Java EE

Java Enterprise Edition

Pierre-Yves Gibello - pierreyves.gibello@experlog.com
(Mise à jour : Septembre 2011)

Java EE - Objectifs

- Faciliter le développement de nouvelles applications à base de composants
- Intégration avec les systèmes d'information existants
- Support pour les applications « critiques » de l'entreprise
  - Disponibilité, tolérance aux pannes, montée en charge, securité ...

Java EE – C'est quoi?

- http://java.sun.com/javaee
  anciennement, J2EE (Java2 Enterprise Edition)
- Spécifications
- Modèle de programmation
- Implémentation de référence
- Suite(s) de tests
- Label Java EE Sun (qualification de plateformes)

Offre commerciale

- Oracle WebLogic (haut de gamme)
- IBM Websphere (no 1)
- Oracle (Sun) Glassfish
- NEC WebOTX
- SAP NetWeaver
- TmaxSoft Jeus, Kingdee Apusic, TongTech TongWeb AS
- …

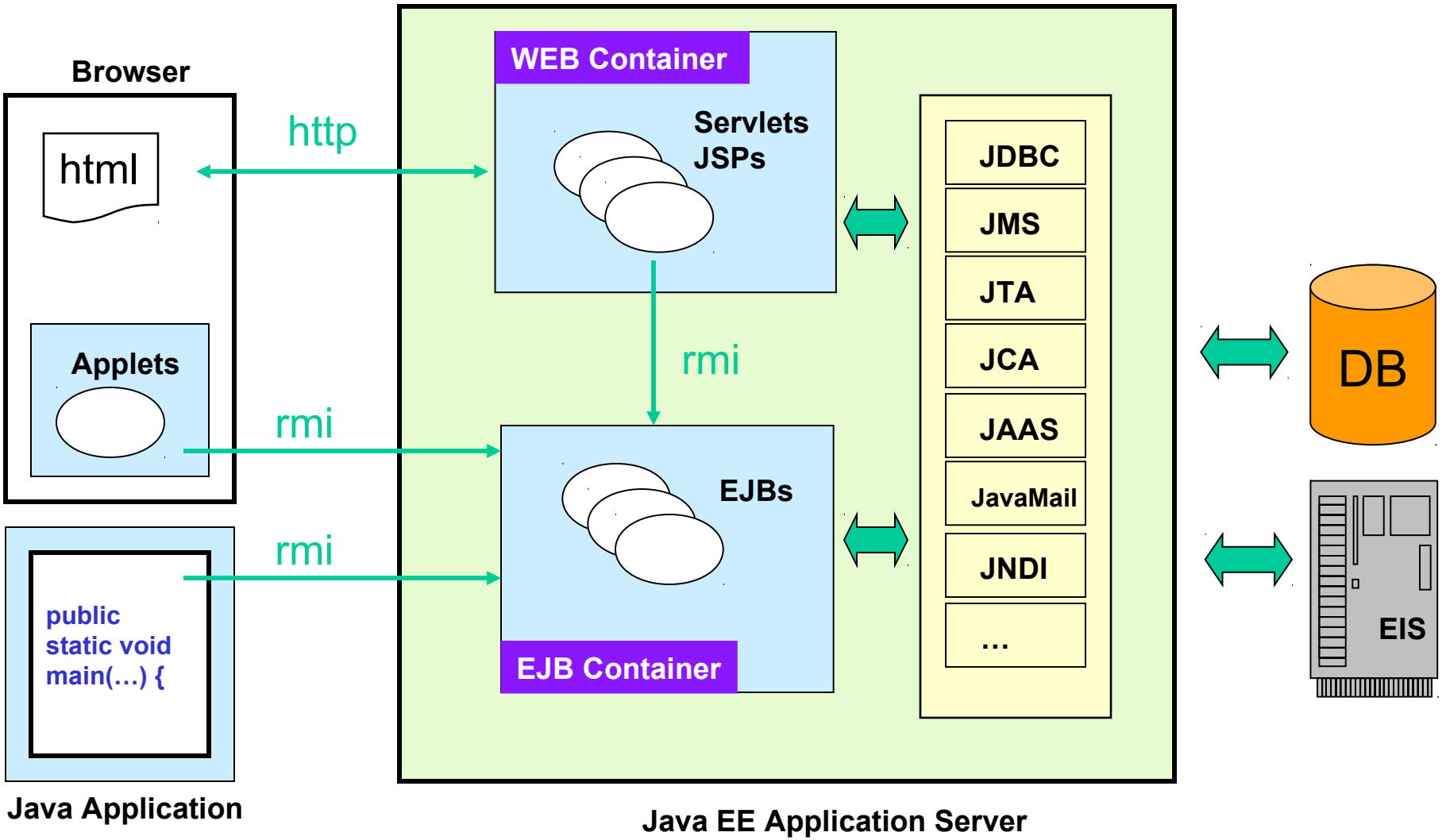
Offre open-source

- JBoss (no 1 en nombre de déploiements)
- OW2 JOnAS
- Oracle (Sun) Glassfish (« Platform edition »)
- Apache Geronimo (« Community edition » de IBM Websphere)
- openEjb
- …

Java EE sous l ’œil de Darwin...

- Standard en évolution depuis 1997
  - J2EE 1.0 à 1.4 en 2003, etc...
- Au départ, applications Web n-tiers
  - Présentation (Servlets puis JSP), essentiellement HTTP
  - Logique métier : EJB
  - Données : JDBC
- Puis infrastructure de support standard pour EAI
  - Facteurs de rationnalisation majeurs (JTA, JMS, JCA, Web Services)
  - Evolution de progiciels existants vers Java EE

Java EE - Architecture

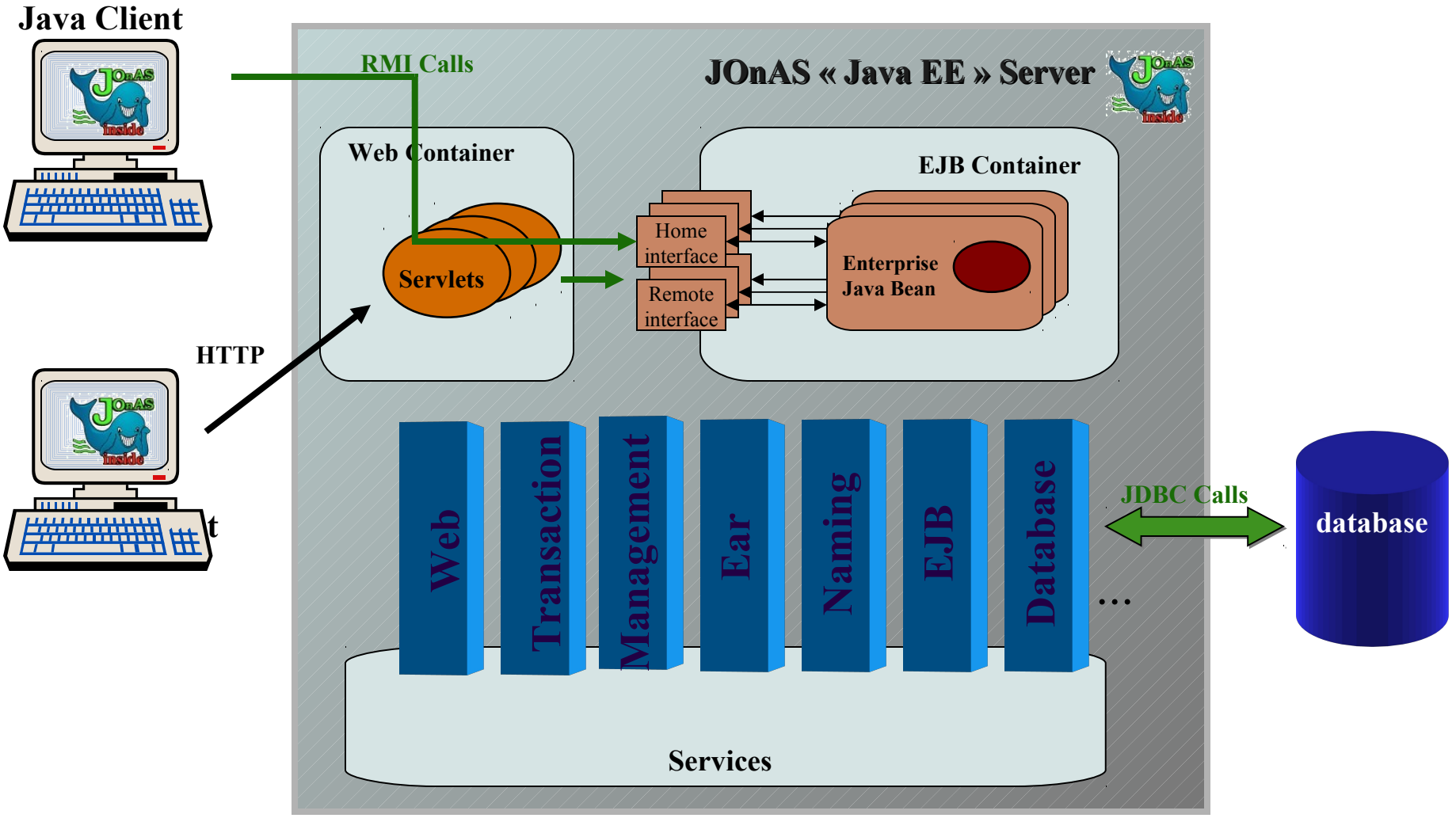


Architecture multi-tiers

- Client
  - Léger (Web, browser)
  - Lourd (Application java, Applet…)
  - Architecture orientée service (Application répartie sans présentation)
- Serveur d ’applications
  - Conteneur EJB + logique métier
  - Services non fonctionnels
- EIS ou Base de données

Un serveur Java EE

Source : Bull/OW2 (JOnAS)



Conteneur Web

- Servlets
  Code java exécuté sur le serveur
  Equivalent du CGI
  Génération de contenu Web dynamique
- JSP: Java Server Pages
  Mélange de HTML/XML et de code java
  Librairies d ’extension (« taglibs »)
  Précompilation en servlet

RMI

- Remote Method Invocation
  - Java seulement, mais passerelles
- « RPC objet » (appels sur objets distants)
- Service de nommage (RMI registry)
- Sécurité paramétrable (SecurityManager)
- Garbage Collection distribuée
- Téléchargement de code
- Fonctions avancées
  - Activation d ’objets persistants, Réplication

JNDI

- Service de nommage / annuaire
  - Java Naming and Directory Interface
- API accès aux annuaires
  - javax.naming
  - « Service Provider » par annuaire cible (LDAP, NIS, RMI registry…)
- Utilisation avec les EJB
  - Accès au bean (stub ou interface locale) pour initialiser
  - Accès à diverses ressources (UserTransaction, Queues JMS, DataSources…)

JMS

- Java Messaging Service
- JMS Provider : inclus dans JavaEE
  - Transport synchrone ou asynchrone, Garantie de livraison
  - « Messaging domains » point à point ou « publish/subscribe »
- Lien avec EJB : « message-driven bean »
  - Pour échanges asynchrones

API JavaEE de transactions : JTA

- Java Transaction API
- Package javax.transaction
  - TransactionManager : begin( ), commit( ), rollback( ) …
  - Transaction : commit( ), rollback( ), enlistResource(XAResource), registerSynchronisation(Synchronization) ...
  - Synchronization : beforeCompletion( ), afterCompletion(commit | rollback)

JTA : Participants

- XAResource
  - Conformes à la spécification XA
  - Enregistrement avec transaction.enlistResource( )
- Synchronization
  - Pour les ressources « non transactionnelles » (EAI…)
  - Participant averti des frontières de transaction
  - enregistrement : transaction.registerSynchronization( )
  - beforeCompletion( ) équivaut à prepare( )
  - afterCompletion(état = commit | rollback) équivaut à commit | rollback

API XA de JavaEE

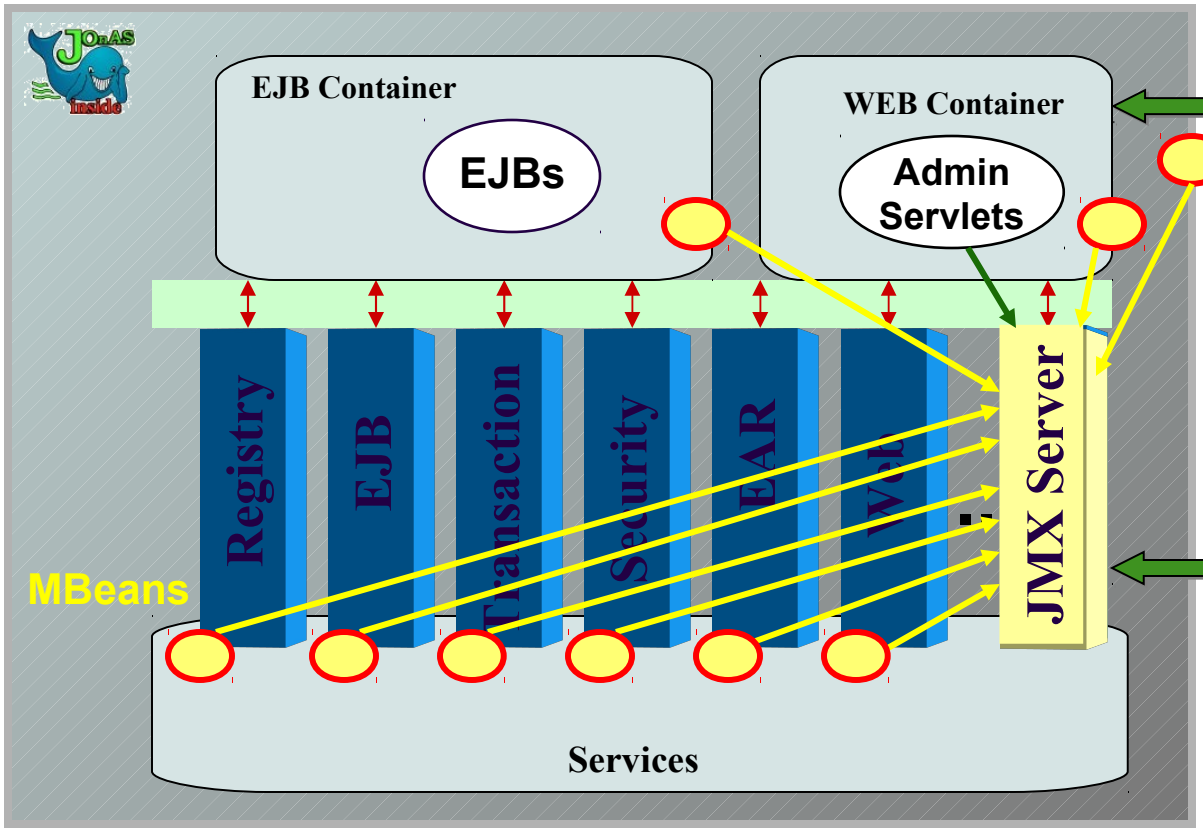
- Package javax.transaction.xa
  - XAResource (prepare( ), commit( ), rollback( )...)
  - Xid (identifiant de transaction XA)
- Package javax.sql
  - Extension de JDBC (bases de données)
  - XADataSource (getXAConnection( ))
  - XAConnection (PooledConnection, avec getConnection( ) et getXAResource( ))

JMX

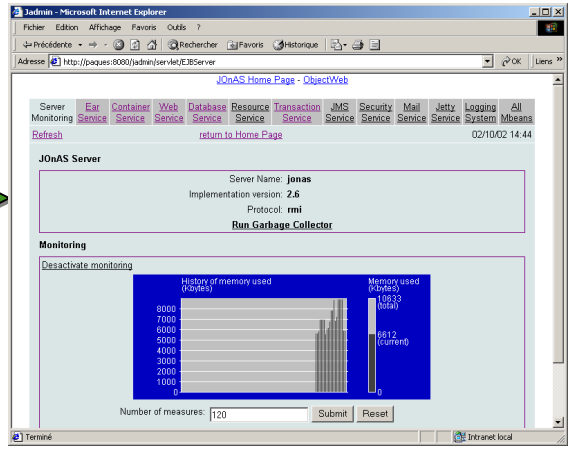
- Java Management eXtensions
  - API unique pour applications de management
- Mbeans avec accesseurs get/set
  - Typage faible, attributs nommés
- Serveur JMX
  - Enregistrement des Mbeans
  - Les applis d ’administration dialoguent avec le serveur JMX
- Instrumenter un composant
  - Fournir un ou des Mbeans
  - Les enregistrer auprès du serveur JMX

JMX : Exemple d ’un serveur JavaEE

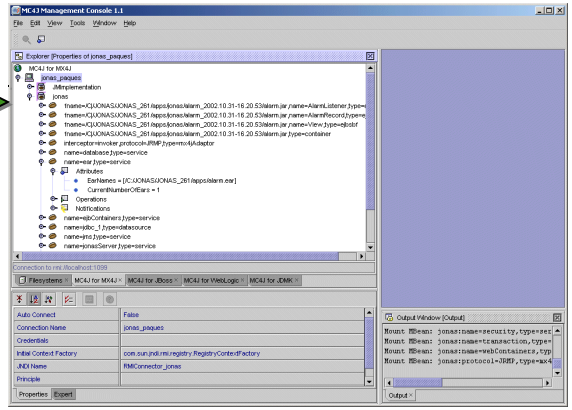
Source : OW2 JOnAS



Admin console



MC4J



Sécurité : JAAS

- Authentification et autorisation
  - Authentification : s'assurer que le client est bien celui qu'il prétend être.
  - Autorisation (ou « habilitation ») : s'assurer qu'il détient les droits de faire ce qu'il demande.
- Clients regroupés par « roles »
  - ex. administrateur, développeur, etc...
  - La gestion des associations nom/ mot de passe ou certificat / rôle est sous-traitée à un « Realm » (peut utiliser différentes techniques : fichier de conf, base de données, LDAP...)
- Intégré aux mécanismes de sécurité du conteneur Web
  - Pour les clients lourds java, APIs clientes JAAS pour implémenter l'intéraction avec l'utilisateur (ex. saisie du mot de passe...)

EJB : Architecture

- JavaBeans pour l' Enterprise
- Pas des JavaBeans (pas de représentation graphique…)
- Logique métier
- S'appuie sur Java SE et les APIs de Java EE
  - JNDI, JTA/JTS, JDBC, JMS , JAAS
- Gestion déclarative : personnalisation par annotations (ou sans toucher au code source : descripteur de déploiement)
- Portable sur les différents conteneurs EJB

EJB : Gamme de services implicites

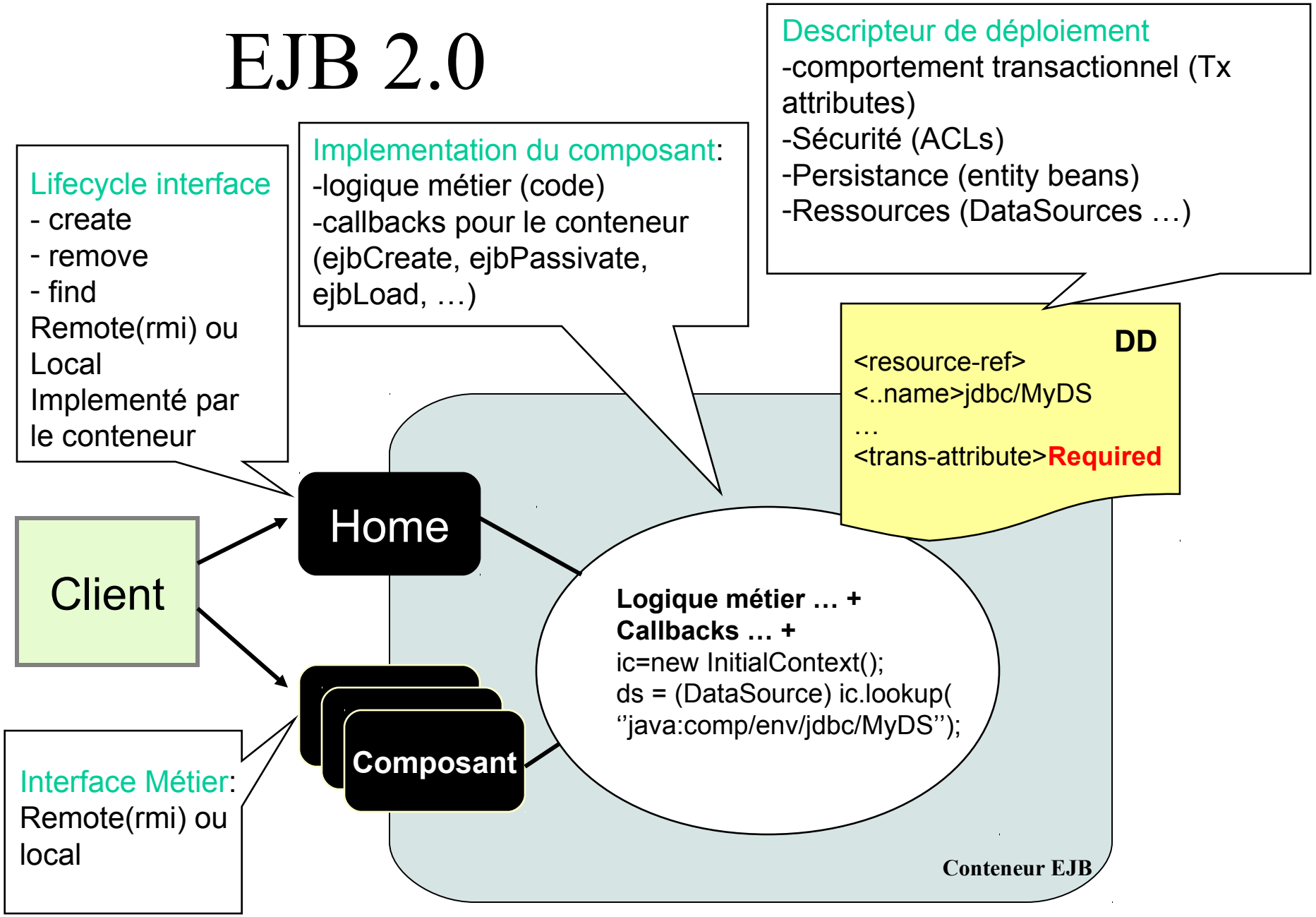
- Gestion du cycle de vie
- Gestion de l'état
- Sécurité
- Transactions
- Persistance
- Localisation des composants transparente

(comparable à objets distribués CORBA)

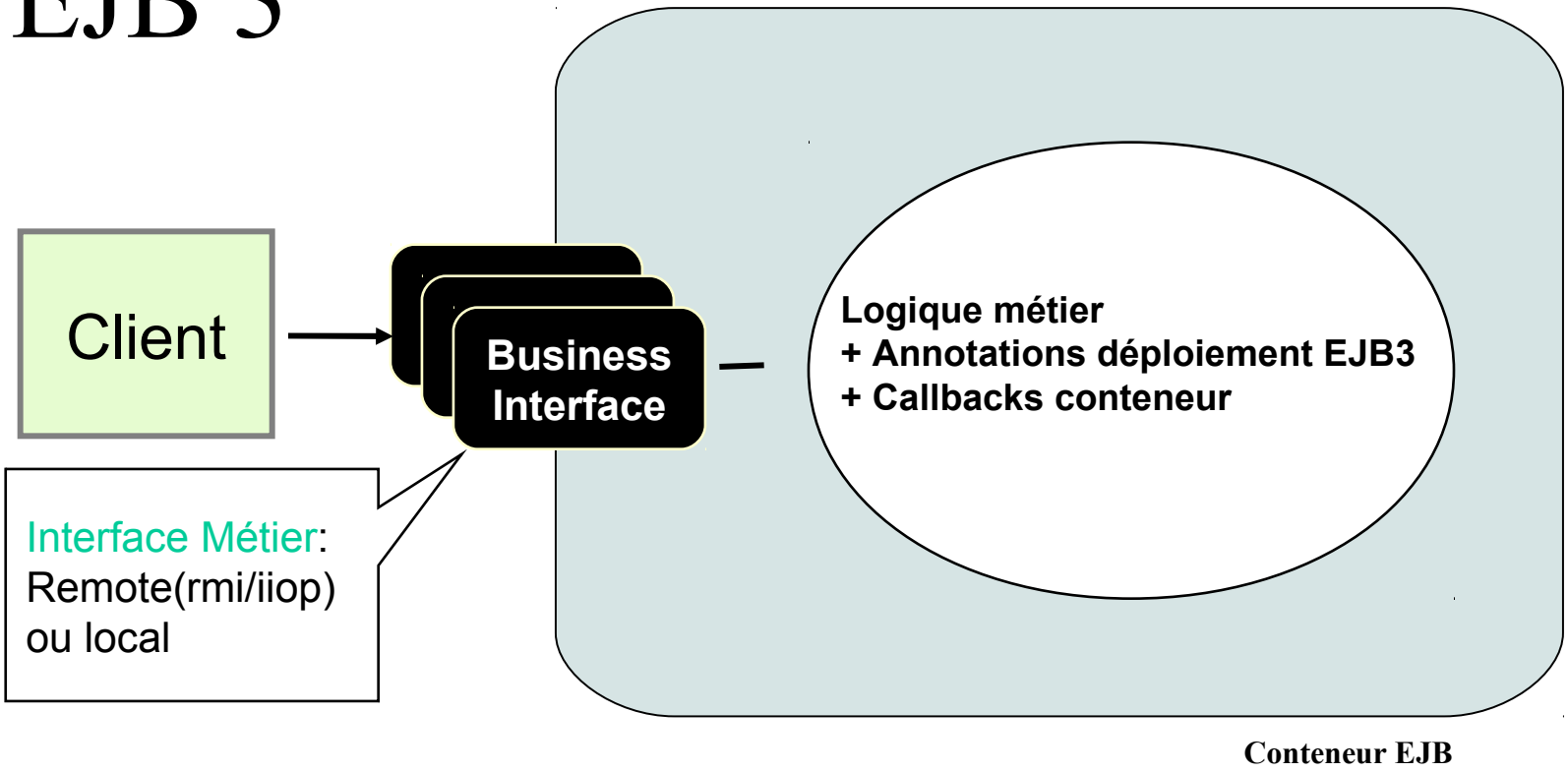
- Répartition de charge, pooling

=> Le développeur se focalise sur les aspects métier

EJB 2.0


Descripteur de déploiement
-comportement transactionnel (Tx attributes)
-Sécurité (ACLs)
-Persistance (entity beans)
-Ressources (DataSources …)
Implementation du composant:
-logique métier (code)
-callbacks pour le conteneur (ejbCreate, ejbPassivate, ejbLoad, …)
Lifecycle interface
- create
- remove
- find
Remote(rmi) ou Local
Implementé par le conteneur
DD
<resource-ref>
<..name>jdbc/MyDS
…
<trans-attribute>Required
Client
Home
Composant
Logique métier … +
Callbacks … +
ic=new InitialContext();
ds = (DataSource) ic.lookup(
''java:comp/env/jdbc/MyDS'');
Interface Métier:
Remote(rmi) ou local
Conteneur EJB

EJB 3



Simplification : 1 classe, 1 ou 2 interfaces
- L'interface Home (cycle de vie) disparaît

Le descripteur de déploiement devient facultatif
- Remplacé par des annotations java dans le bean
- Si présent tout de même, priorité au DD sur les annotations

EJB : Interface métier

- Remote (RMI / IIOP) ou Local
- Vue client de l’ EJB
- Declare les méthodes métier
- Implementée par les outils intégrés à la plateforme EJB - au moment du déploiement

Exemple : interface métier

```
package facturation;

public interface FacturationRemote {
        void init( );
        void creerFacture(String numfact, double montant);
        Facture getFacture(String numfact);
      // …
}
```

EJB : Implémentation du Bean

- Implémente les méthodes de l'interface métier
- Peut hériter d'un autre EJB, ou d'un POJO
- Spécifie ses caractéristiques de déploiement par annotations
  - Type de bean
  - Comportement : transactions, sécurité, persistance…
  - Callbacks conteneur

Exemple : Implém. de Bean

```
package facturation;

@Stateful (mappedName=« Facturation »)
@Remote(FacturationRemote.class)
public class FacturationBean implements FacturationRemote {

        void init( ) {
          // …
        }

        Facture getFacture(String numfact) {
          // …
        }
        // …
}
```

EJB : Code client

```
Context ctx = new InitialContext();

// appel JNDI pour obtenir une
  référence
//à l'interface
FacturationRemote fact =
  (FacturationRemote)ctx.lookup(
    "Facturation");

// appel méthode métier
Facture f = fact.getFacture(numfact);
```

EJB : Entité

- Représente des données dans la base de données
- Persistance gérée par le conteneur
  - Pas d'accès BD dans le code.
  - Pour les EJB3, utilisation de JPA (Java Persistence API)

EJB : Bean Session

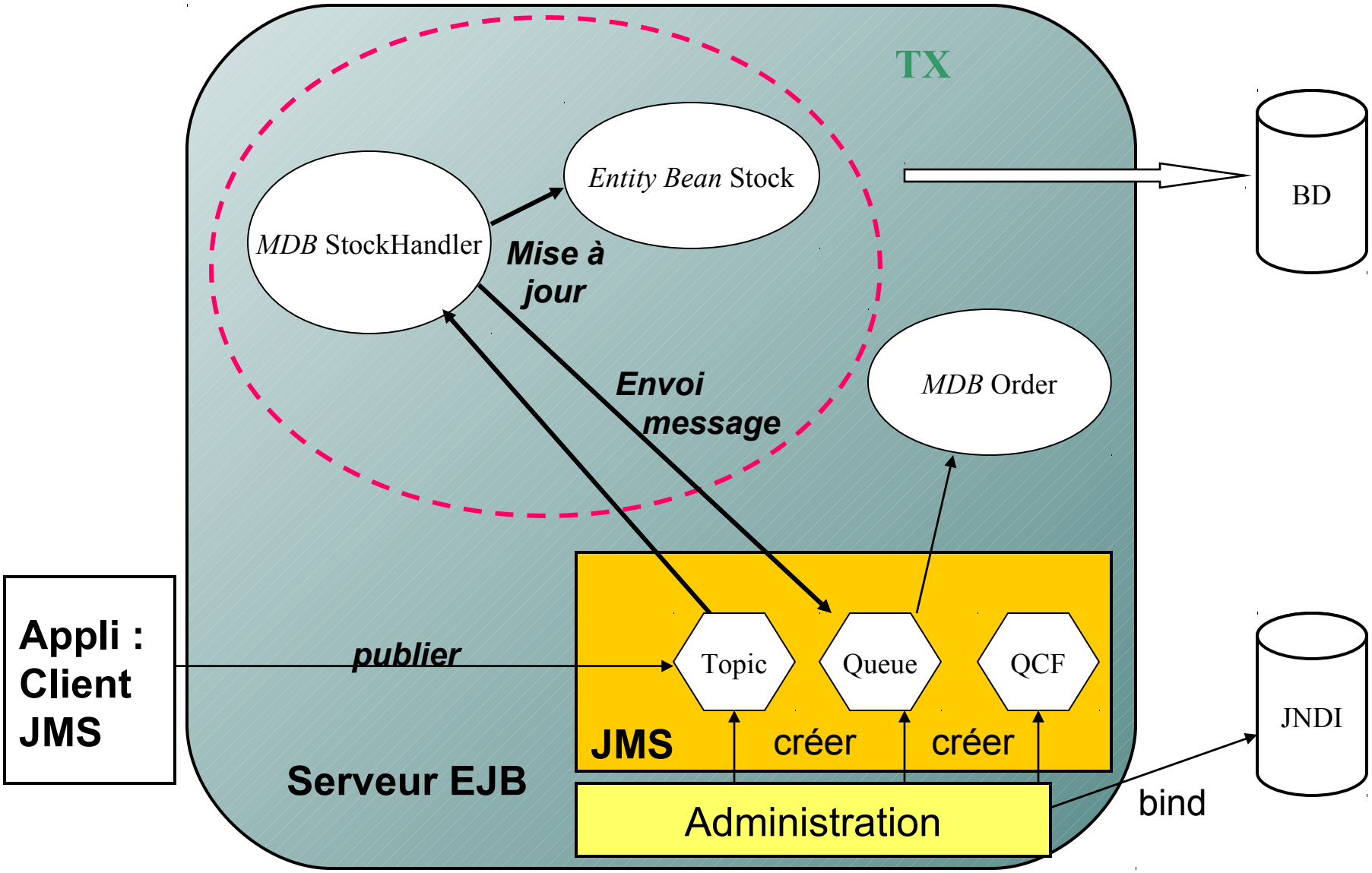
- Gestion des interactions entre beans entité ou session, accès aux ressources, réalisation d'actions sur demande du client
- Objets métier non persistants
- Stateful ou Stateless - maintient ou pas un état interne en mémoire

=> Un Bean Stateful encapsule la logique métier et l'état specifiques à un client

EJB : Message Driven Bean

- Composant asynchrone
- Execution sur réception d'un message JMS
  - Méthode onMessage( )
  - Appels à d'autres beans, etc…
- Descripteur de déploiement / annotations
  - Associations Bean / ressource JMS
  - Ressources JMS (ex. Queue ou Topic)

Message Driven Bean : exemple



Message Driven Bean : exemple (2)

```
public class StockHandlerBean  implements javax.jms.MessageListener {
…
  public void onMessage(Message message) {
     ...
     sh = (StockHome)initialContext.lookup("java:comp/env/ejb/Stock");
     queue = (Queue)initialContext.lookup("java:comp/env/jms/Orders");
     …
     MapMessage msg = (MapMessage)message;
     pid = msg.getString("ProductId");
     qty = msg.getString( "Quantity");
     cid = msg.getString("CustomerId");
     Stock stock = sh.findByPrimaryKey(pid);
     stock.decreaseQuantity(qty);
    …
    qs = session.createSender(queue);
    TextMessage tm = session.createTextMessage();
    String m = "For CustomerId = "+cid+" ProductId= "+pid+" Quantity= "+qty;
    tm.setText(m);
    qs.send(tm);
    ...
    }
}
```

MDB : Annotations

```
@MessageDriven(activationConfig = {
   @ActivationConfigProperty(
      propertyName=« destination »,
      propertyValue=« SampleQueue »),
   @ActivationConfigProperty(
      propertyName=« destinationType »,
      propertyValue=« javax.jms.Queue »)
   }
```

EJB : Configuration & Déploiement

- Interface(s) (Remote et/ou Local), classe qui implémente le Bean
- Déploiement : annotations dans le bean, et/ou descripteur de déploiement (fichier XML)

  <ejb-jar>
  - Description du Bean (Entity ou Session, ...)
  - Ressources (Base de données,...)
  - Securité: permissions et rôles
  - Persistance (BMP, CMP)
  - Attributs transactionnels
  - …

  </ejb-jar>

  Priorité au descripteur de déploiement sur les annotations.

=> Utilisé par l'assembleur d'application et par le conteneur EJB au moment du déploiement

Descripteur de déploiement (optionnel)

```
<enterprise-beans>
  <session>
    <description>EJB Facturation</description>
    <ejb-name>Facturation</ejb-name>
    <business-remote>facturation.FacturationRemote</business-remote>
    <ejb-class>facturation.FacturationBean</ejb-class>
    <session-type>Stateful</session-type>
    <transaction-type>Container</transaction-type>
    <resource-ref>
      <res-ref-name>jdbc/facturationDB</res-ref-name>
      <res-type>javax.sql.DataSource</res-type>
      <res-auth>Container</res-auth>
    </resource-ref>
  </session>
</enterprise-beans>
```

Bean Session

« Indirection » : Lien entre interfaces et implémentation

Ressource : ici, BD

Ressources et JNDI

- Ressources déclarées par annotations ou dans le descripteur de déploiement (accès via JNDI)
- Convention de nommage
  - Noms préfixés par le type de ressource référencée (ejb, jms, jdbc, mail, url…)
- Exemple

```
Fournisseur f =
   (FournisseurRemote)initialContext.lookup(
   "java:comp/env/ejb/Fournisseur");
bd = (DataSource)initialContext.lookup(
   "java:comp/env/jdbc/Compta");
```

Lookup et Stateful Session

- Chaque lookup retourne une nouvelle instance
- Exemple (servlet) :

```
CartBean c =
(CartBean)httpSession.getAttribute(« caddie »);
if(c == null) {
  c = initialContext.lookup(« ShoppingCart »);
  httpSession.setAttribute(« caddie », c);
}
```

Optimisations par le conteneur

- Stateless Session Bean : Sans état
  - Pool d'instances
  - Le serveur peut y accéder via un pool de threads
- Stateful session et Entity beans
  - Activation / Passivation (appel par le container des callbacks annotées @PostActivate après activation / @PrePassivate avant passivation)
- Pour tous les Session Beans
  - Callbacks annotées @PostConstruct et @PreDestroy
  - Gestion possible des callbacks par un Callback Listener (annotation @CallbackListener dans le bean pour spécifier la classe chargée de gérer les callbacks conteneur).

Injection de dépendances

- Variable d'instance initialisée par le conteneur
  - Alternative au lookup JNDI
  - Interne au conteneur (pas client distant !)
- Implicite ou avec spécification du nom (name=« nom JNDI », optionnel)
  - @EJB private CalculatorLocal calc; // Injection d'EJB (local)
  - @Resource javax.sql.DataSource ds; // Injection de ressource
- Egalement utilisé par JPA pour le contexte de persistance

Intercepteurs : @AroundInvoke

- Interception de tous les appels de méthode
  - Identification de l'appel (méthode, classe…) : paramètre InvocationContext
  - **@AroundInvoke**
    public Object intercept(InvocationContext ctx) throws Exception {
      String method = ctx.getMethod().getName();
      //Avant invocation…
      try { **return ctx.proceed();** } catch(Exception e) { throw e; }
      finally {
        // Apres invocation…
      }
    }
- Exemples d'usage : logging, sécurité, pré et post-processing…

Intercepteur : classe dédiée

- Pattern général : séparation cycle de vie / métier
  - Les callbacks d'interception sont regroupées dans une classe dédiée
  - @Stateless
    @Interceptors (Intercept.class)
    public class CalculatorBean { … }
  - public class Intercept {
    @AroundInvoke
    public Object intercept(InvocationContext ctx)
    throws Exception {
    String clazz = ctx.getClass().getName();
    //…
    return ctx.proceed();
    }
    }

Persistance EJB3

- API de persistance : JPA
  - Mapping objet / relationnel
  - Pluggable sur différents frameworks de persistance (JDO, Hibernate, TopLink, etc…) via un « persistence provider ».
- Persistance gérée par le conteneur
  - Gestion déclarative (annotations)
  - Classe persistante : « entité » (@Entity)
  - Champs persistants : variables d'instance (« access=FIELD ») ou propriétés avec méthodes get/set (« access=PROPERTY »).
  - Présence obligatoire d'un constructeur sans argument (et implémentation de java.io.Serializable si transféré côté client).
  - Relations entre instances d'entités (cardinalité 1-1, 1-N, N-P, uni ou bi-directionnel)
- EJB-QL
  - Basé sur SQL92 (« select… from… where »)
  - Méthodes associées à des requêtes avec paramètres

Exemple

```
@Entity
public class Facture implements java.io.Serializable {

    private int numfact;
    private double montant;
    private Client client;

    public Facture() { } // Constructeur par défaut (obligatoire pour Entity bean)
    public Facture(String numfact) { this.numfact = numfact; }

    @Id
    public int getNumfact() { return numfact; }
    public void setNumfact(int nf) { numfact = nf; }

    public void setMontant(double montant) { this.montant = montant; }
    public double getMontant( ) { return montant; }

    @ManyToOne
    @JoinColumn (name = “refclient”)
    public Client getClient( ) { return client; }
    public void setClient(Client client) { this.client = client; }

}
```

Exemple (2)

```
@Stateless
@Remote(FacturationRemote.class)
public class FacturationBean implements FacturationRemote {

	@PersistenceContext
	private EntityManager entityManager = null;

	public void creerFacture(String numfact, double montant) {
		Facture fact = new Facture(numfact);
		fact.setMontant(montant);
		entityManager.persist(fact);
	}

	public Facture getFacture(String numfact) {
		return entityManager.find(Facture.class, numfact);
	}
}
```

Relations

- Part de la classe courante
  - Exemple : classe Client, lien OneToMany avec la classe Facture
- Le « propriétaire » de la relation correspond à la table qui définit la clé étrangère (ne s'applique pas à ManyToMany)
  - Lien Client/Factures : la Facture est « propriétaire » de la relation
  - Attribut « mappedBy » côté Client, et joinColumn côté Facture
- OneToOne
  - Exemple : Fournisseur et Adresse
- OneToMany
  - Exemple : Client et Facture
- ManyToOne
  - Exemple : Facture et Client
- ManyToMany
  - Exemple : Produit et Fournisseur

Relations et matérialisation

- Requête BD : curseur sans matérialisation du dataset = « lazy loading »
  - Buffer avec pré-fetch.
- Eager loading : matérialisation complète des données
- Annotation sur la relation (Défaut : LAZY)
  - @OneToMany(fetch=FetchType.EAGER)

EntityManager

- persist(entité) : sauvegarde
- merge(entité) : sauvegarde après rattachement au contexte (retourne l'entité attachée)
- remove(entité) : suppression
- find(entité.class, clé_primaire) : recherche par clé
- refresh(entité) : annulation modifications
- flush( ) : sauvegarde immédiate modifications
- createQuery(ejbQL)
- createNativeQuery(SQL)

EntityManager.persist (entité)

- Persiste une instance d'entité
  - Nouvelle entité passe en état géré, ignore les entités supprimées
- Ecriture BD au commit( ) transaction, ou sur flush( )
  - Cascade sur relations annotées cascade=PERSIST ou ALL
- Si instance détachée : IllegalArgumentException
  - Envisager un merge ( ) ?

EntityManager.remove (entité)

- Suppression d'une entité
  - Ignore les entités nouvelles ou supprimées
- Ecriture BD au commit( ) transaction, ou sur flush( )
  - Cascade sur relations annotées cascade=REMOVE ou ALL
- Si instance détachée : IllegalArgumentException
  - Envisager un merge ( ) ?

EntityManager.flush( )

- Ecriture modifications (entités attachées seulement).
- Si instance gérée : enregistrement modifs
  - Cascade sur relations annotées cascade=PERSIST ou ALL
- Si instance supprimée : suppression de la BD

EJB-QL

- Dialecte proche de SQL
  - « select… from … where » sur des objets
  - « update » et « delete »
  - « join », « group by… having »
  - Possibilité de requêtes paramétrées
  - Utilisation directe, ou requêtes nommées (@NamedQuery)
- Exemple

```
public List<Facture> listFactures( ) {
      Query qry = entityManager.createQuery(
                  « select f from Facture f »);
      return qry.getResultList();
}
```

Callbacks cycle de vie

- Appelées par le « persistence manager »
  - Annotations @PrePersist, @PostPersist, @PreRemove, @PostRemove, @PreUpdate, @PostUpdate, @PreLoad
  - Gestion possible par une classe dédiée
    @EntityListener(MyListener.class)
    L'instance cible est passée en paramètre aux callbacks
- Exemple

```
@PreRemove
void preRemove() {
      System.out.println(« Avant suppression »);
}
```

Pattern « ValueObject » revisité

- Entité qui implémente java.io.Serializable
  - Récupération de l'objet côté client
  - L'objet est détaché du contexte de persistance (incompatible avec LAZY loading…)
  - Appels méthodes getXXX( ) et/ou setXXX( )
- Au retour de l'objet côté serveur
  - persistentObj = entityManager.merge(obj);
- Attention à la montée en charge…
  - Pas de cascade avec LAZY loading… donc EAGER !
  - Graphe d'objets sérialisé si dépendances (utiliser transient si nécessaire)

Définition fine du mapping O/R

- Table relationnelle : annotation @Table
- Configuration du mapping O/R dans le descripteur de déploiement de persistance (persistence.xml)
- Stratégie de mapping pour l'héritage : attribut strategy de l'annotation @Inheritance
  - ONLY_ONE_TABLE (défaut) : 1 table pour 1 hiérarchie de classes (les colonnes représentent tous les champs persistants possibles), @DiscriminatorColumn et attribut DiscriminatorValue de @Inheritance
  - TABLE_PER_CLASS : 1 table par classe
  - JOINED : Table spécifique pour les champs d'une classe fille, jointure avec celle de la classe parente.

Mapping ONLY_ONE_TABLE

```
@Entity
@Inheritance(discriminatorValue=« P »)
@DiscriminatorColumn(name=« typepersonne »)
public class Personne implements Serializable {
  //…
}

@Entity
@Inheritance(discriminatorValue=« E »)
public class Employe extends Personne {
  //…
}
```

Transactions

- Applicable aux 3 profils de composants
  - Session, Entité, Message driven
    - Limitation pour le MessageDriven (attributs « Required » ou « NotSupported » seulement).
- Gestion explicite
  - Utilisation de javax.transaction.UserTransaction (JTA)
  - Contrôle de la transaction (timeout, « rollbackOnly »)
  - Exemple

```
UserTransaction utx =
  (UserTransaction)ctx.lookup(« java:comp/UserTransaction »);
utx.begin();
…
utx.commit();
```

Gestion déclarative des transactions

- Au niveau de la méthode du bean ! (démarcation)
- Required (valeur par défaut)
  - Si pas de transaction, nouvelle transaction
- Supports
  - Si transaction courante, l'utiliser
- NotSupported
  - Si transaction courante, elle est suspendue
- RequiresNew
  - Nouvelle transaction (si tx courante, suspendue)
- Mandatory
  - Exception si pas de transaction courante
- Never
  - Exception si transaction courante

Annotations : Gestion déclarative des transactions

```
@TransactionManagement(javax.ejb.TransactionManagementType.CONTAINER)
public class Facturation implements FacturationRemote {

    @TransactionAttribute(javax.ejb.TransactionAttributeType.REQUIRED)
    public void creerFacture( ) {
        // …
    }
}
```

Descripteur de déploiement : Gestion déclarative des Transactions

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

 ...
</assembly-descriptor>
```

NotSupported
Required
RequiresNew
Mandatory
Supports
Never

Gestion des événements transactionnels (Session Bean)

- Interception par un EJB des événements transactionnels (produits par le conteneur)
  - Implantation de javax.ejb.SessionSynchronization
- Evénements (appelés par le conteneur)
  - afterBegin : appelé après UserTransaction.begin
  - beforeCompletion : appelé avant UserTransaction.commit
  - afterCompletion(true | false) : appelé après commit ou rollback

Sécurité : gestion déclarative

```
@Stateful
@RolesAllowed( { « comptable », « logisticien » } )
public class Fournisseur implements FournisseurLocal {
  // ...
}
```

- L'identité (nom) d'un client est aussi appelée « Principal »
- Les rôles et les « Realm » (associations nom / mot de passe ou certificat / rôle, avec mécanisme d'accès – type fichier, LDAP ou base de données) sont déclarés par configuration au niveau du serveur (chaque serveur dispose de mécanismes spécifiques).

Sécurité : Descripteur de déploiement

```
<assembly-descriptor>
...
 <security-role>
  <description>Personnel comptable</description>
  <role-name>comptable</role-name>
 </security-role>
 <method-permission>
  <role-name>comptable</role-name>
  <method>
    <ejb-name>Fournisseur</ejb-name>
    <method-name>*</method-name>
  </method>
 </method-permission>
</ assembly-descriptor>
```

Définition de rôle

Permissions accordées à un rôle

Sécurité : domaine de protection

- Domaine de protection (ou de confiance)
  - Le conteneur (ou un ensemble de conteneurs)
  - A l'intérieur, pas de nouvelle authentification
    - Le contexte de sécurité est propagé
    - L'objet invoqué fait confiance à l'appelant
- Mécanisme de changement d'identité
  - Annotation @RunAs (« identité »)

Sécurité : API serveur

- Pour aller plus loin que le mode déclaratif
  - ex. exécuter une action en fonction du rôle du client courant...

```
@Resource SessionContext context;
// ...
if(context.isCallerInRole(« comptable »)) {
   String name = context.getCallerPrincipal( ).getName( );
   // ...
}
```

Sécurité client : conteneur web

- Mapping avec les méthodes classiques
  - BASIC, FORM (avec formulaire), CLIENT-CERT (avec certificat client)...
- Authentification par le conteneur web, et transmission du contexte de sécurité au conteneur EJB
  - Paramétrage du conteneur web pour utiliser le même Realm que le conteneur EJB
    - Exemple : JonAS fournit un gestionnaire de Realm intégrable à Tomcat (d'où propagation du contexte de sécurité de Tomcat à JOnAS).

JCA

- Java Connector Architecture
- Intégration avec les SI d'entreprise (EIS)
  - Applications (ERP, Supply Chain…)
  - Middleware (gestionnaire de transactions…)
- Connecteur composé de :
  - Contrats système
  - API cliente
  - Resource Adapter

Contrats Système

- Interaction entre le SI et le serveur d'applications
  - Gestion des connexions et pooling
  - Gestion des transactions (par le serveur d 'application et/ou le SI)
  - Sécurité (accès sécurisé au SI)

API clientes

- Standard : CCI (Common Client Interface)
  - Avantage : API standard
  - Inconvénient : trop général et peu compréhensible (pas de « sens » métier), implémentation facultative.
- Spécifique : API spécifique au SI, fournie par le « resource adapter »
  - Avantage : API « métier »
  - Inconvénient : spécifique au SI concerné, donc différente selon le connecteur...

Resource Adapter

- Implémentation de l'interfaçage avec le SI
  - Classes d'interface
  - Librairies natives du SI
- Descripteur de déploiement
  - Configuration du connecteur au sein du serveur d 'application
    - Classes d'implémentation des interfaces standard
    - Fonctionnalités supportées ou non (transactions, sécurité…)

Déploiement

- Packaging en fichier .rar (RA archive)
  - Déployé par le serveur d'application
  - Format jar avec organisation standard
  - Contient tous les éléments du connecteur
  - Descripteur de déploiement (ra.xml) dans META-INF : configuration du connecteur

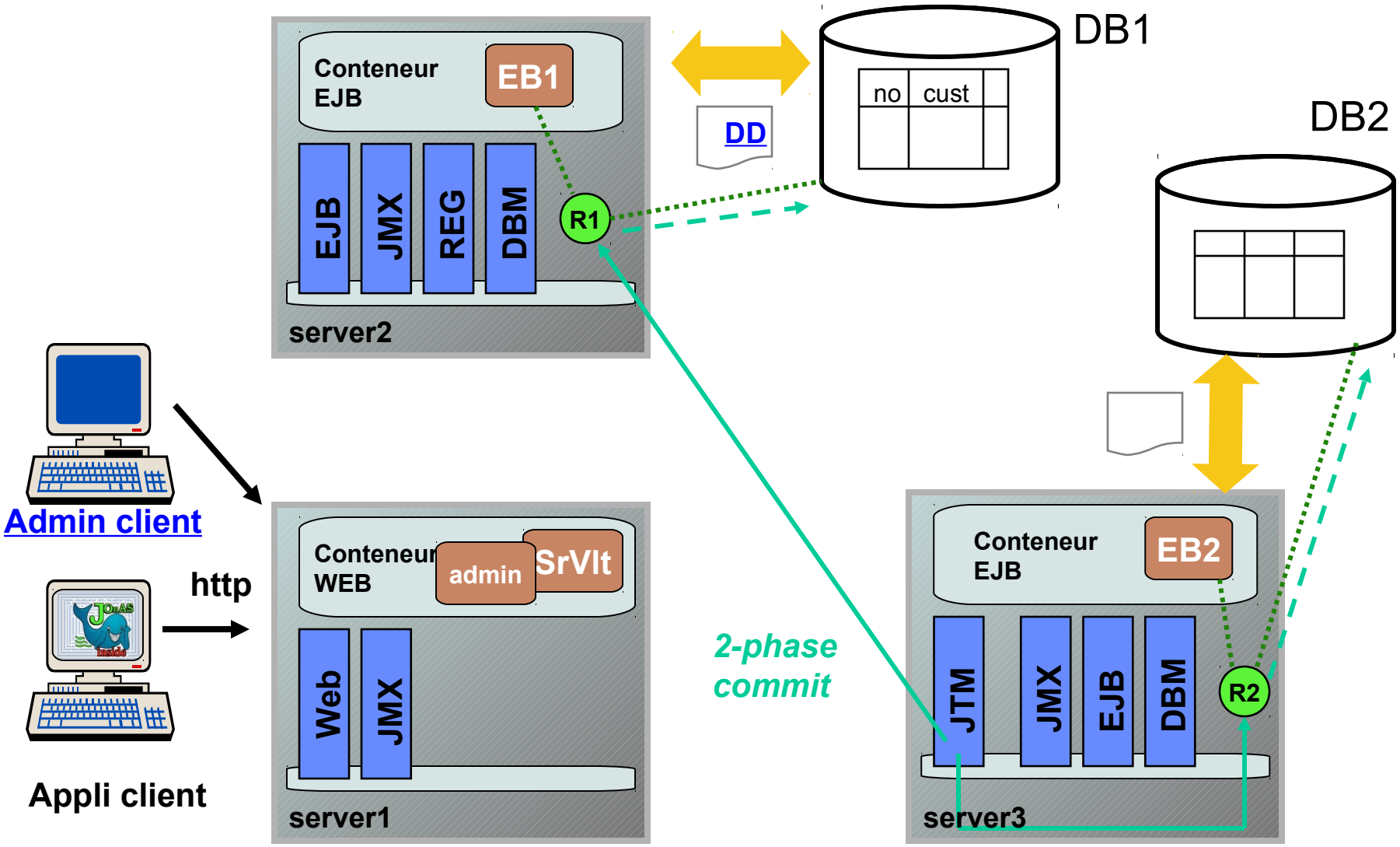
Rôles définis par la spec. EJB

- Fournisseur de beans
  - Développeur qui crée les EJB
- Assembleur d'application
  - Crée l'application par assemblage d'EJB
- Administrateur
  - Déploiement, sécurité, exploitation, montée en charge…
  - Analogue au rôle de DBA pour les BD

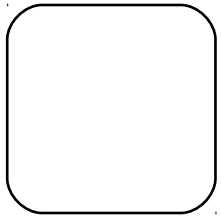
Packaging

- Application JavaEE (agrégation de différents tiers)
  - Fichier « .ear » + descripteur « application.xml »
- Tiers client
  - Web : fichier « .war » + descripteur « web.xml »
  - Application : fichier « .jar » + descripteur « application-client.xml » (lancement du main() de la classe spécifiée dans le « manifest », attribut « Main-Class »)
- Tiers EJB
  - Fichier « .jar » + descripteur « ejb-jar.xml »
- Tiers « EIS » (connecteurs JCA)
  - Fichier « .rar » + descripteur « rar.xml »

Persistance + Transactions : exemple



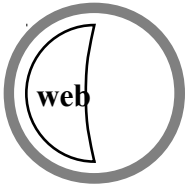
Répartition de charge : notations



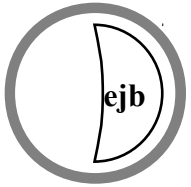
Un noeud (machine) qui héberge un ou plusieurs serveurs

Un conteneur Web

Un conteneur EJB

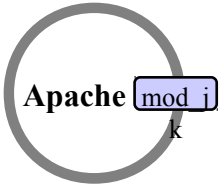


Un serveur qui héberge un conteneur Web



Un serveur qui héberge un conteneur EJB

Un serveur qui héberge un conteneur Web et un conteneur EJB

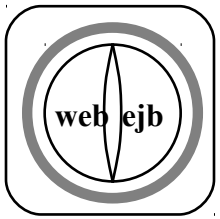


Un serveur Apache avec le module mod_jk

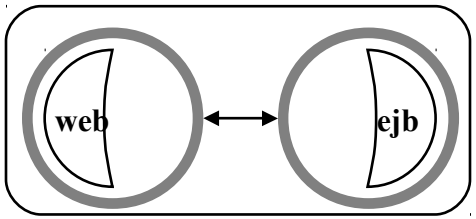
Répartition de charge : scenarii (1)

Répartition du serveur JavaEE

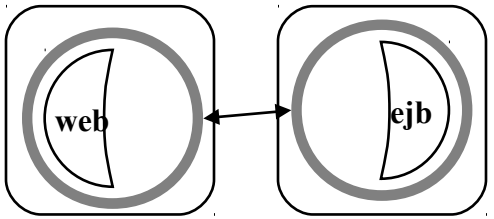
Compact



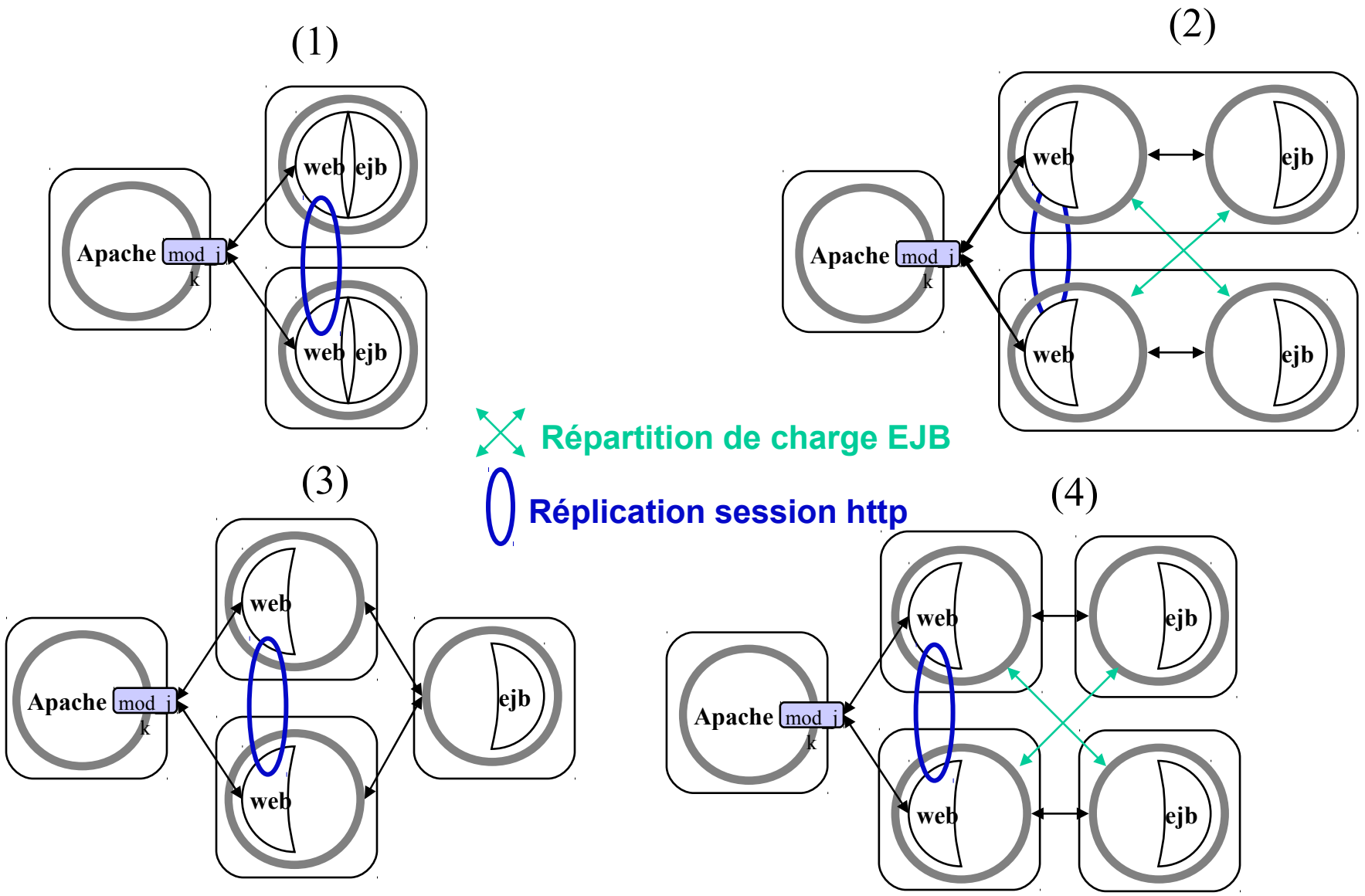
Réparti
(au sein d ’un même nœud)



Réparti



Répartition de charge : scenarii (2)



Répartition de charge et clustering

Source : OW2 JOnAS