, которая здесь поджидает, – необходимость предпринять какие-то действия в случае появления исключения JMSException. Как уже упоминалось, класс JmsTemplate прекрасно справляется со всеми контролируемыми исключениями, наследниками JMSException, возбуждая в ответ соответствующие им неконтролируемые исключения, наследники JmsException. Но это относится только к вызовам методов класса JmsTemplate. Класс JmsTemplate не сможет помочь обработать исключение JMSException, которое может быть возбуждено методом getObject() класса ObjectMessage .

Поэтому необходимо либо перехватить исключение JMSException, либо объявить, что данный метод может возбуждать его. В соответствии с философией фреймворка Spring, которая призывает из- бегать контролируемых исключений, было решено не позволять исключению JMSException покидать пределы этого метода и перехватить его. В блоке catch можно воспользоваться методом convertJmsAccessException() класса JmsUtils , входящего в состав фреймворка Spring, чтобы преобразовать контролируемое исключение JMSException в неконтролируемое исключение JmsException. В действительности имен- но так класс JmsTemplate преобразует исключения.

Самым большим недостатком операции извлечения сообщений с помощью класса JmsTemplate является синхронность метода receive(). Это означает, что приложение вынуждено будет ждать появления сообщения, так как метод receive() окажется заблокированным до момента, пока сообщение не станет доступно (или пока не истечет предельное время ожидания). Не кажется ли вам странной синхронность при приеме сообщения, которое посылается асинхронно?

Решить эту проблему нам помогут простые Java-объекты (POJO), управляемые сообщениями. Посмотрим, как организовать асинхронный прием сообщений с помощью компонентов, реагирующих на сообщения, а не ожидающих их появления.

4 Создание POJO, управляемых сообщениями

Когда программа вызывает метод receive(), он выходит и проверяет наличие сообщения в очереди или в теме, но не возвращается, пока сообщение не поступит или пока не истечет предельное время ожидания. В это время приложение «стоит» и ничего не делает. Разве не было бы лучше, если бы приложение могло продолжать заниматься своими делами и извещалось бы в момент поступления сообщения?

Одним из отличительных моментов спецификации EJB 2 было включение в нее компонентов, управляемых сообщениями (MessageDriven Beans, MDB). Компоненты MDB – это компоненты EJB, обрабатывающие сообщения асинхронно. Иными словами, MDB реагируют на сообщения, появляющиеся в приемнике JMS, как на события. Этим они в корне отличаются от компонентов, выполняющих прием сообщений синхронно, выполнение которых блокируется в отсутствие сообщений.

Компоненты MDB ярко выделялись на общем ландшафте EJB. Даже самые непримиримые критики EJB признавали, что компоненты MDB являются весьма элегантным способом обработки сообщений. Единственным недостатком компонентов MDB в спецификации EJB 2 была необходимость поддержки ими интерфейса javax.ejb.MessageDrivenBean . При этом они также должны были реализовать некоторые методы обратного вызова поддержки жизненного цикла. Проще говоря, компоненты MDB были далеко не простыми Java-объектами.

Компоненты MDB были убраны из спецификации EJB 3, где предпочтение было отдано более простым POJO. Теперь больше не требуется включать реализацию интерфейса MessageDrivenBean. Вместо этого требуется реализовать более обобщенный интерфейс javax.jms.MessageListener и использовать аннотацию @MessageDriven.

Начиная с версии 2.0, фреймворк Spring решил проблему асинхронного чтения сообщений, предоставив собственную реализацию компонентов, управляемых сообщениями, которые очень похожи на компоненты MDB, определяемые спецификацией EJB 3. В этом разделе рассказывается о поддержке асинхронного чтения сообщений в Spring с применением POJO, управляемых сообщениями (назовем их MDP для краткости).

4.1 Создание объекта для чтения сообщений

Если бы потребовалось реализовать прием извещений о новых сообщениях в приложении с использованием модели, определяемой спецификацией EJB, нам пришлось бы задействовать аннотацию @MessageDriven. И реализовать интерфейс MessageListener, хотя это и не обязательно. Результат выглядел бы, как показано ниже:

```
@MessageDriven(mappedName="jms/student.alert.queue")
public class AlertHandler implements MessageListener {
    @Resource
    private MessageDrivenContext mdc;
    public void onMessage(Alert alert) {
        ...
    }
}
```

На мгновение представьте более простой мир, где от компонентов, управляемых сообщениями, не требуется реализовать интерфейс MessageListener. *В этом счастливом мире небо всегда было бы голубым, птицы всегда насвистывали бы вашу любимую песню, и вам не пришлось бы писать реализацию метода onMessage() или внедрять компонент MessageDrivenContext. (Оставим этот замечательный комментарий автора книги)*

Ладно, требования спецификации EJB 3, предъявляемые к MDB, возможно, не самые сложные. Но сам факт, что реализация AlertHandler оказывается слишком тесно связанной с API спецификации EJB и далека от POJO, весьма неприятен. В идеале хотелось бы, чтобы обработчик извещений был способен обрабатывать сообщения, но был реализован так, чтобы не иметь тесной зависимости от конкретного API.

Фреймворк Spring обеспечивает возможность задействовать метод простого объекта POJO для обработки сообщений, извлекаемых из очереди или

из темы JMS. Например, в листинге ниже представлена POJO-реализация класса AlertHandler, которой вполне достаточно для нужд приложения.

```
public class AlertHandler {
        public void processAlert(Alert alert ) { // Метод-обработчик
                // ...здесь находится реализация метода...
        }
}
```

Хотя управление цветом неба и дрессировка птиц вне компетенции фреймворка Spring, тем не менее строки выше демонстрируют, что сказочный мир, описанный выше, не так уж и далек от реальности. Мы еще вернемся к реализации метода processAlert(). А пока обратите внимание, что в классе AlertHandler нет ничего, что говорило бы о связи с JMS. Это самый простой, самый обычный объект POJO во всех смыслах этого слова. Однако он может обрабатывать сообщения, подобно своим собратьям, компонентам EJB. Все, что ему требуется для работы, – это некоторые специальные настройки в конфигурационном файле Spring.

4.2. Настройка обработчиков сообщений

Вся хитрость наделения объекта POJO возможностью получать сообщения заключается в настройке его как обработчика. Пространство имен jms в Spring содержит все необходимое для этого. Сначала обработчик необходимо объявить компонентом:

```
<bean id="alertHandler" class="ru.sfu.students.AlertHandler" />
```

Затем, чтобы превратить AlertHandler в объект POJO, управляемый сообщениями, этот компонент можно объявить обработчиком сообщений:

```
<jms:listener-container connection-factory="connectionFactory">
        <jms:listener destination="students.alert.queue"
                ref="alertHandler" method="processAlert" />
</jms:listener-container>
```

Здесь, внутри контейнера обработчиков сообщений, объявляется обработчик сообщений. *Контейнер обработчиков сообщений* – это специальный компонент, просматривающий приемник JMS в ожидании поступления сообщений. Как только появится новое сообщение, он тут же извлечет его и передаст любому обработчику сообщений, заинтересованному в этом сообщении. Этот порядок взаимодействий изображен на рисунке 7.

Для настройки контейнеров и обработчиков в Spring используются два элемента из пространства имен jms. Элемент <jms:listenercontainer> применяется для включения элементов <jms:listener>.

Рисунок 7 — Контейнер обработчика сообщений в очереди/теме. При появлении сообщения он передает его обработчику (такому как MDP

В этом примере в его атрибуте connectionFactory указывается ссылка на компонент connectionFactory, который будет использоваться всеми вложенными элементами <jms:listener> при приеме сообщений. В данном случае атрибут connectionFactory можно было бы опустить, поскольку здесь он получает значение connectionFactory, предусмотренное по умолчанию.

Элемент <jms:listener> используется для идентификации компонента и его метода, который будет вызываться для обработки входящих сообщений. Чтобы обеспечить обработку извещений о новых сообщениях, в атрибут ref элемента записана ссылка на наш компонент alertHandler. Когда в очереди students.alert.queue (определяется атрибутом destination) появится новое сообщение, будет вызван метод processAlert() компонента alertHandler (согласно атрибуту method).

5. В заключение

Асинхронные механизмы обмена сообщениями имеют ряд преимуществ перед синхронными механизмами RPC. Косвенные взаимодействия ослабляют связь между приложениями и тем самым снижают влияние перерывов в обслуживании на работу клиентских приложений. Кроме того, благодаря тому что сообщения доставляются получателям с помощью посредника, отправителю не приходится ждать ответа. Во многих случаях это может существенно повысить производительность приложения.

Несмотря на то что JMS предоставляет стандартный API, доступный для любых Java-приложений, желающих участвовать в асинхронных взаимодействиях, его использование может оказаться весьма утомительным занятием. Фреймворк Spring устраняет необходимость писать шаблонный код, необходимый для работы с механизмом JMS, и существенно упрощает реализацию асинхронного обмена сообщениями.

В этой главе мы познакомились с несколькими способами организации асинхронных взаимодействий между двумя приложениями, осуществляемых с помощью Spring посредством брокеров сообщений и JMS. Шаблон JMS в Spring позволяет избежать шаблонного кода, который обычно приходится писать при использовании традиционной программной модели JMS. А управляемые сообщениями компоненты в Spring дают возможность объявлять методы, вызываемые автоматически при появлении новых сообщений в очереди или в теме.