Enterprise Java Beans 3

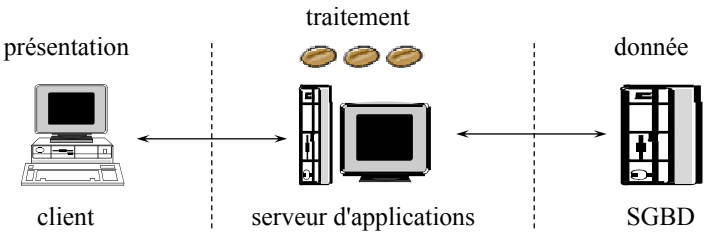
Plan

1. Composant EJB

Enterprise Java Bean (EJB)

Modèle de composants pour le développement d'applications d'entreprises

- sites web (de commerce en ligne, de travail collaboratif, communautaires, …)
- de systèmes informations
- d'applications de gestion
- …



1. Composant EJB

Enterprise Java Bean (EJB)

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

- on se focalise sur la logique applicative
- les services systèmes sont fournis par le conteneur
- la logique de présentation est du ressort du client

Vocabulaire dans ce cours : *bean* = EJB = composant

Types d'EJB

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

Plusieurs versions : actuellement EJB 3 (depuis 2006)

1. Composant EJB

EJB 3

- succès Java EE en général
- mais
  - trop compliqué, lourd, contraignant
  - concepts objets (héritage, typage, polymorphisme, …) difficilement exploitables
  - mapping objet/relationel limité
  - trop de codage XML pénible

⇨ passage EJB 2 vers EJB 3

  - utilisation des annotations et de la généricité Java 5
  - pour simplifier l'écriture des beans

1. Composant EJB

EJB 3

Passage EJB 2 vers EJB 3

⇨ étude réalise par Oracle sur l'application AventureBuilder

Table 1: Summary of Findings

Application Name	Item Measured	J2EE 1.4 Platform	Java EE 5 Platform	Improvement
AdventureBuilder	Number of classes	67	43	36% fewer classes
	Lines of code	3,284	2,777	15% fewer lines of code
RosterApp	Number of classes	17	7	59% fewer classes
	Lines of code	987	716	27% fewer lines of code
	Number of XML files	9	2	78% fewer XML files
	Lines of XML code	792	26	97% fewer lines of XML code

`http://java.sun.com/developer/technicalArticles/J2EE/intro_ee5/`

1. Composant EJB

Annotations

- mécanisme standard dans le langage Java depuis version 5 (1.5)
- idée similaire aux commentaires Javadoc
  - informations attachées à des éléments de programme (classe, méthode, attributs, …)
  - pour ajouter de l'information sur cet élément
  - ex : déclarer méthode distante, déclarer attribut persistant, …
- `@Identificateur`
- éventuellement des paramètres : `@Identificateur(name=value,...)`
- éventuellement plusieurs annotations par éléments

Exemple utilisation annotations :

```
@Resource(name="myDB", type=javax.sql.DataSource.class)
@Stateful
public class ShoppingCartBean implements ShoppingCart {
  ...
}
```


1. Composant EJB

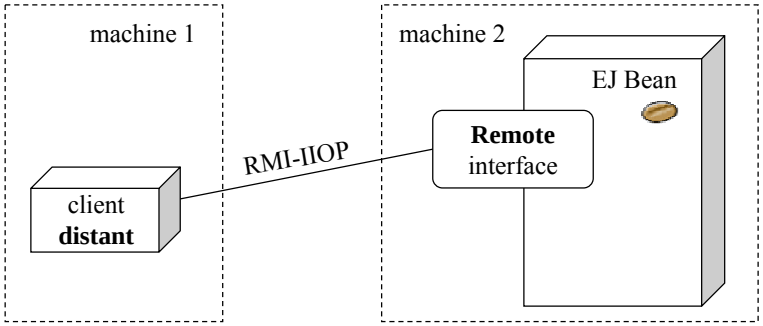
Annotations

- chaque annotation est un type (au même titre qu'une classe ou qu'une interface)
- défini dans un package (ex. : `javax.ejb.Stateless`)
- les annotations existantes peuvent être utilisées
- de nouvelles annotations peuvent être définies par le programmeur
  - nouveau mot clé Java 5 : `@interface`

Pour + d'informations, voir par ex. :
`http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html`

1. Composant EJB

Enterprise Java Bean

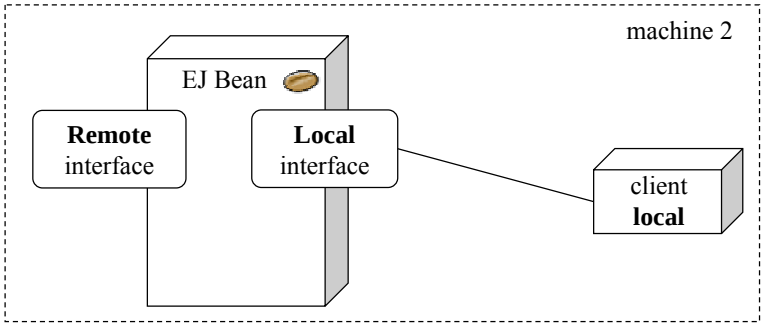


Chaque EJ Bean fournit 1 interface d'accès **distant**

• les services (méthodes) offerts par le bean à ces client

1. Composant EJB

Enterprise Java Bean



+ éventuellement 1 interface d'accès **local** (à partir EJB 2.0)

• les services offerts par le bean à ses clients locaux
• les mêmes (ou d'autres) que ceux offerts à distance
⇨ optimisation

1.1 Session Bean

1. Définition
2. Développement
3. Client local
4. Client distant
5. Stateful session bean

1.1 Session Bean

Définition

Session Bean : représente un traitement (services fournis à un client)

1. Stateless session bean

- sans état
- ne conserve pas d'information entre 2 appels successifs
- 2 instances qqconques d'un tel *bean* sont équivalentes
- 1 instance par invocation

2. Stateful session bean

- avec un état (en mémoire)
- aka servlet/JSP session
- même instance pendant toute la durée d'une session avec un client
- 1 instance par client

1.1 Session Bean

Développement

1 interface (éventuellement 2 : Local + Remote) + 1 classe

Interface

- annotations `@javax.ejb.Local` ou `@javax.ejb.Remote`

```
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);
}
```


1.1 Session Bean

Développement

Classe

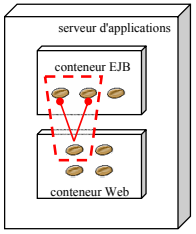
- annotation `@javax.ejb.Stateless` ou `@javax.ejb.stateful`

```
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;}
}
```

- possibilité de nommer les *beans* : `@Stateless(name="foobar")`

1.1 Session Bean

Client local



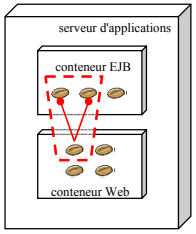
- typiquement une servlet ou une JSP colocalisée sur le même serveur que le *bean*

- mécanisme dit "injection de dépendance"
  1. la classe cliente déclare un attribut
     - du type de l'interface
     - annoté `@EJB`
     - éventuellement `@EJB(name="foobar")`
  2. le conteneur affecte (injecte) la référence du *bean* dans l'attribut

- appel de méthode en utilisant l'attribut

1.1 Session Bean

Client local




```
public class ClientServlet extends HttpServlet {
  @EJB
  private CalculatriceItf myBean;

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

1.1 Session Bean

Client distant

Déclaration d'une interface `@javax.ejb.Remote`
Utilisation de `@javax.ejb.Remote` pour la classe du *bean*

```
@Remote(CalculatriceItf.class)
public class CalculatriceBean implements CalculatriceItf { ... }
```

1. Récupération de la référence vers l'annuaire JNDI
2. Recherche du *bean* dans l'annuaire
3. Appel des méthodes du bean

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


1.1 Session Bean

Stateful Session Bean

- instance du *bean* reste en mémoire tant que le client est présent
- expiration au bout d'un délai d'inactivité
- similaire session JSP/servlet
- utilisation type
  - gestion d'un panier électronique sur un site de commerce en ligne
  - rapport sur l'activité d'un client

2 annotations principales

- `@Stateful` : déclare un *bean* avec état
- `@Remove`
  - définit la méthode de fin de session
  - la session expire à l'issu de l'exécution de cette méthode

1.1 Session Bean

Stateful Session Bean

```
@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() { ... }
}
```


1.2 Entity Bean

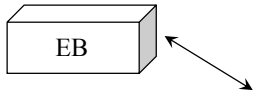
1. Définition
2. Développement
3. Utilisation
4. Relation
5. Clé
6. Règles de *mapping* EB - table
7. Autres annotations

1.2 Entity Bean

Définition

Représentation d'une donnée manipulée par l'application

- donnée typiquement stockée dans un SGBD (ou tout autre support accessible en JDBC)



Nom	Solde
John	100.00
Anne	156.00
Marcel	55.25

- correspondance objet – tuple relationnel (*mapping O/R*)
- possibilité de définir des clés, des relations, des recherches
- avantage : manipulation d'objets Java pluôt que de requêtes SQL

- mis en oeuvre à l'aide
  - d'annotations Java 5
  - de la généricité Java 5
  - de l'API JPA (Java Persistence API)

1.2 Entity Bean

Développement

4 annotations principales

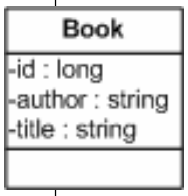
- annotations de classe
  - @Entity : déclare une classe correspondant à un *entity bean* (EB)
  - @NameQuery : associe une requêtre SQL à cet EB

- annotations de méthode *getter* ou d'attribut
  - @Id : définit une clé primaire
  - @GeneratedValue : permet de définir un attribut dont la valeur est générée par le conteneur (ex. : entier auto-incrémenté)

1.2 Entity Bean

Développement

```
@Entity
@NamedQuery(name = "allBooks", query = "select * FROM Book o")
public class Book {

    private long id;
    private String author;
    private String title;

    public Book() {}
    public Book(String author, String title) {
        this.author = author;
        this.title = title; }

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



1.2 Entity Bean

Développement

```
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}
```

1.2 Entity Bean

Utilisation

Notion Entity Manager

- le gestionnaire des entités
- assure le passage entre les objets et les tables relationnelles du SGBD

- permet d'ajouter des enregistrements (INSERT)
- permet d'effectuer des requêtes (SELECT)
  - utilisation de `java.util.List` et de la généricité Java 5 (`List<Book>`)

- attribut de type `javax.persistence.EntityManager`
- annoté par `@PersistenceContext`

1.2 Entity Bean

Utilisation

```
@Stateless
public class MyBean implements MyBeanItf {

    @PersistenceContext
    private EntityManager entityManager = null;

    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");

        entityManager.persist(b1);
        entityManager.persist(b2);
        entityManager.persist(b3);
    }

    public List<Book> listOfBooks() {
        return entityManager.createNamedQuery("allBooks").getResultList();
    }
}
```


1.2 Entity Bean

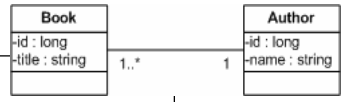
Relation

4 annotations correspondant aux cardinalités envisagées entre entités

- @OneToOne
- @OneToMany
- @ManyToOne
- @ManyToMany

1.2 Entity Bean

Relation




```
@Entity
@NamedQuery(name = "allAuthors",
            query = "select * FROM Author o")
public class Author {

    private long id;
    private String name;
    private Collection<Book> books;

    public Author() { books = new ArrayList<Book>(); }
    public Book(String name) { this.name = name; }

    @OneToMany
    public Collection<Book> getBooks() { return books; }

    public void addBook( String title ) {
      Book b = new Book(this,title);
      getBooks().add(b); }

    public void setBooks( Collection<Book> books ) {
      this.books = books;
    }
```

1.2 Entity Bean

Relation

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

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


1.2 Entity Bean

Relation

```
@Entity
@NamedQuery(name = "allBooks", query = "select * FROM Book o")
public class Book {

    private long id;
    private Author author;
    private String title;

    public Book() {}
    public Book(Author author, String title) {
        this.author = author;
        this.title = title; }

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


1.2 Entity Bean

Relation

```
    @ManyToOne
    @JoinColumn(name="Author_id")
    public String getAuthor() { return author; }
    public void setAuthor(String author) { this.author = author; }

    public String getTitle() { return title; }
    public void setTitle(String title) { this.title = title; }
}
```


1.2 Entity Bean

Clé

Chaque instance de *bean* peut être identifiée par une clé composée

⇨ classe fournie par le développeur du *bean*
- implantant l'interface `java.io.Serializable`
- définissant les méthodes `hashCode` et `equals`

```
public class PersonnePK implements java.io.Serializable {

  public String name; public String firstname;

  public PersonnePK(String n,String p) { name=n; firstname=p; }
  public PersonnePK() {}

  public int hashCode() {
    return name.hashCode() + firstname.hashCode();
  }
  public boolean equals(Object other) {
    return /** name==other.name && firstname==other.firstname */;
} }
```

1.2 Entity Bean

Clé

Utilisation : annotation @IdClass

```
@Entity
@IdClass(PersonnePK.class)
public class Personne {

    private String name;
    private String firstname;

    public Personne() {}
    public Book(String name, String firstname) {
       this.name=name; this.firstname=firstname; }

    @Id
    public String getName() { return name; }
    public void setName(String name) { this.name = name; }

    @Id
    public String getFirstname() { return firstname; }
    public void setFirstname(String fname) { this.firstname = fname; }
}
```


1.2 Entity Bean

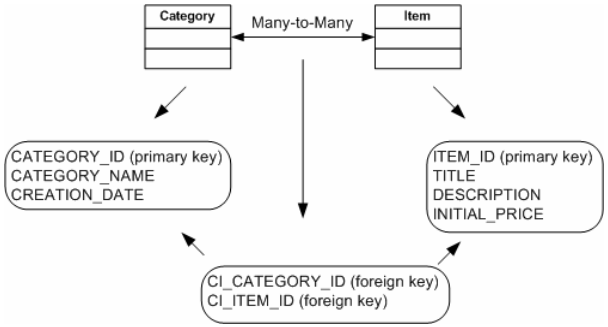
Règles de *mapping* EB - table

- chaque classe de EB est mis en correspondance avec une table
  - par défaut table avec même nom que la classe
  - sauf si annotation classe @Table(name="…")
- 2 modes de définition des colonnes des tables
  - *property-based access* : on annote les méthodes *getter*
  - *field-based access* : on annote les attributs
  - exclusif (choisir)
  - 1 colonne par *field/property*
  - par défaut colonne avec même nom que *field/property*
  - sauf si annotation @Column(name="…")

1.2 Entity Bean

Règles de *mapping* EB - table

- colonne de jointure
  - @JoinColumn(name="…", referencedColumnName="…")
- table de jointure
  - @JoinTable



1.2 Entity Bean

Règles de *mapping* EB - table

- table de jointure
  - @JoinTable

```
@Entity
@Table(name="CATEGORIES")
class Category {
 @Id
 @Column(name="CATEGORY_ID")
 protected long categoryId;

 @ManyToMany
 @JoinTable(name="CATEGORIES_ITEMS",
  joinColumns=
   @JoinColumn(name="CI_CATEGORY_ID", referencedColumnName="CATEGORY_ID"),
  inverseJoinColumns=
   @JoinColumn(name="CI_ITEM_ID", referencedColumnName="ITEM_ID"))
 protected Set<Item> items;

 ...
}
```

```
@Entity
@Table(name="ITEMS")
public class Item {
 @Id
 @Column(name="ITEM_ID")
 protected long itemId;

 @ManyToMany(mappedBy="items")
 protected Set<Category> categories

 ...
}
```

1.2 Entity Bean

Autres annotations

@Enumerated : définit une colonne avec des valeurs énumérées
EnumType : ORDINAL (valeur stockée sous forme int), STRING

```
public enum UserType {STUDENT, TEACHER, SYSADMIN};
```

```
@Enumerated(value=EnumType.ORDINAL)
protected UserType userType;
```

@Lob : données binaires

```
@Lob
protected byte[] picture;
```

@Temporal : dates
TemporalType : DATE (java.sql.Date), TIME (java.sql.Time)
TIMESTAMP (java.sql.Timestamp)

```
@Temporal(TemporalType.DATE)
protected java.util.Date creationDate;
```


1.2 Entity Bean

Autres annotations

@SecondaryTable : mapping d'un EB plusieurs tables

```
@Entity
@Table(name="USERS")
@SecondaryTable(name="USER_PICTURES"
      pkJoinColumns=@PrimaryKeyJoinColumn(name="USER_ID"))
public class User { ... }
```

@Embeddable et @Embedded : embarque les données d'une classe dans une table

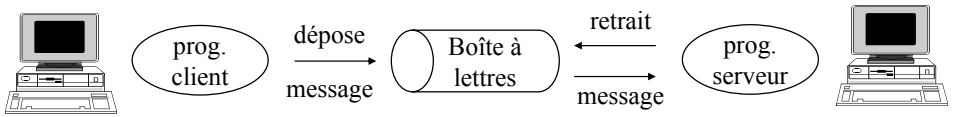
```
@Embeddable
public class Address implements Serializable {
  private String rue; private int codePostal; }
```

```
@Entity
public class User {
  private String nom;

  @Embedded
  private Address adresse;
}
```


1.3 Message-driven Bean

Message-driven bean (MDB)

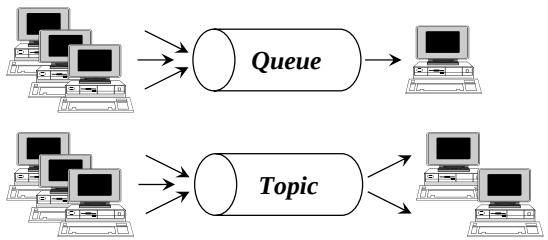
Interaction **par envoi message asynchrone** (MOM : *Message-Oriented Middleware*)



≈ CORBA COSEvent

2 modes

- n vers 1 (*queue*)
- n verrs m (*topic*)



1.3 Message-driven Bean

Caractéristiques

- consomme des messages asynchrones
- pas d'état (≡ *stateless session bean*)
- toutes les instances d'une même classe de MDB sont équivalentes
- peut traiter les messages de clients ≠

Quand utiliser un MDB

• éviter appels bloquants
• découpler clients et serveurs
• besoin de fiabilité : protection *crash* serveurs

Vocabulaire : producteur/consommateur

1.3 Message-driven Bean

Concepts

MDB basé sur les specs Sun Java Messaging Service (JMS) `java.sun.com/jms`

ConnectionFactory	fabrique pour créer des connexions vers *queue/topic*
Connection	une connexion vers *queue/topic*
Session	intervalle de temps pour l'envoi de messages dans *queue/topic*

peut être rendue transactionnelle
similitude avec les notions de sessions JDBC, Hibernate, …

Processus

1. Création d'une connexion
2. Création d'une session (* : éventuellement plusieurs sessions par conn.)
3. Création d'un message associé à un destinataire
4. Envoi du message
5. Fermeture session
6. Fermeture connexion

1.3 Message-driven Bean

Producteur

```
public class MyProducerBean {
 @Resource(name="jms/QueueConnectionFactory")  // l'id de la factory
 private ConnectionFactory connectionFactory;
 @Resource(name="jms/ShippingRequestQueue")  // l'id de la queue
 private Destination destination;
 public void produce() {
  Connection connection = connectionFactory.createConnection();
  Session session = connection.createSession(true,Session.AUTO_ACKNOWLEDGE)
  MessageProducer producer = session.createProducer(destination);
  ObjectMessage message = session.createObjectMessage();
  ShippingRequest request = new ShippingRequest(item,address,amount);
  message.setObject(request);
  producer.send(message);
  session.close();
  connection.close();
} }
```


1.3 Message-driven Bean

Consommateur

MDB = classe
- annotée @MessageDriven
- implantant interface MessageListener
  - méthode void onMessage(Message)

```
@MessageDriven(name="ShippingRequestProcessor",
  activationConfig={
    @ActivationConfigProperty(propertyName="destinationType",
      propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="destinationName",
      propertyValue="jms/ShippingRequestQueue")})
public class MyConsumerBean implements MessageListener {
 public void onMessage(Message message) {
   ObjectMessage objectMessage = (ObjectMessage) message;
   ShippingRequest request = (ShippingRequest) objectMessage.getObject();
   ...
} }
```


1.4 Fonctionnalités avancées

Timer bean

Un *bean* permettant de déclencher des actions périodiquement

- @Timeout : méthode exécutée à échéance du *timer*
  profil de méthode : void <methodname>(javax.ejb.Timer timer)
- @Resource : attribut de type javax.ejb.TimerService
- utilisation des méthodes de TimerService pour créer des *timers*
  - createTimer(long duration, java.io.Serializable info);
  - createTimer(long initialDuration, long period, java.io.Serializable info);

```
public class PlaceBidBean {
  @Resource TimerService ts;
  public void addBid(Bid bid) { ts.createTimer(1000,25000,bid); }
  @Timeout
  public void monitorBid( Timer timer ) {
    Bid bid = (Bid) timer.getInfo(); ...
} }
```

1.4 Fonctionnalités avancées

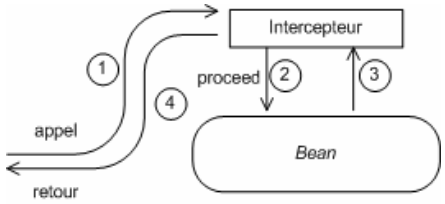
Intercepteurs

Permettent d'implanter des traitements avant/après les méthodes d'un *bean*
- influence de l'AOP (voir AspectJ, JBoss AOP, …)

- @Interceptors : les méthodes devant être interceptées
- @AroundInvoke : les méthodes d'interception
  profil de méthode
  Object <methodname>(javax.interceptor.InvocationContext ctx) throws Exception

InvocationContext

- permet d'obtenir des informations (introspecter) sur les méthodes interceptées
- fournit une méthode proceed() pour exécuter la méthode interceptée



1.4 Fonctionnalités avancées

Intercepteurs

```
public class PlaceBidBean {
  @Interceptors(MyInterceptor.class)
  public void addBid( Bid bid ) {
  ... }
}
```

Plusieurs méthodes dans des classes ≠ peuvent être associées à MyInterceptor

```
public class MyInterceptor {

  @AroundInvoke
  public Object trace( InvocationContext ic ) throws Exception {
    // ... code avant ...

    java.lang.reflect.Method m = ic.getMethod();
    Object bean = ic.getTarget();
    Object[] params = ic.getParameters();
    // éventuellement modif. paramètre avec ic.setParameters(...);

    Object ret = ic.proceed();       // Appel du bean (facultatif)

    // ... code après ...
    return ret; } }
```


Plan

2. Services

Serveur d'application

Services fournis

- cycle de vie
- transaction
- nommage
- sécurité

≠ par rapport à CORBA : services intégrés dès le départ à la plate-forme

2.1 Cycle de vie

Service de cycle de vie des *beans*

• Passivation d'instances

sauvegarde temporaire du bean
lorsque le conteneur a besoin de mémoire

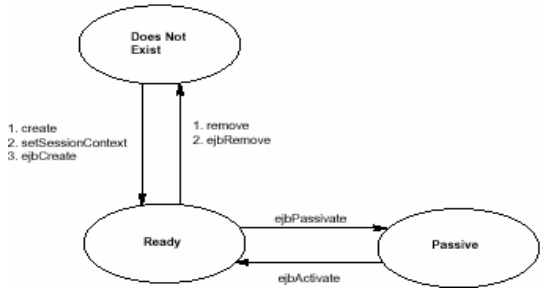
• *Pooling* d'instances

- pour des raisons de performances, le conteneur peut instancier moins de *beans* qu'il n'y a de client
  ⇨ plusieurs clients partagent un même *bean*

2.1 Cycle de vie

Cycle de vie d'un *session bean*

Stateful



Le conteneur peut décider
de "passiver" le *bean*
⇨ sauvegarde de son état
⇨ en général politique *least recently used*

Stateless

même diagramme d'états sauf **pas de** passivation

2.1 Cycle de vie

Annotation cycle de vie d'un *session bean*

Notifications de changement d'état

• méthodes annotées du bean profil : void <methodname>()
• invoquées juste avant le changement d'état
• permet au bean de fermer/ouvrir des ressources (connexions, fichiers, …) "proprement"

@PostConstruct : après la création
@PreDestroy : avant la destruction
@PrePassivate : avant la passivation
@PostActivate : après l'activation

```
public class MyBean {
  @PostConstruct
  public void initialize() {
    ... }
  ... }
```


2.1 Cycle de vie

Cycle de vie d'un *entity bean*

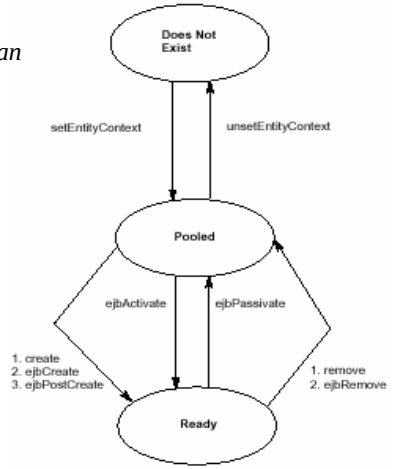
Le conteneur EJB gère un *pool* d'instances de *bean*
Après instanciation, le *bean* est mis dans le pool

⇨ il n'est pas associé à aucune donnée
mais est prêt à servir

`create` et `remove` invoquées par le client

Toutes les autres méthodes

- implantées par le *bean*
- invoquées par le conteneur



2.2 Transactions

Service de transactions

Assure des propriétés **ACID** pour des transactions plates

Exemple classique : un transfert bancaire (débit, crédit)

- atomicité soit les 2 opérations s'effectuent complètement, soit aucune
- cohérence le solde d'un compte ne doit jamais être négatif
- isolation des transferts // doivent fournir le même résultat qu'en séq.
- durabilité les soldes doivent être sauvegardés sur support stable

Support complètement intégré au serveur EJB
Véritable + / aux *middlewares* style CORBA

2.2 Transactions

Granularité des transactions

Comment démarquer (délimiter) les transactions ?

Attribut transactionnel avec 6 valeurs

- REQUIRED
- REQUIRES_NEW
- SUPPORTS
- NOT_SUPPORTED
- MANDATORY
- NEVER

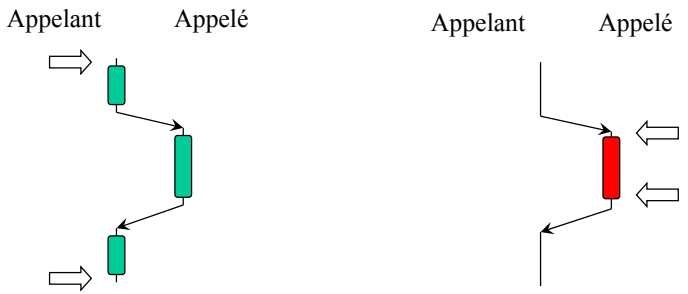
2 cas pour le *bean* appelant

- soit il s'exécute dans une transaction
- soit il s'exécute en dehors de tout contexte transactionnel

2.2 Transactions

Granularité des transactions

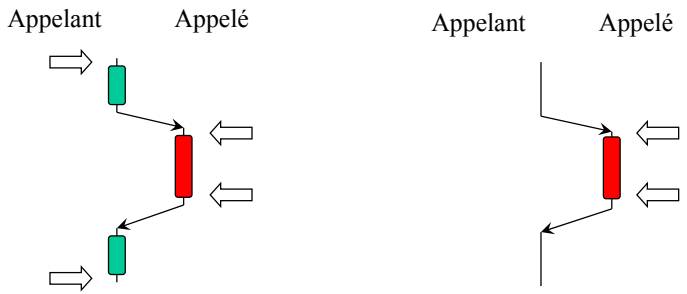
REQUIRED



2.2 Transactions

Granularité des transactions

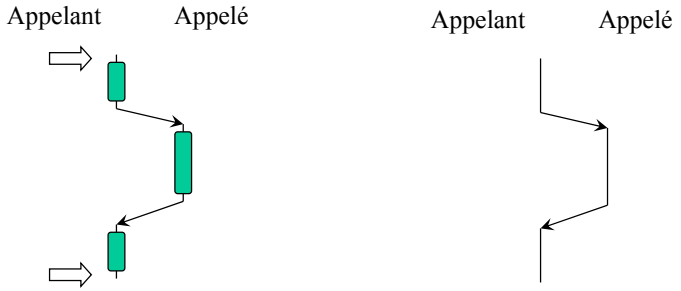
REQUIRES_NEW



2.2 Transactions

Granularité des transactions

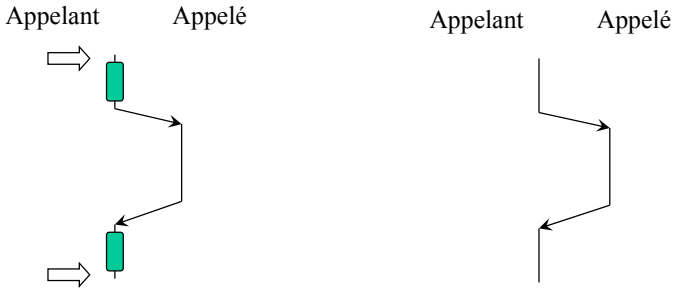
SUPPORTS



2.2 Transactions

Granularité des transactions

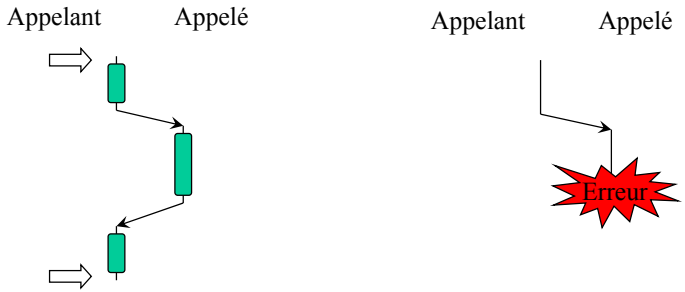
NOT_SUPPORTED



2.2 Transactions

Granularité des transactions

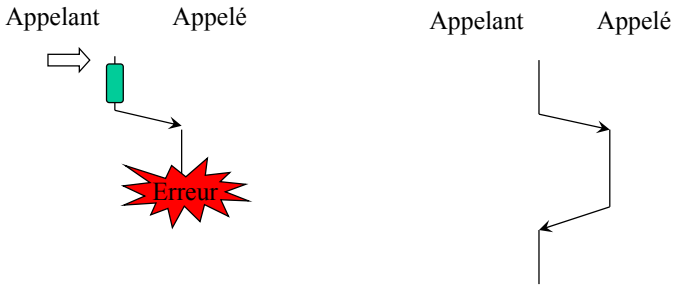
MANDATORY



2.2 Transactions

Granularité des transactions

NEVER



2.2 Transactions

Définition des transactions

2 modes

- CMT (*Container Managed Transaction*) : annotations
- BMT (*Bean Managed Transaction*) : API JTA

CMT

- @TransactionManagement sur la classe
- @TransactionAttribute(TransactionAttributeType.XXX)
  sur les méthodes transactionnelles
  où XXX est 1 des 6 attributs précédents
- toute la méthode est considérée comme un bloc transactionnel
- *commit* par défaut en fin de méthode
- appel setRollbackOnly() pour annuler

2.2 Transactions

Définitions des transactions – CMT

```
@Stateless
@TransactionManagement(TransactionManagementType.CONTAINER)
public class MyBean implements MyBeanItf {

  @TransactionAttribute(TransactionAttributeType.REQUIRED)
  public void transfert() {
    try {
      Account a1 = em.find(Account.class,"Bob");
      Account a2 = em.find(Account.class,"Anne");
      a1.credit(10.5);
      a2.widthdraw(10.5);
    }
    catch( Exception e ) {
      sc.setRollbackOnly();
  } }

  @PersistenceContext
  private EntityManager em;

  @Resource
  private SessionContext sc; }
```


2.2 Transactions

Définition des transactions

BMT

- démarcation explicite avec begin/commit/rollback
- avantage : possibilité granularité plus fine qu'en CMT

2.2 Transactions

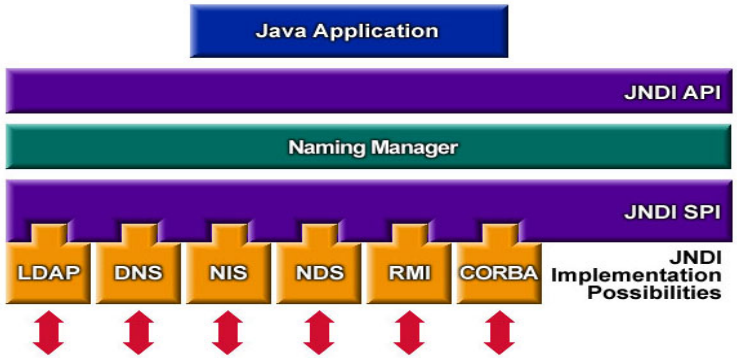
Définition des transactions

```
@Stateless
@TransactionManagement(TransactionManagementType.BEAN)
public class MyBean implements MyBeanItf {

  public void transfert() {
    try {
      ut.begin();
      Account a1 = em.find(Account.class,"Bob");
      Account a2 = em.find(Account.class,"Anne");
      a1.credit(10.5);
      a2.widthdraw(10.5);
      ut.commit();
    }
    catch( Exception e ) {
      ut.rollback();
  } }

  @PersistenceContext
  private EntityManager em;

  @Resource
  private UserTransaction ut; }
```

2.3 Nommage

JNDI (Java Naming and Directory Interface)

- 1 unique API pour accéder à ≠ serveurs de nom
- unification, simplification



2.3 Nommage

JNDI

URL JNDI *`préfixe : schéma`*

- préfixe identifiant unique par type de serveurs de noms
- schéma spécifique à chaque type de serveur de noms

Exemples

- `iiop://foo.com/x` RMI-IIOP
- `ldap://localhost:389` LDAP
- `java:comp/env`

Type `java:`

- serveur de noms en mémoire
- accessible seulement dans le contexte mémoire des *beans*
- stocke des références vers des ressources fournies par J2EE aux *beans* (ex accès à l'API JTA, réf. des interfaces locales, …)

2.3 Nommage

JNDI

Récupération d'un point d'entrée vers JNDI

```
Context ic = new InitialContext();
```

le serveur de noms RMI-IIOP local

```
Properties props = new Properties();
props.put( InitialContext.PROVIDER_URL, "iiop://host.com:1050" );
Context ic = new InitialContext(props);
```

un serveur de noms RMI-IIOP sur la machine host.com et le port 1050

```
Properties props = new Properties();
props.put( InitialContext.INITIAL_CONTEXT_FACTORY,
           "com.sun.jndi.cosnaming.CNCtxFactory" );
props.put( InitialContext.PROVIDER_URL, "..." );
Context ic = new InitialContext(props);
```


Plan

3. Déploiement

Packaging des applications

1 application EJB = 1 archive .ear

- 1 descripteur XML de l'application
- 1 archive .war par *web bean*
- 1 archive .jar par *bean*

1 archive .war

- 1 descripteur XML du *web bean*
- JSP ou servlet.class

1 archive .jar

- 1 descripteur XML du *bean*
- les .class du *bean*

3. Déploiement

Descripteur XML du bean

Fichier XML (ejb-jar.xml) conforme à une DTD définie dans les specs des EJB

⇨ informations pour configurer et déployer le bean
⇨ 46 balises
⇨ plates-formes fournissent des **assistants** pour sa création

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<ejb-jar>
  <description>Beans de gestion de compte bancaire</description>
  <enterprise-beans>          <!-- les beans de l'application -->
    ...
  </enterprise-beans>
  <assembly-descriptor>       <!-- la configuration des services -->
    ...
  </assembly-descriptor>
</ejb-jar>
```


3. Déploiement

Exemple

```
<entity>
  <description>Bean correspondant à un compte</description>
  <ejb-name>Compte</ejb-name>
  <home>mypkg.CompteRemoteHome</home>    <!-- les classes et interf. -->
  <remote>mypkg.CompteRemoteInterface</remote>
  <ejb-class>mypkg.CompteBean</ejb-class>
  <prim-key-class>java.lang.String</prim-key-class>
  <!-- Définition des attributs persistants -->
  <persistence-type>Container</persistence-type>
  <cmp-field> <field-name>nom</field-name> </cmp-field>
  <cmp-field> <field-name>solde</field-name> </cmp-field>
  <reentrant>False</reentrant>
</entity>
```


3. Déploiement

Exemple (suite)

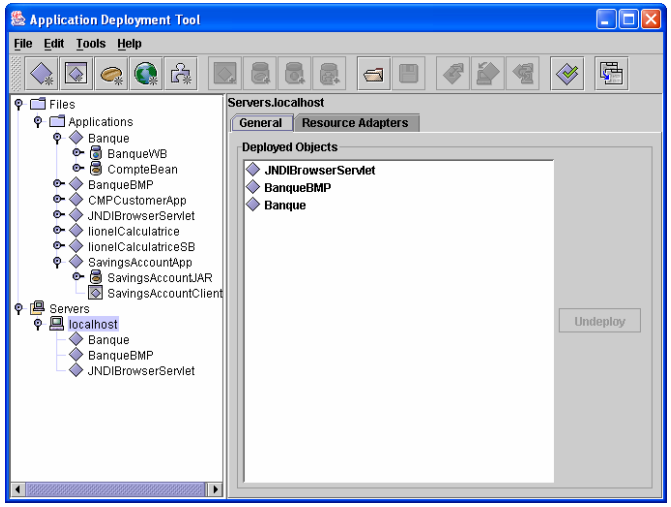
```
<assembly-descriptor>
  <container-transaction>
    <method>
      <ejb-name>Compte</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
</assembly-descriptor>
```

3. Déploiement

Outillage

Exemple : deploytool (Sun RI)

- génère les descripteurs XML
- déploie/retire les applications



4. Design Patterns EJB

Gabarit de conception (*design pattern*)

Problèmes de codage récurrents

- parcourir un arbre de données dont les noeuds sont typés
- maj une fenêtre en fonction de modifications sur des données (et vice-versa)
- …

⇨ design pattern (DP) : solutions reconnues d'organisation du code

But

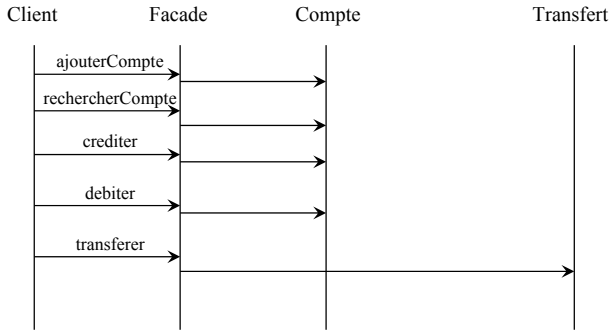
- améliorer la clarté, la compréhension du code
- mettre en avant des éléments d'architecture logicielle

4. Design Patterns EJB

DP Session facade

Pb : nombreuses dépendances entre les clients et les beans

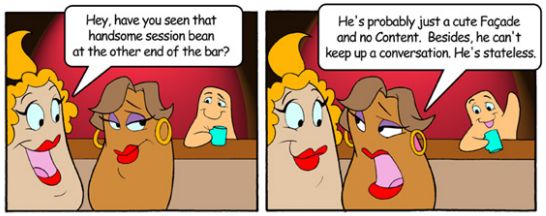
Solution : présenter aux clients **une seule interface façade** (*stateless session bean*)



4. Design Patterns EJB

DP Session facade

- traitements effectués par la façade minimaux essentiellement **délégation**
- éventuellement plusieurs façades sur un même ensemble de beans
  ⇨ différentes vues

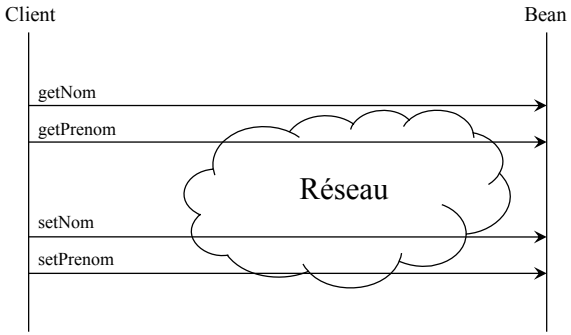


`http://www.theserverside.com/cartoons/TalesFromTheServerSide.tss`

4. Design Patterns EJB

DP Data Transfert Object

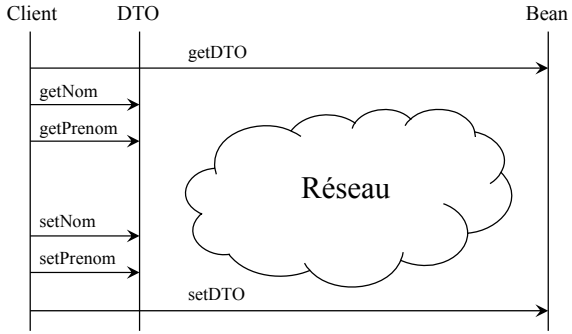
Aussi connu sous le terme : Value Object
Pb : nombreux échanges réseaux pour de simples get/set



4. Design Patterns EJB

DP Data Transfert Object

Solution : transmettre une instance par valeur



4. Design Patterns EJB

Autres DP

- application service
  centraliser un processus métier s'étendant sur +sieurs *beans*
- composite entity
  rassembler dans 1 seul EB les données persistantes de +sieurs EB
- transfert object assembler
  aggregation de +sieurs DTO
- service activator
  invoquer des services de façon asynchrone

5. Web Services

Web Services (WS)

Invocation de traitements à distance via le Web
Solution d'interopérabilité entre systèmes hétérogènes (par ex. Java EE ↔ .NET)
Repose sur HTTP / SOAP / XML / WSDL

- les méthodes fournies par les beans peuvent être exposées comme des WS

@WebService : annotation pour la classe d'un bean
@WebMethod : annotation pour une méthode d'un bean accessible via un WS
@WebResult et @WebParam : annotations pour les paramètres d'une WebMethod

- la plate-forme génére le code (WSDL, *stub*, *skeleton*) nécessaire au fonctionnement WS

5. Web Services

Exemple

```
@WebService(targetNamespace="urn:UserService")
@Stateless
public class MyBean implements MyBeanItf {

  @WebMethod
  @WebResult(name="UserId")
  public long addUser(
    @WebParam(name="UserName") String name,
    @WebParam(name="UserAge") int age )
 { ... }
}
```

Accès (ex. myhost.com 8080) :
http://myhost.com:8080/UserService/MyBean/addUser?UserName=Bob&UserAge=15

5. Web Services

Utilisation de WS

@WebServiceRef : référence un web service

```
public class MyClientBean {

  @WebServiceRef(
    wsdlLocation="http://localhost:8080/UserService/MyBean?WSDL")
  private MyBeanItf service;

  public void foo() {
    long id = service.addUser("Bob",15);
} }
```


6. Conclusion

EJB

Approche intéressante

- prise en charge de services techniques par la plate-forme
- le composant se focalise sur le métier
- *packaging* & déploiement ++

Quelques critiques/pistes d'améliorations possibles

- "uniquement" Java
- domaine applicatif "limité" aux applications Java/Web
  ⇨ contrôle/commande ? temps-réel ? embarqué ? CAO ? infographie ? ...
- déploiement sur un seul serveur à la fois

6. Conclusion

Commerciaux

- WebSphere IBM `www-306.ibm.com/software/webservers`
- WebLogic BEA `www.weblogic.com`
- App Server Borland `www.borland.com/appserver`
- iPortal IONA `www.iona.com/products`
- Oracle, Sybase, HP, Fujitsu, ATG, Hitachi, Macromedia, SilverStream, ...

voir `java.sun.com/javaee/overview/compatibility.jsp`

Open source

- Java EE 5 (Sun) `java.sun.com/javaee`
- JOnAS (ObjectWeb) `jonas.objectweb.org`
- JBoss (Red Hat) `www.jboss.org`
- Geronimo (Apache) `geronimo.apache.org`
- JFox `www.huihoo.org/jfox`

6. Conclusion

Ressources

API Java EE 5 : http://java.sun.com/javaee/5/docs/api/
Tutorial Java EE 5 Sun : http://java.sun.com/javaee/5/docs/tutorial/doc/

Livre Mastering Enterprise Java Beans, 3rd edition
http://www.theserverside.com/tt/books/wiley/masteringEJB/index.tss