Обмен сообщениями в Spring

Составил: Черниговский А.С.
Старший преподаватель кафедры “Информатика”
ИКИТ СФУ

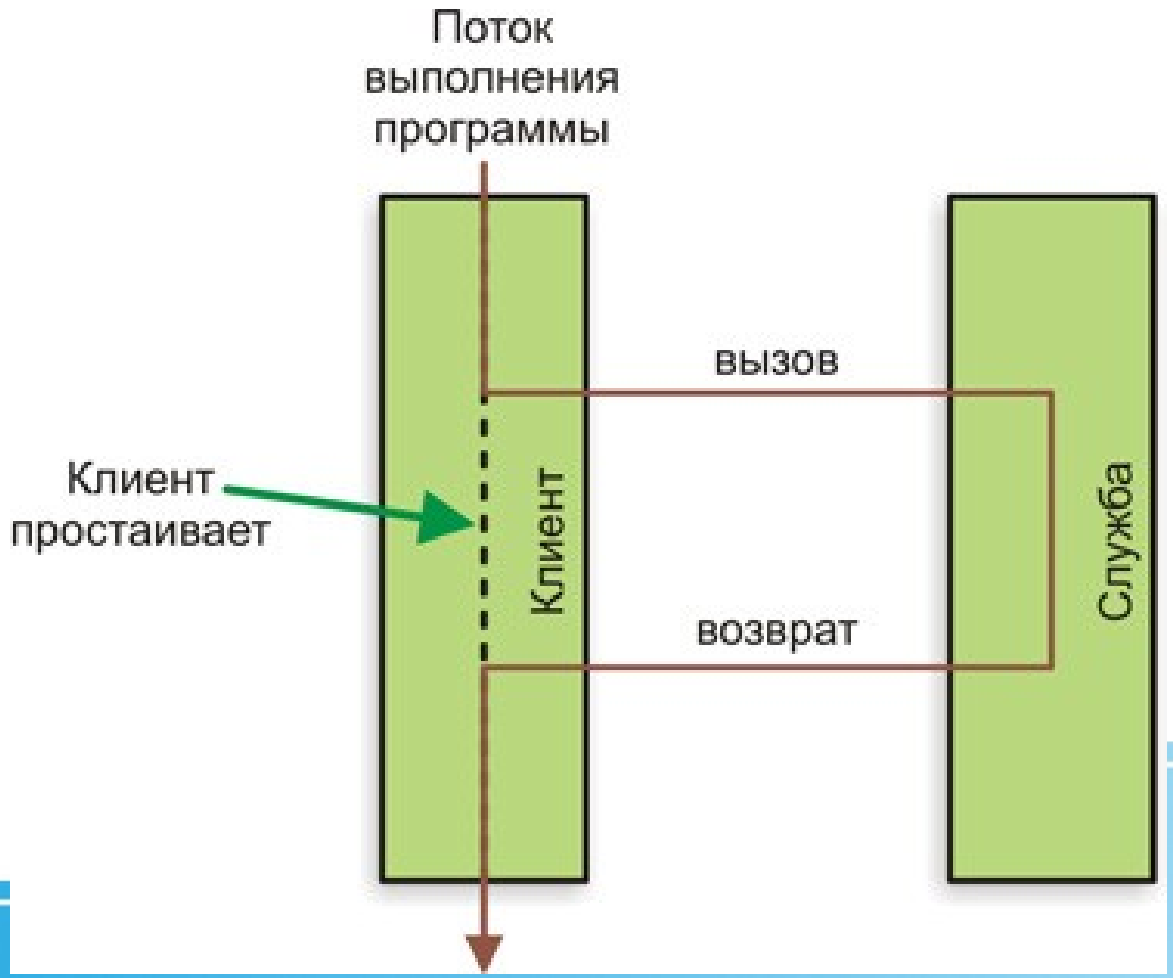
Предисловие

- Механизм взаимодействия между приложениями посредством веб-служб имеет синхронную природу, когда клиентское приложение напрямую взаимодействует с удаленной службой и ожидает, пока удаленная процедура не выполнится, прежде чем продолжить работу.
- Асинхронный обмен сообщениями позволяет отправлять сообщения из одного приложения в другое, не ожидая ответа.
- Java Message Service (JMS) – стандартный API для организации асинхронного обмена сообщениями.

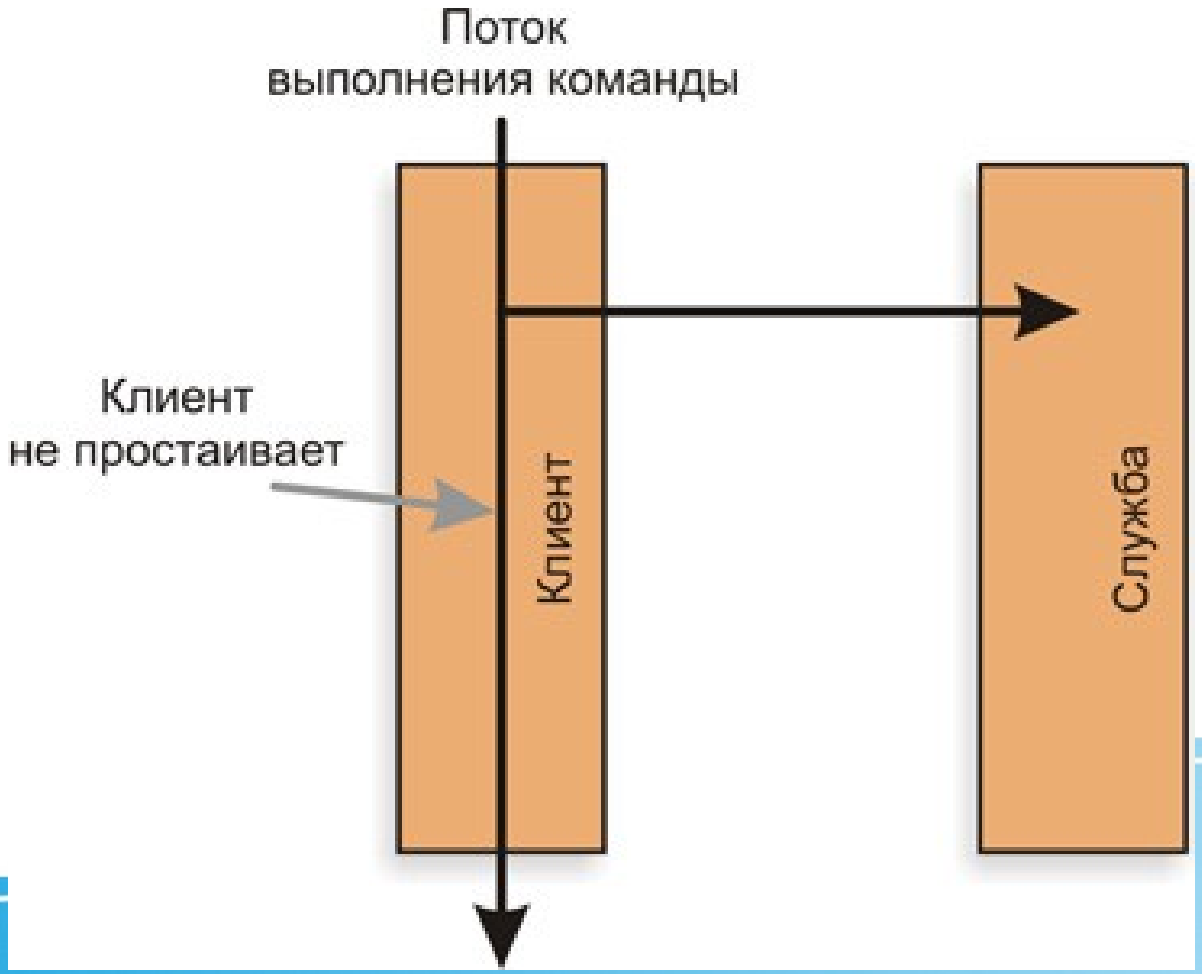
JMS

- 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.

Синхронные взаимодействия



Асинхронные взаимодействия



Архитектура 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. В JMS также имеется понятие адресов, которые определяют приемники. Приемники – это своего рода почтовые ящики, куда будут помещаться сообщения.

Приемники сообщений

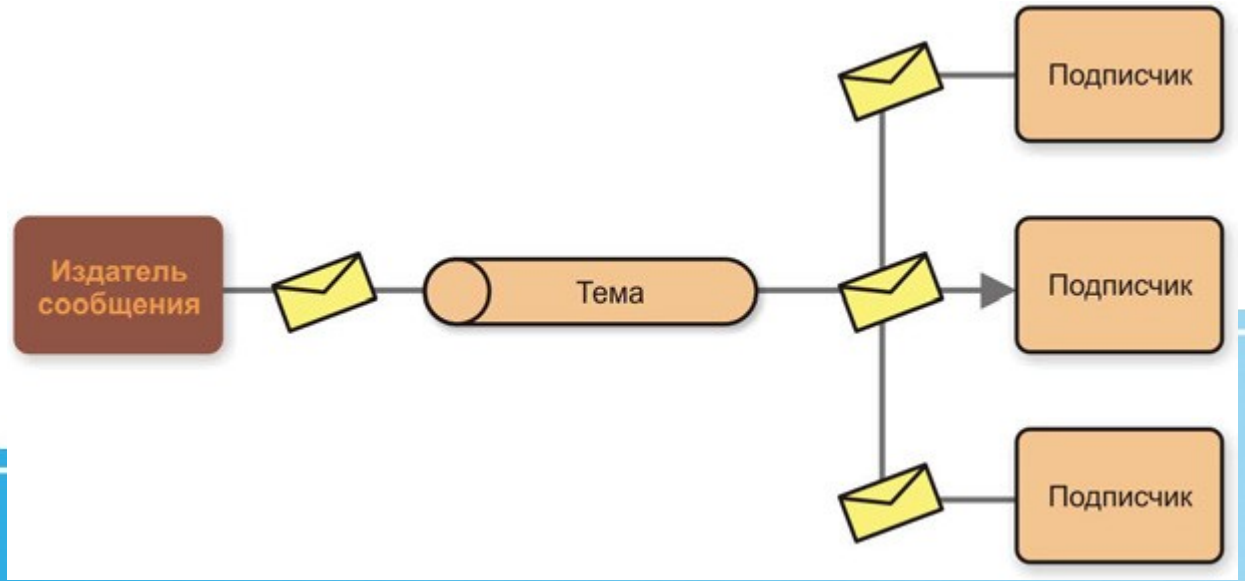
- Адреса в JMS определяют лишь место, где получатель сможет получить сообщение, но не определяют, кто сможет получить его.
- В JMS существуют два типа приемников: очереди и темы . Каждый из них следует определенной модели обмена сообщениями: либо модель «точка–точка» (очереди), либо модель «публикация–подписка» (темы).

Модель точка-точка

- В модели «точка–точка» каждое сообщение имеет точно одного получателя и одного отправителя. 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.

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.



Ограничения на синхронные удаленные службы

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

Преимущества JMS

- Отсутствие ожидания
- Ориентированность на сообщения и независимость
- Независимость от местоположения
- Гарантированная доставка

Брокер сообщений

- ActiveMQ
- http://activemq.apache.org
- org.apache.activemq

Создание фабрики соединений

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

Пространство имен ActiveMQ

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

Объявление приемников ActiveMQ

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

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

Шаблоны JMS

- 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.

Борьба с разбуханием JMS-кода

```
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) {
        }
}
```

Прием сообщений 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) {
        }
}
```

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 .

Внедрение шаблона JMS

```
<bean id="jmsTemplate"

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

Отправка сообщений

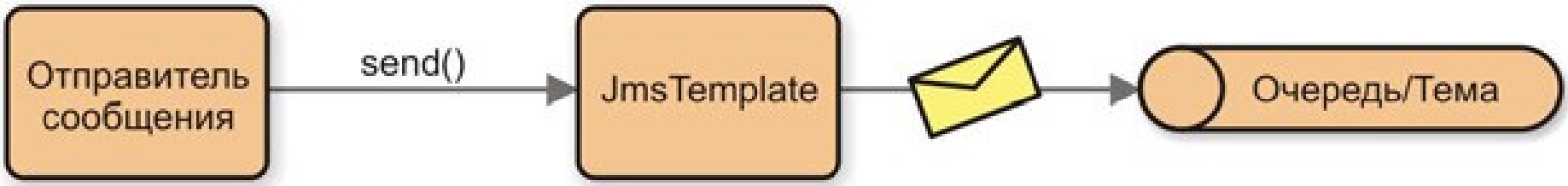
```
<bean id="jmsTemplate"
      class="org.springframework.jms.core.JmsTemplate">
      <property name="connectionFactory"
            ref="connectionFactory" />
 </bean>
```

Отправка сообщений

```
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;
}
```

JmsTemplate



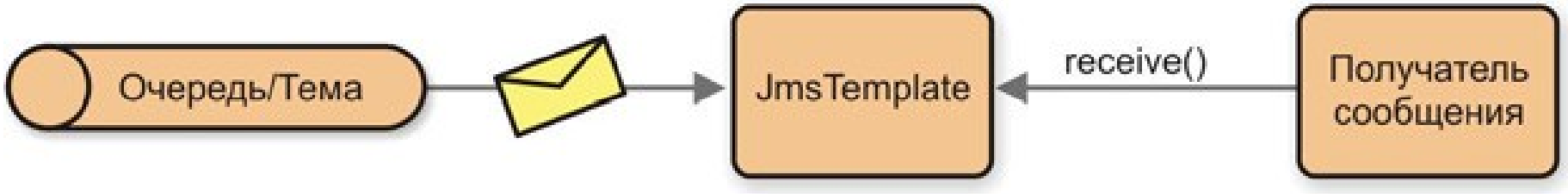
Настройка приемника по умолчанию

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

Прием сообщений с помощью 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);
}
}
```

Прием сообщений с помощью JMS



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.

Создание объекта для чтения сообщений

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

Создание объекта для чтения сообщений

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

Настройка обработчиков сообщений

- <bean id="alertHandler" class="ru.sfu.students.AlertHandler" />
- <jms:listener-container connection-factory="connectionFactory">

  <jms:listener destination="students.alert.queue"

  ref="alertHandler" method="processAlert" />

  </jms:listener-container>

Контейнер обработчика сообщений

Спасибо за внимание!