GLOBAL 2007

Обучение Java

Тема **EJB 3.0.**

Часть **Построение бизнес логики бинами сеансов**

Автор **ASKIL (omendba@gmail.com)**

10.07.2007

В основе любого приложения предприятия лежит его бизнес логика. В идеальном мире прикладные разработчики должны быть заняты только определением и осуществлением бизнес логики, в то время как проблемы представления, постоянства или интеграции должны в значительной степени быть реализованы без их участия. Поэтому сессионные бины – наиболее важная часть технологии EJB, потому что их цель состоит в том, чтобы моделировать бизнес-процессы высокого уровня.

Начинаем изучать сессионные бины

Типичное приложение предприятия будет иметь многочисленные виды бизнес активности или просто бизнес процессов. Сессионные бины могут использоваться для упаковки в капсулу всей бизнес логики этих процессов.

Теория бинов сосредоточена на идее того, что каждый запрос клиента должен быть выполнен в течение сессии. Так, что является сессией? Проще говоря, *сессия* - связь между клиентом и сервером, которая длится конечный промежуток времени.

Сессия может или быть очень недолгой, как запрос HTTP, или занять длительное время. Программа передачи файла – сессия *stateful*, поддерживающая состояние между запросами. Если запрос одноразовый, то данная сессия – *stateless*.

Бины – единственные компоненты EJB, которые вызываются непосредственно клиентами.

Почему используются бины

Бины – намного больше, чем просто исполнители бизнес логики. Самые важные сервисы, которые они предоставляют:

• Параллелизм и безопасность потоков

Поскольку бины определенно предназначены, чтобы обрабатывать запросы клиентов, то они должны поддержать высокую степень параллелизма.

• Remoting и веб-службы

Бины поддерживают обе технологии Java Remote Method Invocation (RMI – native протокол) и Simple Object Access Protocol (SOAP) на основе веб-службы удаленного доступа.

• Транзакции и управление безопасностью

Транзакции и управление безопасностью - два главных кита, на которых строятся приложения уровня предприятия.

• Сервисы таймера и перехватчики

Перехватчики – версия EJB по аспектно-ориентируемому программированию (AOP). AOP – способность изолировать проблему "взаимного сокращения" в отдельный модуль. Данная проблема включает в себя такие вещи как аудит и логирование, которые работают вместе с приложением, но непосредственно не связанны с бизнес-логикой. *Сервисы таймера* - версия EJB прикладных планировщиков. Данные сервисы позволяют легко превращать бины в асинхронную запланированную задачу.

Анатомия бинов

Каждая реализация бинов имеет две различные части: один или более интерфейсов бина и класс, реализующий бин.

Все бины должны быть разделены на эти две части. Это необходимо потому, что клиенты не могут иметь доступа к классу реализации бина непосредственно. Вместо этого они должны использовать бин через бизнес интерфейс.

Бизнес интерфейс бина

Интерфейс, через который клиент вызывает бин, называют *бизнес интерфейсом*. Этот интерфейс по существу определяет методы бина. Например:

```
@Local
public interface IF {
    ...
}
```

Нет ничего странного в этом коде кроме @Local аннотации, определяющей, что интерфейс является локальным интерфейсом. Интересная вещь, которую можно отметить прямо сейчас – факт того, что единственный EJB может иметь множество интерфейсов. Другими словами, реализованные классы EJB могут быть полиморфными, то есть различные клиенты, использующие различные интерфейсы могут использовать классы различными способами.

EJB класс бина

Каждый интерфейс, который бин намеревается поддержать, должен быть явно реализован.

Классы EJB могут иметь private методы, которые не доступны через интерфейс. Кроме того, класс EJB может использовать наследование.

Правила при написании бинов

Правила, которые применяются ко всем типам бинов:

- Бин должен иметь, по крайней мере, один бизнес интерфейс.
- Класс бина должен быть конкретным. Вы не можете определить класс бина как final или abstract, так как контейнер должен управлять им.
- Вы должны иметь конструктор без аргументов в классе бина, поскольку контейнер сам вызывает конструктор, и он не знает, что передавать ему. Компилятор вставляет конструктор без аргументов по умолчанию, если нет никакого конструктора в Java классе.
- Класс бина может наследоваться другими классами бинов. Например, класс stateless по имени Manager может расширить другой класс ManagerTwo следующим образом:

```
@Stateless
public Manager extends ManagerTwo implements IFManager
{
   ...
}
```

• Бизнес методы и callback методы могут быть определены или в классе бина или в суперклассе. Здесь стоит упомянуть об этом, так как наследование аннотаций реализовано с несколькими ограничениями в EJB 3. Например, аннотация бина типа @Stateless или @Stateful определенные в суперклассе будут игнорироваться при реализации.

• Названия бизнес методов не должны начинаться с "ejb". Вы должны определить все бизнес методы как public, но не final или static. Если вы поставляете метод в удаленном бизнес интерфейсе EJB, то удостоверьтесь, что аргументы и возвращаемый тип метода реализуют java.io.Serializable интерфейс.

Типы бинов сессии

Специфический бизнес-процесс может состоять из более одного запроса клиента к бину сессии. В процессе этих запросов, бин может и не может поддерживать общение с клиентом, то есть может запоминать или не запоминать состояние. Бин, который поддерживает диалог, “помнит” результаты предыдущих запросов – *stateful* бин. В java терминологии это означает, что бин будет хранить данные запросов в переменных экземпляра. *Stateless* бины не поддерживают никакого диалога с клиентом. Вообще, stateful бины имеют тенденцию моделировать многошаговые технологические процессы, в то время как stateless бины имеют тенденцию моделировать сервисные услуги общего назначения, используемые клиентом.

Callbacks в жизненном цикле бина

Сессионные бины имеет жизненный цикл. Это означает, что бины проходят предопределенный набор состояний. Контейнер управляет бинами. Это означает, что ни клиент, ни бин не в состоянии управлять созданием, удалением бинов и т.п. действиями. Управление этими действиями позволяет контейнеру обеспечить абстракцию использования EJBs.

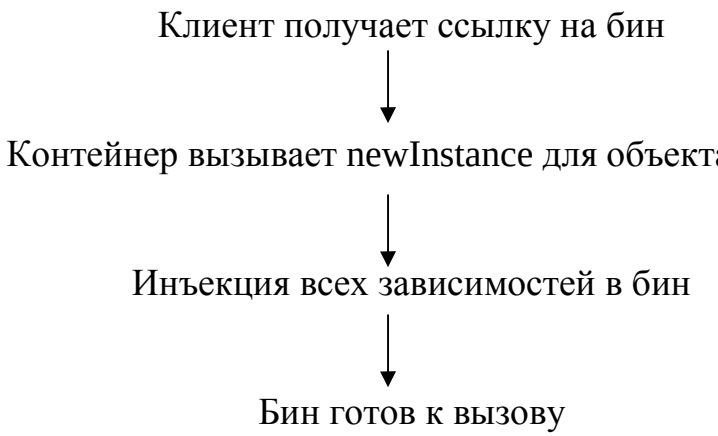
События жизненного цикла

ЖЦ бинов может быть распределен в несколько фаз или событий. Самые очевидные, два события – создание и разрушение. Все бины проходят эти две фазы. Кроме того, stateful бины проходят цикл пассивирования/активации.

ЖЦ бина начинается, когда экземпляр бина создан. Это происходит, когда клиент получает ссылку на бин, ведя поиск через JNDI или при использовании инъекции аннотации @EJB. Последовательность шагов при инициализации бинов:

1. Контейнер вызывает newInstance метод для объекта бина.

2. Если бин использует DI (инъекционные зависимости), все зависимости от ресурсов, других бинов, и компоненты среды будут введены в недавно созданный экземпляр бина.



Callbacks в жизненном цикле

Callbacks методы бина (не объявляемые бизнес интерфейсом) – методы, которые контейнер вызывает при переходе бина из одного состояния в другое. Когда событие происходит, контейнер вызывает соответствующий метод. Вы можете использовать данные методы для выполнения бизнес логики, типа инициализации второстепенных объектов или удалении занимаемых ресурсов.

Callbacks методы – методы бина, которые отмечены аннотациями, типа @PostConstruct и @PreDestroy. Они могут быть public, private или protected.

PostConstruct вызывается только после того, как экземпляр бина создан, и все зависимости введены.

PreDestroy вызывается непосредственно перед тем, как бин удаляется и полезен для того, чтобы очистить ресурсы, используемые бином.

В то время как все бины имеют PostConstruct и PreDestroy, stateful бины имеют два дополнительных: PrePassivate и PostActivate. Для каждого клиента существует свой stateful бин, поддерживающий состояние, но клиентов может быть множество, поэтому и экземпляров может быть много в контейнере. Если до этого доходит, контейнер может решить дезактивировать stateful экземпляр бина временно, если он не используется; этот процесс называют *пассивированием*. Контейнер активизирует экземпляр бина, когда он снова будет нужен клиенту.

@PrePassivate и @PostActivate аннотации применяются для методов пассивирования и активации экземпляров бина.

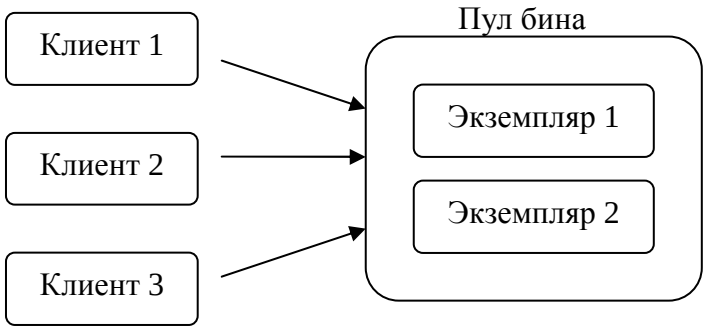
Вы можете определить callback метод или в классе бина или в отдельном классе перехватчика.

Callback аннотация	Тип EJB	Обычно используется для...
javax.annotation. PostConstruct	Stateless, stateful, MDB	Этот аннотируемый метод вызывается после того, как создан экземпляр бина. Метод используется для того, чтобы инициализировать ресурсы.
javax.annotation. PreDestroy	Stateless, stateful, MDB	Этот аннотируемый метод вызывается до разрушения экземпляра бина. Этот метод используется для удаления ресурсов (например, закрытие соединения с базой данных).
javax.ejb.PrePassivate	Stateful	Этот аннотируемый метод вызывается до пассивирования экземпляра бина. Вообще этот вызов используется, чтобы удалить ресурсы, типа соединений базы данных, сокетов TCP/IP, или любых ресурсов, которые не могут быть сериализованы в течение пассивирования.
javax.ejb.PostActivate	Stateful	Этот аннотируемый метод вызывается после того, как экземпляр бина активизирован. Вообще этот вызов используется, чтобы восстановить ресурсы, типа соединений базы данных, которые были удалены при пассивировании.

Stateless бины

Stateless бины моделируют задачи, не поддерживающие состояние. Это означает, что эти бины моделируют задачи, которые могут быть выполнены в единственном запросе. Фактически, stateless бины реальных приложений часто содержат несколько близко связанных бизнес методов. Вообще говоря, stateless бины – самый популярный вид сессионных бинов.

Stateless экземпляры бина группируются в пул. Это значит, что для каждого бина, контейнер хранит определенное число экземпляров, находящихся в пуле бина. Для каждого запроса клиента, контейнер или создает новый экземпляр или назначает один экземпляр из пула. Когда запрос клиента выполнен, экземпляр возвращается в пул для повторного использования. Это означает, что небольшое количество экземпляров бина могут обслуживать относительно большое количество клиентов.



Использование @Stateless аннотации

Спецификация @Stateless аннотации следующая:

```
@Target(TYPE) @Retention(RUNTIME)
public @interface Stateless
{
   String name() default "";
   String mappedName() default "";
   String description() default "";
}
```

Единственный параметр *name* определяет имя бина. Некоторые контейнеры используют этот параметр, чтобы связать EJB с глобальным деревом JNDI. Как показано в определении аннотации, параметр name является дополнительным. Если параметр name опущен, контейнер назначает название класса бину. *mappedName* - определенное для поставщика название, которое вы можете присвоить вашему EJB.

Определение бизнес интерфейсов бина

Приложения-клиенты могут вызвать stateless бины тремя различными способами. В дополнение к local вызову в пределах той же JVM и remote вызову через RMI, stateless бины могут быть также вызваны удаленно как веб-службы.

Три типа бизнес интерфейсов соответствуют различным типам доступа: каждый идентифицирован через свою аннотацию.

Local интерфейс

Локальный интерфейс разработан для клиентов stateless бинов, расположенных в том же самом контейнере JVM экземпляра бина. Вы определяете интерфейс как local бизнес интерфейс при использовании @Local аннотации. Следующее могло быть local интерфейсом для класса Manager:

```
@Local
public interface Manager
{
    void add(int i);
}
```

Remote интерфейс

Клиенты, находящиеся вне JVM контейнера с экземпляром должны использовать удаленный интерфейс. Если клиент также написан на Java, то самым логическим выбором для удаленного доступа будет использование протокола RMI. EJB 3 позволяет stateless бину быть доступным через RMI через @Remote аннотацию. Например:

```
@Remote
public interface BidManager extends Remote {
    ...
}
```

Удаленный бизнес интерфейс может расширить java.rmi.Remote, хотя это является дополнительным. Удаленные бизнес методы интерфейса не обя-

заны обрабатывать java.rmi.RemoteException, если бизнес интерфейс не расширяет java.rmi.Remote интерфейс. Удаленные бизнес интерфейсы имеют одно специальное требование: все параметры и возвращаемый тип методов интерфейса должны быть Serializable. Это необходимо потому, что только Serializable объекты могут быть посланы по сети, используя RMI.

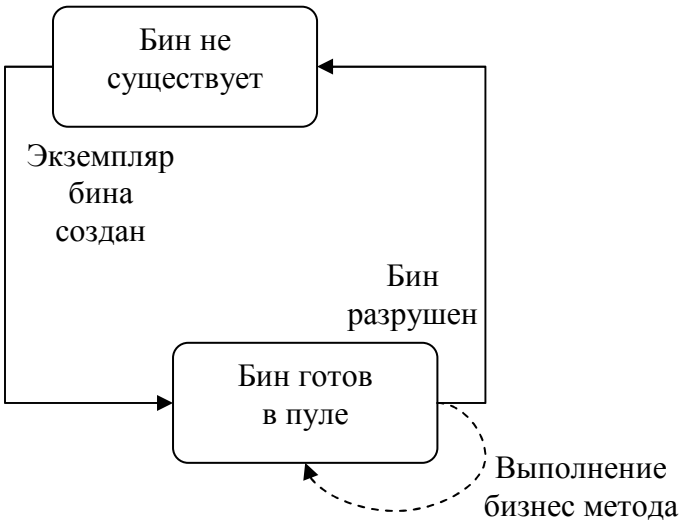
Интерфейс конечной точки веб-службы

Интерфейс конечной точки веб-службы (также известный как SEI) – это способность выставлять stateless бин как веб-службу на основе протокола SOAP, одна из самых мощных особенностей EJB 3. Все что вы должны сделать – написать бизнес интерфейс вместе с @javax.jws.WebService аннотацией. Следующий код определяет простой интерфейс конечной точки веб-службы для класса Manager:

```
@WebService
public interface Manager {
    ...
}
```

@WebService аннотация не накладывает никаких специальных ограничений или на интерфейс или реализацию бина.

Жизненный цикл stateless бина



Контейнер делает следующее:

1. Создается экземпляр бина, с использованием конструктора по умолчанию.
2. Инъекция ресурсов, типа соединений с базой данных.
3. Помещение экземпляров в пул.
4. Выбирает любой экземпляр из пула, когда получен запрос от клиента.
5. Выполняет требуемый бизнес метод, вызванный через бизнес интерфейс.
6. Когда бизнес метод заканчивает выполнение, он кладет обратно бин в пул.
7. Как необходимо удаляет бины из пула.

PostConstruct callback

Пример использования данной аннотации:

```
@PostConstruct
public void initialize() {
   ...
   connection = dataSource.getConnection();
   ...
}
```

В данном случае это соединение теряется после выполнения бизнес метода.

PreDestroy callback

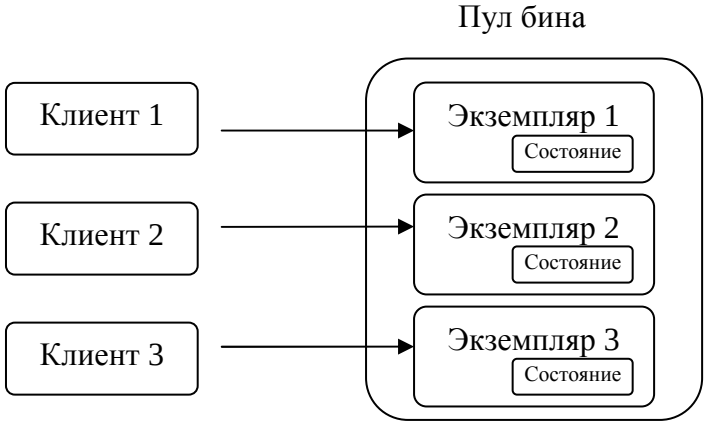
Пример использования данной аннотации:

```
@PreDestroy
public void cleanup() {
    ...
    connection.close();
    connection = null;
    ...
}
```

Сразу после выполнения одношаговой операции, закрываем соединение с базой данных, так как это соединение будет потеряно для данного клиента.

Stateful бины

Бины stateful гарантируют, что они поддерживают состояние. Фактически, единственное различие между stateless и stateful бинами – то, как контейнер управляет их жизненным циклом. Контейнер делает так, чтобы экземпляр stateful бина принимал запросы от одного и того же клиента. Экземпляры бина не могут быть возвращены в пул, пока сессия клиента активна. В результате этого экземпляры stateful бина в большом количестве могут исчерпать всю оперативную память. Техника оптимизации, решающая эту проблему, была названа *пассивированием*. Бины stateful идеальны для многошаговых, ориентируемых на технологический процесс бизнес-процессов. Экземпляр бина существует, пока он не удален клиентом или не прошел таймаут.



Дополнительные правила

• Переменные экземпляров бина, которые используются для сохранения состояния, должны быть примитивами Java или быть объектами Serializable.

• Мы должны определить бизнес метод удаления экземпляра бина клиентом, используя @Remove аннотацию.

• В дополнение к PostConstruct и PreDestroy методам, stateful бины дополняются PrePassivate и PostActivate методами. PrePassivate метод вызывается, перед тем как экземпляр бина пассивируется. PostActivate метод вызывается, после того экземпляр бина вернулся в память и готов обслуживать.

Бизнес интерфейс для stateful бинов

Определение stateful бизнес интерфейса бина почти такое же, лишь с несколькими исключениями. Бины stateful поддерживают local и remote вызовы через @Local и @Remote аннотации. Однако stateful бин не может иметь интерфейса конечной точки веб-службы. Вы должны всегда включать аннотируемый метод, по крайней мере, одного @Remove в интерфейс stateful бина.

Жизненный цикл stateful бина

Контейнер выполняет шаги:

1. Всегда создает новые экземпляры бина, используя конструктор по умолчанию всякий раз, когда создается новая сессия клиента.

2. Инъецирует ресурсы.

3. Хранит экземпляр в памяти.

4. Выполняет требуемый бизнес метод, вызванный через бизнес интерфейс клиентом.

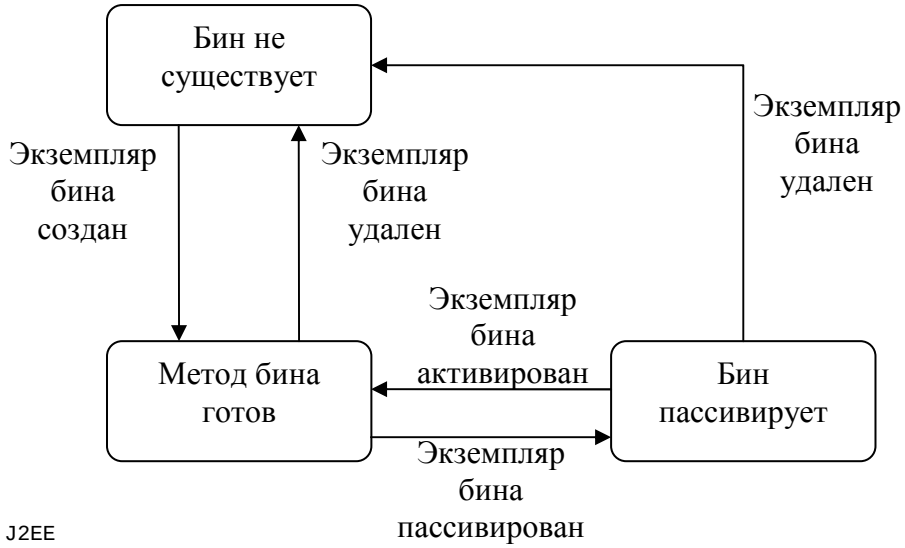
5. Ждет и выполняет последующие запросы клиента.

6. Если клиент не выполняет никаких запросов некоторое время, контейнер пассивирует экземпляр бина. Пассивирование означает, что экземпляр перемещен из активной памяти, сериализован и хранится во внешней памяти.

7. Если клиент вызывает пассивировавший бин, то экземпляр активизируется.

8. Если клиент не вызывает пассивировавший экземпляр бина длительное время, то этот экземпляр удаляется.

9. Если клиент хочет удалить экземпляр бина, то экземпляр сначала активизируется при необходимости, затем удаляется.



PrePassivate callback

Пример использования данной аннотации:

```
@PrePassivate
@PreDestroy
public void cleanup() {
    ...
}
```

PostConstruct callback

Пример использования данной аннотации:

```
@PostConstruct
@PostActivate
public void openConnection() {
    ...
}
```

Пассивирование и активация

Если клиенты не вызывают бин в течение достаточно долгого времени, плохая идея, чтобы продолжать держать их в памяти. Для большого количества бинов это может легко заставить машину исчерпать память. Контейнер использует технику пассивирования, чтобы спасти память, когда это возможно.

Пассивирование по существу означает сохранение экземпляра на диск вместо того, чтобы держать его в памяти. Контейнер выполняет эту задачу, сериализует весь экземпляр бина и перемещает его на диск для хранения как файл. Активация - противоположность пассивирования и выполняется, когда экземпляр бина снова необходим.

Метод PrePassivate должен дать бину шанс подготовиться к сериализации. Данная операция может включать копирование значений несериализуемых переменных в сериализуемые переменные. Чаще всего шаг PrePassivate состоит из освобождения тяжелых ресурсов, типа соединения с базой данных и открытых сокетов, которые не могут быть сериализованы.

Разрушение stateful бина

Бизнес методы, отмеченные клиентом @Remove аннотацией, говорит об его желании закончить сессию. В результате этого метода происходит непосредственное разрушение бина.

Главные различия между stateless и stateful бинами

Возможности	**Stateless**	**Stateful**
Состояние	НЕТ	ДА
Пулинг	ДА	НЕТ
Проблемы в работе	Вряд ли	Возможны
События ЖЦ	PostConstruct, PreDestroy	PostConstruct, PreDestroy, PrePassivate, PostActivate
Таймер	ДА	НЕТ
Синхронизация сессий для транзакций	НЕТ	ДА
Веб-службы	ДА	НЕТ
Расширенный PersistenceContext	НЕТ	ДА

Клиенты бинов

Бин работает для клиента и может быть вызван локальным клиентом, расположенным в той же самой JVM или удаленным клиентом вне JVM. Сначала рассмотрим, как клиент получает доступ к бину, затем рассмотрим @EJB аннотацию.

В EJB 3 доступ к remote и local бину осуществляется одинаково. Клиент любого бина выполняет общие шаги для использования:

1. Клиент получает ссылку на бины прямо или косвенно от JNDI.
2. Доступ осуществляется через интерфейс.
3. Клиент делает много вызовов методов.
4. В случае stateful бина, последний запрос клиента должен быть remove метод.

Самый легкий подход получения ссылки на бин – это использовать инъекционную зависимость через @javax.ejb.EJB аннотацию. В зависимости от вашей окружающей среды клиента, возможно, придется использовать один из двух вариантов, доступных для того, чтобы получить ссылку EJB: использовать EJB контекстный поиск или использовать JNDI lookup.

Использование @EJB аннотации

Вот - спецификация для @EJB аннотации:

```
@Target({TYPE, METHOD, FIELD}) @Retention(RUNTIME)
public @interface EJB {
   String name() default "";
   Class beanInterface() default Object.class;
   String beanName() default "";
}
```

Все три параметра для @EJB аннотации являются дополнительными. Элемент name предлагает имя JNDI. beanInterface определяет бизнес интерфейс, который используется, чтобы получить доступ к EJB. beanName элемент позволяет нам отличать EJBs.

Инъекция и stateful бин

Вы можете внедрить stateful сессию в другую stateful сессию экземпляра бина, ели это необходимо. Имейте в виду, что вы не должны вводить stateful бин в stateless бин, так как stateless бин может быть разделен параллельными клиентами. Однако, инъекция экземпляра stateless бина в stateful бин вполне правильна.

Выбор типа бина для нужд приложения

- Выберите тип бина тщательно. Выбор stateless бина будет подходящим решением в большинстве случаев. Грамотно исследуйте вопрос о применении stateful бинов в вашем приложении.
- Тщательно исследуйте типы интерфейсов для бинов. Remote интерфейсы используют сеть и могут замедлить работу приложения. Если клиент будет всегда работать в пределах той же JVM, что и бин, то используйте local интерфейс.

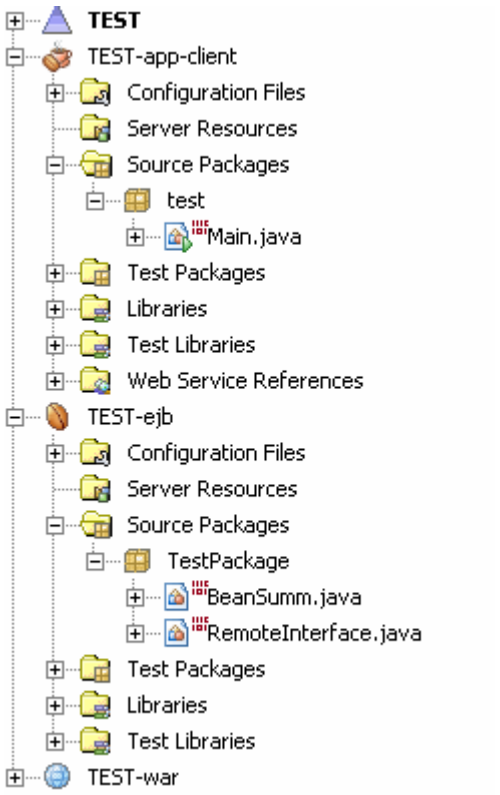
• Если вы используете инъекционную зависимость, удостоверьтесь, что вы не вводите stateful экземпляр в stateless экземпляр или servlet. Экземпляры EJB сохраняются в переменных экземпляров и доступны глобально для клиентов, даже если экземпляр stateless бина возвращается в пул, и инъецированный экземпляр stateful бина может содержать информацию, которая будет доступна для разных клиентов. Правильно было бы ввести stateful экземпляр в другой stateful экземпляр.

• Отделите проблему “взаимного сокращения” с использованием бизнес перехватчиков вместо того, чтобы внедрять ее в бизнес логику.

• Не забывайте определять remove методы в stateful бине.

• Настройте пассивирование и таймаут конфигурацию, чтобы найти оптимальные значение для вашего приложения.

Пример работы Stateless бина




```
package test;
import TestPackage.*;
import javax.ejb.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
public class Main
{
    public static void main(String [] args) throws Exception
    {
      Main c = new Main();
      c.RunTest();
    }
    // Ссылка на удаленный интерфейс - EJB контекстный поиск
    @EJB
    private static RemoteInterface beanSumm1;
    @EJB
    private static RemoteInterface beanSumm2;
```

```
    // Тестовый метод
    private void RunTest() throws Exception
    {
       beanSumm1.AddList(10);
       beanSumm2.AddList(11);
       beanSumm1.AddList(12);
       beanSumm2.AddList(13);
    }
}
```

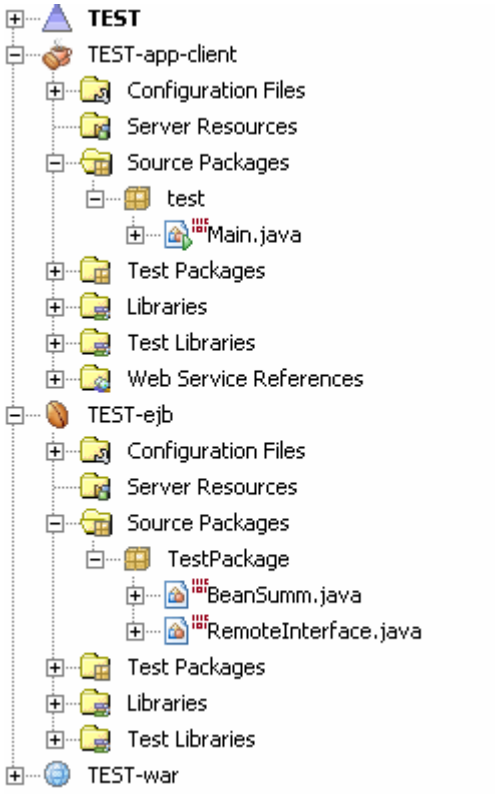
```
package TestPackage;
import java.util.*;
import javax.ejb.Stateless;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
@Stateless
public class BeanSumm implements RemoteInterface
{
    private ArrayList al;
    public void AddList(int a)
    {
      al.add(a);
      System.out.println("Теперь в массиве элементов: "+al.size());
      int sum=0;
      for (int i=0;i<al.size();i++)
          sum+=Integer.valueOf(String.valueOf(al.get(i)));
      System.out.println("Их сумма: "+sum);
    }
    @PostConstruct
    public void Construct()
    {
       al = new ArrayList();
       System.out.println("PostConstruct");
       System.out.println("Количество элементов в массиве: "+al.size());
    }
    @PreDestroy
    public void Destroy()
    {
       System.out.println("PreDestroy");
       al.clear();
       al = null;
    }
}
```

```
package TestPackage;
import javax.ejb.Remote;
@Remote
public interface RemoteInterface
{
   public void AddList(int a);
}
```

В результате работы программы будет напечатано следующее в консоли сервера приложений:

```
PostConstruct
Количество элементов в массиве: 0
Теперь в массиве элементов: 1
Их сумма: 10
Теперь в массиве элементов: 2
Их сумма: 21
Теперь в массиве элементов: 3
Их сумма: 33
Теперь в массиве элементов: 4
Их сумма: 46
```

Пример работы Stateful бина




```
package test;
import TestPackage.*;
import javax.ejb.*;
import javax.naming.InitialContext;
import javax.naming.NamingException;
public class Main
{
    public static void main(String [] args) throws Exception
    {
      Main c = new Main();
      c.RunTest();
    }
    // Ссылка на удаленный интерфейс - EJB контекстный поиск
    @EJB
    private static RemoteInterface beanSumm1;
    @EJB
    private static RemoteInterface beanSumm2;
    // Тестовый метод
    private void RunTest() throws Exception
    {
       beanSumm1.AddList(10);
       beanSumm2.AddList(11);
       beanSumm1.AddList(12);
       beanSumm2.AddList(13);
    }
}
```

```
package TestPackage;
import javax.ejb.Remote;
@Remote
public interface RemoteInterface
{
   public void AddList(int a);
}
```

```
package TestPackage;
import java.util.*;
import javax.ejb.Stateful;
import javax.annotation.PostConstruct;
import javax.annotation.PreDestroy;
import javax.ejb.PostActivate;
import javax.ejb.PrePassivate;
import javax.ejb.Remove;
@Stateful
public class BeanSumm implements RemoteInterface
{
    private ArrayList al;
    public void AddList(int a)
    {
      al.add(a);
      System.out.println("Теперь в массиве элементов: "+al.size());
      int sum=0;
      for (int i=0;i<al.size();i++)
          sum+=Integer.valueOf(String.valueOf(al.get(i)));
      System.out.println("Их сумма: "+sum);
    }
    @PostActivate
    @PostConstruct
    public void Construct()
    {
       al = new ArrayList();
       System.out.println("PostConstruct");
       System.out.println("Количество элементов в массиве: "+al.size());
    }
    @PreDestroy
    @PrePassivate
    public void Destroy()
    {
       System.out.println("PreDestroy");
       //Несериализуемых переменных нет
    }
    @Remove
    public void Cancel()
    {
       al = null;
    }
}
```

В результате работы программы будет напечатано следующее в консоли сервера приложений:

```
PostConstruct
Количество элементов в массиве: 0
PostConstruct
Количество элементов в массиве: 0
Теперь в массиве элементов: 1
Их сумма: 10
Теперь в массиве элементов: 1
Их сумма: 11
Теперь в массиве элементов: 2
Их сумма: 22
Теперь в массиве элементов: 2
Их сумма: 24
```