JAVA MESSAGE SERVICE И MESSAGE-DRIVEN BEANS

Бины, управляемые сообщениями, message-driven beans, чаще всего реализуют технологию JMS, Java Message Service. В этой лабораторной работе мы также будем использовать JMS-технологию, поэтому сначала ознакомимся с базовыми концепциями.

Введение в Java Message Service(JMS) API

Messaging – это метод взаимодействия между программными компонентами или приложениями. Клиент отправляет и получает сообщения.

Messaging делает возможным распределенное взаимодействие между слабо связанными компонентами. Компонент отправляет сообщение в пункт назначения, и получатель может забрать оттуда это сообщение. Однако *отправитель и получатель не обязательно могут быть доступны в одно и то же время*. Ни отправителю ничего не надо знать о получателе, ни получателю – об отправителе. Единственное, что должны знать обе стороны – это ***формат сообщения*** и ***его месторасположение*** (*destination*). В этом смысле, messaging отличается от такой технологии, как RMI, которая требует от приложения вызова метода по его конкретному имени.

Messaging отличается и от электронной почты (e-mail), которая является средством взаимодействия программных продуктов и людей. Messaging же используется для коммуникации между программными приложениями и программными компонентами.

Java Message Service – это Java API, которые позволяют приложениям создавать, отправлять, получать и читать сообщения. JMS API определяет общий набор интерфейсов, который позволяет программам на Java взаимодействовать с другими реализациями messaging.

Коммуникация бывает

- Асинхронной (*asynchronous*). JMS провайдер доставляет сообщения клиенту по мере того, как они будут приходить; клиенту не приходиться запрашивать сообщения, для того чтобы их получить.
- Надежной (*reliable*). JMS API необходимо удостовериться, что сообщение доставлено единожды. Приложениям, которые могут пропустить сообщения или получать дублируемые сообщения, доступен более низкий уровень надежности.

Почему лицо, ответственное за архитектуру корпоративного приложения, выберет скорее messaging API, нежели более тесно связанное API (например, RPC)? Причины следующие:

- Архитектор захочет, чтобы компоненты не зависели от информации о других компонентах. Возможность заменяемости компонент – важное свойство.
- Архитектор захочет, чтобы приложение запускалось, даже если даже работа над некоторыми компонентами не завершена.
- Бизнес-модель приложения позволяет компонентам отправить информацию другим компонентам и продолжить выполнение действий без получения немедленного ответа.

JMS API в составе Java EE обладают следующими особенностями:

- Клиенты, компоненты EJB, web-компоненты могут отправить или синхронно получить JMS сообщения. Клиенты могут вдобавок получать JMS сообщения асинхронно (кроме апплетов).
- Бины, управляемые сообщениями, могут асинхронно получать сообщения. Можно также обеспечить параллельную обработку сообщения MD-бинами.
- Операции отправки и получения сообщений могут участвовать в распределенных транзакциях, которые позволяют проходить JMS операциям и операциям доступа к БД в единой транзакции.

JMS приложение состоит из следующих частей:

1) **JMS провайдер** – это система, управляющая сообщениями, которая реализовывает JMS API и обеспечивает исполнение административных и управленческих задач. JMS провайдер включен в реализацию платформы Java EE.
2) **JMS клиенты** – это программы или компоненты, написанные на Java, которые производят и потребляют сообщения. Любой компонент приложения Java EE может выступать в качестве клиента JMS.
3) **Сообщения** – это объекты, являющиеся носителями информации.
4) **Администрируемые объекты** – это отконфигурированные JMS объекты, создаваемые администратором для использования клиентами. Пункты назначения (*destinations*) и фабрики соединений (*connection factories*) – это администрируемые объекты.

На рис.1 представлено, как эти части взаимодействуют. Административные средства позволяют вам связать пункты назначения (D) и фабрики соединений (CF) в пространстве имен JNDI. JMS клиент может затем использовать включение ресурсов для доступа к администрируемым объектам в данном пространстве имен и установить логическое соединение с теми же самыми объектами с помощью JMS провайдера.

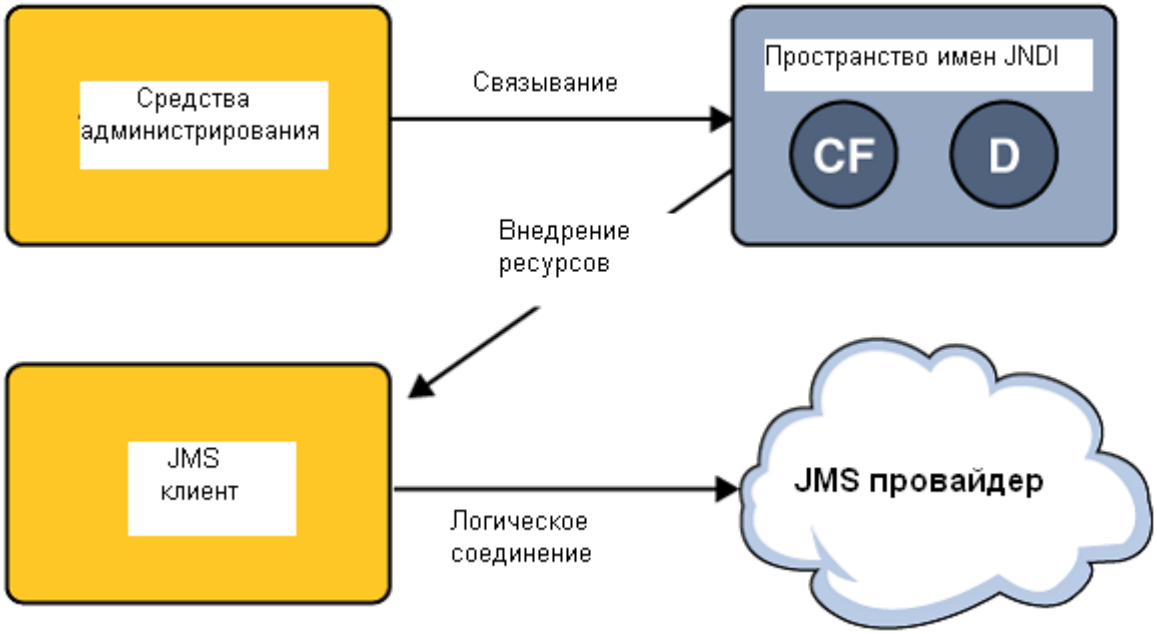


Рис.1. Архитектура JMS API

Перед появлением JMS API большинство продуктов, работающих с сообщениями, поддерживали либо *point-to-point*, либо *publish/subscribe* подходы. Спецификация JMS обеспечивает каждому подходу отдельный домен и определяет совместимость для каждого домена. JMS провайдер в Java EE должен реализовывать оба домена.

Point-to-point.

Подход PTP базируется на концепции очередей сообщений, отправителей и получателей. Каждое сообщение адресуется в определенную очередь, и получающие их клиенты извлекают их из соответствующих очередей. Очереди содержат все отправленные в них сообщения, пока они не будут забраны, либо пока не истечет срок хранения.

Характеристики PTP messaging следующие (рис.2.):

- Каждое сообщение имеет только одного потребителя.

- Отправитель и получатель сообщения не зависят от времени. Получатель может достать сообщение независимо от того, был ли он запущен во время отправки сообщения.
- Получатель узнает об успешной обработке сообщения.

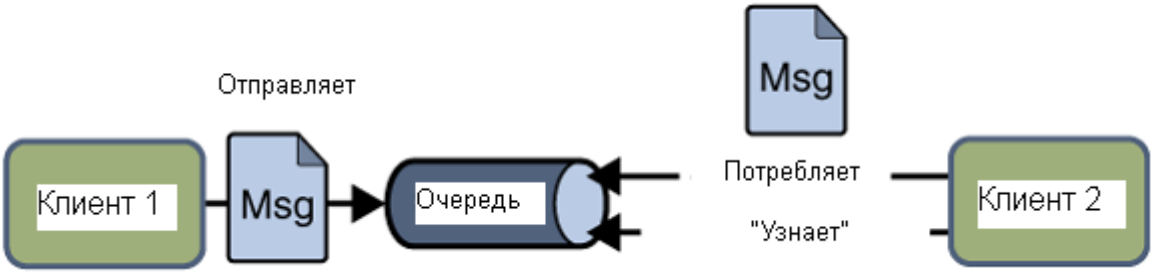


Рис.2. PTP Messaging

Publish/subscribe.

В этом случае клиент адресует сообщение в *topic* (набор тем), функция которого напоминает доски объявлений. Публикаторы и подписчики обычно анонимны и могут динамически публиковать и подписываться на сообщения в пределах иерархии содержания топика. Система заботится о распределении сообщений, прибывших от множества публикаторов, множеству подписчиков. Топики содержат сообщения столько времени, сколько требуется на распределение их подписчикам.

Характеристики (рис.3):

- Каждое сообщение может иметь множество потребителей.
- Публикаторы и подписчики зависят от времени. Клиент, который подписывается на топик, может потребить только то сообщение, которое пришло после его подписки. Для потребления сообщений подписчик должен оставаться активным.

JMS API ослабляет эту временную зависимость, предусматривая создание подписчиками длительных подписок (*durable subscriptions*), которые позволяют получать сообщения, отправленные в то время, когда подписчики были не активны.

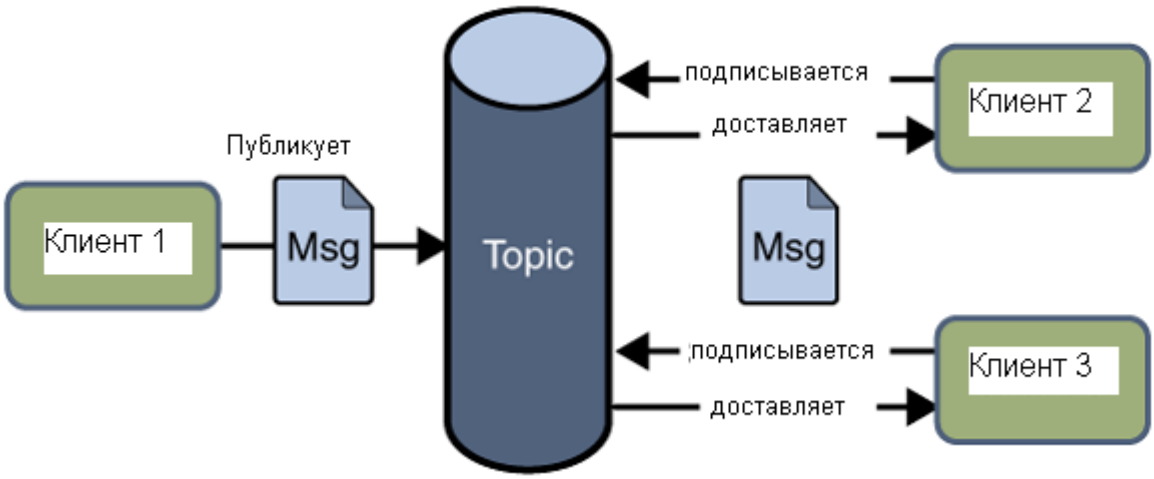


Рис.3. Publish/Subscribe messaging

Сообщения могут потребляться двумя способами:

– **Синхронно**. Подписчик Ии получатель явно забирает сообщение с пункта назначения, вызывая метод *receive*. Этот метод может блокироваться, пока сообщение не придет или не истечет временной лимит.

– **Асинхронно**. Клиент может зарегистрировать у потребителя слушатель сообщений. Как только сообщение приходит на пункт назначения, JMS провайдер доставляет его, вызывая метод *onMessage*, который работает с содержанием сообщения.

Строительные блоки JMS-приложения:

- *Администрируемые объекты.*

1) Destination. Объект, используемый для указания цели для сообщений, что он производит и источник сообщений, которые он потребляет.

2) Connection factory. Объект, используемый клиентом для создания соединения с провайдером.

- *Соединения*

Connection. Виртуальное соединение с JMS-провайдером.

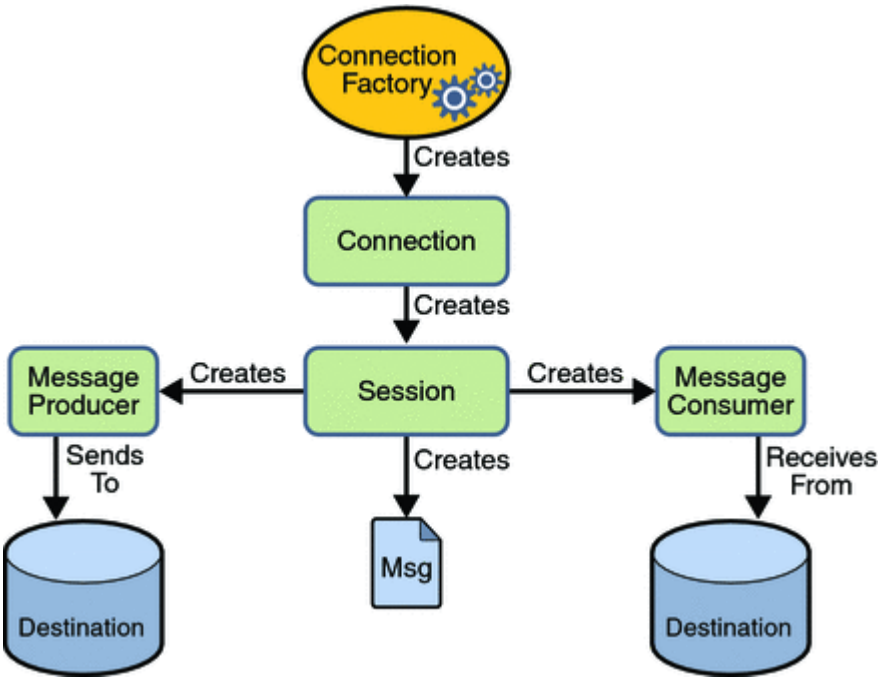
- *Сессии*

Session. Однопоточный контекст для производства и потребления сообщений.

- *Message producers*
- *Message consumers*
- *Message Listeners*
- *Сообщения*

Messages. Состоят из заголовков (обязательно), свойств и тел (опционально).

Модель JMS:



MESSAGE-DRIVEN BEANS

Бин, управляемый сообщениями (далее – *MDB*), – это бин, который позволяет приложению Java EE управлять сообщениями асинхронно. Он обычно выступает в роли слушателя JMS сообщений. Сообщения могут быть отправлены любым Java EE компонентом. MDB может обрабатывать JMS сообщения или другие виды сообщений.

Отличия MDB:

- MDB не сохраняют состояния клиента
- Все MDB равноправны, т.е. EJB контейнер может присвоить сообщение любому бину, управляемому сообщениями.
- Один MDB может обрабатывать сообщения от множества клиентов.

Клиенты не размещают MDB и не вызывают методы бина прямо на нем. Вместо этого, клиент получает доступ к классу бина через JMS, отправляя сообщения в пункт назначения. Присвоение пункта назначения MDB происходит в течение деплоймента с помощью ресурсов сервера приложения.

MDB обладают следующими характеристиками:

1) Они исполняются по единственному клиентскому сообщению
2) Они вызываются асинхронно
3) Они имеют относительно короткое время жизни
4) Они прямо не представляют открытые данные в БД , они могут получить доступ и обновлять эти данные
5) Они могут быть осведомлены о транзакции (“*transaction-aware*”)
6) Они не запоминают своего состояния

Когда приходит сообщение, контейнер вызывает метод *onMessage* для обработки сообщения. Этот метод обычно приводит сообщение к одному из пяти возможных типов JMS сообщения и обрабатывает его в соответствии с бизнес-логикой. Метод *onMessage* может вызывать вспомогательные методы.

Сообщение может быть доставлено бину, управляемому сообщениями, в пределах контекста транзакции, так что все операции внутри метода *onMessage* являются частью единой транзакции. Если обработка сообщения откатывается (выполняется откат – roll back), то и доставка сообщения отменяется.

MDB по сути своей является слушателем сообщений (*message listener*), который может потреблять сообщения из очереди или длительной подписки. Сообщения могут отправляться любым Java EE компонентом, даже приложением, которое не использует технологию Java EE.

Как и слушатель сообщений в автономном JMS клиенте, MDB содержит метод *onMessage*, который вызывается автоматически с приходом сообщения. Особенности MDB-шного слушателя сообщений таковы:

- Некоторые задания выполняются EJB контейнером (см. ниже)
- Класс бина использует аннотацию @MessageDriven, чтобы определить свойства бина или фабрики соединений.

EJB контейнер:

- Создает потребителя сообщений, чтобы он получал сообщения. Вместо создания потребителя сообщений в своем исходном коде, разработчику остается только связать MDB с *destination* и *connection factory* во время деплоймента. При желании можно специфицировать длительную подписку или использовать *message selector* также во время деплоймента.
- Регистрирует слушателя сообщений. Не нужно вызывать *setMessageListener*.
- Специфицирует режим осведомленности. По умолчанию – *AUTO_ACKNOWLEDGE*.

Если JMS интегрировано с сервером приложения с помощью адаптера ресурсов, JMS *resourse adapter* обрабатывает эти задания для EJB контейнера.

Ваш MDB класс должен имплементировать интерфейс *MessageListener* и метод *onMessage*.

Он может имплементировать коллбэк метод @PostConstruct для создания соединения и коллбэк метод @PreDestroy для закрытия соединения. Обычно он реализует эти методы, если прооизводит сообщения или получает синхронно от другого пункта назначения.

MDB не имеет локального либо удаленного интерфейса. MDB – это только сам класс бина.

Чтобы создать экземпляр MDB, контейнер создает экземпляр бина, выполняет внедрение ресурсов, вызывает коллбэк метод @PostConstruct, если тот существует.

На рисунке 1 показан жизненный цикл бина, управляемого сообщениями.

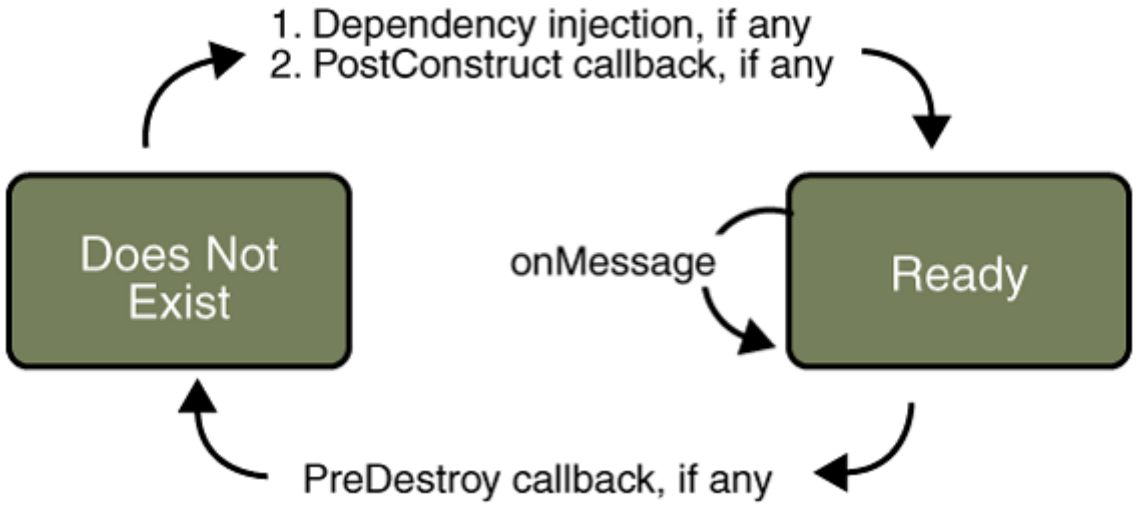


Рис.1. Жизненный цикл бина, управляемого сообщениями

Клиент в Java EE приложении может использовать JMS API таким же образом, что и автономная клиентская программа. Он может производить сообщения и потреблять их, используя либо синхронный механизм либо слушателей сообщений.

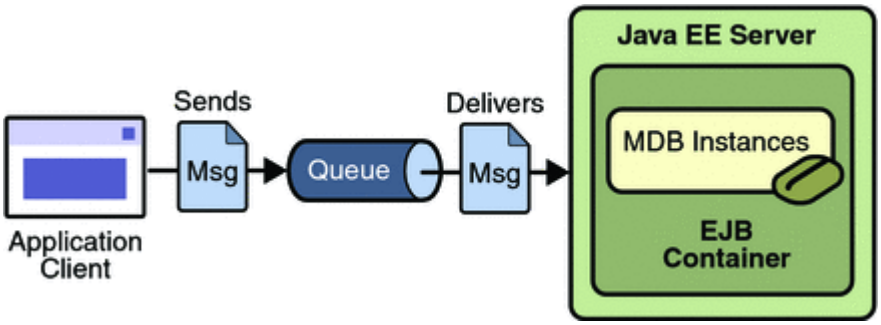
Насчет того как web компонентам лучше использовать JMS, спецификация не накладывает строгих ограничений. На сервере приложений web компонент – единственный тип компонента, использующий либо Java Servlet API, либо технологию JSP – может отправлять и получать сообщения только синхронно.

Т.к. в случае синхронизации использование блокирующего *receive* тратит серверные ресурсы, то считается плохой практикой использование *receive* в web компоненте. Вместо этого лучше использовать синхронное получение с контролем времени (*timed synchronous receive*).

Сделаем упражнение:

Приложение отправляет несколько сообщений в очередь. MDB асинхронно получает и обрабатывает сообщения, которые поступили в очередь.

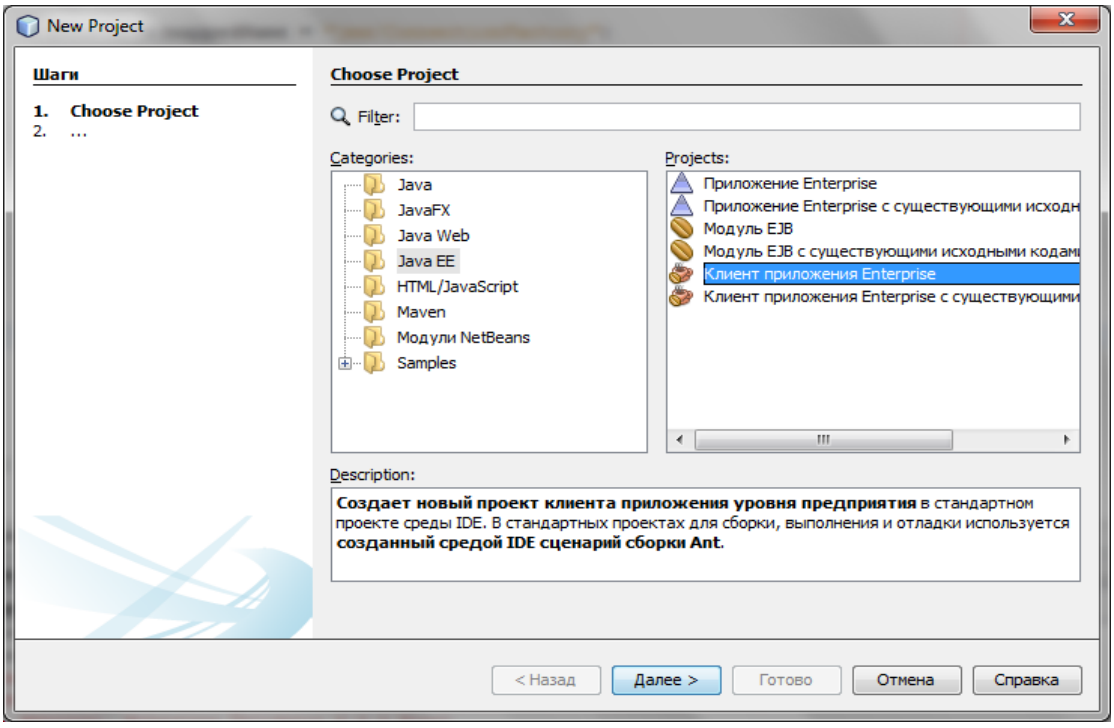
Архитектура приложения нарисована ниже:



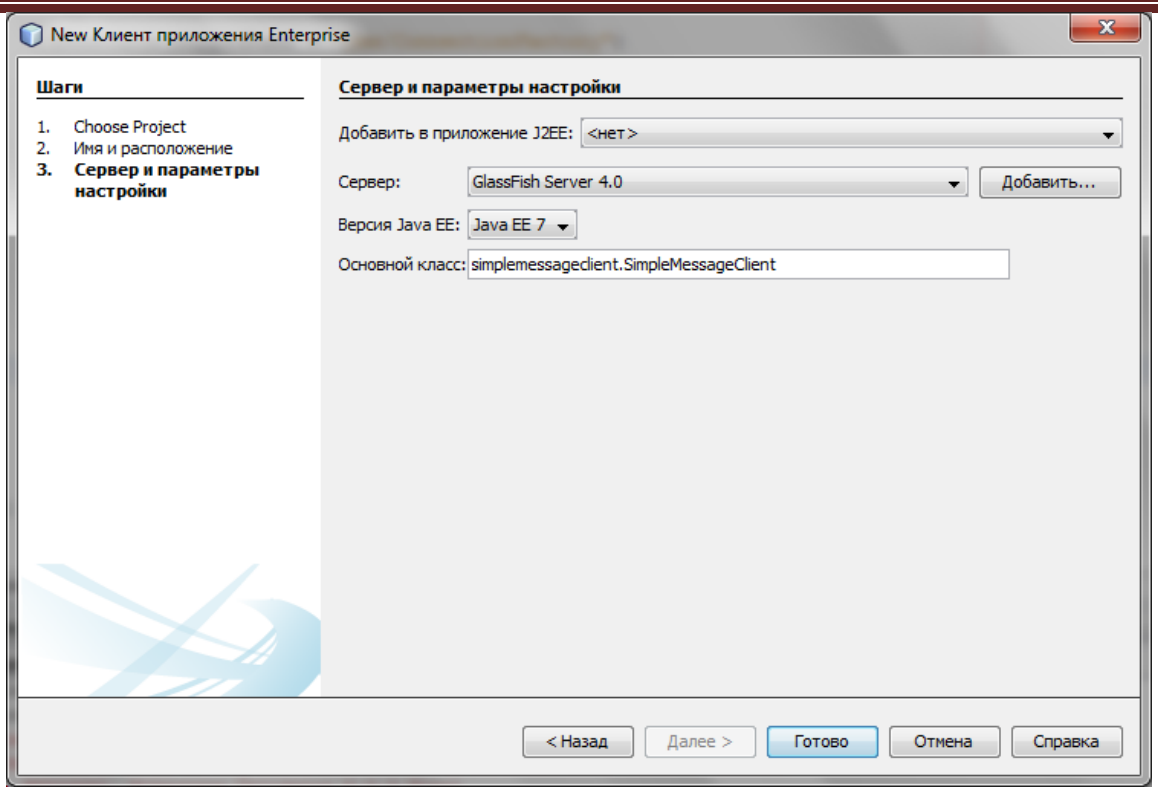
Смотрите на рисунок: клиентское приложение отправляет сообщение в очередь (*queue*), которая была создана в Admin Console (см. в конце пособия). JMS провайдер (у нас в его роли выступает GlassFish) доставляет сообщения экземплярам бина, который уже их и обрабатывает.

Создадим фабрику соединений (connection factory) и пункт назначения (destination) согласно Приложению А.

В NetBeans создаем проект типа Enterprise -> *Enterprise Application Client*.



Класс, содержащий main, назовите SimpleMessageClient .



Он отправляет сообщения в очередь. Код начинается с включения ресурсов как защищенных статических полей класса:

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

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

Затем создаем соединение, сессию и производителя сообщений:

```
Connection connection = connectionFactory.createConnection();
        Session session = connection.createSession(false,
Session.AUTO_ACKNOWLEDGE);
        MessageProducer messageProducer =
session.createProducer(queue);
```

Разумеется, так работать не будет. Надо подключить импорты, добавить обработку исключений.

В конце концов, клиент отправляет несколько сообщений:

```
TextMessage message = session.createTextMessage();
        for (int i = 0; i < 2; i++) {
            message.setText("This is message " + (i + 1));
```

```
                System.out.println("Sending message: " +
message.getText());
                messageProducer.send(message);        }
```

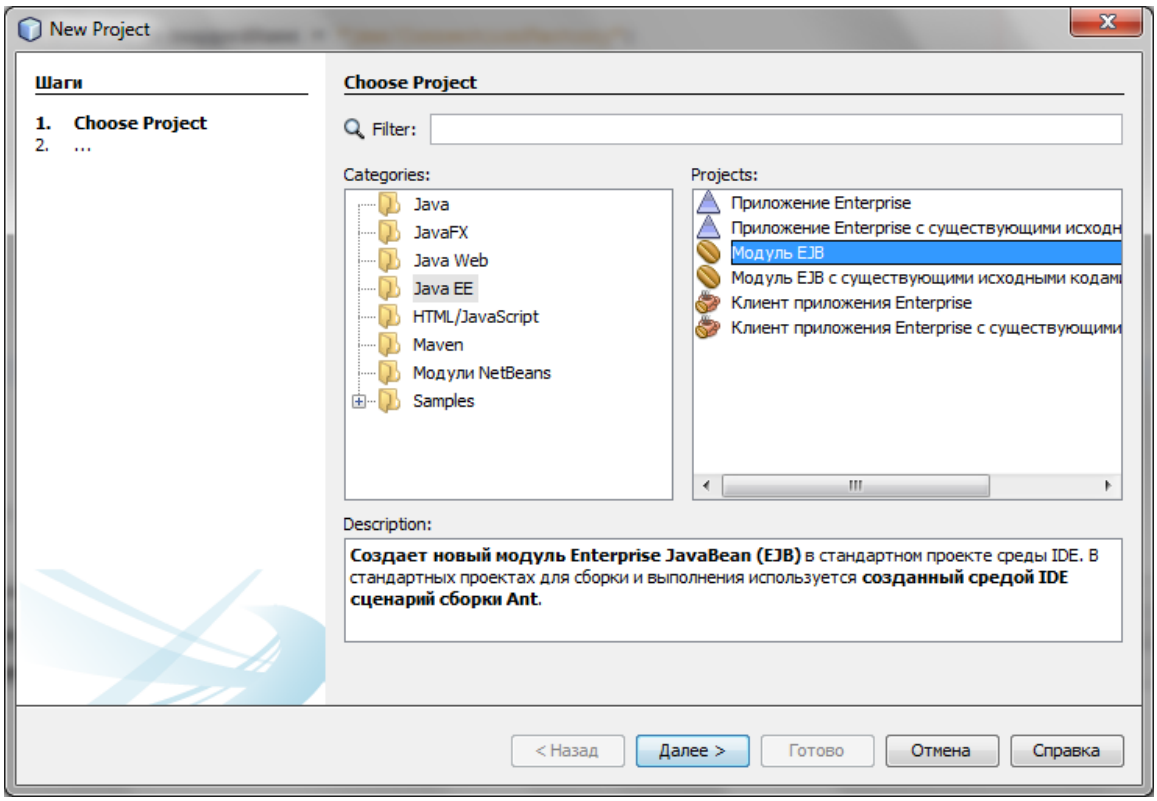
После создания клиентского приложения приступаем к созданию бина.

Класс бина *SimpleMessageBean* иллюстрирует требования, предъявляемые к классу бина MDB:

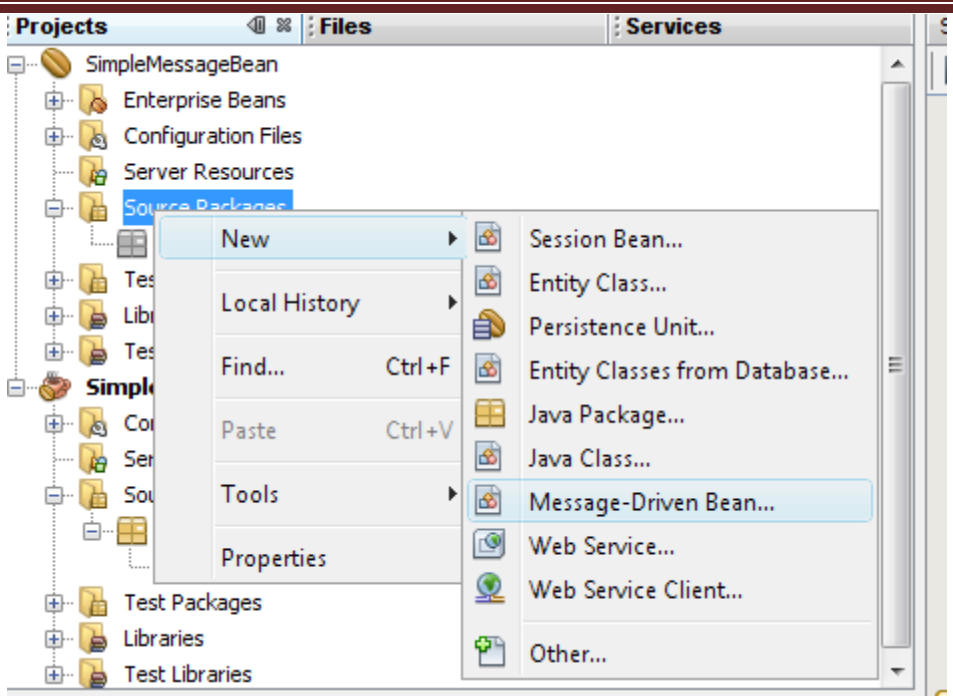
- Он должен реализовывать интерфейс слушателя сообщений.
- Он должен быть предварен аннотацией MessageDriven, если не используется дескриптор развертывания.
- Класс должен быть объявлен как public.
- Он должен содержать конструктор без аргументов.

MDB может также включать ресурс *MessageDrivenContext*. Как правило, этот ресурс используется при обработке исключений.

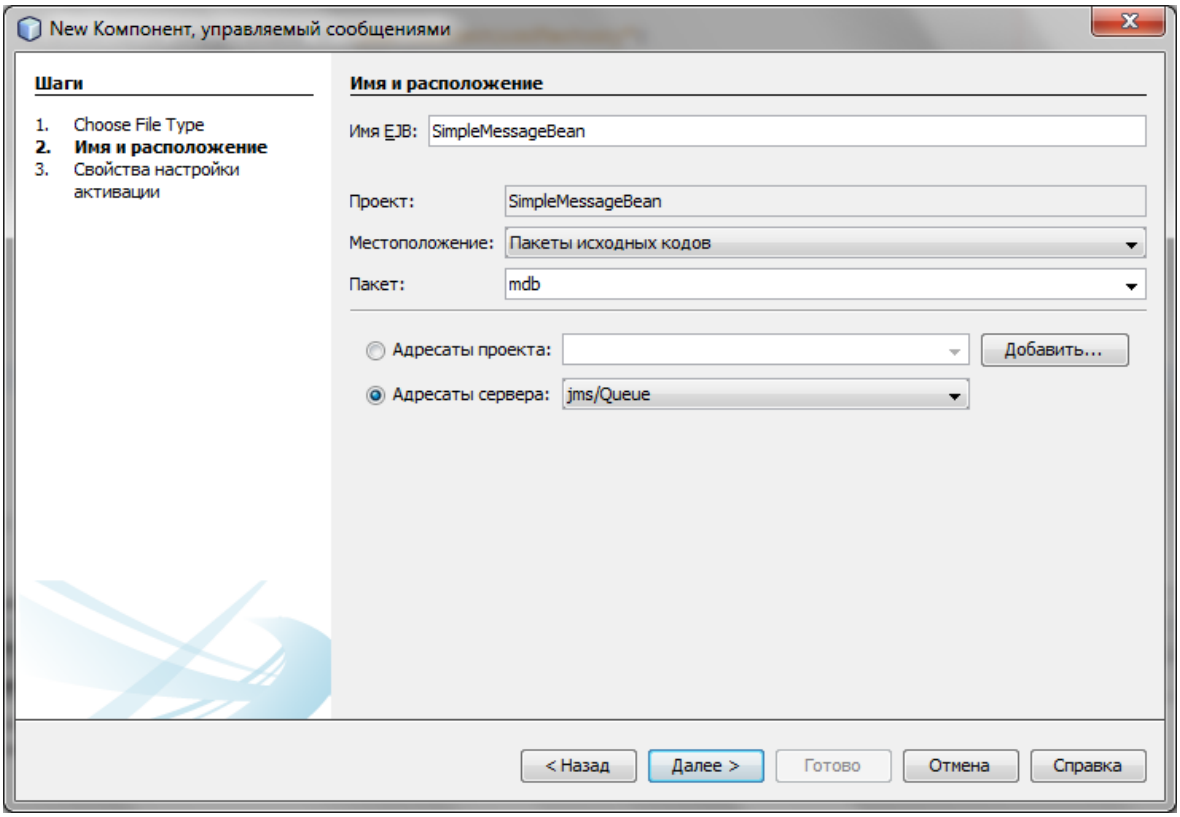
Создаем новый модуль SimpleMessageBean.



А в нем – новый класс бина:



Зададим Server Destination. То есть наша очередь – на сервере, называется она jms/Queue.



Обратимся к методу *onMessage*. Давайте изменим аргумент на inMessage (это делается просто так).

Когда очередь получает сообщение, контейнер вызывает метод слушателя сообщений. Т.к. наш бин использует JMS, метод называется onMessage и он принадлежит интерфейсу MessageListener.

В нем записываем текст:

```
TextMessage msg = null;

        try {
            if (inMessage instanceof TextMessage) {
                msg = (TextMessage) inMessage;
                logger.info("MESSAGE BEAN: Message received: "
                        + msg.getText());
            } else {
                logger.warning("Message of wrong type: "
                        + inMessage.getClass().getName());
            }
        } catch (JMSException e) {
            e.printStackTrace();
            mdc.setRollbackOnly();
        } catch (Throwable te) {
            te.printStackTrace();
        }
    }
```

Также замечаем, что неизвестна переменная logger. Она используется для ведения логов. Объявите такое поле в классе:

```
static final Logger logger = Logger.getLogger("SimpleMessageBean");
```

Для mdc надо внедрить ресурс *MessageDrivenContext*. Напишите перед конструктором:

```
@Resource
private MessageDrivenContext mdc;
```

Импортируем необходимые библиотеки.

На этом все. Развертываем EJB модуль на сервере, запускаем клиента…

Посмотрите что будет в окне output клиента. В логе сервера должно вывестись:

```
MESSAGE BEAN: Message received: This is message 1
MESSAGE BEAN: Message received: This is message 2
MESSAGE BEAN: Message received: This is message 3
```

Следующее задание – самостоятельно (получить у преподавателя)

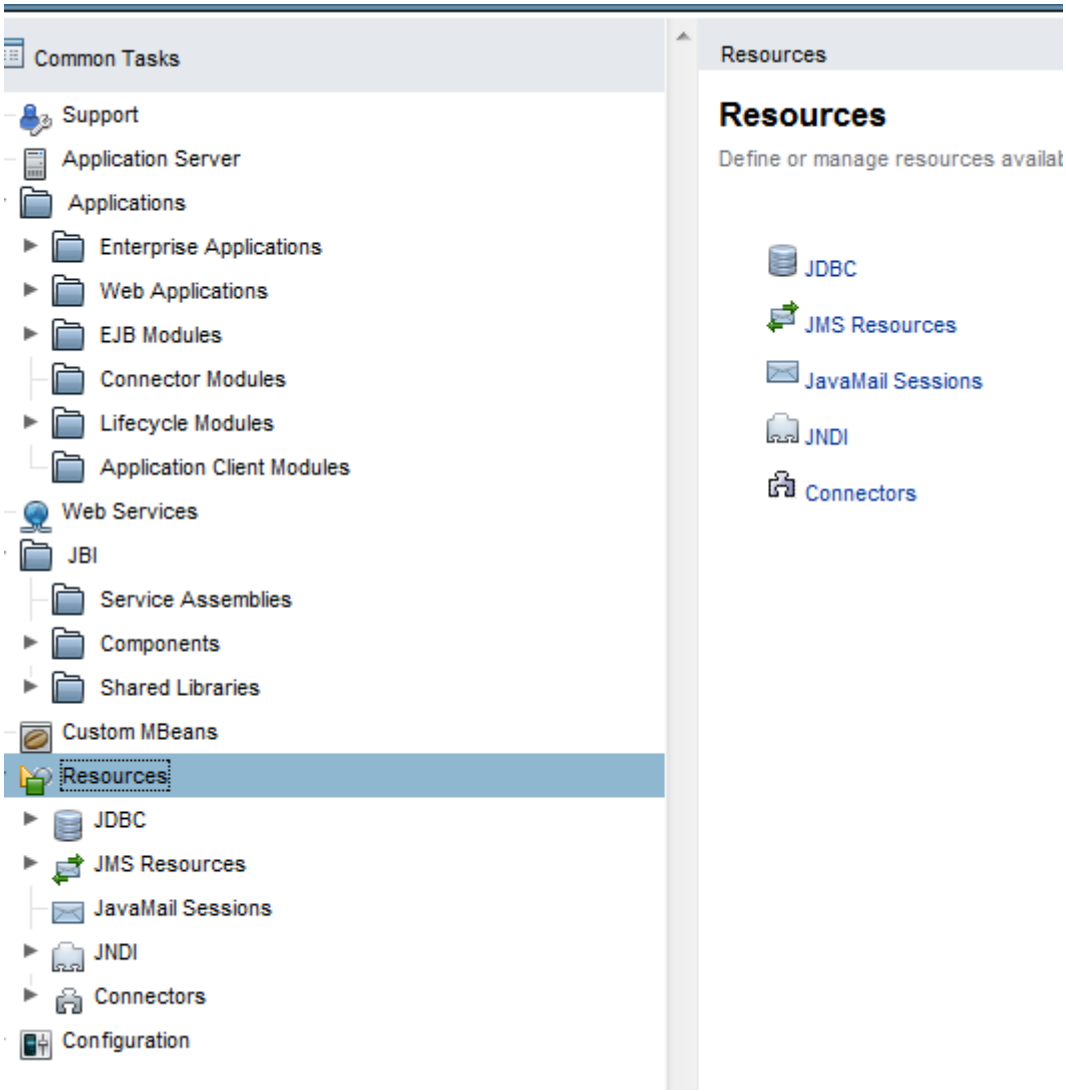
ПРИЛОЖЕНИЕ А

Как создать администрируемые объекты через Admin Console

Нам необходимо создать два ресурса:

- Фабрику соединений (connection factory)
- Пункт назначения (destination)

1) Запустите сервер GlassFish, если он еще не запущен;
2) Через браузер откройте Admin console: localhost:4848;
3) Настроим-ка ресурсы:



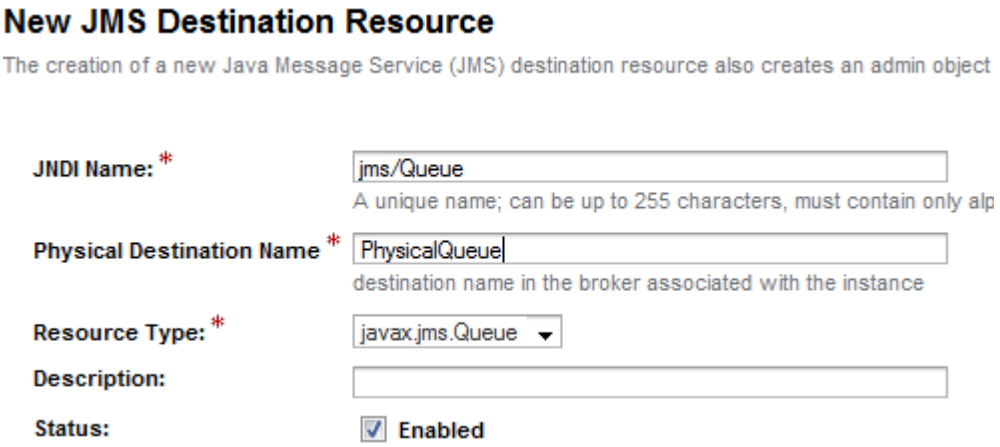
4) Выбираем JMS Resources;
5) Вначале создадим Connection Factory: Connection factories -> New;
6) Заполняем поля со звездочкой, остальное без изменений:

OK.

7) Теперь на дереве слева выбираем узел Destination Resources ->New;

8) Заполняем поля со звездочкой, остальное без изменений:

9)



OK.

Мы создали ресурс-очередь. *Если бы нужен был topic, то вместо queue выбирали бы topic.*

Индивидуальные ЗАДАНИЯ:

Клиентское приложение отправляет несколько (не менее 10) сообщений в QUEUE или TOPIC (нечетный вариант – QUEUE, четный – TOPIC). MDB получает и обрабатывает сообщения следующим образом:

1) Получатель сортирует сообщения в порядке возрастания и таким образом записывает их в текстовый файл.
2) Получатель ищет два одинаковых сообщения и записывает их порядковые номера в файл.
3) Получатель записывает в текстовый файл сообщения с восклицательным знаком.
4) Получатель выводит в консоль сообщения длиной не менее четырех символов отсортированные по алфавиту.
5) Получатель производит поиск сообщения (искать в файле).
6) Получатель записывает приходящие цифры в один файл, буквы – в другой.
7) Получатель записывает полученные сообщения в файл задом наперед.
8) Получатель считает среднюю оценку по трем предметам и записывает её в файл.
9) Получатель сортирует сообщения в порядке возрастания (по длине сообщения) и таким образом записывает их в текстовый файл.
10) Получатель записывает в файл 2 сообщения, которые имеют наименьшее и наибольшее число гласных.
11) Получатель удаляет из файла полученные сообщение.
12) Получатель выводит количество сообщений, содержащих $$.
13) Получатель находит сообщения в файле и заменяет их на *. (количество * равно количеству букв в сообщении, исключая краевые пробелы)

Вопросы для защиты лабораторной работы

1) Что такое Java Message Service (JMS)?
2) JMS приложение состоит из следующих частей: JMS провайдер, JMS клиенты, Сообщения, Администрируемые объекты. Дайте определение этим понятиям.
3) Отличие messaging от RMI?
4) Концепция Point-to-point. Что используется в качестве пункта назначения (destinations) при данном подходе?
5) Концепция Publish/subscribe. Что используется в качестве пункта назначения (destinations) при данном подходе?
6) Что такое Message Driven Bean (MDB)? Какой интерфейс должен реализовывать класс бина? С помощью какого метода бин обрабатывает входящие сообщения?
7) В чем их отличие от сессионных бинов?
8) Для чего служит метод onMessage?
9) В чем преимущество использования MDB для организации обмена сообщениями?
10) Каковы функции слушателя сообщений? В чем сходства и различия его со слушателем событий (event listener)?
11) Может ли MDB взаимодействовать с БД?