



# Enterprise Java Beans


For students of universities

Author: Oxana Dunik

# Технологию EJB (Enterprise Java Beans)

- можно рассматривать с двух точек зрения: как **фреймворк**, и как **компонент**.
  С точки зрения компонента **EJB** - это всего-лишь **надстройка** над POJO-классом, описываемая с помощью **аннотации(**
- **POJO это просто класс Java, который:**
  **1. Содержит только поля, без логики их обработки;**
  **2. Доступ ко все полям только через get/set )**.
- Существует три типа компонентов EJB:

- session beans - используется для описания бизнесс-логики приложения
- message-driven beans - так же используется для бизнесс-логики
- entities - используется для хранения данных
- С точки зрения **фреймворка** EJB - это технология, предоставляющая множество готовых решений (управление транзакциями, безопасность, хранение информации и т.п.) для вашего приложения.

# Основные архитектуры EJB

- Существует 2 основные архитектуры при разработке enterprise-приложений:
- традиционная слоистая архитектура (traditional layered architecture)
- domain-driven design (DDD)

# традиционная слоистая архитектура (traditional layered architecture) EJB

- **4 базовых слоя:** слой презентации, слой бизнесс-логики, слой хранения данных и непосредственно слой самой базы данных.
- Обычно слой **презентации** реализуется через **web-приложение** (т.е. используя **JSP, JSF, GWT** и т.п.) или **web-сервис.**
- Слой **бизнесс-логики** является основой для **enterprise-**приложения. В нем описываются бизнесс-процессы, производится поиск, авторизация и множество других вещей. Слой бизнесс-логики использует механизмы слоя **хранения данных.**
- Чем отличается слой **хранения данных** и слой **базы данных?** Тем, что в первом описываются высокоуровневые объектно-ориентированные механизмы для работы с сущностями БД, в то время как второй - это и есть непосредственно база данных **(Oracle, MySQL** и т.п.)

# domain-driven design (DDD)

- Архитектура **DDD** предпологает, что объекты обладают бизнесс-логикой, а не являются простой репликацией объектов БД. Многие программисты не любят наделять объекты логикой и создают отдельный слой, называемый service layer или application layer. Он похож на слой бизнесс-логики традиционной слоистой архитектуры за тем лишь отличием, что он намного *тоньще*.

# WHAT IS EJB??

- Enterprise JavaBeans (EJB) is a managed, server-side component architecture for modular construction of enterprise applications.
- EJB is a server-side model that encapsulates the business logic of an application.
- The EJB specification was originally developed in 1997 by IBM and later adopted by Sun Microsystems (EJB 1.0 and 1.1) in 1999.


# EJB 3

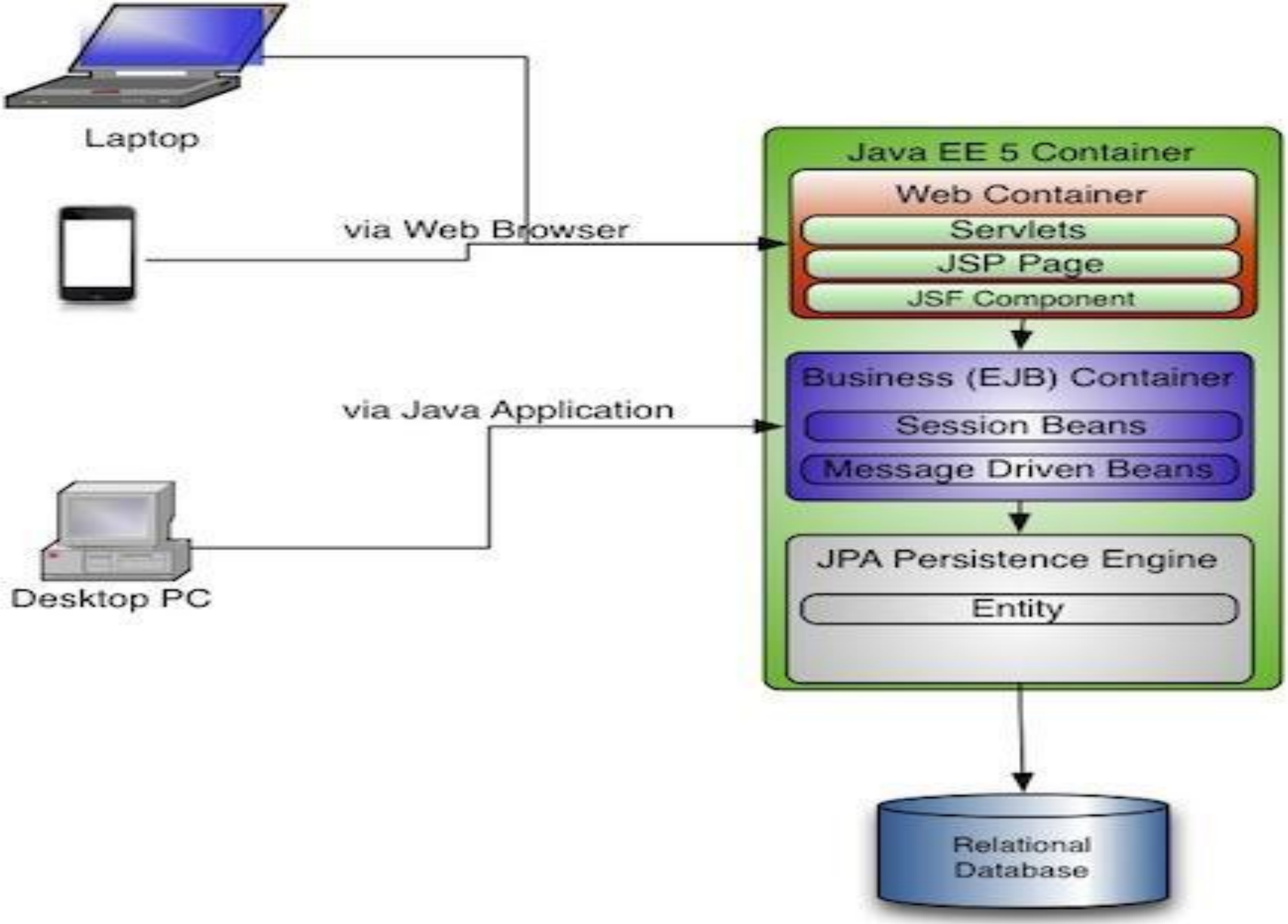
Session bean

Message-driven bean

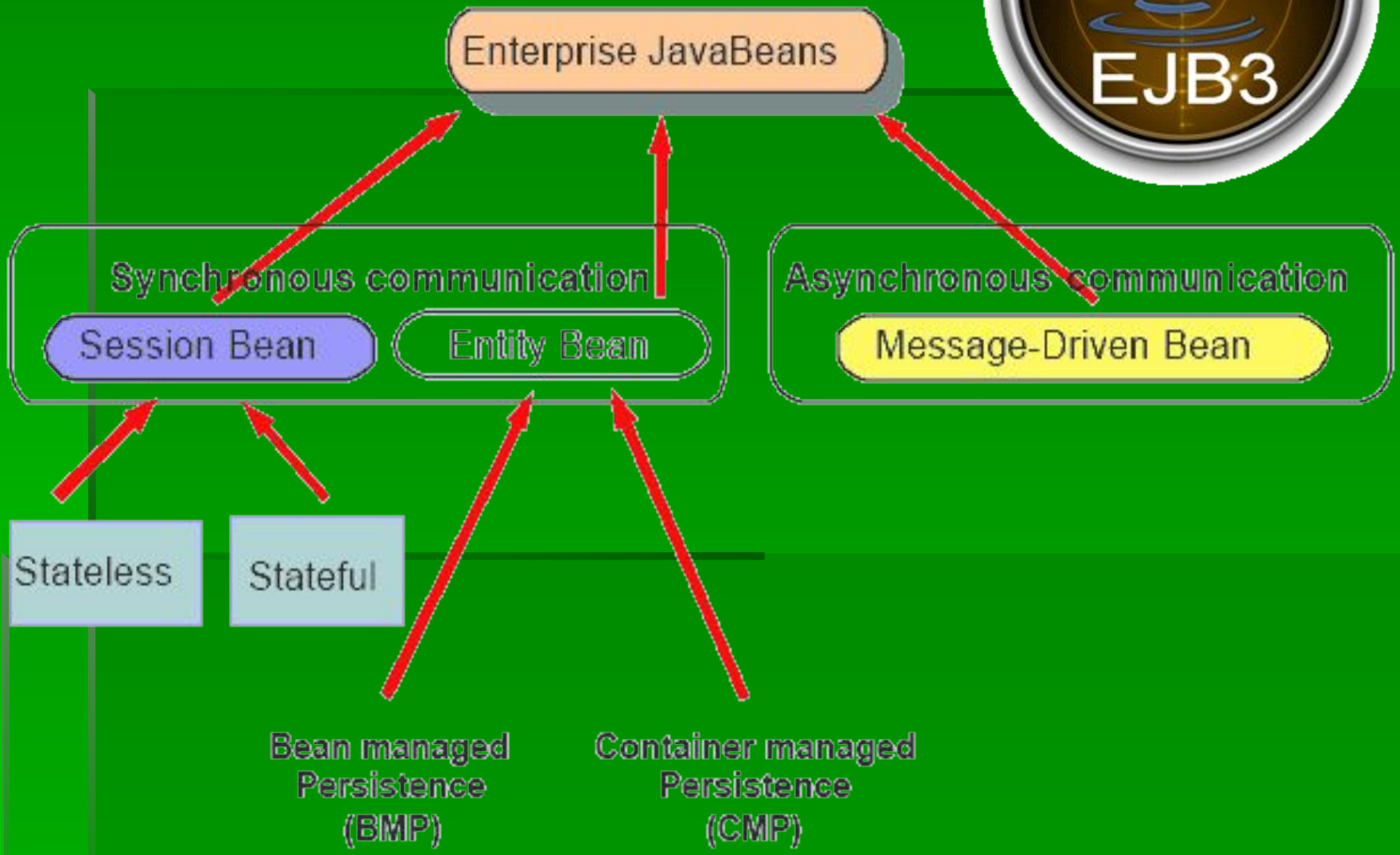
## JPA

Entity bean

- Сервер приложений j2ee состоит из двух основных элементов: **контейнер web-приложения (JSP, JSF и т.д.)** и **EJB-контейнер.** Первый служит для создания пользовательского интерфейса и слабо подходит для описания бизнес-логики приложения. Для этого используется вторая часть J2EE - EJB.


Laptop
via Web Browser
via Java Application
Desktop PC
Java EE 5 Container
Web Container
Servlets
JSP Page
JSF Component
Business (EJB) Container
Session Beans
Message Driven Beans
JPA Persistence Engine
Entity
Relational Database

# Enterprise JavaBeans

# Session bean


- *Session bean* представляет собой EJB-компоненту, связанную с одним клиентом. ``Бины" этого типа, как правило, имеют ограниченный срок жизни (хотя это и не обязательно), и редко участвуют в транзакциях. В частности, они обычно не восстанавливаются после сбоя сервера. В качестве примера session bean можно взять ``бин", который живет в веб-сервере и динамически создает HTML-страницы клиенту, при этом следя за тем, какая именно страница загружена у клиента. Когда же пользователь покидает вэб-узел, или по истечении некоторого времени, session bean уничтожается. Несмотря на то, что в процессе своей работы, session bean мог сохранять некоторую информацию в базе данных, его предназначение заключается все-таки не в отображении состояния или в работе с ``вечными объектами", а просто в выполнении некоторых функций на стороне сервера от имени одного клиента.

# Session bean


В качестве session bean может выступать обычный класс Java удовлетворяющий следующим условиям:

- иметь как минимум один метод
- не должен быть абстрактным
- иметь конструктор по-умолчанию
- методы не должны начинаться с "ejb" (например ejbCreate, ejbDoSomething)

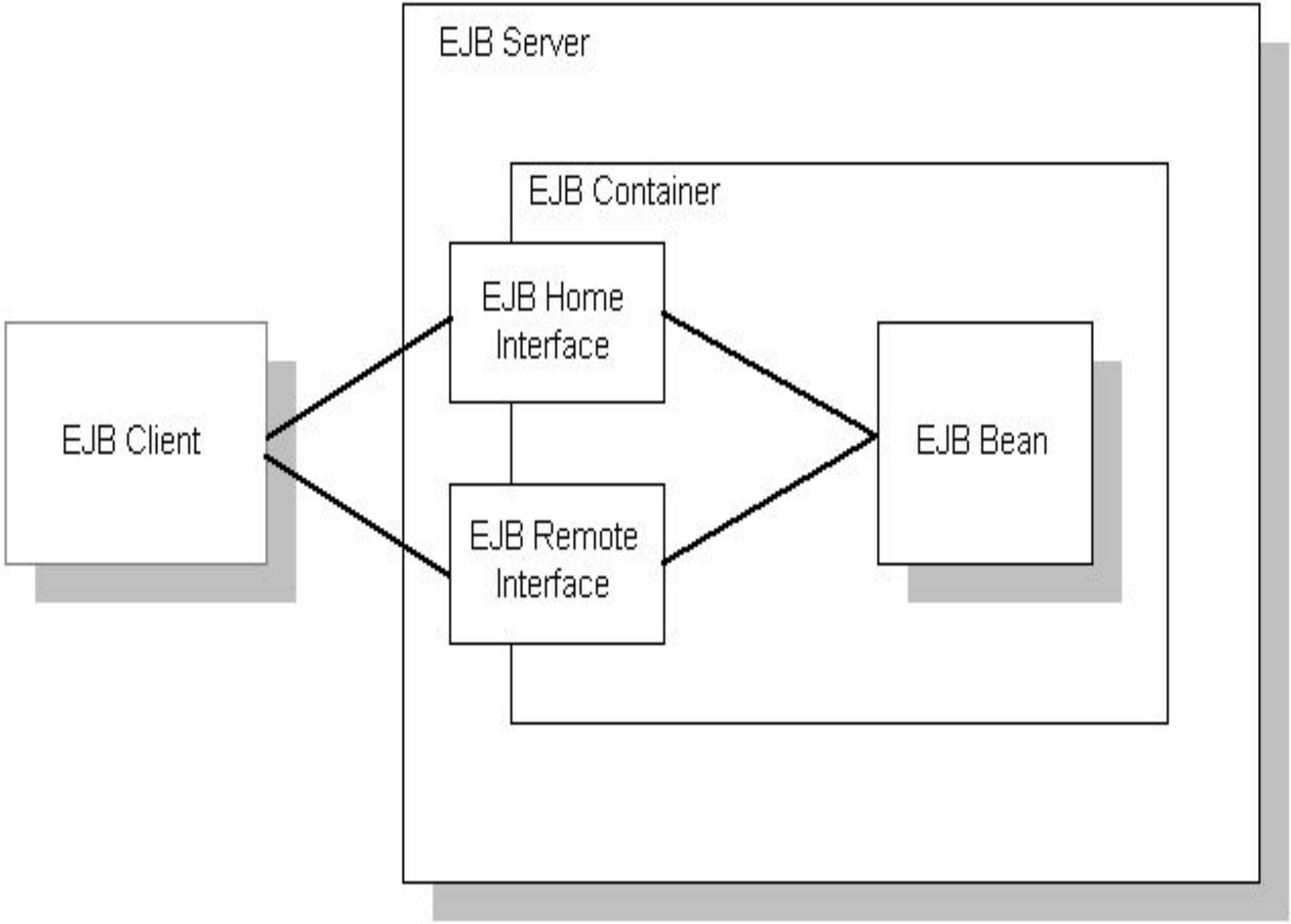
Для stateful-бинов существует еще одно условие: свойства класса должны быть объявлены примитивами или реализовывать интерфейс Serializable.

# Особенности **stateless** и **stateful** бинов


- Один bean может содержать множество клиентских методов. Этот момент является важным для производительности, так как контейнер помещает экземпляры **stateless**-бинов в общее хранилище и множество клиентов могут использовать один экземпляр бина.
  В отличии от stateless **stateful** бины инстанцируются для каждого пользователя отдельно.

# Entity bean


- *Entity bean*, наоборот, представляет собой компоненту, работающую с постоянной (persistent) информацией, хранящейся, например, в базе данных. Entity beans ассоциируются с элементами баз данных и могут быть доступны одновременно нескольким пользователям. Так как информация в базе данных является постоянной, то и entity beans живут постоянно, ``выживая'', тем самым, после сбоев сервера (когда сервер восстанавливается после сбоя, он может восстановить ``бин'' из базы данных).


EJB Server
EJB Container
EJB Home Interface
EJB Client
EJB Remote Interface
EJB Bean

# EJB-компонента (The Enterprise JavaBeans component)


- Отдельная EJB-компонента представляет цобой компоненту в том же смысле что и традиоционный JavaBeans ``bean'' (``зерно''). Компоненты EJB выполняются внутри EJB-*контейнера*, который, в свою очередь, выполняется внутри EJB-*сервера*. Любой сервер, который в состоянии поддерживать EJB-контейнеры и предоставлять им необходимые сервисы, может быть EJB-сервером (то есть многие из существующих серверов могут быть просто расширены до поддержки Enterprise JavaBeans).
- EJB-компонента представляет из себя Java-класс, который реализует некоторую бизнес-логику. Все остальные классы в EJB-системе либо реализуют поддержку клиент/сервер взаимодйествий между компонентами, либо реализуют некоторые сервисы для компонент.

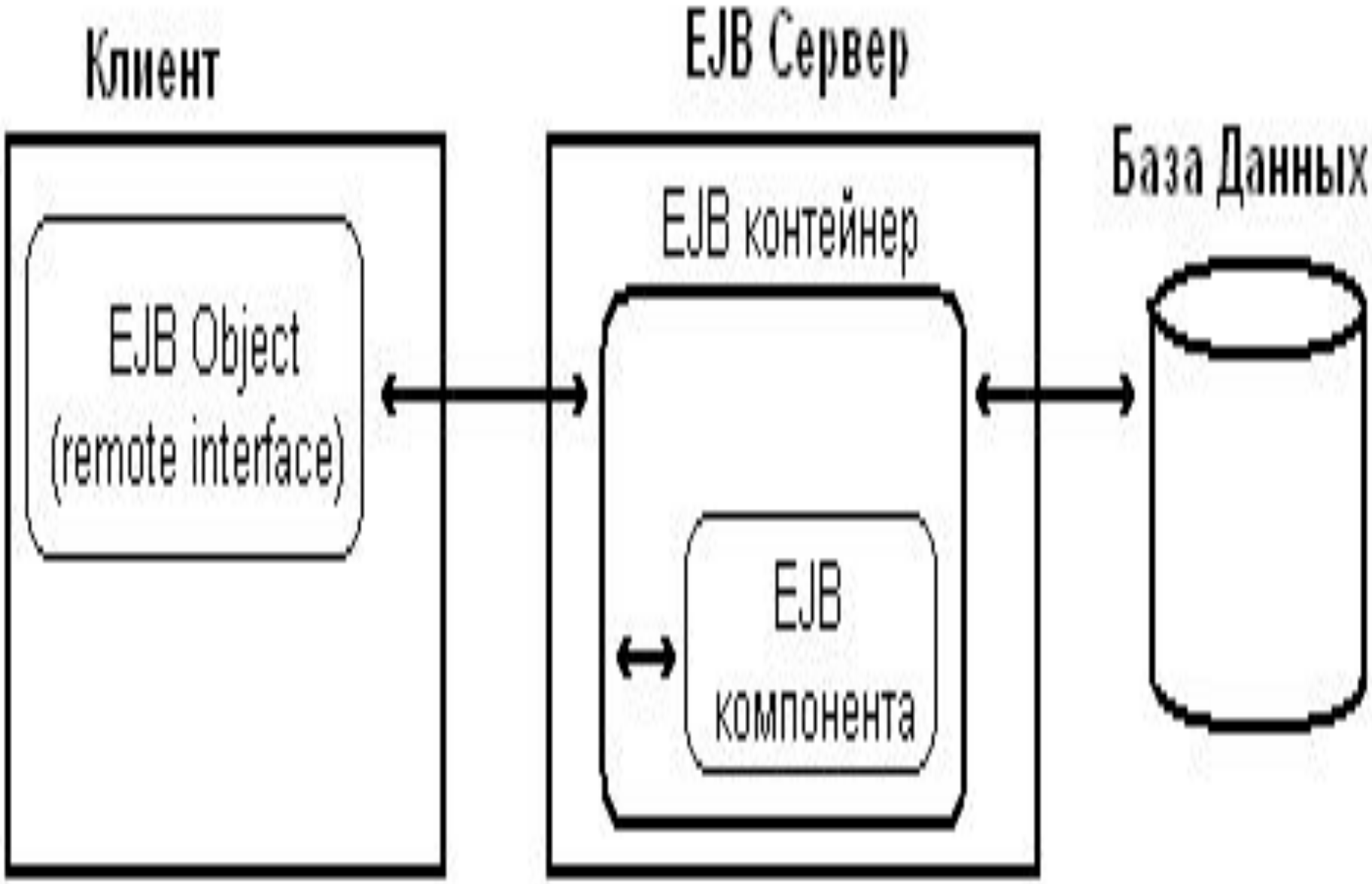
# EJB-контейнер (The Enterp JavaBeans container)


- EJB-контейнер -- это то место, где ``живет'' EJB-компонент. EJB-контейнер реализует для находящихся в нем компонент такие сервисы как транзакции (transaction), управление ресурсами, управление версиями компонент, их мобильностью, настраиваемостью, мобильностью, жизненным циклом. Так как EJB-контейнер реализует все эти функции, то разработчик EJB-компонент может не реализовывать их самостоятельно, а просто вызывать соответсвующие методы у контейнера (правила вызова методов у контейнера описываются в спецификации). Как правило, в одном EJB-контейнере живет несколько однотипных EJB-компонент.

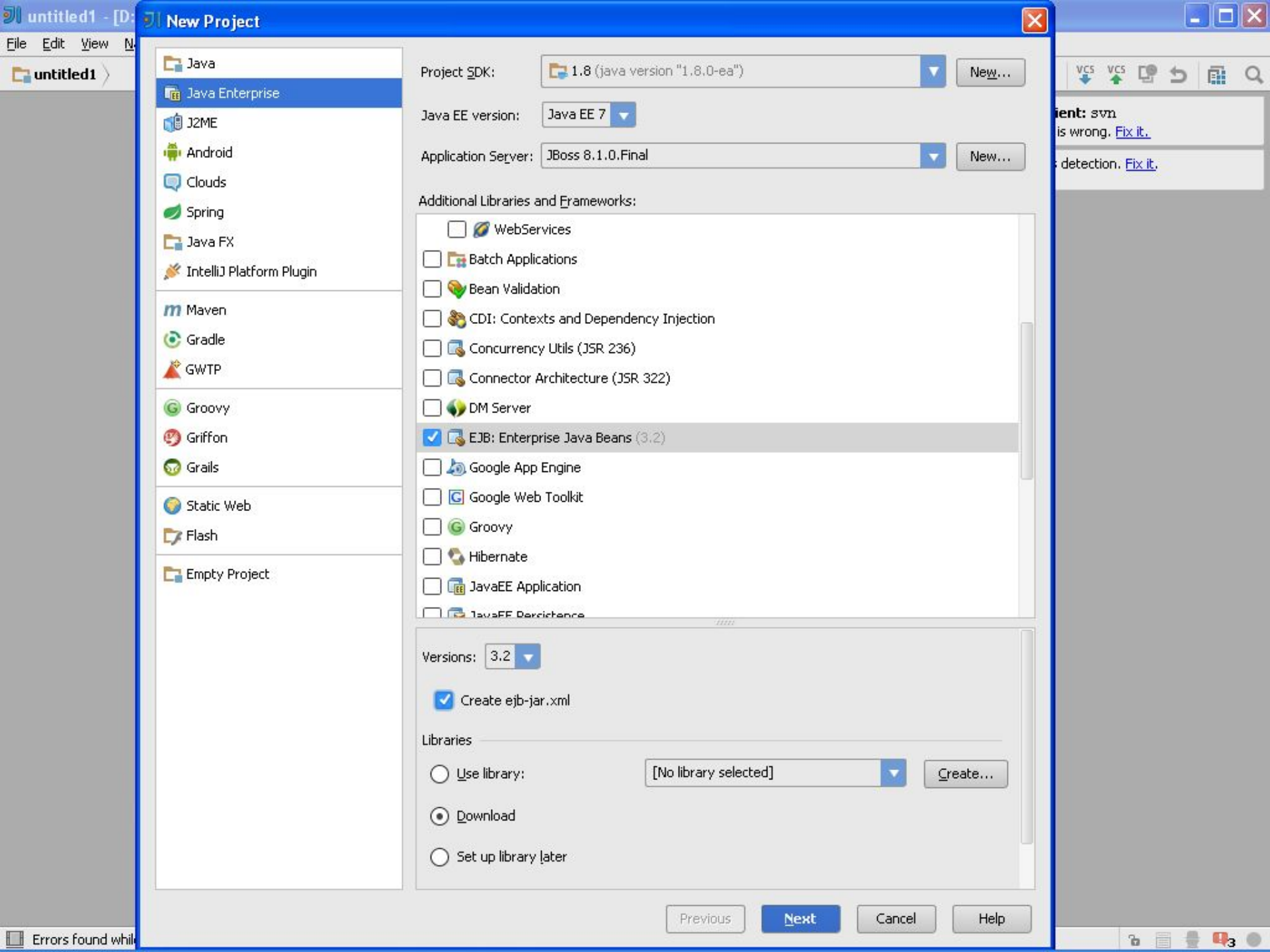
# EJB-объект (EJB-object) и удаленный интерфейс (remot interface)


- Клиентские приложения вызывают методы на удаленных EJB-компонентах через EJB-объект (EJB-object). EJB-объект реализует ``удаленный интерфейс" EJB-компоненты на сервере. Суть в том, что находящаяся на сервере EJB-компонента, помимо бизнес-функций, ради которых она была разработана, должна реализовывать также некоторые функции, определяемые спецификацией, которые служат для ``управления" EJB-компонентой со стороны контейнера. EJB-объект реализует лишь бизнес-интерфейс для EJB-компоненты, являясь, в некотором смысле, ``промежуточным" звеном между клиентом и EJB-компонентой.

**EJB-компонента выполняется на сервере, внутри EJB-контейнера и реализует бизнес-логику, в то время как EJB-объект выполняется у клиента и удаленно вызывает методы у EJB-компоненты.**


Клиент
EJB Сервер
База Данных
EJB контейнер
EJB Object
(remote interface)
EJB
компонента

- Итак, приложение-клиент соединяется с EJB-сервером и посылает ему запрос на создание ``бина'' (Enterprise JavaBean) для обработки своих запросов. Сервер отвечает на такой запрос созданием объекта на стороне сервера (экземпляр EJB-компоненты) и возвращает клиенту прокси-объект (EJB-объект), чей итерфейс совпадает с интерфейсом созданной EJB-компоненты и чьи методы перенаправляют вызовы собственно экземпляру компоненты. После этого приложение-клиент работает с EJB-объектом как с локальным объектом, даже и не подозревая, что всю работу выполняет не EJB-объект, а удаленная компонента на сервере. Необходимо заметить, что созданием и удалением EJB-компонент на сервере занимается EJB-контейнер.


New Project
Java
Java Enterprise
J2ME
Android
Clouds
Spring
Java FX
IntelliJ Platform Plugin
Maven
Gradle
GWTP
Groovy
Griffon
Grails
Static Web
Flash
Empty Project
Project SDK:
1.8 (java version "1.8.0-ea")
New...
Java EE version:
Java EE 7
Application Server:
JBoss 8.1.0.Final
New...
Additional Libraries and Frameworks:
WebServices
Batch Applications
Bean Validation
CDI: Contexts and Dependency Injection
Concurrency Utils (JSR 236)
Connector Architecture (JSR 322)
DM Server
EJB: Enterprise Java Beans (3.2)
Google App Engine
Google Web Toolkit
Groovy
Hibernate
JavaEE Application
Versions:
3.2
Create ejb-jar.xml
Libraries
Use library:
[No library selected]
Create...
Download
Set up library later
Previous
Next
Cancel
Help

# POJO, POJI, annotations


- Теперь рассмотрим реализацию этих сущностей. В EJB3 мы используем **POJO** (Plain Old Java Objects), **POJI** (Plain Old Java Interfaces) и **аннотации**.
- Аннотация записывается так: @<имя аннотации>(<список парамет-значение>)

- Какие аннотации могут быть?

- **Stateless** - говорит контейнеру, что класс будет stateless session bean. Для него контейнер обеспечит безопасность потоков и менеджмент транзакций. Дополнительно, вы можете добавить другие свойства, например прозрачное управление безопасностью и перехватчики событий;
- **Local** - относится к интерфейсу и говорит, что bean реализующий интерфейс доступен локально
- **Remote** - относится к интерфейсу и говорит, что bean доступен через RMI (Remote Method Invocation)
- **EJB** - применятеся в коде, где мы используем bean.
- **Stateful** - говорит контейнеру, что класс будет stateful session bean.
- **Remove** - опциональная аннотация, которая используется с stateful бинами. Метод, помеченный как Remove говорит контейнеру, что после его исполнения нет больше смысла хранить bean, т.е. его состояние сбрасывается. Это бывает критично для производительности.
- **Entity** - говорит контейнеру, что класс будет сущностью БД
- **Table**(name="...") - указывает таблицу для маппинга
- **Id**, **Column** - параметры маппинга
- **WebService** - говорит, что интерфейс или класс будет представлять вэб-сервис.
- и много-много других...

```java
@Entity
@Table(name="book")
@SequenceGenerator(name = "book_sequence", sequenceName = "book_id_seq")
@TableGenerator(  name="book_id", table="primary_keys", pkColumnName="key", pkColumnValue="book",
        valueColumnName="value")
public class Book implements Serializable {
    private static final long serialVersionUID = 7422574264557894633L;
    private Integer id;
    private String title;
    private String author;
    public Book() { super(); }
    public Book(Integer id, String title, String author) {
        super();
        this.id = id;
        this.title = title;
        this.author = author;
    }
    @Override
    public String toString() {
        return "Book: " + getId() + " Title " + getTitle() + " Author "
                + getAuthor();
    }

    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }
    @Id
    @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "book_sequence")
    public Integer getId() { return id; }
    public void setId(Integer id) { this.id = id; }
    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}
```

# POJO, POJI, annotations


- У **stateless** и **MDB** бинов существует **2 события** жизненного цикла, которые мы можем перехватить: создание и удаление бина. Метод, который будет вызываться сразу после создании бина помечается аннотацией javax.annotation.PostConstruct, а перед его удалением - javax.annotation.PreDestroy. **Stateful** бины обладают помимо рассмотреных выше **еще 2 событиями**: при активации (javax.ejb.PostActivate) и при деактивации (javax.ejb.PrePassivate).

- **import** javax.ejb.Local;

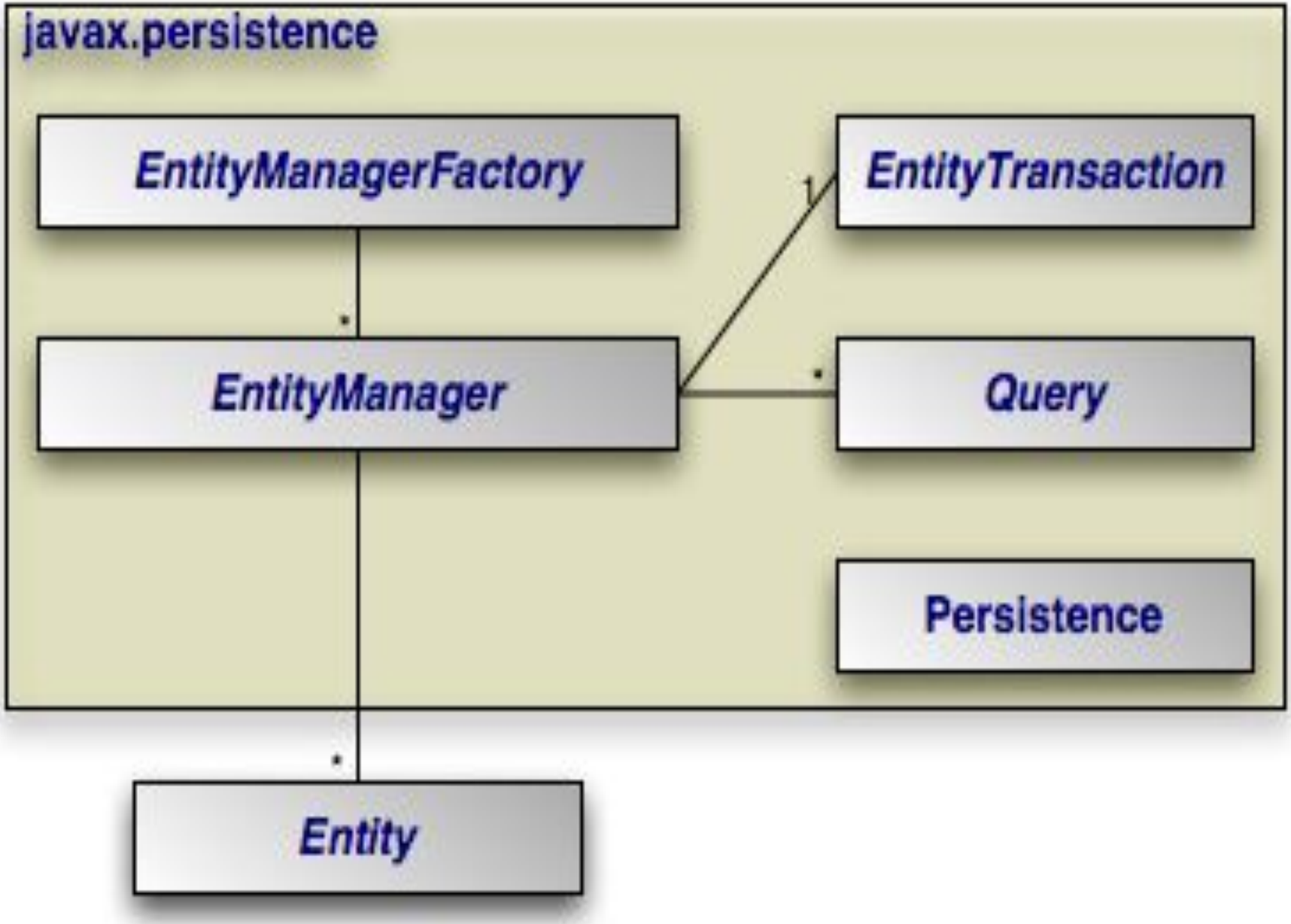
  @Local
  **public interface** BookTestBeanLocal {
  **public void** test();
  }
  ---
- **import** javax.ejb.Remote;

  @Remote
  **public interface** BookTestBeanRemote {
  **public void** test();
  }

```java
import javax.ejb.Stateless;
import javax.persistence.EntityManager;
import javax.persistence.PersistenceContext;
@Stateless
public class BookTestBean implements BookTestBeanLocal, BookTestBeanRemote {
    @PersistenceContext
    EntityManager em;
    public static final String RemoteJNDIName =  BookTestBean.class.getSimpleName() + "/remote";
    public static final String LocalJNDIName =  BookTestBean.class.getSimpleName() + "/local";
    public void test() {
        Book book = new Book(null, "My first bean book", "Sebastian");
        em.persist(book);
        Book book2 = new Book(null, "another book", "Paul");
        em.persist(book2);
        Book book3 = new Book(null, "EJB 3 developer guide, comes soon",
                "Sebastian");
        em.persist(book3);
        System.out.println("list some books");
        List someBooks = em.createQuery("select from Book b where b.author=:name")
                .setParameter("name", "Sebastian").getResultList();
        for (Iterator iter = someBooks.iterator(); iter.hasNext();)
        {
            Book element = (Book) iter.next();
            System.out.println(element);
        }
        System.out.println("List all books");
        List allBooks = em.createQuery("from Book").getResultList();
        for (Iterator iter = allBooks.iterator(); iter.hasNext();)
        {
            Book element = (Book) iter.next();
            System.out.println(element);
        }
        System.out.println("delete a book");
        em.remove(book2);
        System.out.println("List all books");
         allBooks = em.createQuery("select from Book").getResultList();
```

```
        System.out.println("List all books");
        List allBooks = em.createQuery("select from Book").getResultList();
        for (Iterator iter = allBooks.iterator(); iter.hasNext();)
        {
            Book element = (Book) iter.next();
            System.out.println(element);
        }
        System.out.println("delete a book");
        em.remove(book2);
        System.out.println("List all books");
         allBooks = em.createQuery("select from Book").getResultList();
        for (Iterator iter = allBooks.iterator(); iter.hasNext();)
        {
            Book element = (Book) iter.next();
            System.out.println(element);
        }

    }
}
```


javax.persistence
EntityManagerFactory
EntityTransaction
EntityManager
Query
Persistence
Entity

```java
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;

import de.lalilуna.library.BookTestBean;
import de.lalilуna.library.BookTestBeanRemote;
public class FirstEJB3TutorialClient {
    public static void main(String[] args) {
        Context context;
        try
        {
            context = new InitialContext();
            BookTestBeanRemote beanRemote = (BookTestBeanRemote) context.lookup(BookTestBean.RemoteJNDIName);
           beanRemote.test();
        } catch (NamingException e)
        {
           e.printStackTrace();
           throw new RuntimeException(e);
        }
    }
}
```

# Перехватчики(interceptors)

- При создании enterprise-приложений часто возникает необходимость записывать лог вызываемых методов (в целях отладки или для лога безопасности), а так же контролировать доступ пользователей к отдельным частям приложения. Для этого используются **перехватчики** - объекты, методы которых вызываются **автоматически** при **вызове** метода **EJB**-бина.

# Перехватчики(interceptors)

- Объект-перехватчик является POJO, за тем лишь исключением, что метод, который должен вызываться автоматически аннотируется @AroundInvoke

```
package com.sample;

import javax.interceptor.AroundInvoke;
import javax.interceptor.InvocationContext;

public class MyInterceptor {

    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception
    {
        System.out.println("*** TracingInterceptor intercepting " + ctx.getMethod().getName());
        long start = System.currentTimeMillis();
        String param = (String)ctx.getParameters()[0];

        if (param == null)
            ctx.setParameters(new String[]{"default"});

        try
        {
            return ctx.proceed();
        }
        catch(Exception e)
        {
            throw e;
        }
        finally
        {
            long time = System.currentTimeMillis() - start;
            String method = ctx.getClass().getName();
            System.out.println("*** TracingInterceptor invocation of " + method + " took " + time + "ms");
        }
    }
}
```

```
@Stateless(name="SampleEJB")
@RemoteBinding(jndiBinding="SampleEJB")
@Interceptors(value=com.sample.MyInterceptor.class)

public class SampleBean implements Sample {

    public void doSomething(String param) {
        callWebService(param);
    }
    @AroundInvoke
    public Object log(InvocationContext ctx) throws Exception
    {

        try
        {
            return ctx.proceed();
        }
        catch(Exception e)
        {
            throw e;
        }
    }
}
```

## How many kind of Interceptors can you apply ?

You can apply Interceptors at three different level:
1) **Default interceptors**:

These interceptors needs to be declared in your ejb-jar.xml and are valid across **all your EJB deployed** :

```
<assembly-descriptor>
<interceptor-binding>
<ejb-name>*</ejb-name>
<interceptor-class>sample.interceptor.MyDefaultInterceptor</interceptor-class>
</interceptor-binding>
...
</assembly-descriptor>
```

**2) Class level interceptors:**

This is the kind of interceptor we've seen in our example: it is valid across **all methods** of an EJB.

```
@Stateless
    @Interceptors(value=com.sample.SampleInterceptor.class)
    public class StatelessBean
    {
       ...
    }
```

**3) Method level interceptors:**

The method level interceptor only intercepts the single method call :

```
    @Interceptors(value=com.sample.MethodInterceptor.class)
    public void doSomething()
    {
       ...
    }
```

## How do you exclude the DefaultInterceptor ?

If you have defined a default interceptor for all your EJB in a jar file, you can use the following annotation to exclude it and instead execute the Interceptor in the class:

```
1  @Stateless(name="SampleEJB")
2  @ExcludeDefaultInterceptors
3  public class TestEJB implements Test
4  {
5
6      @AroundInvoke
7      public Object customInterceptor(InvocationContext ctx) throws Exception
8      {
9          System.out.println("*** CustomInterceptor intercepting");
10         return ctx.proceed();
11     }
12  }
```

# Literature

- http://j4sq.blogspot.com/2011/10/ejb-1.html
- http://www.laliluna.de/articles/posts/ejb-3-tutorial-jboss.html
- https://www.youtube.com/watch?v=wKwihFhBjHI
- http://www.tutorialspoint.com/ejb/ejb_interceptors.htm