Оглавление

Общее описание технологии EJB

EJB – Enterprise Java Beans (корпоративные Java-компоненты). Это серверные компоненты, которые согласно идеологии JEE должны реализовывать бизнес-логику приложения. Компоненты EJB в приложении должны быть отделены от уровня представления (View), представленного Web-слоем, GUI и т.п. Компоненты EJB в синтаксисе JEE являются глаголами (действиями), тогда как Entity Beans – существительными (объектами).

Rem: при проектировании корпоративных приложений для улучшения модифицируемости, расширяемости и облегчения разработки применяются многоуровневые архитектуры. Популярна классическая 4-уровневая архитектура: представление (Presentation, View) → бизнес-логика (Business logic, Model) → персистентный слой (Persistence) → хранилище (Database). Она хорошо разделяет приложение на слабосвязанные слои, однако, за это приходится платить отходом от принципов ООП: бизнес-логика становится похожа на службу, содержащую набор функций, а персистентный слой представляет собой тупые контейнеры данных. Есть также предметно-ориентированная архитектура (Domain-Driven Design, DDD): представление → прикладная логика (Application/Service logic) → предметная область (Domain) → инфраструктура (Infrastructure). В отличие от традиционной 4-архитектуры, в DDD часть бизнес-логики переезжает на предметный уровень, а оставшаяся часть становится более тонким прикладным уровнем. За счет этого в DDD удается реализовать ООП на предметном уровне. В JEE классикой является традиционная модель, но на предметном уровне в JPA 2 есть все необходимое и для реализации DDD.

Назначение бизнес-слоя, основанного на EJB: реализация бизнес-логики, работающей с серверными ресурсами — с базами данных, многопоточным доступом; взаимодействие со внешними системами, в т.ч. веб-сервисами, асинхронными JMS; формирование транзакционной политики и политики безопасности приложения; реализация точек входа в приложение для различных клиентов от веб-слоя (через сервлеты и подложечные компоненты JSF) до веб-сервисов, JMS, JCA-адаптеров.

Компоненты EJB работают под управлением EJB-контейнера, который наделяет их рядом возможностей при помощи контейнерных служб и встроенной интеграции со внешними технологиями.

История EJB и новшества EJB 3.2

EJB 1.0 появилась в 1998. В 2002 вышла первая стандартизированная (JSR) спека EJB 2.0. В 2003 появилась революционная с т.з. упрощения разработки спецификация EJB 3.0 (JEE5) с концепцией аннотированных POJO-компонентов, JPA, внедрением зависимостей, callback-методами обработки событий жизненного цикла, перехватчиками. В 2009 появилась спека EJB 3.1 (JEE6) с безынтерфейсным представлением EJB, Singleton, встраиваемыми EJB Lite контейнерами, асинхронными вызовами сессионных EJB, переносимыми JNDI-именами, улучшенным планировщиком. В 2013 появилась JEE7 с EJB 3.2 и очередными улучшениями:

- транзакции для callback-методов обработки событий жизненного цикла SFSB
- возможность отключения пассивации SFSB
- упрощение правил определения локальных/удаленных представлений
- разрешение получать экземпляр текущего загрузчика классов (?)
- разрешение использовать java.io (?)
- усовершенствования JMS 2.0
- безынтерфейсные MDB
- поддержка AutoCloseable встраиваемым javax.ejb.embeddable.EJBContainer
- подготовка к отмене требования поддержки протокола IIOP в RMI в JEE8
- распространение транзакций на CDI MBean
- удаление устаревших: EJB 2.x, EJB QL (заменена на JPQL), JAX-RPC (заменена JAX-WS).

API и RI EJB 3.2

Основные соглашения (контракты) между клиентом и EJB-контейнером, а также между EJB-контейнером и EJB-компонентом находятся в пакете javax.ejb. В пакете javax.ejb.embeddable находится API встраивания контейнера EJB Lite. В пакете javax.ejb.spi находится API EJB-контейнера.

Эталонная реализация (Reference Implementation, RI) JEE7 — сервер приложений GlassFish 4.0 (Sun/Oracle).

Общее описание EJB-компонентов

Компонент EJB – совокупность Java-типов (классов и интерфейсов) и метаинформации, работающая в окружении и под управлением EJB-контейнера. С EJB 3.2 (JEE7) EJB-компоненты поддерживают технологию CDI. Т.е. можно считать, что EJB-компоненты являются управляемыми CDI компонентами (CDI MBean) с расширенными возможностями, предоставляемыми спекой EJB.

EJB-компоненты делятся по типу работы на сессионные компоненты и MDB. Сессионный EJB предоставляет клиенту через своего представителя (прокси), называемого также EJB-объектом (EJB Object), бизнес-логику приложения, являясь фактически его серверным продолжением. Сессионные EJB делятся на SLSB (без сохранения состояния), SFSB (с сохранением состояния), Singleton (одиночный экземпляр). MDB выступает в роли слушателя клиентских сообщений, занимаясь их асинхронной обработкой.

Rem: работа сессионных EJB по-умолчанию имеет синхронную природу, но в EJB 3.2 (?) появилась и возможность их асинхронного вызова в отдельном клиентском потоке.

EJB-компоненты начиная с EJB 3.0 состоят из POJO-типов, дополненных метаинформацией. В EJB 3.2 сессионный EJB-компонент состоит из необязательных бизнес-интерфейсов и класса реализации, дополненных метаинформацией посредством аннотаций или необязательного XML-дескриптора ejb-jar.xml.

Сессионный EJB-объект (EJB Object)

EJB-объект — серверный прокси-объект EJB-компонента. Он является связующим звеном между клиентом и некоторым экземпляром (класса реализации) EJB-компонента. Клиент в зависимости от своего типа использует либо локальное, либо удаленное представление EJB-объекта, при помощи которого и взаимодействует с EJB-компонентом. Локальный клиент получает ссылку на (?)сам(?) EJB-объект. Удаленный клиент по протоколу RMI получает заглушечное представление EJB-объекта. Получить клиентское представление EJB-объекта внутри экземпляра EJB-компонента можно из сессионного контекста методом getBusinessObject(...), а извне — посредством JNDI запроса lookup(…) javax.naming.Context или внедрением @EJB, @Inject.

Класс сессионного EJB

Непосредственная реализация бизнес-логики сессионного компонента EJB содержится в POJO-классе, снабженном EJB-метаинформацией и называемом классом EJB. К классам сессионных EJB-компонентов предъявляются следующие требования, позволяющие EJB-контейнеру управлять их экземплярами (путем расширения, создания экземпляров через рефлексию) и организовывать к ним удаленный доступ:

- наличие аннотации @Stateless, @Stateful, @Singleton, либо метаинформации в дескрипторе ejb-jar.xml
- реализация бизнес-интерфейсов, если они существуют
- класс должен быть public и не должен иметь модификаторов final и abstract
- должен существовать публичный конструктор по-умолчанию (без аргументов)
- не должно быть метода финализации finalize()
- все бизнес-методы должны быть public и не должны иметь модификаторы static и final

- аргументы и результат должны быть RMI-совместимыми типами: примитивами, сериализуемыми объектами

На работу классов сессионных EJB-компонентов налагаются следующие ограничения, вызванные делегацией управления ими EJB-контейнеру:

- нельзя создавать или управлять потоками
- нельзя работать с файловой системой посредством java.io (?)снято в EJB 3.2(?)
- нельзя создавать ServerSocket
- нельзя использовать для работы с клиентом API AWT, Swing и т.п.

Представления и интерфейсы сессионных EJB

Клиент EJB работает только с представлениями EJB-компонента. Либо с бизнес-интерфейсами, либо с безынтерфейсным представлением.

- Remote — удаленное представление для вызова бизнес-методов по RMI в виде бизнес-интерфейса. Передача параметров и результата идет в сериализованном виде, т.е. по-значению. Все параметры и результаты методов данного интерфейса должны быть сериализуемыми: быть простыми типами, реализовывать интерфейс-метку java.io.Serializable или же быть помеченными модификатором transient. Удаленный бизнес-интерфейс объявляется при помощи @javax.ejb.Remote в бизнес-интерфейсе или же классе EJB.
- Local — локальное представление для обращений внутри процесса EJB-контейнера в виде бизнес-интерфейса. Передача объектов идет по ссылке. Локальный бизнес-интерфейс объявляется по-умолчанию (без аннотации), либо при помощи @javax.ejb.Local в бизнес-интерфейсе или же классе EJB.
- Безынтерфейсное представление — вариант Local-представления с включением в него всех методов класса EJB. Представляет собой просто класс реализации EJB. Является представлением по-умолчанию для всех классов EJB без @Remote и @Local бизнес-интерфейсов, а также для классов, помеченных @javax.ejb.LocalBean. Допускается внедрение объекта такого класса напрямую.

Rem: в Java все передается по значению, при передаче объекта ссылочного типа создается копия этой ссылки. Но значение, т.е. адрес объекта у копии будет тем же, что и у оригинала. Т.е. обращение через копию будет к тому же объекту, что и через оригинал. Поэтому для упрощения можно считать что объект передается по ссылке.

Rem: интерфейсное представление предпочтительнее в смысле ослабления связанности компонентов.

Удаленный вызов обычно (?) осуществляется клиентом из другого процесса. Это может быть ACC, внешний WEB или EJB контейнер, JSE-приложение.

Локальный вызов доступен внутри процесса EJB-контейнера. Вызывать может другой

EJB или сервлет или JSF на том же JEE-сервере приложений.

Для объявления Local или Remote бизнес-интерфейса достаточно выделить в публичный интерфейс часть методов класса EJB и пометить их @Local либо @Remote, либо перенести аннотацию на класс EJB и указать в ней value = F.Q.N. интерфейса. Последнее удобно при отсутствии исходников интерфейсов.

```
Ex: объявление представлений EJB. Вариант аннотирования класса EJB

public interface ExLocal {…}
public interface ExRemote {…}

@Stateless @Local(ExLocal.class) @Remote(ExRemote.class) @LocalBean
public ExEJB implements ExLocal, ExRemote {…}
```

Rem: в ear-приложениях для избежания возможного конфликта имен Remote-интерфейсов внутри различных jar-модулей, требуется выделять Remote-интерфейсы в отдельный jar-модуль.

Для доступа к SLSB также может применяться представление в виде веб-сервисов: @WebService объявляет SLSB конечной точкой SOAP WS, @Path объявляет ресурс RESTful WS. При этом вместо RMI будет использоваться протокол HTTP.

```
Ex: EJB-представления и WS-представления SLSB. Вариант аннотирования
интерфейсов.

@Local public interface ExLocal {…}
@Remote public interface ExRemote {…}
@WebService public interface ExSOAP {…}
@Path(“/items”) public interface ExREST {…}

@Stateless @LocalBean
public ExEJB implements ExLocal, ExRemote, ExSOAP, ExREST {…}
```

Rem: в отличие от родных EJB-представлений, при вызовах EJB как веб-сервиса по-умолчанию контексты транзакций и безопасности не распространяются.

Компоненты EJB Lite

Компоненты EJB Lite появились в JEE6. Они предназначены для создания WEB-приложений. В JEE7 выделено подмножество EJB API, называемое EJB 3.2 Lite, которое можно запускать во встраиваемом EJB Lite контейнере или на JEE-сервере, реализующем контейнер EJB Lite.

Функция	EJB 3.2 Lite	EJB 3.2
Сессионные EJB-компоненты	+	+

Представление без интерфейса	+	+
Локальный интерфейс	+	+
Перехватчики	+	+
Поддержка транзакций	+	+
Безопасность	+	+
Embeddable API	+	+
Асинхронные вызовы	-	+
MDB	-	+
Удаленный интерфейс	-	+
Веб-службы JAX-WS	-	+
Веб-службы JAX-RS	-	+
TimerService	-	+
Интероперабельность RMI/IIOP	-	+

Rem: вендоры вольны добавлять дополнительный функционал ко встраиваемому EJB Lite контейнеру. Например, встраиваемые EJB-контейнеры GlassFish дополнены до уровня поддержки обычных EJB.

Параллелизм сессионных EJB (Concurrency)

SLSB и SFSB обладают встроенными механизмами распараллеливания запросов. Спецификация требует, чтобы в любой момент времени к любому их экземпляру поступало не более одного запроса. Поскольку каждый запрос живет в отдельном потоке это означает, что никакой дополнительной потокобезопасности SLSB и SFSB не требуется. В случае Singleton все запросы идут параллельно к единственному экземпляру и режим распараллеливания можно задавать.

Окружение (SessionContext) сессионных EJB

Интерфейс javax.ejb.SessionContext, наследующий базовый javax.ejb.EJBContext — интерфейс, позволяющий EJB-компоненту программно (напрямую) связаться с EJB-контейнером в run-time. Методы контекста снабжают объекты EJB во время их выполнения доступом к контейнерным службам (JNDI, таймеров, безопасности, транзакций). В класс EJB контекст может внедряться при помощи @Resource:

```
@Resource
private SessionContext ctx;
```

Общее описание EJB-контейнера

Управлением работой бизнес-слоя EJB занимается EJB-контейнер. Это среда

выполнения EJB-компонентов, снабжающая их набором различных служб и автоматизирующая их работу.

В EJB 3.2 присутствуют следующие EJB-контейнерные службы:

- JNDI / RMI – удаленный вызов, работа с деревом удаленных ресурсов
- Injection – внедрение зависимостей, поддержка CDI
- Поддержка состояния для SFSB
- Поддержка пула для SLSB
- Управление жизненным циклом EJB-компонентов
- JTA/JTS - поддержка транзакций
- Поддержка безопасности (JAAS, JASPIC, JACC)
- Поддержка параллелизма и потокобезопасности, полностью автоматическая для SFSB и SLSB, настраиваемая для Singleton
- Interceptors – перехват
- Timers – планирование
- Поддержка асинхронных вызовов

Также EJB-контейнер обеспечивает EJB-компоненты технологиями:

- организации конечных точек SOAP WS и ресурсов RESTful WS
- обмена сообщениями (JMS)
- JPA – работы с персистентным слоем

Кроме того EJB-компонентам программно доступны технологии Java SE и Java EE вроде JDBC, JavaMail.

Полноценный EJB-контейнер запускается в отдельном процессе (экземпляре JVM).

Начиная с EJB 3.1 (JEE6) также появились встраиваемые EJB-контейнеры с усеченным функционалом EJB Lite, которые можно запускать программно, в JSE-процессе с тем же загрузчиком классов при помощи API javax.ejb.embeddable. Встраиваемый контейнер упрощает и ускоряет модульное и интеграционное тестирования в одном процессе IDE, также позволяет использовать EJB Lite компоненты в standalone-приложениях.

EJB-контейнер может взаимодействовать с:

- WEB-контейнер (сервлеты, jsp, jsf)
- ACC – Application Client Container (толстые standalone клиенты)
- Entity Manager (персистентный слой)
- JMS-поставщикы (система обмена сообщениями (?))

Кластеризация EJB-контейнеров

Кластеризация — технология распространения экземпляров компонентов по кластеру

серверов (процессов), позволяющая распределять нагрузку на приложение. Применение кластеризации к компоненту с сохранением состояния приводит к копированию каждого его экезмпляра с синхронизацией состояния на все сервера кластера. Диспетчер может выбрать наименее загруженный узел (сервер) или переадресовать запрос с аварийного узла на целый, поскольку у всех копий экземпляра компонента будет свежее актуальное состояние. Кластеризация является де-факто стандартом реализации EJB-контейнеров, хотя в спеке EJB и не прописана.

Упаковка EJB-приложения

Контейнеры JEE-сервера требуют предварительной упаковки приложения в архивы специального вида. Существует 3 вида архивов JEE:

- jar – Java-архив, содержащий обязательный каталог META-INF. Он используется для упаковки EJB-компонентов, JSE-библиотек.
- war – WEB-архив. Используется для упаковки сервлетов, jsp и jsf страниц, подложечных MBean и вспомогательных POJO WEB-приложений, работающих в WEB-контейнере. В нем содержится обязательный каталог WEB-INF. Начиная с EJB 3.1 (JEE6) в war-архив можно упаковывать и компоненты EJB-Lite.
- ear – Enterprise-архив. Используется для упаковки сложных JEE-приложений, содержащих множество jar и war архивов.

Rem: при упаковке EJB-компонента в jar-архив, опциональный дескриптор развертки ejb-jar.xml помещается в META-INF, а при упаковке EJB-Lite компонента в war-архив — в WEB-INF.

Развертка (deployment) EJB-приложения

Запуск EJB-приложения возможен только в EJB-контейнере. Сначала запускается процесс контейнера EJB, затем производится установка EJB-компонентов из jar или ear архивов. При этом происходит распаковка и загрузка классов прямо в RAM (перманентную область — permgen или metaspace) загрузчиком (class loader) EJB-контейнера, чтение XML-дескрипторов, заполнение JNDI-дерева из аннотаций и дескрипторов, трудолюбивое создание экземпляров, пулов и т.п. подготовительные действия. Все это называется установкой или разверткой (deployment) EJB-компонентов. Каждому jar-архиву в развернутом виде соответствует свой модуль, свое пространство имен в Global JNDI и свой контекст CDI. После развертки клиент может запрашивать приложение.

При отладке после каждого изменения надо перезапускать контейнерный процесс, что утомляет. В EJB 3.1 появились встраиваемые контейнеры EJB Lite. Они позволяют запускать контейнер в JSE-процессе, например в том же, где идет разработка и отладка, что ускоряет работу.

Rem: в WEB-приложениях можно в горячем режиме менять JSP-странички, поскольку

транслирующий-компилирующий их системный JSP-сервлет, каждый раз проверяет XML-исходник и при необходимости пересобирает его в новый сервлет.

```
Ex: пример использования встраиваемого EJB Lite контейнера

public class Test {
  public static void main(String[] args) throws NamingException {
// Вендор-специфические параметры инициализации встраиваемого контейнера
    Map<String,Object> prop = new HashMap();
// Путь к классам реализации контейнера
    prop.put(EJBContainer.MODULES, new File("target/classes"));
// Создание встраиваемого контейнера с автозакрытием
    try(javax.ejb.embeddable.EJBContainer ec = EJBContainer.createEJBContainer(prop)){
// Получение контекста JNDI интерфейса javax.naming.Context
      Context ctx = ec.getContext();
// Получение локального (безынтерфейсного) представления EJB-компонента
// по переносимому Global JNDI имени (корень дерева — каталог точки входа (?))
      ExEJB ejb = (ExEJB)ctx.lookup("java:global/classes/ExEJB");
      …
    }
  }
}
```

В примере выше создается экземпляр встраиваемого Autocloseable EJB Lite контейнера EJBContainer при помощи статического фабричного метода. При этом для инициализации в связке Map параметров передается путь к классам реализации контейнера. Далее от контейнера получается контекст ctx, связанный с его встроенной JNDI-службой. При помощи ctx по переносимому синтаксису глобальных JNDI-имен запрашивается локальное безынтерфейсное представление EJB Lite - компонента ExEJB, которое в дальнейшем может использоваться для запросов.

Пример развертки приложения в консоли GlassFish:

```
$ asadmin deploy --force=true target\exApp.jar
```

Описание SLSB

Stateless Session Bean (SLSB) – сессионный EJB без сохранения состояния. Под состоянием экземпляра компонента, понимается значение полей экземпляра его класса реализации.

Отказ от сохранения состояния между клиентскими вызовами означает, что SLSB используется по принципу вызвал, получил результат и забыл. Т.е. SLSB для клиента функционально является пакетом статических методов или службой. За один вызов клиентская задача должна решаться целиком. Хотя экземпляр SLSB может поддерживать внутреннее состояние, например, счетчик вызова экземпляров, отладочные серверные логи и т.п., они не должны влиять на бизнес-интерфейс.

SLSB просты и за счет этого эффективны. Они используются для создания отчетов, в качестве пакета методов или веб-служб, выполняющих задачу за один запрос, для реализации операций чтения из РБД и т.д.

Для обслуживания клиентских запросов, как правило, EJB-контейнером создается множество экземпляров SLSB, называемое пулом. В течении клиентской сессии, т.е. периода существования на сервере EJB-объекта, запросы методов SLSB будут обслуживаться разными экземплярами пула, вследствие чего EJB-объект может выдавать разные значения одного и того же поля от вызова к вызову. Обычно пул создается при старте EJB-контейнера. Если при обслуживании множества клиентских запросов он иссякает, то EJB-контейнер может его дополнить. Если число запросов превысило максимальный размер пула, то запросы могут ставиться в очередь. Пул экземпляров не описывается в спеке EJB 3.2, но это де-факто вендорный стандарт EJB-контейнеров. Пул дает возможность переиспользовать экземпляры, что улучшает эффективность сервера в целом:

- устраняет трату времени на создание экземпляров SLSB при обработке запросов
- устраняет трату ресурсов и времени на уничтожение экземпляров SLSB
- реализует полосу пропускания сервера путем задания границ пула

Rem: некоторые вендоры не используют пул, создавая при каждом вызове одноразовый экземпляр. Но это можно считать частным случаем.

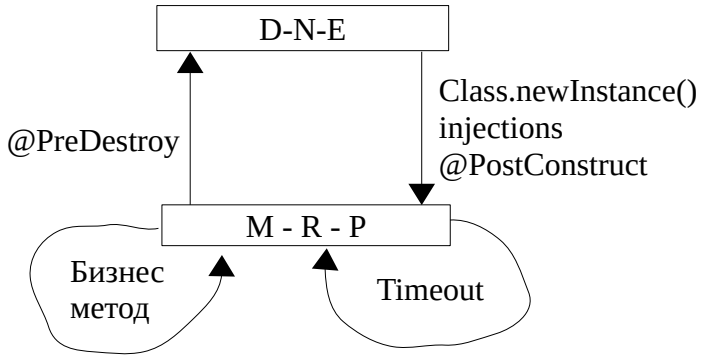
Rem: в GlassFish можно настраивать (в скобках по-умолчанию) минимальный (0 шт.), максимальный (32 шт.) и исходный (0 шт.) объемы пула, а также объем (8 шт.) и таймаут (600 сек.) такта удаления.

В современных JVM создание и удаление простых объектов недорого, поэтому минимальный объем пула по-умолчанию сейчас обычно равен 0. Однако если SLSB тяжеловесны или используют индивидуальные ресурсы вроде TCP-соединений, то имеет смысл установить минимальный объем пула > 0. Максимальный объем пула разумно устанавливать равным ожидаемому пиковому числу пользователей в нормальном режиме функционирования системы.

Жизненный цикл SLSB (SLSB Lifecycle)

В жизненном цикле экземпляра SLSB можно выделить 3 стадии: отсутствие, ожидание и обработка. Контейнер может перехватить только переходы из небытия и обратно, поэтому определено 2 состояния:

- Method-Ready-Pool (M-R-P) – пул ожидающих методов. Это состояние объекта SLSB означает, что он присутствует в пуле экземпляров, но не используется
- Does Not Exist (D-N-E) – объект не существует в RAM



Rem: временные события timeout возникают при использовании службы таймеров.

Жизненный цикл SLSB. Переход D-N-E → M-R-P. Создание экземпляра

- EJB-контейнер выполняет рефлексивный метод Class.newInstance(), создающий при помощи конструктора по-умолчанию класса SLSB его экземпляр.

  Rem: класс SLSB должен обладать конструктором по-умолчанию для рефлексивного вызова Class.newInstance().

- Заполнение EJB-контейнером экземпляра ресурсами, внедряемыми при помощи метаинформации: аннотаций внедрения или XML-дескриптора развертки.
- Активация EJB-контейнером события PostConstruct после инициализации экземпляра SLSB. Вызов callback-метода обработки @javax.annotation.PostConstruct (JSE), если он определен в SLSB.

Rem: разрешается определять только один callback-метод обработки PostConstruct или не определять его вовсе

Callback-метод PostConstruct не должен пробрасывать проверяемых исключений, его сигнатура имеет вид:

```
void exMethod() {...}
```

В качестве альтернативы аннотированию @PostConstruct можно использовать дескриптор развертки ejb-jar.xml

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>ExEJB</ejb-name>
```

```
    <post-construct>
      <lifecycle-callback-method>exMethod</lifecycle-callback-method>
    </post-construct>
   </session>
  </enterprise-beans>
</ejb-jar>
```

Rem: в обработчике PostConstruct можно использовать внедренные ресурсы, что удобно для инициализации, поскольку в конструкторе этого сделать еще нельзя (ресурсы внедряет внешний контейнер после создания конструктором объекта).

Жизненный цикл SLSB. M-R-P. Ожидание и обработка

Когда клиент вызывает метод EJB-объекта (серверный прокси), этот вызов передается одному из экземпляров в M-R-P. С этого момента и до окончания вызова экземпляр выводится из M-R-P в неявное состояние обработки и становится недоступным для других клиентских потоков. Таким образом один запрос соответствует одному экземпляру одного компонента. Когда экземпляр SLSB подгружается из пула, его SessionContext меняется путем учета связи между EJB-объектом и вызывающим клиентом. Экземпляру через контекст становится доступна информация, относящаяся к клиентскому вызову (окружение безопасности и транзакций). Когда вызов завершается, экземпляр отвязывается от EJB-объекта и возвращается в M-R-P.

Клиент может получить @Remote или @Local ссылку на SLSB через JNDI. Однако это не является достаточной причиной создания экземпляра SLSB или изъятия его из M-R-P. Достаточным является вызов бизнес-метода.

@PostConstruct вызывается EJB-контейнером один раз при переходе из Not Exist → M-R-P. При повторных вызовах с подгрузкой из пула этот callback не вызывается.

Жизненный цикл SLSB. Переход M-R-P → D-N-E. Ликвидация экземпляра

Ликвидация экземпляров начинается тогда, когда EJB-контейнер решает почистить память. При этом для каждого уничтожаемого экземпляра возбуждается событие PreDestroy. В SLSB можно объявить один необязательный обработчик этого события — callback-метод, аннотированный @PreDestroy с сигнатурой вида:

```
void exMethod() {...}
```

Этот метод вызывается EJB-контейнером только один раз при переходе M-R-P → Does Not Exist. Обычно там проводится освобождение ресурсов — соединений с РБД, потоков и т.п.

Определить метод обработки PreDestroy можно также в стандартном ejb-jar.xml

```
<ejb-jar>
 <enterprise-beans>
  <session>
   <ejb-name>ExEJB</ejb-name>
   <pre-destroy>
    <lifecycle-callback-method>clean</lifecycle-callback-method>
   </pre-destroy>
  </session>
 </enterprise-beans>
</ejb-jar>
```

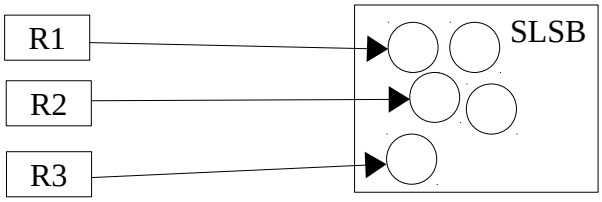
Rem: во время выполнения обработчика PreDestroy все еще доступны SessionContext и JNDI ENC.

После выполнения обработчика PreDestroy зачищаются все ссылки на экземпляр SLSB и он удаляется при активации сборщика мусора.

Параллелизм и потокобезопасность SLSB (Concurrency)

SLSB обладает встроенным механизмом распараллеливания запросов. Спецификация требует, чтобы в любой момент времени к любому его экземпляру поступало не более одного запроса. Поскольку каждый запрос живет в отдельном потоке это означает, что никакой дополнительной потокобезопасности SLSB не требуется.

Контейнер просто распределяет параллельные запросы к SLSB по отдельным экземплярам из пула.



Применение SLSB. Аннотация @Stateless

Аннотация @javax.ejb.Stateless уровня TYPE делает POJO-класс сессионным EJB без поддержки состояния, т.е. SLSB.

```
@Stateless(name = "ExEJB")
@Local(ExLocal.class)
@Remote(ExRemote.class)
public class ExBean implements ExLocal, ExRemote { … }
```

Атрибуты @Stateless:

- name - имя EJB-компонента (компонентная часть глобального JNDI-имени в

переносимом синтаксисе), используемое при развертке приложения в deploy-time, соответствующее <ejb-name> стандартного дескриптора ejb-jar.xml. По-умолчанию это короткое имя класса ("ExBean").
- mappedName – вендор-специфическое (непереносимое) глобальное JNDI-имя
- description – строковое описание компонента

Применение SLSB на примере EncryptionEJB

В качестве примера рассматривается задача шифровки учетной информации (паролей, номеров кредиток и т.п.). В локальном приложении задача решается обычными статическими методами, не требующими состояния. В распределенной среде аналогом такого подхода будет использование SLSB, которое затем можно оформить в виде веб-службы. В примере будут применены техники криптографического хэширования для проверки учетной информации и симметричного шифрования с запароленным ключом для возвращения зашифрованного результата.

Бизнес-интерфейсы (контракты). @Local и @Remote

В начале определяются задачи и описываются методами, т.е. создается базовый интерфейс.

Ex.: базовый интерфейс

```
public interface EncryptionCommon {
// Шифровка строки
  String encrypt(String input);
// Расшифровка строки
  String decrypt(String input);
// Получение дайджеста (digest) – хэш-слепка строки
  String hash(String input);
// Сравнение с заданным hash хэш-слепка input
  boolean compare(String hash, String input);
}
```

На основе базового интерфейса создаются бизнес-интерфейсы: локальный (Local), для внутреннего использования приложением, и удаленный (Remote) для внешних сетевых клиентов. В данном примере они не отличаются от базового. Их можно пометить @Local или @Remote, но в данном случае предполагается переложить это на класс реализации.

Ex.: бизнес-интерфейсы

```
public interface EncryptionLocal extends EncryptionCommon {}
public interface EncryptionRemote extends EncryptionCommon {}
```

Rem: все параметры и результаты бизнес-методов удаленного интерфейса должны быть сериализуемыми. В данном случае используются простые типы, все сериализуемо.

Rem: бизнес-интерфейс может быть либо локальным, либо удаленным, но не тем и другим одновременно (?)

Rem: при запросе по локальному интерфейсу параметры и результат передаются по ссылке (call-by-reference) внутри одного процесса JVM.

Rem: при запросе по удаленному интерфейсу параметры и результат передаются по значению (call-by-vlalue), т.е. копируются (клонируются) при помощи сериализации безотносительно того, где проходил вызов — внутри процесса или вне его.

Класс реализации компонента. @Stateless, @Local, @Remote, @PostConstruct

```
@Stateless(name = "EncryptionEJB")
@Local(EncryptionLocal.class)
@Remote(EncryptionRemote.class)
public class EncryptionBean implements EncryptionLocal, EncryptionRemote { … }
```

Аннотация @javax.ejb.Stateless делает EncryptionBean SLSB с компонентным JNDI-именем EncryptionEJB. Аннотации @javax.ejb.Local и @javax.ejb.Remote задают локальный и удаленный интерфейсы SLSB. Их параметры value задают соответствующие интерфейсные классы. Эти аннотации можно выносить в сами интерфейсы:

```
@Local
public interface EncryptionLocal extends EncryptionCommon {}
@Remote
public interface EncryptionRemote extends EncryptionCommon {}
```

Помимо бизнес-методов в классе SLSB можно определить и @PostConstruct, удобный для инициализации, поскольку в этом методе уже доступны внедренные объекты.

```
@PostConstruct
public void initialize() throws Exception {
// Загрузка пароля ключа из JNDI (Environment Entries) через lookup внедренного SessionContext
   final String ciphersPassphrase = this.getCiphersPassphrase();
…
```

Описание SFSB

SFSB – сессионный EJB с сохранением состояния, поддерживаемым на время сессии с клиентом.

SFSB по сути является серверным продолжением клиента на время сессии. Одному экземпляру SFSB на всем протяжении его жизненного цикла соответствует один клиент. Кроме этого EJB-контейнер гарантирует изолированность экземпляра SFSB от других

клиентов. Экземпляры SFSB не хранятся в пулах и поддерживают сессию с клиентом путем сохранения диалогового состояния. Диалоговое состояние формируется как клиентом, так и компонентом, в качестве примера можно привести GUI, где заполнение предыдущей формы влияет на последующие. Это диалоговое состояние не сохраняется персистентно, а живет в RAM на период сессии или серверного процесса.

Rem: клиентская сессия соответствует EJB-объекту (серверному прокси), чья ссылка (Local представление) или заглушка (Remote представление) посылается клиенту при внедрении или JNDI-запросе EJB-компонента. Если клиент делает рядом 2 JNDI-запроса или 2 внедрения одного и того же типа SFSB, то он получит 2 разных экземпляра прокси, связанных с 2 разными экземплярами SFSB на сервере.

SFSB можно рассматривать как механизм инкапсуляции и перемещения бизнес-логики из клиента на сервер. Взамен клиент получает простой интерфейс и тем самым разгружает свой код, система становится более управляемой и модифицируемой.

Жизненный цикл SFSB

В отличие от SLSB, в SFSB нет пула, а каждому клиенту соответствует свой экземпляр. Зато экземпляры компонента SFSB могут пассивироваться, т.е. выгружаться из RAM на какой-либо носитель, если они неактивны, что позволяет экономить ресурсы в условиях толпы клиентов. При этом EJB-объект экземпляра остается в RAM, а сохраненное диалоговое состояние вместе с экземпляром восстанавливается при первом же клиентском запросе.

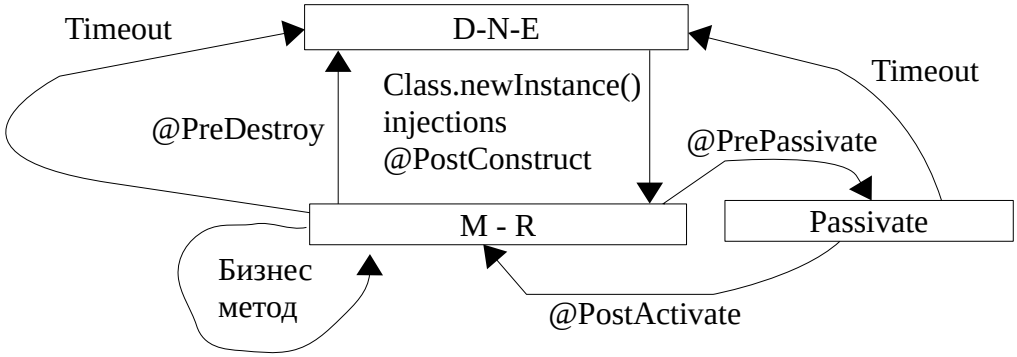
Rem: пассивация как и пулы не регламентируется спекой EJB 3.2, однако является де-факто стандартом реализации EJB-контейнеров.

Rem: жизненный цикл SFSB зависит и от того, реализует ли он интерфейс javax.ejb.SessionSynchronization. Этот интерфейс добавляет несколько callback-обработчиков событий, связанных с транзакциями. Ниже описывается стандартный жизненный цикл без SessionSynchronization.

В жизненном цикле SFSB можно выделить стадии: небытие, ожидание, выполнение, пассивное состояние. Можно перехватить переходы из небытия и обратно, а также переходы к пассивному состоянию и обратно, поэтому вводятся 3 состояния:

- Does Not Exist – экземпляр SFSB еще не создан, в RAM его нет
- Method Ready – экземпляр существует в RAM и готов работать с клиентом

- Passivated – экземпляр сохранен в ROM



В EJB 3.2 (JEE7) появилась возможность запрета пассивации, что иногда удобно, т.к. не нужно каждый раз закрывать и открывать вновь ресурсы, соединения с РБД и т.д. Для этого в объявлении SFSB надо указать: @Stateful(PassivationCapable=false).

Жизненный цикл SFSB. Переход D-N-E → M-R. Создание экземпляра

Когда клиент впервые запрашивает метод SFSB, EJB-контейнер начинает подготавливать новый экземпляр:

- вызывает Class.newInstance()
- внедряет все требуемые зависимости
- связывает новый экземпляр с клиентом изменяя SessionContext
- вызывает PostConstruct-метод, если тот определен

После этого SFSB находится в состоянии M-R и EJB-объект передает его экземпляру целевой вызов метода.

Жизненный цикл SFSB. M-R. Ожидание и обработка

В M-R экземпляр SFSB может принимать клиентские запросы, обращаться к другим EJB, открывать соединения с РБД. В этот период он может поддерживать диалоговое состояние с клиентом и открытые ресурсы в своих переменных.

Жизненный цикл SFSB. Выход из M-R. Пассивация и ликвидация экземпляра

Из M-R SFSB может перейти либо в состояние D-N-E, либо в Passivated.

Пассивация экземпляра зависит от способа использования компонента клиентом, от способа его загрузки EJB-контейнером и от алгоритма пассивации, применяемого вендором. Пассивация может происходить 0 или более раз за жизненный цикл SFSB.

Переход из M-R в D-N-E возможен с использованием @Remove – метода, помеченного @javax.ejb.Remove или описанного в дескрипторе. Также такой переход возможен при

устаревании SFSB. Время жизни SFSB устанавливается при развертывании приложения в вендор-специфической форме. SFSB не может устареть до завершения транзакции. Для обработки события timeout, которое может случиться в M-R, можно определить callback-метод, помеченный @PreDestroy или описанный в дескрипторе. Наконец последний вариант перехода в D-N-E возникает при выбросе в экземпляре системного исключения, при этом @PreDestroy не вызывается, EJB-объект более не имеет жизненной силы и сессия гибнет.

Rem: крайне рекомендуется задавать не менее одного терминального бизнес-метода с @Remove. Это дает возможность оперативно разрывать сессию и удалять из памяти те экземпляры SFSB, клиентская работа с которыми завершена.

Жизненный цикл SFSB. Состояние Passivate

Passivate – состояние жизненного цикла SFSB, в котором экземпляр удален из RAM, но сохраняет диалоговое состояние в некотором персистентном хранилище. Пассивация применяется при достаточно длительном неиспользовании SFSB клиентом.

Rem: спецификация EJB 3.1 (JEE6) не определяет частоту и механизм пассивации/активации SFSB.

Сохраняемое диалоговое состояние может включать:

- примитивы
- java.io.Serializable
- javax.ejb.SessionContext
- javax.jta.UserTransaction
- javax.naming.Context
- javax.persistence.EntityManager
- javax.persistence.EntityManagerFactory
- фабрики управляемых ресурсов вроде javax.sql.DataSource
- ссылки на другие EJB

Сохранение диалогового состояния проводится механизмом пассивации. При этом сериализация экземпляра компонента требуется не всегда. Восстановление состояния проводится автоматически.

Можно определить callback-метод обработки события PrePassivate, возникающего перед пассивацией, при помощи @PrePassivate или дескриптора. В этом методе можно занулить все не-transient не-Serializable поля компонента для избежания выброса исключений. Также в этом методе можно закрыть соединения.

Rem: transient-поля игнорируются при сериализации, но вендор-специфические механизмы пассивации могут работать по-другому, поэтому лучше занулять такие поля вручную.

Механизм пассивации зависит от вендора. Он может использовать как стандартный

механизм сериализации (байтовыми потоками), так и собственные средства кэширования значений. При активации экземпляра компонента все пассивированные значения должны быть восстановлены.

Жизненный цикл SFSB. Переход Passivate → M-R. Активация экземпляра

Активация экземпляра компонента и переход SFSB из Passivate к состоянию M-R проводится при повторном запросе клиентом этого SFSB. EJB-контейнер проводит восстановление диалогового состояния при помощи десериализации или своих инструментов, воссоздает ссылки на SessionContext, другие EJB, менеджеры ресурсов. Также автоматически проводится перевнедрение ресурсов, в т.ч. других SFSB.

Когда диалоговое состояние восстановлено, EJB-контейнером выбрасывается событие PostActivate. Оно может быть обработано callback-методом SFSB, помеченным @PostActivate или описанным в дескрипторе. В этом методе можно открыть соединения, инициализировать несохраненные поля (transient или специально зануленные не-Serializable). После этого SFSB переходит в M-R состояние.

Rem: transient-поля по стандарту десериализации инициализируются значениями по-умолчанию: null, 0, false. Однако вендор-специфические механизмы пассивации могут делать это по-своему, поэтому лучше явно инициализировать такие поля в обработчике PostActivate.

Жизненный цикл SFSB. Переход Passivate → D-N-E. Уничтожение пассивного экземпляра

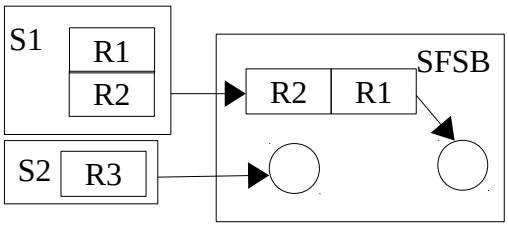
При устаревании SFSB, т.е. при истечении timeout, заданного в deploy-time, его пассивированный экземпляр попросту ликвидируется без вызова @PreDestroy. После этого SFSB переходит в состояние D-N-E.

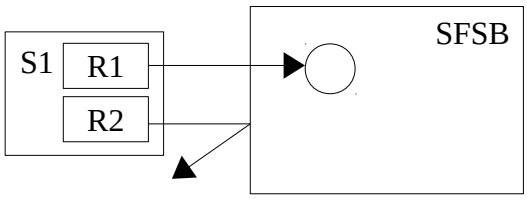
Клиентский вызов терминального @Remove метода к экземпляру в пассивном состоянии приведет к его активации, отработке метода, выбросу события PreDestroy и переходу в D-N-E.

Параллелизм и потокобезопасность SFSB (Concurrency)

SFSB обладает встроенным механизмом распараллеливания запросов. Спецификация требует, чтобы в любой момент времени к любому его экземпляру поступало не более одного запроса. Поскольку каждый запрос живет в отдельном потоке это означает, что никакой дополнительной потокобезопасности SFSB не требуется.

EJB-контейнер может сериализовывать параллельные запросы к SFSB и ждать, пока экземпляр вновь не станет доступным. Разработчик SFSB также может при желании запретить параллельный доступ к нему при помощи @javax.ejb.ConcurrencyManagement.





Применение SFSB. Аннотация @javax.ejb.Stateful

Аннотация @javax.ejb.Stateful уровня TYPE делает POJO-класс сессионным EJB с сохранением состояния, т.е. SFSB.

```
@Stateful(name = "ExEJB")
@Local(ExLocal.class)
@Remote(ExRemote.class)
public class ExBean implements ExLocal, ExRemote { … }
```

Атрибуты @Stateful:

- name - имя EJB-компонента (компонентная часть глобального JNDI-имени в переносимом синтаксисе), используемое при развертке приложения в deploy-time, соответствующее <ejb-name> стандартного дескриптора ejb-jar.xml. По-умолчанию это короткое имя класса ("ExBean").
- mappedName – вендор-специфическое (непереносимое) глобальное JNDI-имя
- passivationCapable – атрибут запрета пассивации (false), по-умолчанию равен true, появился в EJB 3.2 (JEE7)
- description – строковое описание компонента

Применение SFSB на примере FileTransferEJB

Рассмотрим пример создания FTP-клиента. Реализация работы по протоколу FTP требует диалогового состояния. Действия FTP-клиента зависит от сессионной истории вызовов, поэтому SLSB для реализации файлообмена не подходит, требуется SFSB.

В примере реализуются некоторые бизнес-функции: print (pwd), make (mkdir), change (cd). В состоянии компонента будут присутствовать несериализуемые поля и потребность в явной настройке соединения с сервером. Вместо попытки сериализации этих полей в течении пассивации, они будут переинициализироваться программно в момент активации.

Rem: для отмены пассивации вместо ненадежного transient лучше использовать явное зануливание объекта

Также потребуется безопасно закрывать открытые соединения. Пассивация и активация должны быть полностью незаметны для клиента, с точки зрения вызывающей стороны

работа должна строиться также, как если бы их не было. Необходимо реализовать и сигнализацию клиентом окончания рабочей сессии, чтобы запускать методы очистки, снижающие издержки от висения в памяти экземпляров более ненужных EJB.

Бизнес-интерфейсы (контракты) @Local и @Remote

Общий интерфейс FTP-клиента будет иметь вид:

```
pubic interface FileTransferCommon {
  void mkdir(String directory) throws IllegalStateException;
  void cd(String directory) throws IllegalStateException;
  String pwd() throws IllegalStateException;
// Очистка ресурсов по завершению сессии клиентом
  void disconnect();
// Открытие соединений, аутентификация и т.п. подготовка в начале новой клиентской сессии
  void connect() throws IllegalStateException;
}
```

На основе общего создается удаленный бизнес-интерфейс:

```
pubic interface FileTransferReomte extends FileTransferCommon {
// Будет использоваться как Remove-метод в SFSB
  void endSession();
}
```

Их можно сразу пометить @Local или @Remote, но в данном случае предполагается переложить это на класс реализации.

Исключения

Для описания ошибок, возникающих при выполнении команд FTP-клиента, определяется исключение. Поскольку клиент не может исправить подобные ошибки, исключение будет системным (непроверяемым):

```
public class FileTransferException extends RuntimeException {…}
```

Класс реализации компонента. @Local, @Remote, @PostConstruct, @PostActivate, @PrePassivate, @PreDestroy

```
// Имя ejb (компонентная часть Global JNDI имени) берется из статической константы класса
@Stateful(name = FileTransferBean.EJB_NAME)
@Remote(FileTransferRemote.class)
public class FileTransferBean implements FileTransferRemote, Serializable {
// Имя ejb
   public static final String EJB_NAME = "FileTransferEJB";
// Хост FTP-сервера. На боевой системе обычно внедряется из переменных среды JNDI
   private static String CONNECT_HOST = "localhost";
```

```
// Порт FTP-сервера. На боевой системе обычно берется из JNDI. Станадртный = 21, он требует root
  private static int CONNECT_PORT = 12345;
// FTP-клиент из библиотеки Apache. Для надежности пассивации он и его соединения
// зануливаются/воссоздаются без использования сериализации
  private FTPClient client;
// Текущая рабочая директория. Пассивируется автоматом
  private String presentWorkingDirectory;
// Инициализация/реинициализация. Выбрасывает системные исключения
  @PostConstruct
  @PostActivate
  @Override
  public void connect() throws IllegalStateException, FileTransferException {
    final FTPClient clientBefore = this.getClient();
    if (clientBefore != null && clientBefore.isConnected()) {
       throw new IllegalStateException("FTP Client is already initialized");
     }
     final String connectHost = this.getConnectHost();
     final int connectPort = this.getConnectPort();
     final FTPClient client = new FTPClient();
     // Соединение, аутентификация, перехода в текущую рабочую директорию
     …
  }
// Деактивация/подготовка к пассивации. FTPClient зануливается явно вместо transient
  @PrePassivate
  @PreDestroy
  @Override
  public void disconnect() {
    final FTPClient client = this.getClient();
    if (client != null) {
      if (client.isConnected()) {
         // Завершение сессии с FTP-сервером, разрыв соединения
         …
       }
       // Зануливание поля FTPClient, чтобы избежать сериализации
       this.client = null;
    }
  }
// Вспомогательные методы
  public String getConnectHost() { … }
  public void setConnectHost(final String connectHost) { … }
  public int getConnectPort() { … }
  public void setConnectPort(final int connectPort) { … }
// Бизнес-методы
  @Override
  void mkdir(String directory) throws IllegalStateException { … }
  @Override
  void cd(String directory) throws IllegalStateException { … }
  @Override
  String pwd() throws IllegalStateException { … }
}
```

Модульное тестирование (JUnit)

Модульное тестирование выполняется вне сервера, т.е. EJB в этих тестах превращаются в POJO. Наличие диалогового состояния требует иммитации событий жизненного цикла: PostConstruct, PreDestroy. Механизм пассивации/активации требует имитации событий PrePassivate и PostActivate.

Для тестирования экземпляр SFSB создается программно, а роль EJB-контейнера играет JUnit. Класс компонента JUnit эмулирует EJB-контейнер, позволяя задавать тестовое состояние компонента, используя обработчики событий своего жизненного цикла @Before и @After для подыгрыша событий жизненного цикла SFSB PreDestroy и PostConstruct, соответственно. Для проведения тестов вызовов и спонтанной пассивации/активации можно использовать @Test методы JUnit. Там же можно проверять результаты выполнения тестовых методов при помощи заранее предопределенных ответов (org.junit.assertEquals(…)). Пассивацию/активацию можно эмулировать при помощи стандартной сериализации/десериализации.

Ex.: модульное тестирование SFSB локальным клиентом при помощи JUnit

```
public class FileTransferUnitTest {
…
  @Before
  public void createFtpClient() throws Exception {
    // Создание EJB
    final FileTransferBean ftpClient = new FileTransferBean();
    // Имитация выброса PostConstruct – вызов @PostConstruct - метода
    ftpClient.connect();
    // Сохранение состояния компонента
    this.ftpClient = ftpClient;
    log.info("Set FTP Client: " + ftpClient);
  }
 @After
  public void cleanup() throws Exception {
    // Получение сохраненного состояния компонента
    final FileTransferBean ftpClient = this.ftpClient;
    if (ftpClient != null) {
      // Имитация выброса PreDestroy – вызов @PreDestroy - метода
      ftpClient.disconnect();
      // Уничтожение состояния компонента / сессии
      this.ftpClient = null;
    }
  }
  @Test
  public void testPassivationAndActivation() throws Exception {
    // Получение сохраненного состояния компонента в виде бизнес-интерфейса
    final FileTransferCommon client = this.getClient();
    // Получение начального тестового значения рабочего каталога и переход в него
    final String home = getFtpHome().getAbsolutePath();
    client.cd(home);
```

```
    // Тестирование печати имени рабочего каталога
    final String pwdBefore = client.pwd();
    assertEquals("Present working directory should be set to home", home, pwdBefore);
    //  Имитация выброса PrePassivate
    client.disconnect();
    // Эмуляция пассивации компонента
    final ByteArrayOutputStream outStream = new ByteArrayOutputStream();
    final ObjectOutput objectOut = new ObjectOutputStream(outStream);
    objectOut.writeObject(client);
    objectOut.close();
    // Эмуляция активации компонента
    final InputStream inStream = new ByteArrayInputStream(outStream.toByteArray());
    final ObjectInput objectIn = new ObjectInputStream(inStream);
    final FileTransferCommon serializedClient = (FileTransferCommon) objectIn.readObject();
    objectIn.close();
    // Имитация выброса PostActivate
    serializedClient.connect();
    // Тестирование печати имени текущего рабочего каталога после пассивации/активации
    final String pwdAfter = serializedClient.pwd();
    assertEquals("Рабочий каталог до и после пассивации должен совпасть", home, pwdAfter);
  }
}
```

Интеграционное тестирование

Для проверки взаимодействия множества экземпляров SFSB необходимо развернуть тестовый JEE-сервер и создать удаленный тестовый клиент. В простейшем случае можно протестировать корректность подключения, удаления сессии, изолированности сессионных контекстов и поддержки диалогового состояния экземпляров.

```
Ex.: интеграционное тестирование удаленным клиентом при помощи JUnit

public class FileTransferIntegrationTest {
  …
// Метод получения заглушки SFSB на удаленном клиенте через global JNDI
   private FileTransferRemote createNewSession() throws Exception {
     …
   }
// Тестирование завершения сессии с SFSB при помощи @Remove – метода endSession
 @Test
   public void testSfsbRemoval() throws Exception {
     // Новая сессия
     final FileTransferRemoteBusiness sfsb = this.createNewSession();
     // Переход в начальный рабочий каталог
     final String ftpHome = "myHome";
     sfsb.cd(ftpHome);
     // Проверка корректности перехода
     final String pwdBefore = sfsb.pwd();
     assertEquals("Session should be in the FTP Home directory", ftpHome, pwdBefore);
     // Завершение сессии клиентом @Remove - методом endSession
```

```
    sfsb.endSession();
    boolean gotExpectedException = false;
    try {
      sfsb.pwd();
    }
    catch (final NoSuchEJBException nsee) {
      gotExpectedException = true;
    }
    assertTrue("Вызов метода удаления привел к разрушению сессии и SFSB",gotExpectedException);
  }
// Тестирование поддержки диалоговых состояний и их изоляции (независимости)
  @Test
  public void testSessionIsolation() throws Exception {
    // Создание 2-х клиентстких сессий с SFSB
    final FileTransferRemoteBusiness session1 = this.createNewSession();
    final FileTransferRemoteBusiness session2 = this.createNewSession();
    // Переходы в начальный рабочий каталог
    final String ftpHome = "myHome";
    session1.cd(ftpHome);
    session2.cd(ftpHome);
    // Создание каталога и переход в него в 1-й сессии
    final String newDirSession1 = "newDirSession1";
    session1.mkdir(newDirSession1);
    session1.cd(newDirSession1);
    // Создание каталога и переход в него во 2-й сессии
    final String newDirSession2 = "newDirSession2";
    session2.mkdir(newDirSession2);
    session2.cd(newDirSession2);
    // Печать имен текущих каталогов
    final String pwdSession1 = session1.pwd();
    final String pwdSession2 = session2.pwd();
    // Проверка нахождения в созданных каталогах
    assertEquals("Сессия 1 корректна", ftpHome + File.separator + newDirSession1, pwdSession1);
    assertEquals("Сессия 1 корректна", ftpHome + File.separator + newDirSession2, pwdSession2);
    // Завершение клиентских сессий
    session1.endSession();
    session2.endSession();
}
```

Первый тест testSfsbRemoval — на корректность удаления сессии: при корректном удалении сессии, экземпляр SFSB переходит в D-N-E состояние и повторный его запрос через удаленную клиентскую заглушку приведет к выбросу исключения. С SLSB такое действие невозможно.

Второй тест testSessionIsolation — на независимость результатов работы в разных сессиях и их поддержки. Каждую сессию поддерживает свой экземпляр SFSB, который помнит историю предыдущих вызовов. Состояния SFSB взаимонезависимы (только если не конфликтуют из-за общих ресурсов, например, одного и того же каталога). В случае SLSB каждый запрос на печать текущего рабочего каталога мог бы выдавать различные результаты в зависимости от состояния экземпляра, подгруженного из пула,

переход в новую директорию вообще мог бы вызвать ошибку.

Описание Singleton

В EJB 3.1 (JEE6) появился новый сессионный EJB – Singleton (одиночка) Session Bean, позволяющий всем клиентам работать с единственным экземпляром. Примерами применения Singleton являются создание объектов мыши, принтера, единого кэша, системного календаря.

Концепция Singleton удовлетворяет следующим требованиям:

- внутреннее состояние принадлежит всем клиентам
- в любой момент любой запрос обрабатывается единственным экземпляром
- все бизнес-методы потокобезопасны

Эти требования могут приводить к следующим издержкам:

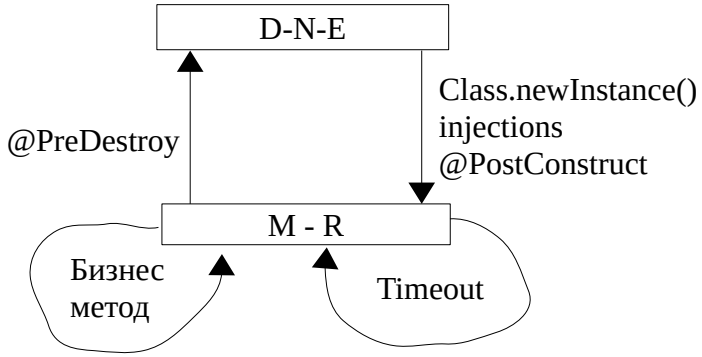
- В следствие потокобезопасности возможны задержки обработки клиентских запросов
- Проблемы (тупики) в параллельных запросах невозможно отловить в однопоточном тесте

Rem: можно создать Singleton вручную: закрытый конструктор по-умолчанию, закрытое статическое поле instance экземпляра (инициализирующееся статически), статический публичный фабричный метод getInstance(), возвращающий instance. Для обеспечения потокобезопасности можно синхронизировать (synchronized) метод getInstance, при этом в роли mutex (семафор) будет выступать объект class синглетона.

Поэтому Singleton EJB дополнили настройкой параллельного доступа, в отличие от остальных сессионных EJB-компонентов. При аккуратном использовании компонент Singleton может дать серьезную экономию RAM в определенных задачах. Вместе с этим неаккуратное применение способно привести к зависанию и краху всего приложения.

Жизненный цикл Singleton

Жизненный цикл напоминает жизненный цикл SLSB: стадии небытия, ожидания и обработки; актуальные для перехвата контейнером состояния небытия и ожидания. EJB-контейнер создает единственный экземпляр Singleton до момента первого вызова, который живет вплоть до завершения работы приложения. Все запросы идут к этому экземпляру.



Параллелизм Singleton (Concurrency) и потокобезопасность

В случае Singleton все запросы идут параллельно к единственному экземпляру и режим распараллеливания можно задавать. Существует 3 режима управления параллельной обработкой, которые можно задать для Singleton EJB: контейнерный (CONTAINER), программный (компонентный, BEAN), синхронизированный (CONCURRENCY_NOT_SUPPORTED). Синглетоны с контейнерным типом называют CMC (Container-Managed Concurency), а с программным — BMC (Bean-Managed Concurency). По-умолчанию синглетоны являются CMC.

Потокобезопасность — способность системы к корректной в некотором смысле одновременной работе во множестве потоков. Корректность может иметь смысл логической согласованности операций, каждая из которых выполняется пусть даже с реальным (свежим) состоянием системы. А может иметь смысл реальности (свежести) состояния системы, над которыми выполняются операции, пусть даже и логически-связанные. Обычно под потокобезопасностью имеют ввиду логически-согласованную работу кода в каждом потоке с реальным (свежим) состоянием системы при наличии общих (разделяемых) ресурсов.

В Java существует 2 механизма обеспечения потокобезопасности:

- Атомарность, которая гарантирует логическую согласованность простейших операций с реальным состоянием системы. Реализуется в пакете java.util.concurrent.atomic
- Механизм разграничения синхронизации доступа к общим ресурсам, гарантирующий согласованную работу потока с реальным состоянием системы. Реализуется это разграничение встроенными в каждый Java-объект механизмом блокировки доступа под названием монитор и конструкцией Java-языка synchronized(mutex){…}, называющейся также критической секцией. В роли объекта mutex (mutual exclusion, взаимное исключение, аналог семафора) может выступать любой объект, доступный в this. Объект mutex – фактически метка (семафор), блокировка монитора которой на время выполнения критической секции также блокирует остальные помеченные ею же критические секции this. Модификатор synchronized может применяться к нестатическому методу, это аналогично критической секции, включающей все тело метода с mutex=this.

Применение synchronized к статическому методу аналогично критической секции, включающей тело метода с mutex=this.getClass().

Rem: более глубокое изучение потокобезопасности и параллелизма связано с понятиями видимости потоков, активными и пассивными блокировками (livelock, deadlock), состоянии гонки (race conditions).

Rem: Singleton не поддерживает кластеризацию. Т.е. в различных экземплярах EJB-контейнеров, объединенных в один кластер, будут различные экземпляры Singleton.

Применение Singleton

Аннотация @Singleton

Аннотация @javax.ejb.Singleton уровня TYPE делает POJO-класс сессионным компонентом-одиночкой, т.е. синглетоном.

```
@Singleton(name = "ExEJB")
@Local(ExLocal.class)
@Remote(ExRemote.class)
public class ExBean implements ExLocal, ExRemote { … }
```

Атрибуты @Singleton:

- name - имя EJB-компонента (компонентная часть глобального JNDI-имени в переносимом синтаксисе), используемое при развертке приложения в deploy-time, соответствующее <ejb-name> стандартного дескриптора ejb-jar.xml. По-умолчанию это короткое имя класса ("ExBean").
- mappedName – вендор-специфическое (непереносимое) глобальное JNDI-имя
- description – строковое описание компонента

Пример RSSCacheEJB

RSSCacheEJB – пример компонента, каэширующего содержимое RSS-канала, который постоянно читают клиенты и который относительно редко обновляется по сети.

Чтение — скоростная операция без блокировок. Обновление кэша идет с Write-блокировкой, в течении которой запросы на чтение накапливаются в EJB-контейнере. Также, чтобы не терять темп при первом обращении, необходимо делать трудолюбивую инициализацию компонента.

Singleton хорош для операций многопоточного чтения, где не нужно выделять отдельные потоковые экземпляры, как это делается для SLSB, или поддерживать сессии, как в случае SFSB. Он как раз и подходит для реализации RSSCacheEJB.

Для передачи создается интерфейс DTO типа RSSEntry. Выделение интерфейса DTO

позволяет приложению использовать различные библиотеки работы с RSS. В данном примере предполагается использование библиотеку Rome.

```
public interface RssEntry {
  String getAuthor();
  String getTitle();
  URL getUrl();
  String getDescription();
}
```

Для обеспечения защиты состояния от записи локальным клиентом создается класс-утилита ProtectExportUtil, которая копирует заданный объект.

Rem: локальные клиенты (работающие с представлением @Local и безынтерфейсным LocalBean), живущие в одном процессе с приложением, получают значения по ссылке и могут менять мутирующие объекты состояния.

Rem: при использовании клиентом EJB удаленного интерфейса (@Remote), все передается при помощи RMI в сериализованном виде, т.е. по-значению и клиент работает с десериализованными копиями.

```
class ProtectExportUtil {
  private ProtectExportUtil() {}
  static URL copyUrl(final URL url) {
    if (url == null){
      return url;
    }
    return new URL(url.toExternalForm());
  }
}
```

Далее создается реализация контейнера данных RSSEntry, совместимая с библиотекой Rome (SyndEntry). Это шаблонный JSE bean, хранящий свое состояние в атрибутах - приватных полях с доступом через get/set.

```
public class RomeRssEntry implements RssEntry {
  private String author;
  private String description;
  private String title;
  private URL url;
// Конструктор-преобразователь SyndEntry → RomeRssEntry
  RomeRssEntry(final SyndEntry entry) throws IllegalStateException {
    this.author = entry.getAuthor();
    final SyndContent content = entry.getDescription();
    this.description = content.getValue();
    this.title = entry.getTitle();
    final String urlString = entry.getLink();
    this.url = new URL(urlString));
```

```
  @Override
   public String getAuthor() {
     return this.author;
   }
   @Override
   public String getDescription() {
     return this.description;
   }
   @Override
   public String getTitle()
   {
     return this.title;
   }
   @Override
   public URL getUrl() {
     return ProtectExportUtil.copyUrl(this.url);
   }
}
```

Теперь надо спроектировать бизнес-интерфейс компонента (или более общо контракт), которым будут пользоваться клиенты.

```
public interface RssCacheCommonBusiness {
   // Возврат кэша всех элементов RSS-канала точки входа с текущим url. Список read-only
   List<RssEntry> getEntries();
   // Возврат текущего url
   URL getUrl();
   // Очистка кэша в памяти и обновление всех элементов канала по текущему url
   void refresh();
}
```

Наконец нужно создать основной класс реализации компонента. Выше был определен тип компонента — Singleton. Также была определена необходимость трудолюбивой его загрузки и типы блокировок методов, обеспечивающие эффективность и потокобезопасность.

```
@Singleton
@Startup
@Remote(RssCacheCommonBusiness.class) {
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class RssCacheBean implements RssCacheCommonBusiness {
   private static final Logger log = Logger.getLogger(RssCacheBean.class);
// Конечная точка RSS-канала
   private URL url;
   private List<RssEntry> entries;

// Полностью параллельный метод чтения
   @Override
   @Lock(LockType.READ)
```

```
  public List<RssEntry> getEntries() {
    return entries;
  }

  @Lock(LockType.READ)
  @Override
  public URL getUrl() {
// Всем клиентам выдается копия, this.url становится read-only
    return ProtectExportUtil.copyUrl(this.url);
  }

  @PostConstruct
  @Override
  @Lock(LockType.WRITE)
  public void refresh() throws IllegalStateException {
    final URL url = this.url;
    if (url == null) {
      throw new IllegalStateException("The Feed URL has not been set");
    }
    log.info("Requested: " + url);
// API Rome
    final FeedFetcher feedFetcher = new HttpClientFeedFetcher();
    SyndFeed feed = null;
    try {
      feed = feedFetcher.retrieveFeed(url);
    }
    catch (final FeedException | final FetcherException | final IOException fe) {
      throw new RuntimeException(fe);
    }
    catch (final IOException ioe) {
      throw new RuntimeException(ioe);
    }
    final List<RssEntry> rssEntries = new ArrayList<RssEntry>();
    final List<SyndEntry> list = (List<SyndEntry>) feed.getEntries();
    for (final SyndEntry entry : list) {
      final RssEntry rssEntry = new RomeRssEntry(entry);
      rssEntries.add(rssEntry);
      log.debug("Found new RSS Entry: " + rssEntry);
    }
    // Защита от модификации клиентом. Read-only кэш
    final List<RssEntry> protectedEntries = Collections.unmodifiableList (rssEntries);
    this.entries = protectedEntries;
  }
}
```

Можно отметить в данном коде защиту от изменения пользователем read-only данных, что обеспечивает полностью параллельный доступ на чтение кэша RSS-канала множеству пользователей. Для этого использовалась как самопальное глубокое клонирование объекта ProtectExportUtil.copyUrl(...), так и неизменяемая реализация List - Collections.unmodifiableList(…).

Также можно отметить сочетание @Startup, @Singleton и @PostConstruct, делающие метод refresh() инициализирующим для всего приложения.

Управление загрузкой синглетонов. @Startup, @DependsOn

EJB-контейнер может создавать экземпляр Singleton лениво (при первом запросе) или трудолюбиво (при развертке в deploy-time). Ленивое поведение — по умолчанию, трудолюбивое при пометке типа синглетона @javax.ejb.Startup. Трудолюбивый способ целесообразен при затратной инициализации. По сути это автозапуск компонента.

При этом следует помнить, что экземпляры SFSB и SLSB компонентов в deploy-time еще могут быть недоступными, поэтому обращение к ним в @PostConstruct методе автозапускающегося синглетона может привести к java.lang.IllegalAccessException. Зато можно обращаться к JPA и БД.

Порядок загрузки синглетонов можно задать при помощи аннотирования типа синглетона @javax.ejb.DependsOn. Атрибут value = {“SingletonName1”, …,”SingletonNameN”} содержит строковый массив имен синглетонов. Инициализация будет (?)выполняться в порядке перечисления (?) не нужно надеяться на порядок перечисления. (?) После чего будет загружен сам аннотированный синглетон. Имена синглетонов в случае ear-приложения можно уточнить содержащим их jar-модулем: @DependsOn({“ejb1.jar#SingletonName1”,…}) . При сочетании @DependsOn @Startup все перечисленные синглетоны также будут трудолюбиво загружены.

Rem: стоит избегать циклических зависимостей

Управление параллелизмом Singleton. @ConcurrencyManagement

Существует 3 режима управления параллельной обработкой синглетона, которые можно задать для Singleton EJB аннотацией уровня класса @javax.ejb.ConcurrencyManagement.

Rem: @ConcurrencyManagement применяется только к классу компонента, а не к бизнес-интерфейсам.

Ее значениями могут быть константы-перечисления javax.ejb.ConcurrencyManagementType {CONTAINER, BEAN, CONCURRENCY_NOT_SUPPORTED}. По-умолчанию используется режим CONTAINER.

Контейнерное управление параллелизмом (CMC). @Lock

Управлением множественными запросами к EJB может заниматься EJB-контейнер. Это называется контейнерным управлением параллелизмом (Container-Managed Concurency, CMC). Регулировать CMC можно при помощи метаинформации. Для Singleton EJB такой тип управления (javax.ejb.ConcurrencyManagementType.CONTAINER)

используется по-умолчанию. EJB-контейнер управляет синхронизацией при помощи Read и Write блокировок методов.

```
@ConcurrencyManagement(ConcurrencyManagementType.CONTAINER)
public class ExBean {...}
```

Каждый метод Singleton EJB помечается Read или Write блокировкой. Write-блокировка метода аналогична установке модификатора synchronized (?). Она дает доступ к методу только одному потоку, при этом блокируется весь синглетон, т.к. семафором (mutex) для синхрометода является сам объект (this), который к тому же является единственным экземпляром Singleton. Read-блокировка дает полный параллельный доступ к методу, при условии, что в текущий момент Singleton не заблокирован запросом к какому-либо другому методу с Write-блокировкой (?). По-умолчанию используется Write-блокировка. Изменить тип блокировки можно аннотацией уровня метода или класса @javax.ejb.Lock, значением которой может быть перечисление-константа javax.ejb.LockType {READ, WRITE}.

```
@Lock(LockType.READ)
public String getValue(){ … }
```

```
@Lock(LockType.WRITE)
public void setValue(String value){ … }
```

Также можно задать период ожидания доступа к методу с любым типом блокировки при помощи @javax.ejb.AccessTimeout. По истечению этого периода ожидания, EJB-контейнер выбросит непроверяемое исключение javax.ejb.ConcurrentAccessTimeoutException. Это полезно также в диагностических целях при анализе производительности приложения.

В примере ниже установлен период ожидания доступа к методу с Read-блокировкой, равный 15 секундам. Если весь этот период Singleton будет заблокирован более 15 секунд Write-блокировкой какого-то другого метода, то в качестве ответа на запрос будет выброшено исключение.

```
@Lock(LockType.READ)
@AccessTimeout(timeout=15, unit=java.util.concurrent.TimeUnit.SECONDS)
public String getValue() { … }
```

Ниже пример установки таймаута для метода с Write-блокировкой. Поток запроса будет ждать доступа 5 секунд до выброса исключения.

```
@Lock(LockType.WRITE)
@AccessTimeout(timeout=5, unit=java.util.concurrent.TimeUnit.SECONDS)
public void setValue(String value) { … }
```

Значение timeout=-1 соответствует бесконечному периоду, т.е. ожиданию снятия блокировки. Значение timeout=0 означает запрет на доступ при блокировке.

Программное управление параллелизмом (BMC)

Программное управление параллелизмом (Bean Managment Concurency, BMC) применяется для более сложного управления синхронизацией Singleton. В этом случае распараллеливанием занимается разработчик компонента при помощи средств разграничения ресурсов JSE (synchronized, volatile), библиотеки Concurrent и т.д.

```
@ConcurrencyManagement(ConcurrencyManagementType.BEAN)
public class ExBean {...}
```

Отказ от параллелизма

При отказе от обслуживания параллельных запросов, EJB-контейнер будет заворачивать все запросы пока экземпляр EJB (SFSB(?) или Singleton) не обработает текущий.

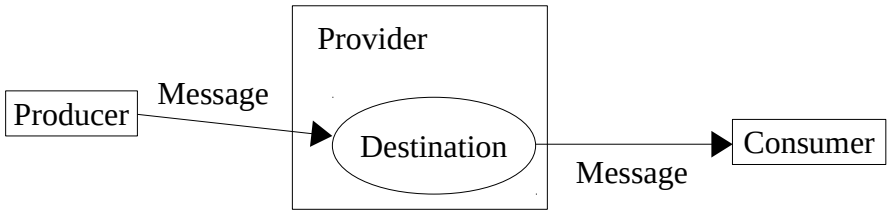
```
@ConcurrencyManagement(ConcurrencyManagementType.CONCURRENCY_NOT_SUPPORTED)
public class ExBean {...}
```

Описание MDB

MOM

Для обмена информацией между слабо-связанными и разнородными системами удобно использовать асинхронный обмен сообщениями: клиенты не обязаны знать ничего ни друг про друга, ни про то, каким образом обрабатываются или генерируются их сообщения на другом конце канала связи. Единственное, что требуется от клиентов в этой схеме — знать формат сообщений и согласовать промежуточный пункт назначения, где они будут их принимать и куда будут отправлять. Примерами таких клиентов могут быть системы, работающие в разном темпе, например, одна регулярно, другая — периодически.

Класс интеграционных систем, обеспечивающих подобный обмен сообщениями, называется MOM (Message Oriented Middleware). Идея MOM состоит в создании общего посредника между клиентами, который и берет на себя все проблемы обмена. Этот посредник называется поставщиком сообщений (Message Provider или Message Broker). Клиент, который отправляет сообщение поставщику, называется производителем сообщения (Message Producer), а клиент, который принимает от поставщика сообщение — потребителем (Message Consumer). Промежуточное хранилище сообщений и порядок обращения к нему через поставщика обобщаются в понятии источника (Message Destination).



Система обмена сообщениями (Message Service) является реализацией MOM, устанавливающей правила доступа к поставщику, регламентирует создание, отправку и получение сообщений. Основными требованиями к системам обмена сообщений являются:

- асинхронность
- слабая связанность клиентов
- надежность обмена

Асинхронность означает временную независимость клиентов обмена, в т.ч. их on-line/off-line режимы. Слабая связанность клиентов обусловлена наличием посредника, устанавливающего относительно универсальные правила приема и отправки сообщений. Надежность системы обмена означает способность обеспечивать доставку сообщений в асинхронных режимах работы, т.е. в условиях задержек или off-line состояний клиентов, при сбоях передачи или обработки сообщений.

Системы сообщений в Java

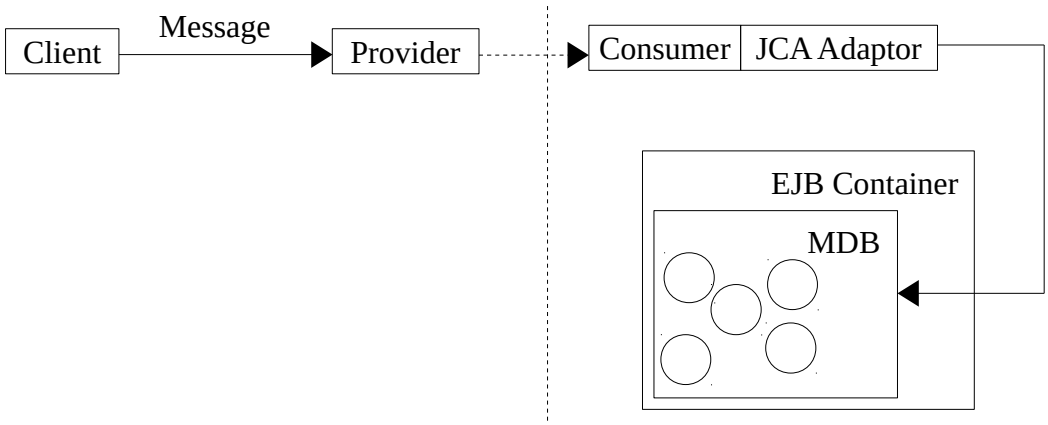
Штатная система сообщений в JEE описывается API JMS, появившимся в 1998 году. JMS 1.1 был штатной системой сообщений JEE до версии JEE7, где его сменил JMS 2.0. Вообще API JMS вылазит за пределы JEE-контейнеров и требует от вендоров также и реализации функционала для работы в JSE.

При этом JMS не является единственной системой сообщений, поддерживаемой JEE. Начиная с EJB 2.1 (J2EE 1.4) спека предусматривает использование более широкого класса EIS (Enterprise Information System, промышленные системы обмена сообщениями) через переходник, называемый JCA (Java Connector Architecture, java-архитектура). JCA — JEE-стандарт, описывающий соединение JEE-серверов приложений с EIS. JCA включает стандартный интерфейс поставщика службы (Service Provider Interface, SPI). Этот интерфейс позволяет любой EIS встраиваться в любой JEE-контейнер. Версия JCA 1.6 позволяет JEE-серверам работать с любыми асинхронными EIS. При взаимодействие основано лишь на получении сообщения, предварительной подготовки не требуется.

Компоненты MDB

MDB (Message Driven Bean, управляемый сообщениями компонент) – это асинхронный прослушиватель сообщений, являющийся EJB. В терминах MOM это асинхронный

потребитель. Клиентами MDB являются системы, которым не особо важно то, как будут обработаны их сообщения и которые не будут дожидаться ответа.



MDB не работает напрямую со своими клиентами, а использует посредника — JMS-поставщика (JMS provider), управляющего отправкой и доставкой сообщений.

Короче MDB является прослушивателем любых систем сообщений с правильным JCA-адаптером. JCA обеспечивает абстракцию доставки сообщений любых типов. Родной для MDB системой сообщений является JMS, ее обязаны поддерживать все реализации EJB-контейнера.

MDB является по своей природе компонентом без сохранения состояния. Для обработки сообщений, как правило, EJB-контейнером создается множество экземпляров MDB, называемое пулом. Обычно это делается при старте приложения, затем, при необходимости, пул может быть увеличен или уменьшен.

MDB появился в EJB 2.0 как средство обработки сообщений JMS-поставщика. Но уже в EJB 2.1 MDB могут обрабатывать любые сообщения любых систем, оснащенных JCA-адаптерами, а не только JMS. В EJB 3.0 MDB были дополнены средствами настройки через аннотации.

MDB напоминает местами SLSB. В отличие от сессионных EJB-компонентов, его методы не доступны клиентам, все взаимодействие происходит от лица посредника — поставщика сообщений. MDB выгодно отличается от простых потребителей, выполненных на базе POJO или CDI MBean возможностями EJB-компонента: автоматизацией параллелизма (сам код MDB не содержит реализацию многопоточности), транзакционным механизмом JTA, использованием пула. Также допускается использование стандартной для EJB аутентификации и авторизации, однако, в силу слабой связанности клиентов через поставщика сообщений и возможной поддержки самых разнородных производителей этим поставщиком, стандартного механизма передачи учетной информации от клиента к MDB не описано.

Класс MDB должен удовлетворять следующим требованиям:

- помечаться @javax.jms.MessageDriven или XML-эквивалентом
- (для JMS MDB) реализовывать интерфейс асинхронного прослушивателя javax.jms.MessageListener
- быть доступным для создания экземпляров извне и расширения: public, не abstract, не final, иметь публичный конструктор по-умолчанию
- не реализовывать метод finalize()

Поскольку единственный требующийся для JMS MDB интерфейс MessageListener можно задать через метаинформацию, класс MDB-компонента можно считать POJO. MDB-компонент имеет поведение по-умолчанию и настраивается посредством метаинформации — аннотаций или их XML-эквивалента.

Для развертки MDB требуется полновесный JEE-сервер, в состав EJB Lite эти компоненты не включены спецификацией (по факту вендоры включают).

JMS

Штатная система сообщений в JEE описывается API JMS, появившимся в 1998 году. JMS 1.1 был штатной системой сообщений JEE до версии JEE7, где его сменил JMS 2.0. Вообще API JMS вылазит за пределы JEE-контейнеров и требует от вендоров также и реализации функционала для работы в JSE.

JMS (Java Message Service) – стандартное в JEE7 API системы сообщений. Вообще JMS предназначена для работы как в составе EJB-контейнера, так и вне, например в JSE-приложениях.

JMS является вендорно-независимым API, которое может использоваться для доступа к более общему классу EIS (Enterprise Information System, промышленные системы обмена сообщениями). JMS играет роль JDBC, обеспечивая приложению вендорно-нейтральный доступ к EIS через стандартное JMS API.

Rem: JDBC использует JMS API для организации доступа к РБД

В JEE-контейнеры исторически большинство вендоров встраивало собственную систему сообщений, оставляя возможность интеграции внешних продуктов. Несмотря на это, экспертной группой было решено гарантировать разработчикам EJB наличие реализации штатной JMS, обеспечивающей отправку и прием сообщений. Все реализации контейнеров EJB 3.0 должны поддерживать JMS-поставщика.

Реализацию JMS также называют брокером (Message Broker) или Message Router. Все типы EJB могут использовать JMS для отправки сообщений. Эти сообщения далее могут потребляться другими приложениями или MDB.

В архитектуре JMS используются следующие термины:

- JMS-клиент (JMS Client) — приложение, использующее JMS, объединяет и производителя и потребителя сообщений;
- JMS-производитель (JMS Producer) – JMS-клиент, отправляющий сообщения;
- JMS-потребитель (JMS Consumer) – JMS-клиент, принимающий сообщения;
- JMS-поставщик (JMS Provider) — реальная система, которая управляет отправкой и доставкой сообщений согласно JMS API;
- JMS-сообщение (JMS Message) — объект, который JMS-клиент отправляет или получает от поставщика сообщений;
- JMS-источник (JMS Destination) — хранилище поставщика для сообщений, которые ему отправляют производители и принимают потребители. В JMS существует 2 типа источников: темы (Topic) и очереди (Queue).
- Администрируемые объекты JMS — фабрики соединений (Connection Factory) и источники (Destination), которые создаются брокером административно, настраиваются и помещаются в JNDI-дерево для последующих JNDI-запросов или внедрения в run-time;
- JMS-приложение — бизнес-система, состоящая из одного или более JMS-клиента и одного (обычно) или более JMS-поставщика.

Rem: любой JMS-клиент может одновременно быть и производителем и потребителем сообщений.

Rem: различные Message Broker появились десятки лет назад, наиболее популярными были IBM MQ Series. JMS появился относительно недавно, он специально разрабатывался для обмена различными сообщениями между java-приложениями.

JMS 2.0 появился в JEE7, сменив JMS 1.1. Из нового появилась поддержка интерфейса AutoCloseable для администрируемых объектов. К классическому API добавлен новый упрощенный API (JMSContext и т.д.). Добавлены системные исключения JMSRuntimeExcepiton. Добавлены методы асинхронной отправки сообщений с использованием интерфейса прослушивателя MessageListener.

API JMS требует реализации, т.е. JMS-поставщика. Эталонным (RI) JMS-поставщиком является система Open Message Queue (Open MQ), входящая в состав сервера приложений GlassFish.

Асинхронность JMS

Принципиальной чертой JMS является асинхронность. В отличие от стандартного взаимодействия с сессионными EJB по RMI или HTTP (WS-представление EJB), где каждый вызов блокирует поток до полного выполнения запроса, JMS-клиент не ждет ответа на посланный им запрос. Т.е. под асинхронностью подразумевается временная независимость производителя и потребителя сообщения, при этом связь клиентов с источником может быть как асинхронной, так и синхронной.

Клиенты-производители отправляют запросы в т.н. темы (topic) или очереди (queue), откуда их могут получить другие клиенты-потребители. При этом ответственность за доставку лежит целиком на брокере, а отправитель продолжает выполнять свою работу безотносительно потребителя.

В некоторых случаях использование JMS выгоднее использования RMI. Получив от JMS-поставщика по JNDI JMS-соединение, EJB-компонент может использовать JMS для доставки сообщений другим Java-приложениям асинхронным образом.

В качестве примера можно рассмотреть регистрацию: пусть приложению требуется оповещать новых пользователей при их регистрации, но при этом не отвлекаться на это. Регистрацией может заниматься некоторый сессионный EJB — UserRegistrationEJB, отсылающий JMS-сообщения о завершении некоторой системе, слушающей регистрационные события. При этом сам UserRegistrationEJB может сразу же продолжать работу, не дожидаясь подтверждения приема или ответа. В качестве системы, прослушивающей регистрационные события-сообщения, может выступать MDB, другой EJB или вообще приложение на другой платформе, заинтересованное в оповещении о результате регистрации. Пример может включать рассылку электронных писем для подтверждения регистрации, включение пользователя в списки рассылки, причем все это может делаться отдельно от основного потока регистрации, не влияя на него.

JMS-клиенты не общаются напрямую, а используют для этого JMS-поставщика. Общение между JMS-клиентами и JMS-поставщиком уже более полноценно.

Контекст безопасности не передается от отправителя сообщений к получателю. Хотя JMS-поставщик может аутентифицировать пользователя по его сообщению, он не передаст эту информацию потребителю. Это связано с тем, что архитектура MOM рассчитана на разнородных производителей, которые могут работать на множестве платформ и использовать совершенно разные схемы безопасности, поэтому стандартизовать это в JEE тяжело.

Транзакции также не распространяются от отправителя получателю, т.к. неясно, кто будет получать очередное сообщение. А распределенная по тысяче получателей транзакция не является разумной. К тому же неясно, какие задержки будут при получении, будет ли вообще доступен получатель и т.д. Такое поведение с возможностью подвисания противоречит концепции оперативного выполнения транзакции, в процессе которой могут блокироваться общеприкладные и системные ресурсы. Но распределенные транзакции возможны между JMS-клиентами и JMS-поставщиком. Например, JMS-поставщик может отменить рассылку сообщений, если у отправителя произошел сбой и он решил откатить транзакцию.

Модели обмена JMS

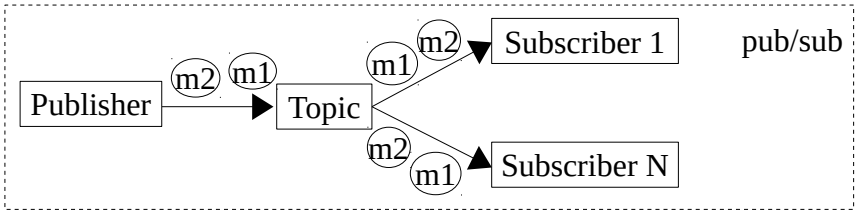
В JMS существует 2 модели обмена сообщениями (в спеке JMS 1.1 это messaging domains).

- publish-and-subsribe (pub/sub) — публикация/подписка
- point-to-point (p2p) — точка-точка

С JMS 1.1 введено общее API для обеих моделей: pub/sub и p2p.

Модель pub/sub (JMS Topic)

Модель pub/sub используется для рассылок 1:N.

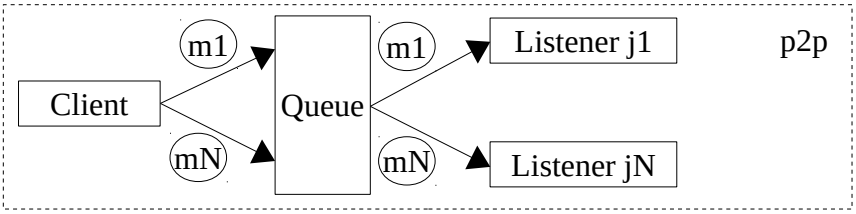


В модели публикация/подписка (pub/sub) производитель (producer) отправляет сообщение в виртуальный канал, называемый темой (topic). Потребители (consumers) могут подписать на эту тему и автоматически получать из нее сообщения. Такую систему рассылки называют еще основанной на проталкивании (push-based): потребителю не нужно слушать или запрашивать тему (topic) для получения сообщения, он делегирует это самой теме (topic) поставщика.

Производители сообщений не зависят от их потребителей. Потребляющие JMS-клиенты, использующие тему (topic), могут устанавливать длительную подписку, в течении которой могут отсоединяться и подсоединяться вновь, собирая непрочитанные сообщения. Причем сообщения каждому из подписчиков приходят в неопределенном порядке

Модель p2p (JMS Queue)

Модель p2p используется для сообщений, обрабатываемых однократно 1:1.



Модель точка-точка (p2p) позволяет клиентам отправлять и принимать сообщения в синхронном и асинхронном режимах через виртуальный канал, называемый очередью (queue). Модель точка-точка (p2p) по типу является системой опросного типа (pull/polling-based model), где сообщение надо запрашивать из очереди. К очереди может быть подключено множество потребителей, но каждое сообщение может быть

прочитано только одним из них, после чего оно удаляется. При этом название «очередь» имеет весьма символический характер: в JMS-спецификации не оговорен порядок распределения доступа потребителей к очереди, JMS гарантирует лишь то, что каждое сообщение будет читать единственный потребитель.

Производитель и потребитель никак не связаны во времени, потребитель может получать сообщения, присланные до его создания.

Схемы использования JMS

Модель pub/sub применяется там, где производителю нужно передавать копии сообщения n>>1 пользователям. И при этом его не должна заботить степень готовности потребителей принять эти сообщения. Также используя различные темы (topic) можно организовывать фильтрацию сообщений даже в схемах общения 1:1. При этом потребитель организовывает прием каждого вида сообщений через отдельный свой прослушиватель onMessage(). Классическим примером pub/sub является обычная почтовая рассылка подписчикам каких-то новостей.

Модель p2p применяется для схемы общения 1:1, между определенными клиентами, когда каждое сообщение имеет реальное значение. Также может иметь значение диапазон и вариативность данных в сообщениях. Радикальное отличие p2p от pub/sub в однократном потреблении каждого сообщения. Это может быть особенно важно тогда, когда необходимо обрабатывать сообщения отдельно, но последовательно. Классическим примером очереди сообщений (p2p) является минное поле, где на каждую мину может наступить любой, но только один раз. Или менеджер, победивший в драке за очередной звонок клиента.

Модель запрос-ответ предполагает определенную (событийную) синхронизацию: после доставки запроса на 1-м этапе обмена, производитель и потребитель меняются местами с целью доставки ответа. Часто ее реализуют при помощи p2p или pub/sub. Например, для ответа организуется ответная очередь, которую слушает производитель запроса, привязка ответа к запросу производится меткой сообщений correlationID.

Rem: спека JMS 1.1 не регламентирует конкретно как должны быть реализованы модели pub/sub и p2p. Каждая из моделей может использовать систему push или pull, но концептуально для pub/sub ближе push, а для p2p ближе pull.

JMS MDB

JMS MDB – асинхронный серверный компонент без сохранения состояния, служащий для обработки сообщений, доставляемых JMS. Это стандартный для JEE компонент, поэтому далее используется его сокращенное название — MDB. В то время как экземпляр MDB обрабатывает сообщения, EJB-контейнер управляет его окружением: транзакциями, распараллеливанием запросов, подтверждением приема сообщений и ресурсами.

MDB не сохраняет состояние между вызовами для обработки сообщений. MDB автоматом поддерживает многопоточность и потокобезопасность, в результате чего множество экземпляров MDB может параллельно обрабатывать огромное множество сообщений в каждый момент времени.

MDB – полноценный EJB. Он похож на SLSB, но в отличие от последнего, не обладает клиентскими представлениями, в т.ч. бизнес-интерфейсами, поскольку MDB работает напрямую только с контейнером и сообщениями, а не с клиентскими запросами.

JCA MDB

JMS MDB несмотря на все свои преимущества имеет ряд ограничений. Одним из основных является поддержка EJB-вендорами лишь малого количества сторонних JMS-поставщиков. Это связано с необходимостью поддержки EJB-контейнером единой транзакции, включающей все операции от передачи JMS к MDB сообщения и до его обработки. Для этого, перед началом реальной доставки сообщения, EJB-контейнер должен получить от JMS-поставщика уведомление о ее подготовке и открыть транзакцию. Но механизм предварительного оповещения EJB-контейнера со стороны JMS-поставщика не прописан в спеке JMS 1.1. Поэтому EJB-вендоры должны сами реализовывать эту интеграцию и поддерживают обычно только одного JMS-поставщика (а до EJB 2.1 вообще требовался свой JMS). Еще одним ограничением JMS MDB является отсутствие поддержки других EIS-систем, например, SOAP, email, CORBA Messaging, проприетарных EMS в составе ERP-SAP, PeopleSoft и т.д.

Спецификация EJB, начиная с EJB 2.1 дает более расширенное определение MDB, которое позволяет им быть обработчиками любых EIS любых вендоров. Единственным требованием к таким MDB является поддержка жизненного цикла MDB. EJB-вендоры могут создавать произвольный код для поддержки других EIS, но вместе с тем они должны обязательно поддерживать любые типы MDB, основанные на спецификации JCA 1.6.

JCA предусматривает стандартный интерфейс поставщика службы (Service Provider Interface, SPI). Этот интерфейс позволяет любой EIS встраиваться в любой JEE-контейнер. Версия JCA 1.6 позволяет JEE-серверам работать с любыми асинхронными EIS. При этом взаимодействие основано лишь на получении сообщения, предварительной подготовки не требуется.

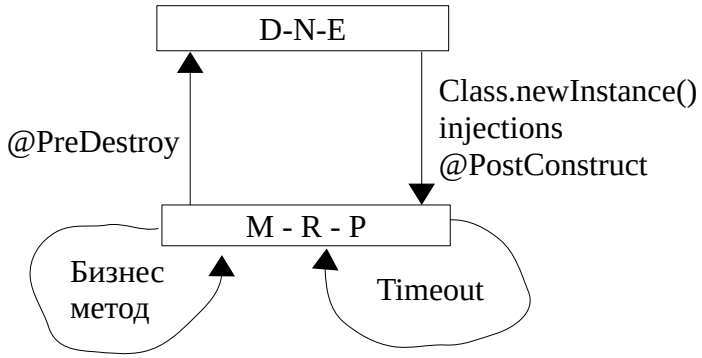
JCA определяет соглашение по обмену сообщениями (messaging contract), специально подогнанное под MDB. Оно определяет контракты между EJB-контейнером и асинхронным коннектором (JEE-интерфейс javax.resource.spi.Connector), таким образом, что MDB могут напрямую обрабатывать сообщения, полученные от EIS. JCA MDB могут реализовывать специфичный JEE-интерфейс, определенный на основе javax.resource.spi.Connector. Вместо javax.jms.MessageDriven MDB реализовывает некоторый другой интерфейс, подходящий для соответствующей EIS.

<u>Rem</u>: службы сообщений на базе JCA не обязаны реализовывать специфичные для JMS модели обмена Topic и Queue.

Жизненный цикл MDB

Жизненный цикл MDB, как и у всех компонентов без поддержки состояния, состоит из 3 стадий: отсутствие, ожидание и обработка. EJB-контейнер может перехватить только переходы из небытия и обратно, поэтому определено 2 состояния:

- Method-Ready-Pool (M-R-P) – пул ожидающих методов. Это состояние объекта MDB означает, что он присутствует в пуле экземпляров, но не используется
- Does Not Exist (D-N-E) – объект не существует в RAM



<u>Rem</u>: временные события timeout возникают при использовании службы таймеров.

Жизненный цикл MDB. D-N-E → M-R-P. Создание экземпляра

- EJB-контейнер вызывает конструктор по-умолчанию (void без аргументов) и создает экземпляр в RAM
- EJB-контейнер производит внедрение в экземпляр объектов и обработку метаинформации
- EJB-контейнер вызывает метод @PostConstruct, если он определен

Жизненный цикл MDB. M-R-P. Ожидание и обработка

После получения EJB-контейнером сообщения от JMS-поставщика, оно переадресуется одному из экземпляров MDB в пуле (M-R-P). Таким образом может параллельно обрабатываться множество сообщений. При этом для задействованного экземпляра создается новое транзакционное окружение и тем самым меняется его MessageDrivenContext. Во врем обработки сообщения экземпляр недоступен, сразу по окончанию экземпляр возвращается в пул.

Жизненный цикл MDB. M-R-P → D-N-E. Ликвидация экземпляра

EJB-контейнер может очистить RAM от одного или более экземпляров MDB из пула M-

R-P. При этом может быть выполнен метод PreDestroy: допускается создание единственного метода без аргументов, возвращающего void и помеченного аннотацией @PreDestroy. Обычно в нем закрываются незакрытые соединения, в его коде еще доступен MessageDrivenContext и JNDI ENC. После этого обнуляются все ссылки на экземпляр MDB и он становится доступен для сборщика мусора.

Rem: в жизненный цикл EJB, в т.ч. MDB, могут вмешиваться перехватчики конструкторов, вызовов методов.

Параллелизм и потокобезопасность MDB

MDB-компонент похож на SLSB и обладает встроенным механизмом распараллеливания приема сообщений. Спецификация требует, чтобы в любой момент времени любой его экземпляр принимал одно сообщение от одного однопоточного контекста JMS-поставщика. Это означает, что никакой дополнительной потокобезопасности для приема сообщений в классе MDB обеспечивать не нужно. EJB-контейнер просто распределяет доставку сообщений от JMS-поставщика по отдельным экземплярам из пула.

JMS MDB vs Session EJB vs CDI

JMS отвечает основным требованиям MOM-архитектуры: асинхронность, слабая связанность, надежность.

Если нужна просто асинхронность, а надежность и слабая связанность не очень важны, то проще использовать асинхронные методы (@Asynchronous) сессионных EJB. При этом клиент будет жестко привязан к представлению EJB, а сбой при вызове приведет к потере экземпляра EJB или сессии.

Если нужна слабая связанность и асинхронность без отказоустойчивости, то можно использовать CDI MBean с механизмом внедрения объектных событий (Event<T>) и организацией их приемников (@Observes). При этом можно использовать @Asynchronous. Но без встроенной отказоустойчивости.

Когда нужно все вместе, то удобнее всего использовать специализированную систему обмена сообщениями вроде JMS: рассылки, очереди, поддержка off-line режима клиентов, дублирование запросов при сбоях, фильтрации и т.д. В API JMS 2.0 (JEE7) появилась возможность использовать асинхронный прослушиватель сообщений MessageListener. Можно использовать API JMS для организации синхронного или асинхронного приема в сессионных EJB, но при этом придется заниматься программной настройкой и получением администрируемых объектов источника, фабрики или JMS-контекста, писать код для их использования. Наиболее удобным для организации в JEE асинхронного, слабосвязанного, надежного и параллельного обмена является MDB-компоненты.

Применение MDB. Аннотация @MessageDriven

MDB является EJB-компонентом, осуществляющим потребление и производство сообщений различных EIS, в т.ч. коробочной JMS.

Аннотация @javax.ejb.MessageDriven помечает класс как реализацию MDB-компонента. EE аналогом в XML-дескрипторе ejb-jar.xml является элемент <message-driven>. Компонент MDB может развертываться EJB-контейнером отдельно, но обычно это происходит вместе с другими связанными с ним EJB.

Аннотация MessageDriven содержит следующие необязательные атрибуты:

- name - имя EJB-компонента (компонентная часть глобального JNDI-имени в переносимом синтаксисе), используемое при развертке приложения в deploy-time, соответствующее <ejb-name> стандартного дескриптора ejb-jar.xml. По-умолчанию это короткое имя класса ("ExBean").
- mappedName – JNDI-имя (произвольное глобальное) источника
- messageListenerInterface – F.Q.N. интерфейса асинхронного прослушивателя, который указывается для JMS MDB в случае множественной реализации интерфейсов классом MDB. Он позволяет избавиться от реализации MessageListener классом MDB, т.е. превратить его в класс POJO.
- description – строковое описание компонента
- activationConfig – набор строковых параметров для настройки поставщика сообщений

У атрибутов этой аннотации есть аналоги в дескрипторе ejb-jar.xml:

Ex: определение в ejb-jar.xml асинхронного прослушивателя MessageListener. в MDB

```
<ejb-jar ...>
<enterprise-beans>
<message-driven>
…
<messaging-type>javax.jms.MessageListener</messaging-type>
</message-driven>
</enterprise-beans>
</ejb-jar>
```

MDB должен уметь обрабатывать различные форматы сообщений, его настройка должна быть достаточно гибкой для того, чтобы описывать множество проприетарных свойств, встречающиеся у разных поставщиков как JMS, так и EIS (для JCA MDB). Поэтому для настройки MDB в @MessageDriven применяется гибкий атрибут — activationConfig, содержащий массив свойств — строковых пар имя-значение, оформленных в виде @javax.ejb.ActivationConfigProperty. Эти свойства сильно зависят от типа и реализации используемой EIS.

Ex: пример настройки MDB для JMS-поставщика Open MQ 5.1 (GlassFish 4.1)

```
@MessageDriven(activationConfig={
  @ActivationConfigProperty(propertyName="destinationLookup", propertyValue="xxx/Queue"),
  @ActivationConfigProperty(propertyName="destinationType",propertyValue="javax.jms.Queue"),
  @ActivationConfigProperty(propertyName="messageSelector",
    propertyValue="MessageFormat = 'Version 3.4'"),
  @ActivationConfigProperty(propertyName="acknowledgeMode",
    propertyValue="Auto-acknowledge")})
public class ExMDB implements javax.jms.MessageListener {
  …
}
```

В JMS 2.0 стандартизованы следующие свойства (для настройки EJB-контейнером JMS-контекста и создания требуемого JMS-потребителя в MDB-компоненте):

- acknowledgeMode – режим подтверждения приема сообщения (AUTO_ACKNOWLEDGE)
- messageSelector – селектор сообщения MDB
- destinationType – тип источника (QUEUE, TOPIC)
- destinationLookup – произвольное глобальное JNDI-имя административно-созданного источника потребления MDB
- connectionFactoryLookup – произвольное глобальное JNDI-имя административно-созданной фабрики соединений
- subscriptionDurability – длительность подписки (NON_DURABLE) на Topic
- subscriptionName – имя потребителя для длительной подписки на Topic
- clientId – идентификатор клиента, используемый для идентификации потребителя длительной подписки на Topic
- shareDescriptions – флаг использования одинаковых имен подписки серверными процессами в кластере

Свойство messageSelector

Именованное свойство messageSelector позволяет JMS-поставщику распознавать типы сообщений, поступающие в некоторый источник и фильтровать их доставку для MDB. Информация о типе берется из свойств и заголовка входного объекта-сообщения интерфейса javax.jms.Message. JMS-поставщик сравнивает эти параметры со значениями, полученными им в deploy-time от EJB-контейнера из метаинформационных свойств messageSelector MDB, и при помощи булевой логики определяет тип доставленного сообщения.

Селекторы сообщений представляют собою строковые логические выражения, использующие синтаксис оператора WHERE SQL92. Селекторов может быть много, они объединяются конъюнктивно. Их операндами выступают имена свойств и частей заголовка простых типов, литералы. Допускаются логические операторы (NOT, AND,

OR), сравнения (=, <>, >,<,>=,<=), скобки, арифметические (+,-,*,/) и выражения ([NOT] BETWEEN, [NOT] IN, [NOT] LIKE, IS [NOT] NULL).

```
Ex: пример селекторов MDB для фильтрации JMS-поставщиком доставляемых сообщений по заголовочному приоритету и прикладным свойствам price и MessageFormat:

@ActivationConfigProperty(propertyName="messageSelector", propertyValue="MessageFormat = 'Version 3.4' AND (price BETWEEN 10 AND 20)"),
@ActivationConfigProperty(propertyName=”messageSelector”, propertyValue=”JMSPriority < 6”)
```

Rem: активное применение селекторов может ухудшить производительность. Иногда лучше развести потоки сообщений по разным источникам

Свойство acknowledgeMode

В JMS существует понятие подтверждения доставки сообщения (acknowledgment). JMS-клиент уведомляет JMS-поставщика о том, что он получил сообщение. Если такого подтверждения нет, то JMS-поставщик может отправить сообщение вновь.

При использовании MDB в качестве потребляющего JMS-клиента, ответственность за подтверждение берет на себя EJB-контейнер. Если MDB CMT, то EJB-контейнер включает в глобальную операцию доставки метод onMessage(…) и в зависимости от ее фиксации или отката подтверждает или не подтверждает JMS-поставщику доставку. Если же в MDB вместо CMT используется BMT или транзакции отключены, то EJB-контейнер может подтверждать доставку в случае успешного выполнения onMessage(…), а в случае выброса непроверяемого исключения не подтверждать, но это зависит от вендора.

Rem: тест в Glassfish 4.1 (Open MQ 5.1) показал дублирование отправки сообщения при выбросе системного исключения в onMessage(...) вне транзакции.

Режим подтверждения успешной доставки задается именованной аннотацией-свойством с допустимыми значениями Auto-acknowledge и Dups-ok-acknowledge. По-умолчанию используется автоподтверждение Auto-acknowledge.

```
@ActivationConfigProperty(propertyName="acknowledgeMode", propertyValue="Auto-acknowledge")
@ActivationConfigProperty(propertyName="acknowledgeMode", propertyValue="Dups-ok-acknowledge")
```

Auto-acknowledge – режим, означающий, что EJB-контейнер вышлет подтверждение после успешного завершения метода onMessage() MDB без задержек.

Dups-ok-acknowledge означает, что EJB-контейнер может подтверждать доставку по своему усмотрению, причем подразумевается, что задержка может спровоцировать JMS-поставщика на повторную отправку дубликата сообщения (и этот дубликат надо уметь обрабатывать в MDB). Теоретический смысл Dups-ok-acknowledge в разгрузке сети, но на практике он применяется редко.

Свойство subscriptionDurability

Для тех JMS MDB, что используют рассылочную схему Topic, должен быть описан тип подписки:

- Durable (длительная подписка)
- NonDurable (краткосрочная подписка)

По-умолчанию используется краткосрочная подписка NonDurable. Задается тип подписки в именованном свойстве-аннотации со значениями Durable, NonDurable.

```
@ActivationConfigProperty(propertyName = "subscriptionDurability", propertyValue = "Durable"),
```

Durable – тип длительной подписки, при которой JMS-поставщик сохраняет сообщения, которые не были отправлены MDB в результате разрыва связи, т.е. off-line клиентам. Как только это соединение будет восстановлено, JMS-поставщик передаст все сохраненные сообщения MDB через EJB-контейнер. Длительная подписка доступна только одному потребителю, идентифицируемому JMS-поставщиком по clientID и имени подписки subscriptionDurability , заданных в соседних свойствах @ActivationConfigProperty. После создания, такая подписка начинает сохранят off-line сообщения (?)до завершения жизни приложения (?).

Ex: пример описания длительной подписки:

```
@ActivationConfigProperty(propertyName = "destinationType", propertyValue = "javax.jms.Topic"),
@ActivationConfigProperty(propertyName = "destinationLookup", propertyValue = "xxx/Topic"),
@ActivationConfigProperty(propertyName = "subscriptionDurability", propertyValue = "Durable"),
@ActivationConfigProperty(propertyName = "subscriptionName", propertyValue = "xxx/Topic"),
@ActivationConfigProperty(propertyName = "clientId", propertyValue = "xxx/Topic")
```

NonDurable – тип краткосрочной подписки, при которой теряются все сообщения, не переданные из-за разрыва связи. Т.е. получать сообщения можно только on-line. Это позволяет повысить производительность, вместе с тем ухудшает надежность обработки MDB. И рекомендуется только в том случае, когда потеря сообщений не является критичной.

Для очередей MDB с рассылочной схемой Queue тип подписки не нужен, т.к. очереди имеют другую политику рассылки.

Применение MDB. Интерфейс MessageDrivenContext

Интерфейс javax.ejb.MessageDrivenContext – еще один наследник базового окружения EJB – интерфейса javax.ejb.EJBContext. Получается он, как и прочие контексты EJB, через JNDI или внедрение, например, при помощи @Resource.

В отличие от контекста сессионных EJB, реализация контекста MDB должна глушить выбросом непроверяемых (RuntimeException) исключений методы EJBContext, связанные с клиентскими представлениями. У MDB-компонентов нет представителей, т.к. их единственным посредником является JMS-поставщик, поэтому методы контекста getEJB*() не нужны.

В области защиты доступа к MDB возможно получение из контекста JAAS-представления отправителя сообщения методом getCallerPrincipal() и проверка принадлежности его к заданной роли методом isCallerInRole(). Однако, по идеологии MOM, производитель и потребитель связаны слабо через посредника. Т.е. непосредственным отправителем сообщений для MDB является JMS-поставщик (или другая EIS с JCA-адапетром), а не производитель. В добавок производители могут иметь самую разную природу и формы аутентификации в EIS, поэтому стандартных механизмов аутентификации и авторизации производителя сообщений в MDB нет.

Rem: в Glassfish 4.1 (Open MQ 5.1), в контекст MDB попадает принципал ANONYMOUS с пустой ролью null.

Транзакции не распространяются на MDB от производителя сообщения. Они могут либо инициироваться EJB-контейнером (для CMT MDB) и охватывать операцию доставки или отправки сообщений из MDB, либо задаваться в самом коде MDB (javax.transaction.UserTransaction для BMT MDB). Метод получения компонентной транзакции getUserTransaction() может применяться для BMT MDB, а getRollbackOnly() и setRollbackOnly() для CMT MDB.

Компоненты MDB, как и прочие EJB, обладают технологией JNDI ENC, позволяющей внедрять и получать ссылки на различные управляемые ресурсы. Прямые JNDI-запросы можно делать через метод контекста lookup(…).

Компонентам MDB доступна служба таймеров EJB. MDB-таймеры похожи на SLSB-таймеры. Основным отличием от SLSB-таймера является способ инициализации: либо при получении сообщения, в коде его обработчика, либо, если дополнительно допускается вендором, через конфигурационный файл в deploy-time. В коде класса реализации MDB-компонента таймер можно получить методом контекста getTimerService().

Применение MDB. Интерфейс MessageListener. Метод onMessage. Потребление сообщений

Интерфейс асинхронного прослушивателя javax.jms.MessageListener реализуют JMS MDB. Этот интерфейс содержит только один метод, в котором и должно обрабатываться сообщение:

```
public void onMessage(Message message);
```

Rem: никто не запрещает MDB интегрироваться с другими EIS, которые могут определять свои интерфейсы, отличные от MessageListener.

Пример onMessage() для задач B2B

В методе onMessage() сосредоточена вся бизнес-логика JMS MDB. EJB-контейнер передает все принятые сообщения именно в этот метод. После завершения работы onMessage() экземпляр MDB вновь возвращается в пул и вновь ожидает сообщений.

```
Ex: пример реализации бизнес-логики MDB

public void onMessage(Message message) {
  try {
    // Извлечение информации из MapMessage
    MapMessage mapMessage = (MapMessage)message;
    int userId = mapMessage.getInt("userId");
    String email = mapMessage.getInt("email");
    // Обработка тела сообщения
}
```

Функционально MDB часто используется как интегратор (точка сборки) B2B-приложений: MDB принимает сообщения от внешних бизнес-партнеров вроде сторонних обработчиков или систем проверки и обрабатывает их. При этом MDB может получать доступ к сессионным EJB, взаимодействовать с ними, управлять ресурсами, например, использовать JPA для связывания сообщений с РБД.

Потребление сообщений при помощи API JMS

Никто не запрещает организовывать потребление сообщений в EJB, в частности в MDB, при помощи API JMS. При помощи объекта производителя JMSConsumer можно организовать как асинхронный, так и синхронный прием из источника. Для организации асинхронного режима регистрируется прослушиватель MessageListener (в MDB он уже встроен). Для создания синхронного приема запускается бесконечный цикл, в котором вызывается метод recieve(...) производителя, запрашивающий сообщения из источника.

Синхронный прием не рекомендован в MDB или SLSB, т.к. подвисшие циклы приема могут привести к исчерпанию пулов.

Применение MDB. Производство сообщений в MDB

MDB может также отправлять сообщения через API JMS. Ниже используется упрощенное API, появившееся в JMS 2.0, реализующее текучий интерфейс.

Перед отправкой надо получить от JMS-поставщика администрируемые (созданные вне

приложения) объекты: контекста javax.jms.JMSContext и источника javax.jms.Destination. Контекст является прокси JMS-поставщика, от лица которого и идет обмен с клиентами. Источник — прокси того хранилища JMS-поставщика, в котором накапливаются сообщения при обмене между заданными клиентами. Источники в зависимости от схемы обмена делятся на Topic (тема, схема подписки pub/sum) и Queue (очередь, схема точка-точка p2p), но в программном коде достаточно использования их обобщенного интерфейса Destination. В MDB для получения администрируемых объектов лучше всего использовать внедрение. Для внедрения контекста можно использовать @Inject с уточняющими аннотациями вроде @JMSConnectionFactory, @JMSSessionMode, @JMSPasswordCredential. А для внедрения источника используется @Resource.

Ex: внедрение в MDB администрируемых объектов JMS-поставщика

```
@MessageDriven
public class ExMDB implements MessageListener {
  @Inject
  @JMSConnectionFactory("xxx/Factory")
  private JMSContext jmsContext;
  @Resource(lookup="xxx/Queue")
  private Destination destination;
...
}
```

При помощи контекста и источника создается производитель сообщений javax.jms.JMSProducer и сами сообщения java.jms.Message. Производитель сообщения является прокси отправителя сообщений JMS-поставщика. Сообщения — единица обмена между клиентами, производитель посылает их в источник, а потребитель их оттуда получает.

Сообщения создаются либо явно при помощи контекста, либо неявно при отправке методами send(…) производителя. Каждое сообщение JMS можно разбить на 3 части:

- заголовок (header) — совокупность предопределенных именованных свойств (get/set) Message, в которых хранится стандартизованная информация для доставки и идентификации сообщения
- свойства (properties) — пары имя-значение, несущие произвольную прикладную информацию простых типов, в частности, позволяющие источнику фильтровать сообщения по своим значениям
- тело (body) — полезная прикладная нагрузка сообщения, записанная в одном из допустимых форматов

Доступ к частям заголовка сообщения обеспечивают методы getXxx/setXxx интерфейса Message. Заголовок содержится информация о маршруте, типе сообщения и политике доставки. Части заголовка либо устанавливаются вручную в объекте Message в коде MDB, либо JMS-поставщиком в клиентском вызове (через прокси производителя) его

метода send(), либо самостоятельно JMS-поставщиком:

```
Ex: установка в заголовке JMSCorrelationID клиентом вручную

Message msg = jmsContext.createTextMessage("Test text");
msg.setJMSCorrelationID("xxx");
```

```
Ex: установка в заголовке приоритета через интерфейс производителя

jmsContext.createProducer().setPriority(4).send(destination, msg);
```

Часть заголовка	**Описание**	**Установщик**
JMSDestination	источник	send(...)
JMSDeliveryMode	Указывает JMS-поставщику режим доставки: PERSISTENT — сохранять сообщение в ROM в рамках операции send() для восстановления при сбоях; NON_PERSISTENT — экономный режим, не сохраняющий сообщение в ROM, гарантирующий одноразовость (без дублей при сбое)	send(...)
JMSMessageID	UID сообщения, посылаемого поставщиком. Уникальность должна обеспечиваться в рамках сообщений, передаваемых подсоединенными роутерами (брокерами) сообщений используемого JMS-поставщика. UID должен ставить клиент, некоторые JMS-поставщики допускают null и ставят его сами для экономии трафика	send(...)
JMSTimeStamp	Время поступления сообщения поставщику, msec.	send(...)
JMSCorrelationID	Метка для связывания сообщений. Полезна в режиме запрос-ответ	send(...)
JMSReplyTo	источник (Destination) для перенаправления ответа. Полезен в режиме запрос-ответ, для указания адресата ответа	клиент
JMSRedelivered	Флаг повторной доставки, сообщающий потребителю, что, возможно, это сообщение уже посылалось ему ранее, но тогда подтверждения приема не последовало	поставщик
JMSType	Вендор-специфический идентификатор типа сообщения	клиент
JMSExpiration	Время истечения сообщения, устанавливаемое JMS-поставщиком путем сложения текущего времени получения и времени жизни сообщения, установленного клиентом	send(...)
JMSPriority	Приоритет сообщения. Может влиять на скорость доставки сообщений JMS-поставщиком. Всего 10 уровней: 0-4 нормальные, 5-9 срочные	send(...)

Свойства Message — это дополнительные типизированные свойства, дополняющие

стандартные именованные части заголовка сообщения. Они несут прикладную информацию простых типов boolean, byte, short, int, long, float, double, String. Доступ к ним обеспечивают методы Message setXxxProperty(name,value) и getXxxProperty(name), где Xxx — один из простых типов, name — строковое имя, а value — значение типа Xxx.

Ex: примеры задания свойств и части заголовка (приоритета) сообщения с отправкой

```
message.setStringProperty(“MessageFormat”,”Version 3.4”);
message.setIntProperty(“price”,12.5);
jmsContext.createProducer().setPriority(4).send(destination,message);
```

Тело Message — полезная нагрузка сообщения в заданном формате: текст, параметры, сериализуемые объекты, байтовые потоки и т.п. Для каждого типа предусмотрен свой тип сообщения: TextMessage, MapMessage, ObjectMessage, StreamMessage, BytesMessage.

В заголовке присутствует часть JMSReplyTo. Отправитель сообщения может задать в нем адрес любого источника, доступного JMS-поставщику. MDB может считать его и использовать для отправки своего сообщения.

В частности, подобную схему можно использовать для информирования об ошибках, возникающих при бизнес-обработке сообщений.

Ex: фрагменты MDB, занимающегося переотправлением писем

```
@MessageDriven
public class ExMDB implements MessageListener {
  @Inject
  @JMSConnectionFactory(“xxx/Factory”)
  private JMSContext jmsContext;
  @Resource(lookup=”xxx/Queue”)
  private Destination queue;
```

Ex: шаблонный пример реализации бизнес-логики MDB

```
  public void onMessage(Message message) {
      // Извлечение заголовков из сообщения
      MapMessage mapMessage = (MapMessage)message;
      int userId = mapMessage.getInt("userId");
      String email = mapMessage.getInt("email");
      // Обработка тела сообщения
      …
      // Создание некоторого сериализуемого объекта - письма
      Ticket t = …;
      // Отправка письма по адресу JMSReplyTo входного message
      exMethod(message,t);
  }
```

```
 ...
 public void exMethod(Message message, Ticket t) throws JMSException{
    Destination destination = message.getJMSReplyTo();
    jmsContext.createProducer().send(destination,t);
 }
}
```

Применение MDB. Пример JMS-приложения

Для связи с JMS-поставщиком MDB использует API JMS. В т.ч. ряд управляемых EJB-контейнером ресурсов — фабрик, мест назначения и т.п. Все это позволяет MDB как принимать сообщения, так и отправлять их.

В качестве примера рассматривается система оповещения об изменении статуса. Некоторое клиентское Java-приложение шлет сообщение об изменении статуса на JMS Topic. Рассылка с Topic идет в различные MDB: 1-й MDB просто ведет журнал изменений статуса; 2-й MDB передает изменения статуса внешней стороне, скажем Twitter.

Rem: полный текст примера для EJB 3.1 / JMS 1.1 -

Класс StatusUpdate – POJO-класс-контейнер, где хранится и обновляется статусная строка.

```
public class StatusUpdate {
  private final String status;
  public StatusUpdate(final String status) throws IllegalArgumentException {
    this.status = status;
  }
  public String getText() {
    return status;
  }
}
```

Клиентское приложение использует классическое API (JMS 1.1). Через JNDI-запросы запрашиваются прокси источника JMS Topic и ConnectionFactory. Объекты Connection, Session, MessageProducer создаются в try-с-ресурсами, что обеспечивает их автоматическое закрытие. После чего создается объектное сообщение Message и начинается рассылка через topic вызовом send(message):

```
...
void sendStatusUpdate(final StatusUpdate status) throws Exception {
//   java.util.Hashtable<String,String> ht = …;
...
  final Context ctx = new InitialContext(ht);
  final Destination topic = (Destination) ctx.lookup("TopicJNDIName");
  final ConnectionFactory factory = (ConnectionFactory) ctx.lookup("FactoryJNDIName");
```

```
  try(final Connection con = factory.createConnection();
      final Session session = con.createSession(false, TopicSession.AUTO_ACKNOWLEDGE);
      final MessageProducer producer = session.createProducer(topic)) {
    final Message message = session.createObjectMessage(status);
    producer.send(message);
  }
}
```

На стороне сервера находится 2 прослушивателя JMS Topic: журналирующий (logger) и публикующий в twitter MDB-компоненты. Поскольку эти компоненты различаются лишь дальнейшей обработкой полученного сообщения, то целесообразно вынести общий код прослушивания сообщений в абстрактный суперкласс, реализующий базовый для JMS MDB интерфейс MessageListener.

```
public abstract class BaseBean implements MessageListener {
  public abstract void updateStatus(StatusUpdate newStatus) throws IllegalArgumentException;
  @Override
  public void onMessage(final Message message) {
// Проверка типа входящего сообщения
    final ObjectMessage objMessage;
    if (message instanceof ObjectMessage) {
      objMessage = (ObjectMessage) message;
    }
    else {
      throw new IllegalArgumentException("Specified message must be of type " +
        ObjectMessage.class.getName());
    }
// Попытка извлечения тела сообщения
    final Serializable obj;
    try {
      obj = objMessage.getObject();
    }
    catch (final JMSException jmse) {
      throw new IllegalArgumentException("Could not obtain contents of message " + objMessage);
    }
// Проверка типа объекта из сообщения
    final StatusUpdate status;
    if (obj instanceof StatusUpdate) {
      status = (StatusUpdate) obj;
    }
    else {
      throw new IllegalArgumentException("Contents of message should be of type " +
        StatusUpdate.class.getName() + "; was instead " + obj);
    }
// Обработка полученного из сообщения нового состояния статуса
    this.updateStatus(status);
  }
}
```

Этот абстрактный прослушиватель расширяется в 2 MDB: LoggingMDB и TwitterMDB.

Ex: журналирующий MDB (интерфейс прослушивателя BaseBean тут определен в свойстве MessageListenerInterface аннотации @MessageDriven)

```
@MessageDriven(MessageListenerInterface = BaseBean.class,
  activationConfig = { @ActivationConfigProperty(propertyName=”destinationType”,
propertyValue=”javax.jms.Topic”),
  @ActivationConfigProperty(propertyName=”destination”, propertyValue=”TopicJNDIName”)})
public class LoggingMDB {
  private static final Logger log =Logger.getLogger(LoggingMDB.class.getName());

   public void updateStatus(final StatusUpdate newStatus) throws IllegalArgumentException {
      final String status = newStatus.getText();
      log.info("New status received: \"" + status + "\"");
   }
}
```

Ex: публикующий в twitter MDB

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(name=”destinationType”,value=”javax.jms.Topic”),
  @ActivationConfigProperty(name=”destination”,value=”TopicJNDIName”)})
public class TwitterMDB implements BaseBean {
   private Twitter client;
// Инициализация экземпляра MDB
   @PostConstruct
   void createTwitterClient() {
// Получение клиента Twitter при помощи внешнего API
     client = …;
   }

   @Override
   public void updateStatus(final StatusUpdate newStatus)  throws IllegalArgumentException {
      final String status = newStatus.getText();
      client.updateStatus(status);
   }
}
```

Применение JCA MDB

В качестве гипотетического примера можно рассмотреть обмен почтовыми сообщениями: некоторый Email Connector и соответствующие ему JCA MDB обрабатывают письма так же, как JMS позволяет JMS MDB обрабатывать JMS-сообщения. Вендор создает этот JCA-коннектор, пакует его в jar-архив, называемый ресурсным rar-архивом. Далее этот rar-архив, содержащий код Email Connector и необходимые дескрипторы развертки встраивается в EJB-контейнер. Также rar-архив содержит интерфейс обмена сообщениями (messaging interface), который должен реализовываться всеми EmailMDB.

Ex.: пример интерфейса обмена почтовыми сообщениями

```
public interface EmailListener {
   public void onMessage(javax.mail.Message message);
}
```

Ex.: гипотетический пример обработчика почтовых сообщений

```
@MessageDriven(activationConfig={
  @ActivationConfigProperty(propertyName="mailServer", propertyValue="mail.ispx.com"),
  @ActivationConfigProperty(propertyName="serverType", propertyValue="POP3 "),
  @ActivationConfigProperty(propertyName="messageFilter", propertyValue="to='ex@xxx.yyy'")})
public class EmailMDB implements EmailListener {
   public void onMessage(javax.mail.Message message) {
     //  бизнес-логика обработки письма
   }
}
```

В примере EJB-контейнер вызывает onMessage(…) EmailMDB при доставке почтового сообщения типа javax.mail.Message.

Rem: сигнатуры и имена методов интерфейса обмена могут быть любыми подходящими для соответствующей EIS. Метод обработки сообщения может даже возвращать результат.

В качестве еще более необычного примера можно рассмотреть пример обмена SOAP-сообщениями в стиле запрос/отклик. Некоторый SOAP Connector может использовать интерфейс ReqRespListener из JAXM (Java API for XML Messaging) – API обмена SOAP-сообщениями, разработанного Sun, но не вошедшего в JEE.

Ex.: пример интерфейса обмена SOAP-сообщениями (XML из SOAP WS)

```
public interface ReqRespListener {
  public SOAPMessage onMessage(SOAPMessage message);
}
```

В этом примере onMessage(…) не только обрабатывает входное сообщение но и посылает SOAPMessage в качестве отклика, при этом предполагается, что EJB-контейнер и Connector ответственны за доставку этого отклика отправителю сообщения.

Rem: помимо различий в сигнатурах, каждый интерфейс обмена сообщений может определять и различные методы для обработки различных видов сообщений.

EJB-контейнер может поддерживать неограниченное количество новых видов JCA MDB. При этом JCA MDB полностью независимы от EJB-вендора. Т.е. для переноса какой-либо системы обмена сообщения, основанной на JCA в другой EJB-контейнер, достаточно перетащить rar-архив и код соответствующего JCA MDB.

Метаинформация, определяющая поведение JCA MDB-компонента, имеет ту же структуру, что и для стандартного JMS MDB, но содержание зависит от типа использующегося JCA-коннектора и его требований.

```
@MessageDriven(activationConfig = {
  @ActivationConfigProperty(propertyName = "mailServer", propertyValue="mail.ispx.com"),
  @ActivationConfigProperty(propertyName = "serverType", propertyValue="POP3 "),
  @ActivationConfigProperty(propertyName = "messageFilter", propertyValue="to='ex@xxx.yyy'")})
```

Применение MDB. Отправка сообщений из EJB. Message Linking

Связывание сообщений (message linking) – метод, позволяющий направлять сообщения, посланные некоторым EJB-компонентом к заданному MDB-компоненту того же приложения. Это достигается применением виртуальных конечных точек (endpoint) вместо реальных JCA-endpoint или источников. Данная техника позволяет посредством логических имен источников ослаблять связь между EJB-компонентами и поставщиком сообщений.

Логическое имя (ссылка на источник) может определяться в стандартном дескрипторе ejb-jar.xml (или web.xml для WEB-контейнера) в элементе <message-destination-link> и привязываться к реальному источнику через <message-destination-name>. Это логическое имя и используется в MDB, при внедрении или поиске источника через JNDI ENC. При развертке приложения логическое имя связывается с настоящим именем источника.

```
Ex: пример отправки SLSB сообщения по логическому имени, связанному посредством message
linking с физическим источником

@Stateless(name = "ExEJB")
@Local(ExLocal.class)
public class ExEJB implements ExLocal {
//Внедрение очереди для посылки сообщений, ее логическое имя связано с физическим
// источником посредством ENC JNDI <message-destination-link> (?)
   @Resource(name = "jms/MessageDestinationLinkQueue")
   private Queue queue;
…
}
```

Применение MDB. Транзакции

По причине слабой связанности клиентов в МОМ через посредника — поставщика сообщений, транзакционный контекст не передается от производителя сообщения к потребителю. Т.е. MDB ничего не знает о клиентской транзакции. От лица клиента выступает JMS-поставщик, вот с ним и возможно транзакционное взаимодействие.

JMS-поставщик ответственен за доставку сообщения потребителю. В JMS определен механизм локальных транзакций и механизм подтверждения доставки сообщения (acknowledgement), которыми можно явно пользоваться в JSE-клиентах. В случае MDB-клиента подтверждением доставки заведует EJB-контейнер. С точки зрения EJB-контейнера доставка MDB-компоненту сообщения происходит успешно в случае успешного приема объекта Message из источника и его безошибочной обработки в методе потребления onMessage(…).

EJB-контейнер может контролировать доставку включением приема и обработки сообщения в глобальную контейнерную транзакцию (CMT). При отсутствии транзакции, EJB-контейнер может использовать проверку выброса исключений и механизм подтверждения JMS-поставщика. Отсюда возникает 2 транзакционных режима метода потребления onMessage() MDB-компонента: либо метод не включается в транзакции, либо использует глобальную транзакцию операции доставки. В результате для onMessage(…) допускаются два транзакционных режима CMT: NOT_SUPPORTED (не использовать транзакции) и REQUIRED (включаться в глобальную транзакцию доставки, режим по-умолчанию).Использование BMT для onMessage(...) при доставке игнорируется и приравнивается к NOT_SUPPORTED CMT.

Из MDB-компонента также возможна отправка сообщений при помощи JMS API. Отправка сообщений MDB-кмпонентом может вовлекаться в глобальную JTA-транзакцию, распространяющуюся и на JMS-поставщика. При этом можно использовать как контейнерную (CMT), так и компонентную (BMT) транзакции.

Отсутствие транзакции

При отсутствии BMT или неактивной (NOT_SUPPORTED) CMT подтверждение доставки делается EJB-контейнером в случае успешного завершения метода onMessage(), в противном случае JMS-поставщик, в зависимости от реализации, может переотправить дубль сообщения. Отправка сообщений из MDB вне транзакции идет сразу, без задержек.

Активная (REQUIRED) CMT

Использование контейнерной транзакции в режиме REQUIRED – это транзакционный режим MDB по-умолчанию. В этом случае EJB-контейнер открывает глобальную транзакцию еще до получения сообщения из источника JMS-поставщика и вызова метода потребления onMessage(…) экземпляра MDB. В эту транзакцию вовлекаются и весь метод onMessage().

Фиксация CMT по завершению метода onMessage(...) подтвердит JMS-поставщику успешную доставку. Также при фиксации будут отправлены в источники JMS-поставщика сообщения, произведенные в этом методе.

Откат CMT приведет к неподтверждению доставки JMS-поставщику, при этом

произведенные в onMessage(…) сообщения никуда отправляться не будут. В ответ на неуспешную доставку JMS-поставщик может отправить дубль сообщения.

```
// Ex.: откат глобальной транзакции, JMS-поставщик может в ответ выслать дубль
// @TransactionManagement(TransactionManagementType.CONTAINER)
@MessageDriven(…)
public class ExMDB implements MessageListener {
  @Resource private MessageDrivenContext mdbContext;

  @Override
// @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void onMessage(Message message) {
    ...
    mdbContext.setRollbackOnly();
    ...
  }
}
```

BMT

Использование компонентной транзакции (BMT) UserTransaction приводит к тому, что onMessage(…) не включается в транзакцию операции доставки, т.е. JMS-поставщик использует собственную локальную транзакцию. При откате BMT в методе onMessage() локальная транзакция JMS-поставщика не откатится и автоматической переотправки сообщения не будет.

Однако вызовы транзакционных методов в рамках UserTransaction, в т.ч. отправка сообщений методом send(…) JMS-производителя, включаются в глобальную транзакцию. Поэтому реальная отправка в JMS-источник сообщений в рамках BMT начнется только при фиксации UserTransaction, а при ее откате она будет отменена.

Rem: сообщения, отправленные вне границ UserTransaction отправятся без задержек.

```
// Ex.: откат userTransaction не повлияет на подтверждение доставки и ее дублирования не будет,
при этом отправка сообщения в источник exQueue будет отменена, поскольку она входит в границы
этой BMT. При фиксации userTransaction в exQueue будет отправлено сообщение.

@TransactionManagement(TransactionManagementType.BEAN)
@MessageDriven(…)
public class ExMDB implements MessageListener {
  @Resource private MessageDrivenContext mdbContext;
  @Inject @JMSConnectionFactory("xxx/ExFactory") private JMSContext jmsContext;
  @Resource(lookup="xxx/ExQueue") private Destination exQueue;

  @Override
  public void onMessage(Message message) {
    try {
      mdbContext.getUserTransaction().begin();
      jmsContext.createProducer().send(exQueue,"ExMDB test message");
```

```
    mdbContext.getUserTransaction().commit();
   } catch(Exception ex) {
     mdbContext.getUserTransaction().rollback();
   }
   ...
  }
}
```

Как будет вести себя JMS-поставщик при выбросе исключений в методе onMessage(…) зависит от вендора . В Open MQ 5.1 (Glassfish 4.1) проброс из метода onMessage(...) непроверяемого исключения приводит к уведомлению JMS-поставщика о сбое доставки и повторной отправке сообщения. Поэтому, при необходимости переотправки сообщения в случае сбоя его обработки, для BMT или NOT_SUPPORTED CMT рекомендуется выбрасывать в onMessage() непроверяемое EJBException или JMSRuntimeException.

```
// Ex.: выброс исключения уведомит EJB-контейнер о сбое обработки, JMS-поставщик не получит
/ /подтверждения доставки и отправит дубль

@MessageDriven(…)
public class ExMDB implements MessageListener {
  @Resource private MessageDrivenContext mdbContext;

  @Override
  @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
  public void onMessage(Message message) {
    ...
    throw new JMSRuntimeException("Consumption failed!");
    ...
  }
}
```

Rem: вендоры используют собственные механизмы определения числа повторных попыток отправки неподтвержденных сообщений для BMT и NOT_SUPPORTED CMT. JMS-провайдер также может определить зону «мертвых сообщений», куда будут помещаться те из них, что превысили планку попыток повторной отправки. Эту мертвую зону можно затем обрабатывать вручную. Все остальные объекты кроме сообщений включаются в транзакционные границы onMessage().

Применение JMS. Администрируемые объекты

Администрируемые объекты — объекты, которые создаются вне приложения (административно) для дальнейшего программного применения в run-time. Брокер (реализация JMS) создает и настраивает эти объекты один раз, после чего помещает их в JNDI-дерево, доступное клиентам. Существует 2 вида администрируемых объектов:

- Connection Factory – фабрики соединений, которые используются клиентами для подключений к источникам

- Destination – источники, которые получают, хранят и распространяют сообщения клиентов. В JMS существуют источники 2-х видов: Topic (тема, pub/sub) и Queue (очередь, p2p).

Такие объекты создаются вне приложения при помощи JMS-поставщика (сервера приложений в JEE). В JEE (WEB и EJB контейнерах) допускается также их задание при помощи метаинформации с последующим созданием JEE-контейнером в deploy-time .

Создание администрируемых объектов JMS в консоли GlassFish

Можно сделать это через консоль Glassfish: перейти в его каталог bin, запустить там консольный скрипт asadmin и выполнить в нем команды:

```
asadmin> create-jms-resource –restype javax.jms.ConnectionFactory xxx/ExFactory
asadmin> create-jms-resource –restype javax.jms.Topic xxx/ExTopic
asadmin> create-jms-resource –restype javax.jms.Queue xxx/ExQueue
```

Для просмотра созданных администрируемых объектов (ресурсов JMS) можно использовать команду:

```
asadmin> list-jms-resources
```

Задание администрируемых объектов JMS аннотациями

Администрируемые объекты в JMS 2.0 также можно определять посредством метаинформации, в аннотациях @javax.jms.JMSConnectionFactoryDefinitiion и @javax.jms.JMSDestinationDefinition, помечающих класс MDB :

```
Ex: пример в Glassfish 4.1

@JMSDestinationDefinition(name = "java:app/ExTopic", interfaceName = "javax.jms.Topic",
resourceAdapter = "jmsra", destinationName = "ExTopic")
@JMSConnectionFactoryDefinition(name = "jms/example/ExConnectionFactory", className =
"javax.jms.ConnectionFactory")
@MessageDriven(…)
public class ...
```

Rem: GlassFish 4.1 при использовании этих аннотаций требует применения портируемых глобальных JNDI-имен. JSE-клиент почему-то не мог найти через JNDI-запрос созданные таким образом объекты.

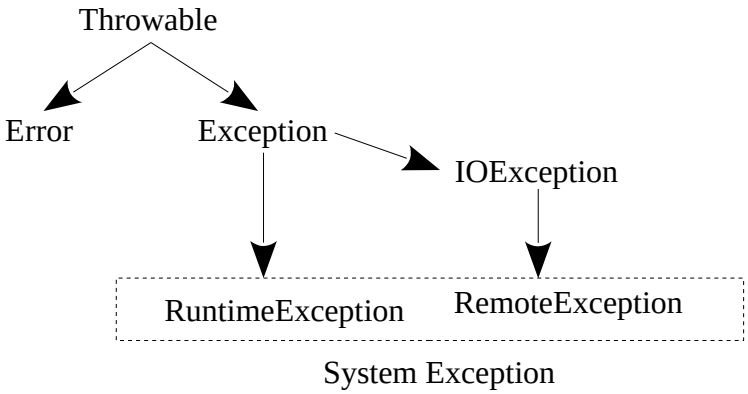
Клиентский доступ к администрируемым объектам JMS

Клиент получает доступ к этим объектам через интерфейсы JMS запросами JNDI или внедрением. Т.е. работает с ними через прокси.

Объекты, осуществляющие обмен, производятся администрируемыми объектами «по-ссылке» (?). Т.е. они также являются прокси соответствующих объектов брокера. И клиентские обращения к их методам — это обращение к методам соответствующих объектов JMS-поставщика, делегировавшего управление клиенту.

Применение EJB. Исключения

Исключения в Java порождаются классом java.lang.Throwable. Они делятся на подклассы java.lang.Error (серьезные системные сбои, используемые на этапе компиляции) и java.lang.Exception (исключения приложений, используемые в run-time). Исключения Exception делятся на непроверяемые (unchecked) — те, что JVM пробрасывает из метода автоматом и проверяемые (checked) – те, которые надо проверять в коде при помощи try/catch или пробрасывать из метода самостоятельно (оператором throws). К непроверяемым исключениям относится лишь подкласс java.lang.RuntimeException и его наследники, все остальные исключения, включая корневой Exception, относятся к проверяемым.



EJB-контейнер делит исключения на системные (system) и прикладные (application). Системное исключение — сбой внутри EJB-контейнера, выброс такого исключения прерывает текущий бизнес-процесс. Предполагается, что системные исключения не будут обрабатываться клиентом, поэтому они оборачиваются EJB-контейнером в одну из исключений-оберток и в таком виде возвращаются клиенту. Прикладные исключения трактуются как сбои в бизнес-логике и не являются основанием для прерывания или отката бизнес-процесса. Предполагается, что прикладные исключения будут обрабатываться клиентом, которому они и передаются в исходном виде.

К системным исключениям с т.з. EJB-контейнера по-умолчанию относятся непроверяемые RuntimeException (в т.ч. подкласс javax.ejb.EJBException) и класс java.rmi.RemoteException с наследниками. Для RemoteException и RuntimeException различаются механизмы преобразования системных исключений в прикладные при помощи @javax.ejb.ApplicationException.

При возникновении в EJB-компоненте системного исключения откатывается транзакция (если была), EJB-контейнер перехватывает это исключение и повторно выбрасывает клиенту одно из подходящих непроверяемых (RuntimeException) исключений, невзирая на его тип — удаленный или локальный. Дальнейшая реакция

зависит от типа сессионного EJB.

В случае SLSB и SFSB экземпляр компонента уничтожается, аннулируются все зависимости от него (ссылки на EJB-объект) и собирается мусор, поскольку EJB-контейнер считает такие компоненты нестабильными и небезопасными. Для SLSB потеря экземпляра из пула несущественна. Для SFSB выброс экземпляра будет означать потерю диалогового состояния, т.е. разрыв сессии с клиентом и аннуляцию клиентской ссылки на компонент. При этом SFSB переходит из состояния M-R в D-N-E (т.е. уничтожается) без вызова обработчика PreDestroy. При повторном обращении клиента по этой же ссылке, клиент получит от EJB-контейнера непроверяемое (RuntimeException) исключение javax.ejb.NoSuchEJBException.

В случае синглетона, при возникновении системного (прикладные запрещены) исключения в @PostConstruct методе, экземпляр компонента будет уничтожен, а если дополнительно этот синглетон был помечен @Startup, то развертка приложения вообще может быть сорвана. При выбросе исключения любым другим бизнес-методом ничего с экземпляром компонента не произойдет и он будет жить до завершения приложения.

С MDB-компонентом взаимодействует не сам клиент, а поставщик сообщений от его лица, не умеющий обрабатывать прикладные исключения. Поэтому исключение, выброшенное onMessage() или callback-методами @PostConstruct и @PreDestroy, может быть только системным RuntimeException. Оно приведет к ликвидации экземпляра MDB. Для MDB/BMT переотправка сообщения JMS-поставщиком зависит от того, когда было послано сообщение о неподтверждении приема. В случае MDB/CMT EJB-контейнер не подтвердит доставку и JMS-поставщик отправит сообщение вновь.

Rem: как будет вести себя удаленная клиентская ссылка (прокси) при удалении экземпляра ejb, зависит от вендора.

Rem: при использовании Entity в EJB надо помнить, что JPA генерирует только RuntimeException

Прикладные исключения выбрасываются в ответ на сбои в бизнес-логике. Они приходят к клиенту без перепаковки EJB-контейнером, в отличие от системных, в сериализованном виде (RMI). Прикладные исключения часто используются для оповещения клиента об ошибках валидации данных. В этом случае исключение выбрасывается до выполнения основных задач и ясно дает указывает, что это не ошибка подсистем вроде JDBC, JMS, RMI, JNDI и т.д.

Аннотация уровня класса @javax.ejb.ApplicationException служит для объявления класса исключения прикладным. Ее параметр rollback=true заставляет контейнер откатывать транзакцию при выбросе, по-умолчанию rollback=false.

```
Ex.:
@ApplicationException(rollback=true)
```

```
public class ExApplicationException extends Exception {...}
```

Аннотацией @ApplicationException можно помечать и классы системных исключений, т.е. наследников RuntimeException и RemoteException, превращая их в прикладные исключения. Этом может быть полезно при нежелании автоматической перепаковки отдельных исключений в EJBException или отката транзакции в ответ на их выброс.

Rem: в качестве аналога @ApplicationException можно использовать <application-exception> в стандартном дескрипторе ejb-jar.xml

Применение EJB. JNDI-запросы и вндрение

Для организации удаленного и локального взаимодействия с компонентами, EJB-контейнер применяет службу JNDI (Java Naming & Directory Interface, интерфейс службы имен и каталогов). JNDI представляет собой Java API службы каталогов, связывающей дерево символьных имен с Java-объектами.

В EJB-модели различается Global JNDI и локальный JNDI ENC (Environment Naming Context). Global JNDI используется где угодно для удаленного получения объектов с использованием протокола RMI. Локальный JNDI ENC привязан к экземпляру MBean, используется для внедрения в него объектов из домена и последующего их запросом внутри. Проще говоря JNDI ENC является динамическим карманным справочником экземпляра MBean, в т.ч. EJB.

Global JNDI работает с полными именами. В ранних версиях JEE у каждого вендора существовал свой синтаксис глобальных имен. Начиная с EJB 3.1 для унификации удаленных вызовов был введен также синтаксис переносимых глобальных JNDI-имен. В этом синтаксисе используются 4 пространства имен для разграничения их области действия: java:global, java:app, java:module и java:comp. Они образуют матрешку, т.е. имя в более узком пространстве может быть дополнено до более широкого. Это дополнение проводится автоматически с использованием текущего контекста в котором используется имя. Для доступа к корню JNDI-дерева можно использовать javax.naming.InitialContext (предварительно настроенный на соответствующую JNDI-службу).

JNDI ENC используется внутри связанного с ним экземпляра MBean при помощи внутренних имен, которые автоматически отображаются контейнером JEE-сервера на имена Global JNDI в контексте (узле JNDI-дерева) компонента: java:comp/env

Rem: по сути JNDI ENC обслуживает элементы внедрения ejb-jar.xml и аннотации внедрения @EJB, @PersistenceUnit, @PersistenceContext, @Resource, чьи атрибуты name являются ENC-именами точек внедрения объектов соответствующих типов (сессионных EJB, EMF, EM, SessionContext, DataSource и т.д.). В спецификации CDI ушли от символьных имен для обеспечения типобезопасности, поэтому внедренные через @Inject объекты в JNDI ENC не попадают.

Rem: заполнение дерева JNDI-именами происходит на этапе развертки (в deploy-time).

Пространство глобальных имен

Обычно используется для удаленных запросов EJB-компонентов из вне приложения:

```
java:global[/application-name]/module-name/ejb-name[!interface-name]

java:global/exmodule/ExEJB
java:global/exapp/exmodule/ExEJB!xxx.yyy.ExRemote
```

- application-name используется только при enterprise-архивации приложений и является именем ear модуля без расширения
- module-name – имя war или jar модуля
- ejb-name – компонентное имя EJB-компонента, эквивалентное атрибуту name @Stateful, @Stateless, @Singleton, атрибуту beanName @EJB, элементу <ejb-name> ejb-jar.xml
- interface-name – F.Q.N. бизнес-интерфейса, необходимо только, если в EJB-компоненте присутствует более одного бизнес-интерфейса

Пространство имен приложения

Усеченный вариант java:global, используемый для запросов EJB-компонентов внутри одного приложения:

```
java:app/module-name/ejb-name[!interface name]

java:app/exmodule/ExEJB
java:app/exmodule/ExEJB!xxx.yyy.ExLocal
```

Пространство имен модуля

Усеченный вариант java:global, используемый для запросов EJB-компонентов внутри одного модуля:

```
java:module/ejb-name[!interface name]

java:module/ExEJB
java:module/ExEJB!xxx.yyy.ExLocal
```

Пространство имен компонента. JNDI ENC

Это самое маленькое пространство имен, предназначенное для запросов внутри MBean. В нем определено пространство имен JNDI ENC java:comp/env.

```
java:comp/env/ENC-name

java:comp/env/ejb/ExBean
java:comp/env/jdbc/ExDataSource
```

Пространство JNDI ENC java:comp/env связано с узлом контекста MBean JNDI-дерева.

```
@Stateful
public class ExSFSB {
  @Resource(name=”jdbc/ExDataSource”) DataSource ds;
// Внедрение удаленного представления  ExEJB в JNDI ENC текущего ExSFSB, ENC-имя
// точки внедрения - ”ejb/ExBean”
  @EJB(name=”ejb/ExBean”) ExRemote exEJB;
// Внедрение безынтерфейсного @LocalBean представления ExEJB, поиск по типу
  @EJB ExEJB exLocalBeanEJB;

  public void exMethod() {
    // Получение представления корня  JNDI-дерева
    javax.naming.InitialContext initialContext = new InitialContext();
    // Получение представления узла окружения текущего компонента ExSFSB
    javax.naming.Context ctx = (Context) initialContext.lookup(“java:comp/env”);
    // Получение ссылки на источник данных (ранеее внедренный в JNDI ENC @Resource)
    DataSource dataSource = (DataSource) ctx.lookup(“jdbc/ExDataSource”);
    // Получение ссылки на exEJB (ранее внедренной @EJB в JNDI ENC) по полному имени
    ExRemote ejb = (ExRemote) initialContext.lookup(“java:comp/env/ejb/ExBean”);
    …
  }
}
```

ENC-имена задаются в атрибуте name аннотациях внедрения @EJB, @PersistenceUnit, @PersistenceContext, @Resource. Также в элементах <ejb-ref-name>, <persistence-unit-ref-name>, <persistence-context-ref-name>.

Для упрощения внутреннего получения по ENC-именам уже внедренных в EJB-компонент других MBean, можно использовать EJBContext или SessionContext, которые сразу связаны с текущим java:comp/env:

```
Ex: пример неявного внедрения (на уровне класса) EJB-компонента ExEJB в компонент ExSFSB,
получить неявно-внедренный экземпляр ExEJB можно только через JNDI-запрос

@EJB(name="XXX", beanInterface=ExLocal.class, beanName="ExEJB")
@Stateful
public class ExSFSB {
@Resource private javax.ejb.SessionContext ctx;
…
public void ExMethod() {
  ExLocal ejb = (ExLocal)ctx.lookup("XXX");
```

Rem: атрибут beanName – компонентная часть переносимого Global JNDI-имени

внедряемого EJB, идет только в паре с атрибутом beanInterface, задающим клиентское представление. Атрибут name задает ENC-имя точки внедрения EJB. При применении @EJB к классу, можно считать, что точка внедрения неявная, т.е. внедренный EJB доступен только через JNDI ENC.

Environment Entry

Environment Entry (переменные среды) — объекты-константы простых типов (String, Enum, оболочек примитивов Boolean, Byte, Short, Integer, Long, Float, Double, Characer, String), инициализирующиеся при развертке приложения.

Обычно они используются в качестве конфигурационных параметров EJB, когда неохота использовать и дергать для этого РБД.

В deploy-time переменные среды создаются в ENC JNDI контекста соответствующего компонента и заполняются из дескриптора развертки ejb-jar.xml. Получить их в компоненте можно через аннотацию @Resource или вручную через lookup(...).

В run-time при создании экземпляра EJB, после отработки конструктора, EJB-контейнер заполняет точки внедрения соответствующими значениями из ENC JNDI. В примере ниже поле passphrase по-умолчанию = “defaultPassphrase”, но это значение не попадет в ENC JNDI, а напротив, будет заменено EJB-контейнером на “overriddenPassword” после отработки конструктора и внедрения в passphrase значения переменной среды <env-entry-name>ciphersPassphrase</env-entry-name> из ejb-jar.xml.

```
Ex: переопределение переменной среды  passphrase через ENC JNDI
…
@Resource(name="ciphersPassphrase") private String passphrase = "defaultPassphrase";
...
<ejb-jar>
<enterprise-beans>
<session>
<ejb-name>EncryptionEJB</ejb-name>
<!-- Переопределение пароля -->
<env-entry>
<env-entry-name>ciphersPassphrase</env-entry-name>
<env-entry-type>java.lang.String</env-entry-type>
<env-entry-value>overriddenPassword</env-entry-value>
<injection-target>
<injection-target-class>
org.ejb3book.examples.MyEJBBean
</injection-target-class>
<injection-target-name>ciphersPassphrase</injection-target-name>
</injection-target>
</env-entry>
</session>
</enterprise-beans>
</ejb-jar>
```

Для явного получения значения Environment Entry можно использовать JNDI-поиск:

```
final String ciphersPassphrase = context.lookup(“ciphersPassphrase”);
```

Внедрение зависимостей

Внедрение зависимостей является callback-операцией MBean, выполняемой EJB-контейнером. Это противоположность прямому JNDI-поиску, выполняемому в прикладном коде. Внедрение удобно при обязательной инициализации внедряемого компонента контейнером, если же есть вероятность >0, что объект не потребуется, то для доступа к нему целесообразно применять прямой JNDI-поиск.

Для внедрения в MBean (сервлеты, JSF MBean, EJB) в EJB 3.0 (JEE5) была определен ряд специфичных аннотаций, работающих с JNDI ENC. Они были заточены под внедрение объектов строго-определенных типов, существующих в домене приложения.

В JEE6 спекой DI (CDI) была добавлена универсальная аннотация внедрения MBean @javax.inject.Inject. Она дает расширенный контроль за внедренными объектами, позволяет внедрять что угодно и куда угодно в рамках модуля, но имеет ограничения: может применяться только локально, в рамках модуля, не может обеспечить символьную параметризацию как, например, @PersistenceContext(unitName=”xxx”). Для обхода этих ограничений вместе с @Inject можно использовать костыли в виде CDI-управляемых оболочек с производителями (@Produces) нужных типов.

- @javax.ejb.EJB – внедряет ссылку на Local, Remote и LocalBean (безынтерфейсные) представления EJB-компонентов
- @PersistenceContext – внедряет ссылку на EM (JPA)
- @PersistenceUnit – внедряет ссылку на EMF (JPA)
- @Resource – внедряет ссылку на ряд типов: javax.sql.DataSource, SessionContext, JMS (?), Environment Entry (примитивы, оболочки и String), TimerService, UserTransaction и др.
- WebServiceRef – внедряет ссылку на JAX-WS
- @Inject – типобезопасно (автоопределение по типу с возможностью уточнения квалификаторами) внедряет ссылку на любой тип компонента, принадлежащий текущему модулю. При необходимости внедрить ссылку на внешний или параметризованный объект, можно создать производителя @Produces.

Помечать аннотациями внедрения можно поля, сеттеры, т.е. свойства MBean.

```
@Stateless
public ExEJB {
  @PersistenceContext(unitName=”ExPU”) private EntityManager em;
  @EJB private ExLocal exEJB;
  @EJB private ExRemote  exEJB2;
```

```
  @EJB private ExEjb3 exEJB3;
  @Inject private ExPOJO exPOJO;
  @WebServiceRef private ExWS  exSOAPWS;
  private SessionContext ctx;
  @Resource public void setContext(SessionContext ctx) {this.ctx = ctx;}
…
}
```

Применение EJB. Асинхронные методы. Библиотека Concurrent

По-умолчанию все вызовы сессионных EJB являются для клиентов синхронными. В EJB 3.1 (JEE6) наряду с JMS/MDB появилась возможность вызывать сессионные EJB в отдельных потоках, т.е. асинхронно. Это делается при помощи библиотеки java.util.concurent.*

Асинхронные вызовы полезны для медленных методов, когда результат или подтверждение завершения обработки не требуется немедленно: печать, фоновое кэширование данных, пакетные задачи и т.п.

Для задания асинхронного поведения можно использовать аннотацию @javax.ejb.Asynchronous. Помечать ею можно как отдельные методы, так и типы сразу.

При асинхронном вызове управление вызывающим потоком сразу же возвращается клиенту, без ожидания вызова экземпляра EJB через EJB-объект. EJB-контейнер также запускает отдельный поток выполнения.

Асинхронный метод может возвращать void, тогда полностью реализуется принцип запустил-забыл, полезный для неких необязательных уведомительных задач.

В случае, когда требуется контроль за выполнением асинхронного запроса или собственно результат, метод должен возвращать асинхронную оболочку из библиотеки Concurrent – интерфейса java.util.concurrent.Future<V>, где V – целевой тип. На серверной стороне удобно использовать его реализацию javax.ejb.AsyncResult<V>, появившуюся в EJB 3.1 (JEE6). Клиент может управлять асинхронным вызовом метода с результатом при помощи методов get(...) и cancel(...) объекта-оболочки результата — Future. При вызове клиентом cancel(…), EJB-контейнер может попытаться остановить асинхронный вызов, если соответствующий асинхронный метод еще не успел запуститься, а при выполнении асинхронного метода, в его коде можно выполнять проверку на клиентскую остановку вызовом метода wasCancelCalled() сессионного контекста. Метод Future.get(…) позволяет извлечь результат из оболочки или получить исключение, если при выполнении произошел сбой.

```
Ex: пример асинхронного метода EJB

@Asynchronous
@Override
```

```
public Future<String> hashAsync(final String input) throws IllegalArgumentException,
EncryptionException {
// получение дайджеста
  final String hash = this.hash(input);
// обертывание асинхронным объектом-оболочкой и возврат
  return new AsyncResult<String>(hash);
}
```

Пример клиентского кода, делающего асинхронный вызов и получающего результат:

```
Ex: пример вызова асинхронного метода EJB

final String input = "input test";
// асинхронный запрос
final Future<String> hashFuture = ejbProxy.hashAsync(input);
// выполнение кода в прежнем потоке
…
// запрос ответа от асинхронного объекта с блокировкой потока на 10 секунда
final String hash = hashFuture.get(10,TimeUnit.SECONDS);
```

Пример клиентского управления асинхронным вызовом сессионного EJB:

```
Ex: SLSB с асинхронным методом

@Stateless @Asynchronous
public class ExEJB {
  @Resource private SessionContext ctx;
   public Future<Integer> getStatus(){
     Integer result = 0;
     // …
     result = 1;
     if(ctx.wasCancelCalled()) return new AsyncResult(2);
     // …
     return new AsyncResult(result);
   }
}
```

```
Ex: фрагмент клиента, управляющий асинхронным вызовом к SLSB
...
Future<Integer> status = exEJB.getStatus();
Integer statusValue = status.get();
```

Асинхронный метод getStatus() выполняет некоторую задачу, возвращая статус завершения. Если бы клиент вызвал метод status.cancel() до проверки ctx.wasCancelCalled(), то метод бы завершился досрочно с кодом 2. Если бы метод завершился штатно, то код был бы 1.

При асинхронном вызове транзакционный контекст не передается, контекст

безопасности передается.

Применение EJB. SessionContext & EJBContext

Интерфейс javax.ejb.SessionContext, наследующий javax.ejb.EJBContext — интерфейс, позволяющий классу EJB напрямую связаться с EJB-контейнером. В класс EJB он может внедряться при помощи @Resource:

```
@Resource
private SessionContext ctx;
```

Актуальные для EJB 3.2 (JEE7) методы SessionContext:

- <T>T getBusinessObject(Class<T> businessInterface) – метод получения представления EJB-объекта — прокси, реализующего заданный бизнес-интерфейс для связи с текущим экземпляром EJB. Это своеобразный аналог Java this в виде прокси, который можно передавать другим клиентам, локальным или удаленным в зависимости от типа бизнес-интерфейса.
- getInvokedBusinessInterface – выдает описание (Class) того бизнес-интерфейса, через экземпляр которого был сделан вызов экземпляра EJB, в т.ч. позволяет определить его тип: Local, Remote или Endpoint (WS) (?)
- wasCalledCancel – проверка вызова cancel() в клиентском объекте Future, соответствующем выполняемому в данный момент асинхронному методу.

```
Ex: пример передачи интерфейсной ссылки EJB A в EJB B

@Stateless
public class A extends ARemote {
  @Resource
  private SessionContext ctx;
  public void exMethod() {
// Получение прокси для удаленного общения с чужим EJB-компонентом
    BRemote b = (BRemote)ctx.lookup(…);
// Получение через контекст прокси для удаленного общения с текущим экз. SLSB
    ARemote a = ctx.getBusinessObject(ARemote.class);
    b.setA(a);
  }
}
```

javax.ejb.EJBContext – интерфейс базового окружения EJB-контейнера, доступный объекту EJB в run-time. Он содержит несколько методов, снабжающих объекты EJB информацией во время их выполнения. Основные методы, использующиеся в EJB 3.2 (JEE7):

- lookup(String) – метод поиска элементов в JNDI ENC
- getTimerService – метод, дающий ссылку на службу таймера EJB-контейнера. Это

позволяет SLSB и Singleton задавать обработчик временных событий, которые будут выбрасываться EJB-контейнером при наступлении заданного момента времени или истечении временного интервала. Объект реализации javax.ejb.TimerService может быть также внедрен в свойство при помощи @Resource

- getCallerPrincipal – метод получения объекта java.security.Principal (JAAS, JSE), который описывает аутентифицированного системой безопасности клиента SLSB.

```
Principal p = ctx.getCallerPrincipal(); String name = p.getName();
```

- isCallerInRole(String) – метод проверки принадлежности клиента SLSB заданной роли

```
boolean adminFlag = ctx.isCallerInRole(“admin”);
```

- getUserTransaction – метод, дающий явную программно-управляемую транзакцию (BMT) интерфейса javax.transacton.UserTransacrion

```
UserTransaction utx = ctx.getUserTransaction(); utx.begin(); …; utx.commit();
```

- setRollbackOnly — метод, помечающий текущую контейнерно-управляемую транзакцию (CMT) на откат при попытке фиксации

```
UserTransaction utx = ctx.getUserTransaction(); utx.begin(); try {…} catch(Exception e) {utx.setRollbackOnly();} finally {utx.commit();}
```

- getRollbackOnly — метод проверки пометки на откат для текущей CMT

Применение EJB. XML-дескриптор развертывания ejb-jar.xml

XML-дескриптор развертки EJB — XML-документ, определенной спекой EJB схемы, находящийся в файле META-INF/ejb-jar.xml jar-модуля приложения.

Это необязательный конфигурационный файл, используемый EJB-контейнером на этапе развертки приложения, который служит альтернативой, дополнением или замещением (в силу большего приоритета) аннотаций EJB. В отличие от аннотаций, XML не внедряется в байт-код.

В нем целесообразно описывать вендор-специфические параметры, например, для development и production серверов.

- <ejb> - корневой элемент
- <enterprise-beans> - перечень компонентов для развертки
- <session> - идентификатор сессионности компонента

- <ejb-name> - имя EJB-компонента (компонентная часть глобального JNDI-имени в переносимом синтаксисе), используемое при развертке приложения в deploy-time, соответствующее атрибуту name @Stateless, @Stateful, @Singleton. По-умолчанию это короткое имя класса ("ExBean").
- <local>, <remote> - описание бизнес-интерфейсов
- <ejb-class> - класс EJB
- <session-type> - тип сессионного компонента, по-умолчанию stateless (?)
- <env-entry> - инициализирует значения полей, предоставляемые классом компонента для заполнения контейнером (?)

Rem: значения XML-дескриптора приоритетнее значений, заданных аннотациями, что позволяет перенастраивать поведение EJB без затрагивания исходников или байт-кода.

Применение EJB. Клиенты EJB-компонентов

Можно разделить клиентов EJB-компонентов на локальные и удаленные. Первые живут в одном процессе JVM с EJB-контейнером, вторые — в разных. Примером локального клиента можно считать другой EJB-компонент или WEB-компонент (сервлет, CDI MBean) в рамках одного ear-модуля на одном хосте. Примером удаленного клиента является любой Java-компонент, работающий в другом процессе JVM от обычных POJO JSE-приложения до MBean в JEE-контейнерах.

Rem: начиная с EJB 3.1 (JEE6) EJB-компоненты можно использовать непосредственно в JSF-страницах. Однако это считается дурным тоном из-за смешения уровней бизнес-логики и представления.

Получить доступ к EJB-компоненту можно через внедрение или прямой JNDI-поиск. Все вызовы к стандартным сессионным EJB идут в том же потоке клиента, для @Asynchronous EJB создается отдельный поток.

Внедрение можно использовать лишь в контейнерах JEE: EJB, WEB, ACC. Для внедрения EJB-компонентов можно использовать аннотации @EJB и @Inject. Аннотация @EJB может внедрять удаленные и локальные представления (Local, LocalBean, Remote), используя символьные имена: имена JNDI ENC (атрибут beanName), либо переносимые имена Global JNDI (атрибут lookup), либо непереносимые вендор-специфические имена (mappedName), а при отсутствии оных может использовать поиск по типу внедрения. Аннотация @Inject использует строгий поиск по типам вместо символьных имен, дает расширенные возможности внедрения (квалификаторы, альтернативы и др.), но ограничена локальными представлениями только тех EJB-компонентов, что находятся в одном модуле с точкой внедрения и не поддерживает параметры.

JNDI – JSE API службы имен и каталогов. Реализация самой службы не важна, важен ее адрес, реализация класса подключения, задающиеся при создании объекта корня JNDI-дерева — javax.naming.InitialContext. В JEE-контейнерах эти параметры настраиваются

в deploy-time и разработчику компонентов о них не нужно заботиться.

Прямой JNDI-поиск возможен из любого типа клиента по именам Global JNDI. Локальные EJB-клиенты также могут использовать символьные имена JNDI ENC (java:comp/env/...) внедренных в них объектов.

Внедрение является предпочтительным способом получения доступа к EJB-компонентам в JEE MBean. Прямой JNDI-поиск может применяться везде. Он целесообразен в клиентах без поддержки внедрения и там, где вероятность использования внедряемого EJB-компонента < 1.

Основной контекст (InitialContext) указывает на корень JNDI-дерева (”java:global/”):

```
Context ctx = new InitialContext();
ExRemote exEJB = (ExRemote) ctx.lookup(“java:global/exapp/ExEJB!xxx.yyy.zzz.ExRemote”);
```

В EJB-компонентах также можно использовать EJBContext, который связан с JNDI-узлом текущего компонента:

```
@Resource private javax.ejb.EJBContext ctx;
…
ExRemote exEJB = (ExRemote) ctx.lookup(“ExEJB!xxx.yyy.zzz.ExRemote”);
```

При локальных вызовах объекты фактически передаются по-ссылке. При внешних — копии, т.е. по значению.

Применение EJB. Callback-методы обработки событий ЖЦ

При переходе из одного состояния MBean в другое происходят события жизненного цикла:

- PostConstruct — событие возникает после создания экземпляра MBean и внедрения зависимостей
- PreDestroy — событие возникает перед уничтожением экземпляра, для SLSB это может наступить по таймауту или в результате вызова @Remove метода.
- PrePassivate — событие наступает только для SFSB перед пассивацией экземпляра
- PostActivate — событие наступает только для SFSB сразу после восстановления экземпляра из пассивированного состояния

На любое из событий жизненного цикла MBean можно поставить обработчик — метод MBean, который будет вызываться контейнером (обратный вызов, callback), удовлетворяющий следующим условиям:

- сигнатура void <exMethod>()
- должен быть снабжен метаинформацией о событии, например, одной или более аннотациями @PostConstruct, @PreDestroy, @PostActivate, @PrePassivate
- (?) должен быть уникальным обработчиком события (?) можно пометить одной аннотацией несколько методов(?)
- не должен выбрасывать проверяемых (прикладных) исключений
- не должно быть модификаторов final и static

В EJB-компонентах callback-методу доступен SessionContext. Как правило эти методы используются для закрытия и открытия соединений и ресурсов.

В JEE7 появилась возможность вовлекать в транзакцию callback-методы при помощи @TransactionalAttribute(REQUIRES_NEW), причем REQUIRES_NEW – единственное возможное значение этой аннотации.

Ругательства

- MBean = Managed Bean = управляемый компонент, работающий в некотором контейнере. В JEE5 это сервлеты, JSF MBean (Backing Bean), EJB. В JEE6 все контейнеры поддерживают CDI и любой компонент, включая POJO, можно сделать CDI-управляемым MBean. EJB-компоненты также являются CDI MBean с расширенными возможностями.
- EJB Object = EJB-объект = серверный посредник EJB, который связывает клиента с экземпляром EJB-компонента. Удаленное (заглушка RMI) или локальное (сслыка) представление текущего EJB-объекта экземпляра EJB-компонента можно получить методом getBusinessObject(…) интерфейса SessionContext.
- SLSB = Stateless Session Bean
- M-R-P = Method-Ready-Pool — пул ожидающих методов — состояние объекта SLSB, означающее, что он существует в пуле и готов принимать запрос
- Not Exist = не существует — состояние объекта SLSB, означающее, что экземпляр SLSB не существует в RAM.
- CMC = Container-Managed Concurency = контейнерное управление параллелизмом
- SFSB = Stateful Session Bean
- Message Service = система сообщений, отвечающая за их отправку и доставку
- Message Broker = Message Service
- MDB = Message Driven Bean = управляемый сообщениями компонент
- EIS = Enterprise Information System — промышленная информационная система
- JCA = Java Connector Architecture = java-архитектура соединителя — JEE-стандарт, описывающий соединение JEE-серверов приложений с EIS.
- JCA Resource Adapter = Resource Adapter = JCA Adapter — компонент, инкапсулирующий алгоритмы соединения с определенной EIS по правилам спецификации JCA.
- JEE-коннектор = JCA Resource Adapter
- MOM = Message Oriented Middleware
- Message Provider = поставщик сообщений — посредник, организующий обмен

сообщениями между клиентами MOM
- Message Service = служба сообщений — абстракция MOM, регламентирующая порядок доступа к поставщику, создание, отправку и получение сообщений.
- Message Broker = конкретная реализация Message Service
- Message Router = Message Broker
- JMS = Java Message Service = служба сообщений JEE
- JMS-клиент — приложение, использующее JMS.
- JMS-поставщик — система, которая управляет отправкой и доставкой сообщений согласно JMS API.
- JMS-приложение — бизнес-система, состоящая из одного или более JMS-клиента и одного (обычно) или более JMS-поставщика.
- JMS-производитель (producer) – JMS-клиент, отправляющий сообщения.
- JMS-потребитель (consumer) – JMS-клиент, принимающий сообщения.

Литература

Enterprise JavaBeans 3.1 (6th edition), Andrew Lee Rubinger, Bill Burke, стр. nnn

Изучаем Java EE 7, Энтони Гонсалвес, стр. nnn

EJB3 в действии, (2е издание), Дебу Панда, Реза Рахман, Райан Куправк, Майкл Ремижан, стр. nnn

http://www.interface.ru/home.asp?artId=21612

http://www.ibm.com/developerworks/ru/library/j-jca1/index.html

https://glassfish.java.net/docs/4.0/mq-tech-over.pdf

https://www.ibm.com/support/knowledgecenter/SSEQTP_7.0.0/com.ibm.websphere.nd.doc/info/ae/ae/cmb_trans.html