Components Based Applications
Example with JEE-EJB3.0

Reda Bendraou
Reda.Bendraou@lip6.fr
This course is inspired by the readings/sources listed in the last slide
JEE Architecture

The EJB architecture is THE Java standard for the design and realization of distributed enterprise applications.
- Comes in form of an API

Aspects addressed by the standard:
- Design
- Deployment
- Life cycle management of application components at runtime

« In programming with the EJB 3.0 API, the developer typically uses the enterprise bean class as the primary programming artifact »

EJB: When?

If the application:
- Has to be scalable
- Need of a (distributed) transactional context
- Diversity of clients

Why using Enterprise Java Beans 3?

Encapsulating business logic
- Business logic separated from control and presentation

Remote access
- Multiple apps on different servers can access EJBs

Simplicity
- Relatively easy to use compared to other remote-object systems

Broad vendor support
- JBoss, Oracle AS, WebLogic, WebSphere, Glassfish, etc.

Scalability
- Virtually all Java EE app servers support clustering, load balancing, and failover
Why using Enterprise Java Beans 3?

EJB Servers provide a built-in solution that discharge/unburden the developer of the task of explicitly coding essential services such as:

- Network connections between the clients and the EJBs
- Naming services (JNDI)
- Transactions
- Persistence and the management of DB pool of connections
- Distribution
- Security
- Management of component’s life cycle

EJB Version 3 are much more readable and easier to implement than previous versions (see next slide)

Disadvantages of EJB

- Complexity
  - Although EJB3 might be simpler than other remote object systems, remote-object frameworks are much more complex than local-object approaches.
  - Spring is easier and more powerful for local access
- Requires Java EE server
  - Can’t run on Tomcat, Jetty, Resin, JRun
  - Java EE servers are usually much harder to configure, dramatically slower to start/restart during development and testing, and usually cost money
- Requires latest Java EE version
  - Must upgrade to latest server releases
- Bad reputation due to earlier releases
  - EJB2 was so complex that EJB has bad rap to this day (see next slide)

Why the industry was disappointed by EJBs 1.x & 2.x

EJB 1.x and 2.x

- Too complicated, heavy, constraining
- Difficulty to use some basic OO Concepts (inheritance, polymorphism, …)
- limited support of objet/relational mapping
- Lot of XML files, hard to write/maintain/understand

From EJB 2 to EJB 3

- Use of Java annotations and genericity in order to simplify Beans writing
- Important reduction of XML files (see next slide)

Simplicity of EJB3 (wrt. EJB2.x)

<table>
<thead>
<tr>
<th>Application Name</th>
<th>Item Measured</th>
<th>JEE 1.4 Platform</th>
<th>Java EE 5 Platform</th>
<th>Improvement</th>
</tr>
</thead>
<tbody>
<tr>
<td>AdventureBuilder</td>
<td>Number of classes</td>
<td>87</td>
<td>43</td>
<td>50% fewer classes</td>
</tr>
<tr>
<td></td>
<td>Lines of code</td>
<td>3,284</td>
<td>2,777</td>
<td>15% fewer lines of code</td>
</tr>
<tr>
<td>RemoteApp</td>
<td>Number of classes</td>
<td>17</td>
<td>7</td>
<td>60% fewer classes</td>
</tr>
<tr>
<td></td>
<td>Lines of code</td>
<td>167</td>
<td>716</td>
<td>27% fewer lines of code</td>
</tr>
<tr>
<td></td>
<td>Number of XML files</td>
<td>9</td>
<td>2</td>
<td>95% fewer XML files</td>
</tr>
<tr>
<td></td>
<td>Lines of XML code</td>
<td>702</td>
<td>34</td>
<td>97% fewer lines of XML code</td>
</tr>
</tbody>
</table>
Notion of implicit Middleware

**Principle**
- Writing your business object without worrying about distribution, transactions, persistence aspects
  - Some aspects can be customized using configuration files (jndi.properties, persistence.xml, etc.)

**Purpose**
- More readable code/components
- Your component => (almost) POJO
- Reusable solutions => Portable from one server into another

Enterprise Java Beans

"server-side component that encapsulates the business logic of an application"

The Triad of Beans
- **Session Beans**: performs a task for a client
- **Entity Beans**: represents a business entity object that exists in persistent storage
- **Message-Driven Beans**: listening processing messages asynchronously

Accessing the Beans

Each EJ Bean may provide remote interfaces
- Client runs in a different JVM
- Transparent Distribution

Each EJ Bean may provide local interfaces
- Web components or beans in the same JVM

Local Vs. Distant Interfaces

**Beans strongly or loosely coupled**
- Strong coupling: two interdependent beans => local

**Type of the client**
- Clients can be:
  - Applications located in a client machine (heavy client) => distant
  - Web Components (jsp, servlet) => distant/local
  - Other beans => distant/local

Clustering and load balancing
- Beans on the server side located in different machines => distant
Session Beans

Definition: Session Beans are reusable components that contain logic for business processes.

A session bean can perform:
- Banking transaction, stock trades, complex calculation, a workflow, etc.

Session Bean Types

1. Stateless session bean
   - without a state
   - Information is not persistent between two successive calls
   - 2 instances of a given bean are equivalent

2. Stateful session bean
   - Has a state (in memory)
   - Similar to Servlet/JSP session
   - The same instance along a client’s session
   - 1 instance per client
   - Heavy!!! Be careful!

Stateless Session Beans: Coding

1 interface (or 2 : Local + Remote) + 1 class

The Interface

- annotations @javax.ejb.Local or @javax.ejb.Remote

```java
import javax.ejb.Local;
@Local
public interface CalculatriceItf {
    public double add(double v1, double v2);
    public double sub(double v1, double v2);
    public double mul(double v1, double v2);
    public double div(double v1, double v2);
}
```

Stateful Session Beans: Coding

Class (that implements the interface)

- annotation @javax.ejb.Stateless or @javax.ejb.stateful

```java
import javax.ejb.Stateless;
@Stateless
public class CalculatriceBean implements CalculatriceItf {
    public double add(double v1, double v2) {return v1 + v2;}
    public double sub(double v1, double v2) {return v1 - v2;}
    public double mul(double v1, double v2) {return v1 * v2;}
    public double div(double v1, double v2) {return v1 / v2;}
}
```
Remote Client’s Code for Stateless Beans

Clients find the bean via JNDI
✓ Client Java code doesn’t even know the machine on which the bean is deployed

Clients use the bean like a normal POJO
✓ But arguments and return values are sent across network
✓ So, custom classes should be Serializable

Core code

```java
InitialContext context = new InitialContext();
InterfaceName bean = (InterfaceName) context.lookup("JNDI-Name");
```

`jndi.properties`
✓ Text file in classpath; gives remote URL and other info

Local Client’s Code for Stateless Beans

- Could be a servlet or a JSP
- Located in the same server as the bean
- Use of Dependency Injection mechanism
✓ Property (variable) typed by the Interface
✓ annotated with `@EJB` eventually `@EJB(name="foobar")`

```java
public class ClientServlet extends HttpServlet {
    @EJB(name="foobar")
    private CalculatriceItf myBean;

    public void service(HttpServletRequest req, HttpServletResponse response) {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        double result = myBean.add(12, 4.75);
        out.println("<html><body>"+result+"</body></html>");
    }
}
```

Remote Client’s Code for Stateless Beans

Code of a distant client

```java
public class Client {
    public static void main(String args[]) throws Exception {
        javax.naming.Context ic = new javax.naming.InitialContext();
        CalculatriceItf bean = (CalculatriceItf) ic.lookup("foobar");
        double res = bean.add(3, 6);
    }
}
```

Local Client’s Code for Stateless Beans

Restrictions

– Before instance variables, not local variables

– Both classes must be part of the same EAR on the same server
✓ In Eclipse, all classes in a single EJB project satisfy this
✓ If you use an EJB project (EJBs) and Dynamic Web projects (classes that use the EJBs), you must create an Enterprise Application Project, then add to it both projects i.e., (EJB (Jar) and Dynamic Web Project (WAR))
Stateless Session Beans: Lifecycle

- Container decides it needs more instances in the pool to service clients.
- Bean Instance does not exist.
- New instance created.
- Dependecy Injection, if any.
- PostConstruct Callback, if any.
- Pool of equivalent, method-ready bean instances.
- Any client calls a business method on any bean's business interface.
- Bean Instance does not exist.
- PreDestroy Callback, if any.

Stateful Session Beans: Lifecycle

- Bean Instance does not exist.
- New instance created.
- Dependency Injection, if any.
- PostConstruct Callback, if any.
- Pool of equivalent, method-ready bean instances.
- Any client calls a business method on any bean's business interface.
- Bean Instance does not exist.
- PreDestroy Callback, if any.

Stateful Session Beans: Definition

**POJOs**

- Instance of the Bean relates to a specific client (in memory while he/she is connected).
- Expires in case of inactivity (similar to session in Servlet/Jsp).
- Ordinary Java classes; no special interfaces or parent classes.
- E.g. e-commerce applications with shopping cart.

**Local or remote access**

- Can be accessed either on local app server or remote app server.

**Annotations**

- `@Stateful` : declaring a Stateful bean
- `@Remove` : defines the methods that ends the session
  - The Session expires when the method annotated with `@Remove` is executed.

Stateful Session Beans: Coding

For the interfaces and Client coding=> same as Stateless beans

```java
@Stateful
public class CartBean implements CartItf {
  private List items = new ArrayList();
  private List quantities = new ArrayList();
  public void addItem( int ref, int qte ) { ... }
  public void removeItem( int ref ) { ... }
  @Remove
  public void confirmOrder() { ... }
}
```
**Callback Methods**

Management of Bean’s lifecycle

A Callback method is:
- Decorated with an annotation
- Of type void, and without arguments

Example:

```java
@Stateful public class ShoppingCartBean implements ShoppingCart {
    private float total;
    private Vector productCodes;
    public int someShoppingMethod(){...};
    ...
    @PreDestroy void endShoppingCart() {...};
}
```

**Callback Annotations**

@PostConstruct
```java
public void initialise() { ... at Bean’s initialization ... }
```

@PreDestroy
```java
public void dtruir() { ... destruction of Bean ... }
```

@PrePassivate //only for Stateful beans
```java
public void avantSwap() { ... to do before Bean is swapped ... }
```

@PostActivate //only for Stateful beans
```java
public void apresSwap() { ... to do after Bean is activated ... }
```

**Session Beans...**

Compilation : javac
    Add javaee.jar in the classpath

Packaging : in a .jar file
    /test/HelloWorld.class
    /test/HelloWorldBean.class
    /test/Test.class
    /test/TestBean.class

Deployment : in the server’s deploy folder

**Advanced Concepts: Interceptors**

Executing a behavior before/after bean’s methods
- AOP inspiration (cf. AspectJ, Spring AOP, ...)
- @Interceptors : methods to be intercepted
- @AroundInvoke : interception methods

Syntax

```java
Object <methodname>( InvocationContext ctx ) throws Exception javax.interceptor.InvocationContext
```

- Allow obtaining information over the intercepted methods
- Provide a proceed() method to pursue the execution of the intercepted method
Advanced Concepts: Interceptors

```java
public class EnchereBean {
    @Interceptors(MyInterceptor.class)
    public void ajouterEnchere(Bid bid) {... }
}
```

```java
public class MyInterceptor {
    @AroundInvoke
    public Object trace(InvocationContext ic) throws Exception {
        // ... code before...
        java.lang.reflect.Method m = ic.getMethod();
        Object bean = ic.getTarget();
        Object[] params = ic.getParameters();
        // eventually modification of the parameters with ic.setParameters(...)
        Object ret = ic.proceed(); // calling the bean (optional)
        // ... code after ...
        return ret; }
}
```

Session Beans: Summary

**Stateless session beans**
- Interface: mark with `@Remote`
- Class: mark with `@Stateless(mappedName="blah")`

**Stateful session beans**
- Mark class with `@Stateful` instead of `@Stateless`
- Mark a method with `@Remove`

**Session bean clients**
- `InitialContext context = new InitialContext();`
- `InterfaceType var = (InterfaceType)context.lookup("blah");`
- `var.someMethod(args);`
- For stateful beans, call specially marked method when done
- Need jndi.properties specific to server type

**Local access to beans**
- `@EJB private InterfaceType var;`

Entity Beans

**Entity Bean** = A tuple in a database (RDB)

- **Entity Bean**: POJO
  - (Plain Old Java Object)
  - POJO property = a column in a table

Entity Bean API: JPA (Java Persistence API)

Inspired from Hibernate, TopLink…

Managing persistency in a transparent way

Advantage: using objects instead of SQL requests
Entity Beans

annotation `@Entity` => a class corresponding to the entity bean (EB)

each EB class => table
  • Default: the name of the class=name of the table
  • Except if annotation `@Table(name="...")`

Entity Beans

2 (exclusive) modes for the definition of table’s columns
  ✓ property-based access: annotate getter methods
  ✓ field-based access: annotate attributes

Default: column name=field/property class
  Except if annotation `@Column(name="...")`

annotation `@Id`: defines a primary key

Entity Beans: Example

```java
@Entity
@Table(name = "FILMS")
public class Film implements java.io.Serializable {
    private int id;
    private String name;

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    public int getId() { return id; }
    public void setId(int id) { this.id = id; }

    public String getName() { return name; }
    public void setName(final String name) { this.name = name; }
}
```

Entity Beans

`@Basic` or nothing: indicates that a field is persistent

All fields are persistent
  Except if annotated `@Transient`

Primary Key is mandatory: primitive type or composed
  `@GeneratedValue(strategy=?)`: indicates how IDs are generated
  `Auto, Identity, Sequence, etc.`
### Entity Beans: Composed Primary Key

```java
@Entity public class Etudiant {
  private String nomId;
  @Id
  public String getNomId() { return nomId; }
  public void setNomId(String nomId) { this.nomId = nomId; }
  private String prenomId;
  @Id
  public String getPrenomId() { return prenomId; }
  public void setPrenom(String prenomId) { this.prenomId = prenomId; }
}
```

### Entity Beans: Two classes in one table

**@Embeddable & @Embedded**: fields of two classes into one table

```java
@Entity public class User {
  private String nom;
  @Embedded
  private Address adresse;
}
```

### Entity Beans: Associations

Entity Beans are linked with each other through Associations

**Association Multiplicities**

- 1 - 1 (one-to-one)
- 1 - n (one-to-many)
- n - 1 (many-to-one)
- n - n (many-to-many)

**Navigability of Associations (Direction)**

- Bi-directional: two sides: an owner side and an inverse side
- Unidirectional: one side: the owner

### Unidirectional: OneToOne

```java
@Entity public class Employe {
  private ProfilVoyage pv;
  @OneToOne
  public ProfilVoyage getPv() { return pv; }
  public void setPv(ProfilVoyage profil) { this.pv = profil; }
}
```
Unidirectional : ManyToOne

@Entity
public class Employe {
    private Adresse ad;
    @ManyToOne
    public Adresse getAd() { return ad; }
    public void setAd(Adresse a) { this.ad = a; }
    ...
}

Employe entity ➔ Employe table

Adresse entity ➔ Adresse table with Id_ad as PK

Employe table owns a foreign key to Adresse, ad

2009-2010

Unidirectional : ManyToMany

@Entity
public class Employe {
    private Collection<Adresse> adresses;
    @ManyToMany
    public Collection<Adresse> getAdresses() { return adresses; }
    public void setAdresses(Collection<Adresse> adresses) { this.adresses = adresses; }
    ...
}

@Entity
public class Adresse {
    ...
}

Employe entity ➔ Employe table

Adresse entity ➔ Adresse table with Id_ad as PK

Creation of a join table Employe_Adresse with two columns (i.e. Employe_PkLoc & Adresse_PKAdresse, each column represents a PK to each table

2009-2010

Unidirectional : OneToMany

@Entity
public class Employe {
    private Collection<Adresse> adresses;
    @OneToMany
    public Collection<Adresse> getAdresses() { return adresses; }
    public void setAdresses(Collection<Adresse> adresses) { this.adresses = adresses; }
    ...
}

@Entity
public class Adresse {
    ...
}

Employe entity ➔ Employe table

Adresse entity ➔ Adresse table with Id_ad as PK

Creation of a join table Employe_Adresse with two columns (i.e. Employe_PkLoc & Adresse_PKAdresse, each column represents a PK to each table

2009-2010

Bidirectional : OneToOne/OneToOne

@Entity
public class Employe {
    private Casier monCasier;
    @OneToOne
    public Casier getMonCasier() { return monCasier; }
    public void setMonCasier(Casier c) { this.monCasier = c; }
    ...
}

@Entity
public class Casier {
    private Employe monEmploye;
    @OneToOne(mappedBy="monCasier")
    public Employe getMonEmploye() { return monEmploye; }
    public void setMonEmploye(Employe e) { this.monEmploye = e; }
    ...
}

Employe entity ➔ Employe table

Casier entity ➔ Casier table with Id_ad as PK

Employe table owns a foreign key to Casier, monCasier

2009-2010
Bidirectional : ManyToOne/OneToMany

@Entity
public class Employe {
    private Casier monCasier;
    @ManyToOne
    public Casier getMonCasier()  { return monCasier; } public void setMoncasier(Casier c)  { this.monCasier = c; } ... }

@Entity
public class Casier {
    private Collection<Employe> mesEmployes;
    @OneToMany(
        mappedBy="monCasier"
    )
    public Collection<Employe> getMesEmployes()  { return mesEmployes; } public void setMesEmployes  (Collection<Employe> e) { this.mesEmployes = e; } ... }

Employe entity ➔ Employe table

Casier entity ➔ Casier table with Id_ad as PK

Employe table owns a foreign key to Casier . monCasier

---

Bidirectional : ManyToMany/ManyToMany

@Entity
public class Projet {
    Collection<Employe> mesEmployes;
    @ManyToMany
    public Collection<Employe> getMesEmployes()  { return mesEmployes; } public void setMesEmployes  (Collection<Employe> e) { this.mesEmployes = e; } ... }

@Entity
public class Employe {
    private Collection<Projet> mesProjets;
    @ManyToMany(
        mappedBy= "mesEmployes"
    )
    public Collection<Projet> getMesProjets()  { return mesProjets; } public void setMesProjets  (Collection<Projet> p) { this.mesProjets = p; } ... }

Projet entity ➔ Projet table

Employe entity ➔ Employe table

Creation of a join table Projet_Employe with two columns (i.e. mesProjets_PKProjet & mesEmployes_Pkemploye, each column represents a PK to each table

---

Entity Beans: Inheritance

Entities support inheritance and polymorphism

Entities may be concrete or abstract

An Entity can inherit a non-entity class

A non-entity class can inherit an entity class

---

Inheriting abstract class

@Entity
public abstract class Personne{
    @Id
    protected String numSecuSociale;
}

@Entity
public class Employe extends Personne{
    protected float salaire;
}
Entity Beans: Inheritance Strategies

One Table by classes hierarchy (Default)

@Inheritance(strategy=SINGLE_TABLE)

One Table by concrete class

@Inheritance(strategy=TABLE_PER_CLASS)

Join Strategy: a join between the concrete class and the super class tables

• No duplication of the fields, a Join operation to get the info

@Inheritance(strategy=JOINED)

Inheritance Strategies : One table

A discriminator column is used

Possible Types

- DiscriminatorType.STRING (Default)
- DiscriminatorType.CHAR
- DiscriminatorType.INTEGER

Example

@Entity
@DiscriminatorColumn(name="DISCRIMINATEUR_PERSONNE"
 discriminatorType=DiscriminatorType.STRING)
@DiscriminatorValue("Personne")
public class Personne{
 ...
}

@Entity
@DiscriminatorValue("Player")
public class Player extends Personne {
 ...
}

Inheritance : MappedSuperClass

Entities can inherit non-entity beans

MappedSuperClasses are not accessible to the Entity Manager

Not considered as an Entity (no table in the DB)

@MappedSuperclass
public class BaseEntity{
   public String baseattribute1;
   public String baseattribute2;
}

@Entity
public class Entity extends BaseEntity{
   @Id
   protected int id;
   protected float attribute;
   }

2009-2010
Entity Bean : Complements

**Fetch** : option for loading the graph of objects
- FetchType.EAGER : loads all the tree (required if Serializable)
- FetchType.LAZY : only on demand (unusable with Serializable)

**Cascade** : transitivity of operations over the beans
- CascadeType.ALL : every operation is propagated
- CascadeType.MERGE : in case of a merge
- CascadeType.PERSIST : Film becomes persistent ⇒ List<SalleProg> too
- CascadeType_REFRESH : loading from the DB
- CascadeType.REMOVE : delete in cascade

Entity Beans: Entity Manager

**Utilization of Entity Beans inside Session Beans**

```java
@Stateless
public class MyBean implements MyBeanItf {
    @PersistenceContext
    private EntityManager em;
    public void init() {
        Book b1 = new Book("Honore de Balzac","Le Pere Goriot");
        Book b2 = new Book("Honore de Balzac","Les Chouans");
        Book b3 = new Book("Victor Hugo","Les Miserables");
        em.persist(b1);
        em.persist(b2);
        em.persist(b3);
    }
}
```

Research by id

```java
Book myBook = em.find(Book.class,12);
```

- Returns null if the key does not exist in the table
- IllegalArgumentException
  - If first parameter is not a EB class
  - If second parameter is not of the same type as the Id’s type

Entity Beans: Entity Manager

Managing Entities: **EntityManager**

Ensure synchronization between Java objects and DB tables

In charge of adding/updating/deleting records

**Accessible through dependency injection**

- type of the attribute javax.persistence.EntityManager
- annotated by @PersistenceContext

Entity Beans: Entity Manager

Managing Entities: **EntityManager**

Ensure synchronization between Java objects and DB tables

In charge of adding/updating/deleting records

**Accessible through dependency injection**

- type of the attribute javax.persistence.EntityManager
- annotated by @PersistenceContext
Entity Beans: Lifecycle

- **new**: The entity instance was created in memory, but is not yet associated with either a persistent identity in the database or a persistence context. This is the state that our Account entity was in right after creation. Changes in the entity state are not synchronized with the database at this stage.
- **managed**: The entity has a persistent identity in the database and is currently associated with a persistence context. Our Account entity was in the managed state after the `persist()` method was called. Changes to the entity will be synchronized with the database when transactions are committed or when synchronization is explicitly triggered using the `flush()` operation.
- **detached**: The entity does have a persistent identity but is not or is no longer associated with the persistence context.
- **removed**: The entity is currently associated with a persistence context but has been scheduled for removal from the database.

### Attaching & detaching beans

- **void merge(Object entity)**: Merge the state of the given entity into the current persistent context
  - Used with a detached bean
  - E.g. the Bean is modified at the client and sent back to the server

- **void persist(Object entity)**: persist and attach the bean
  - Usable over a new Bean (after a new)

```java
Film createFilm(String name) {
    Film film = new Film();
    res.setName(name);
    em.persist(film); // attach the bean + makes it persistent
    return film; // the copy is detached
}
```

### Principal methods of Entity Manager

- **Object find(Class cl, Object key)**: Find an EntityBean by its id
- **boolean contains(Object entity)**: True if the entity is attached to the EntityManager
- **Query createQuery(String qlString)**: Creating a query in EJB-QL
- **Query createNamedQuery(String name)**: Creating a named query
- **Query createNativeQuery(String sqlQuery)**: Creating an SQL query

- **void remove(Object entity)**: Remove the entity from the base
- **void refresh(Object entity)**: Recharging the bean from the DB
Entity Beans…

Examples

Finding a film by its id
public void findFilm(int id) {
    return em.find(Film.class, Integer.valueOf(id));
}

Removing a film:
public void removeFilm(int id) {
    em.remove(findFilm(id));
}

Entity Beans : the Query Language

« EJB-Query Language » : Close to SQL
- Selection from the Bean’s name
- Parameters indicated by : pram-name (fname in the example)
- Request in the Query object
- Result from Query

Example:
public Film findFilmByName(String name) {
    Query q = em.createQuery("select f from Film where f.name = :fname"ffname");
    q.setParameter("fname", name);
    List<Film> res = q.getResultList();
    return res.size() == 0 ? null : res.get(0);
}

getSingleResult() in case of a unique result
• NonUniqueResultException in case of no unique result

Entity Beans : Named Queries

A named query attached to the EB

@NamedQuery(name="allbooks",query="select OBJECT(b) from Book b")
public class Book { ...
Query q = em.createNamedQuery("allbooks");
List<Book> list = (List<Book>) q resultList();

Entity Beans : Named Queries

Multiple Queries

@NamedQueries({
    @NamedQuery(name = "findAllFilms", query = "select f from Film f")
    @NamedQuery(name = "findFilmByName", query = "select f from Film f WHERE f.name = :fname")
})
public class Film implements java.io.Serializable { …

Used in a SessionBean :
void findFilmByName() {
    Query q = em.createNamedQuery("findFilmByName"); …
Entity Beans : Native Queries

**Query** `createNativeQuery(String sqlString)`

- Create an instance of Query for executing a native SQL statement, e.g., for update or delete.

**Parameters:**
- `sqlString` - a native SQL query string

**Returns:** the new query instance

**Throws:**
- `IllegalStateException` - if this EntityManager has been closed.

Entity Beans : Lifecycle- Interceptors

Interception of state changes

- Around the creation (em.persist) :
  - `@PrePersist`
  - `@PostPersist`

- At loading time from DB (em.find, Query.getResultList)
  - `@PostLoad`

- Around updates (modification of a field, em.merge)
  - `@PreUpdate`
  - `@PostUpdate`

- Around a remove action (em.remove)
  - `@PreRemove`
  - `@PostRemove`

Deployment Descriptor: Persistence.xml

Specifies advanced mapping concepts, the data source (here the JBoss default DB)

In the **META-INF** folder in the EJB Project

```xml
<persistence>
  <persistence-unit name="IntroEJB3">
    <jta-data-source>
      java:/DefaultDS
    </jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="update"/>
    </properties>
  </persistence-unit>
</persistence>
```

Message-Driven Beans
Message-Driven Beans

Message-Driven Bean: interaction by messages
MOM: Message Oriented Middleware

Two modes of communication
✓ N vers 1: Queue
✓ N vers M: Topic

Message Driven Bean: JMS-based specification
JMS: Java Message Service

Principles of Message Driven Beans
✓ Consume asynchronous messages
✓ Stateless (all instances of the same MDB are equivalent)
✓ Handle client messages
✓ 1 business method (onMessage)
  • Fixed Parameters
  • No return value
  • No exception

When to use a MDB
✓ Avoid Blocking calls
✓ When you have clients (producers) and servers (consumers)

Message-Driven Beans: Architecture

Architecture of Java Message Service
(for queue, replace Topic by Queue, Publisher by Sender, Subscriber by Receiver)

JMS (java.sun.com/jms)
Queue: Thread of discussion (one consumer)
Topic: Topic of discussion (diffusion)

ConnectionFactory: Factory of connections towards queue/topic
Connection: connection towards queue/topic

Session:
✓ Creation of a sender and of a receiver
✓ Can be transactional
Example of Message-Driven Bean

```java
@MessageDriven(activationConfig = {
    @ActivationConfigProperty(propertyName = "destination",
        propertyValue = "topic_rigolo"),
    @ActivationConfigProperty(propertyName = "destinationType",
        propertyValue = "javax.jms.Topic")
})
public class Mdb implements MessageListener {
    public void onMessage(Message inMessage) {
        System.out.println(((TextMessage)msg).getText());
    }
}
```

Example of a sender

```java
@Resource(name="rigolo", mappedName="topic_rigolo")
Topic topic;
@Resource(name="factory", mappedName="JTCF")
TopicConnectionFactory factory;
TopicSession session;
TopicPublisher sender;

public void publish(String value) {
    TopicConnection tc = factory.createTopicConnection();
    session = tc.createTopicSession(false,
        Session.AUTO_ACKNOWLEDGE);
    sender = session.createPublisher(topic);
    TextMessage msg = session.createTextMessage();
    msg.setText("MDB: " + value);
    sender.publish(msg);
}
```

EJB3: Conclusion

Les EJBs: une technologie qui a encore toute sa place dans les besoins de performance, de distribution, de contexte transactionnel des applications d’entreprise aujourd’hui

La version 3.x a sauvé cette technologie, face à la concurrence féroce livrée par Spring

Une demande toujours présente de la part de l’industrie de profils JEE/EJB3

Readings/Sources

- The JEE specification: https://java.sun.com/
- Lionel Seinturier Courses (slides in French at http://www2.lifl.fr/~seinturi/)
- Marie-Pierre Gervais courses (slides in French not provided online)