Java Enterprise Edition

CARA M2 Miage FA-FC

Jean-François Roos - bâtiment M3 extension - bureau 218 - Jean-Francois.Roos@univ-lille1.fr

3 janvier 2016

Sommaire

1 Introduction
- JEE
- 3 tiers
- Application JEE
- Client
- Composants web
- Composants métiers et tier système d'information
- Composants métiers
- Serveur d'applications et containers
- Assemblage et déploiement
- Rôles dans le développement

2 Composants EJB

3 Design Patterns EJB

Ensemble de concepts pour le développement d'applications réparties

- défini par Sun , ORACLE
- basé sur Java
- un ensemble de spécifications JSR
- en évolution permanente depuis 1996/1997
  J2EE 1.0 (servlet+EJB+JDBC),...,1.3,1.4 Java EE 5 depuis 2006 (annotations), Java EE7 depuis juin 2013
- domaines applicatifs visés :
  - e-commerce (B2B,B2C)
  - systèmes d'information
  - sites web
  - plate-formes de services (audiovisuel, télécom, ...)

Plates-formes existantes :

- implémentation de référence : Oracle GlassFish Server OpenSource Edition 4.03,4
- commerciales certifiées Java EE 7 :
  - TmaxSoft TMAX JEUS 85
  - RedHat Wildfly 8.0.06
- commerciales certifiées Java EE 6
  - Oracle GlassFish Enterprise Server v3
  - Oracle WebLogic Server 12c de Oracle Corporation
  - JEUS 7, un serveur d'applications de TmaxSoft
  - IBM WebSphere Application Server 8.0
  - Fujitsu Interstage
- open source certifiées Java EE 6 :
  - JBoss AS 7.x (profil web uniquement pour la version 7.0.x)
  - Apache Geronimo 3.0

Processus de certification mis en place par Sun

- TCK (Test Compatibility Kit)
- payant sauf pour plates-formes open-source
- assez lourd (20 000 tests) à mettre en oeuvre

communications distantes :

- RMI-IIOP : requête/réponse (TCP + IIOP + sérialisation Java)
- JAX-WS : requête/réponse Web Service (HTTP + SOAP + XML)
- JMS : MOM (message oriented middleware) : message + boîte à lettres

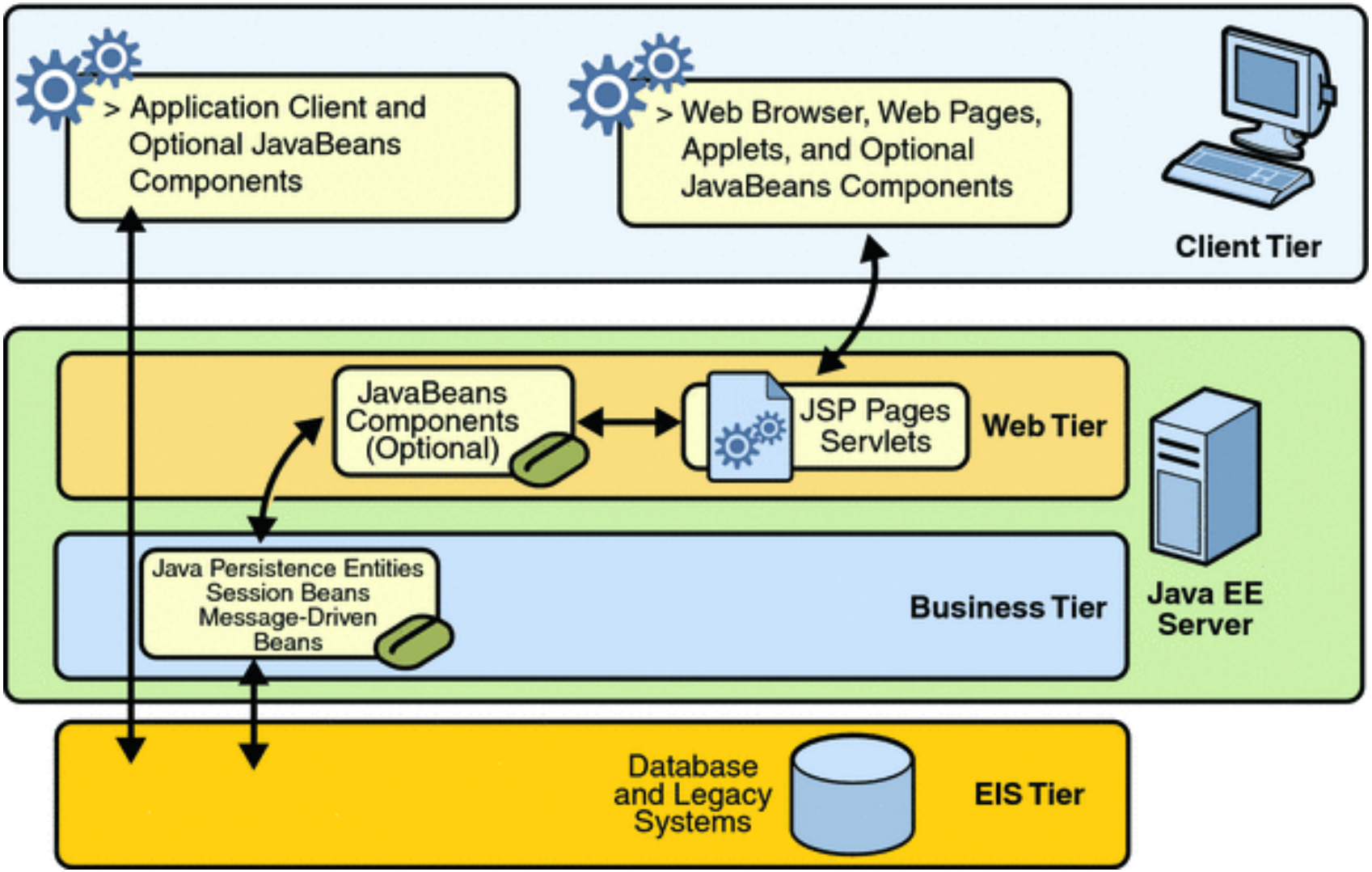
services systèmes :

- JNDI (Java Naming and Directory Interface) : annuaire
- JTA (Java Transaction API) : gestion de transactions
- JPA (Java Persistence API) : gestion de la persistance des données

un système de connecteurs (JCA) pour permettre à la plate-forme d'interagir :

- JDBC (Java Data Base Connectivity) : accès client/serveur aux SGBD
- JavaMail : envoi/réception de mail

support des web services


> Application Client and Optional JavaBeans Components
> Web Browser, Web Pages, Applets, and Optional JavaBeans Components
Client Tier
JavaBeans Components (Optional)
JSP Pages Servlets
Web Tier
Java Persistence Entities
Session Beans
Message-Driven Beans
Business Tier
Java EE Server
Database and Legacy Systems
EIS Tier

Mise en œuvre du principe des architectures 3 tiers

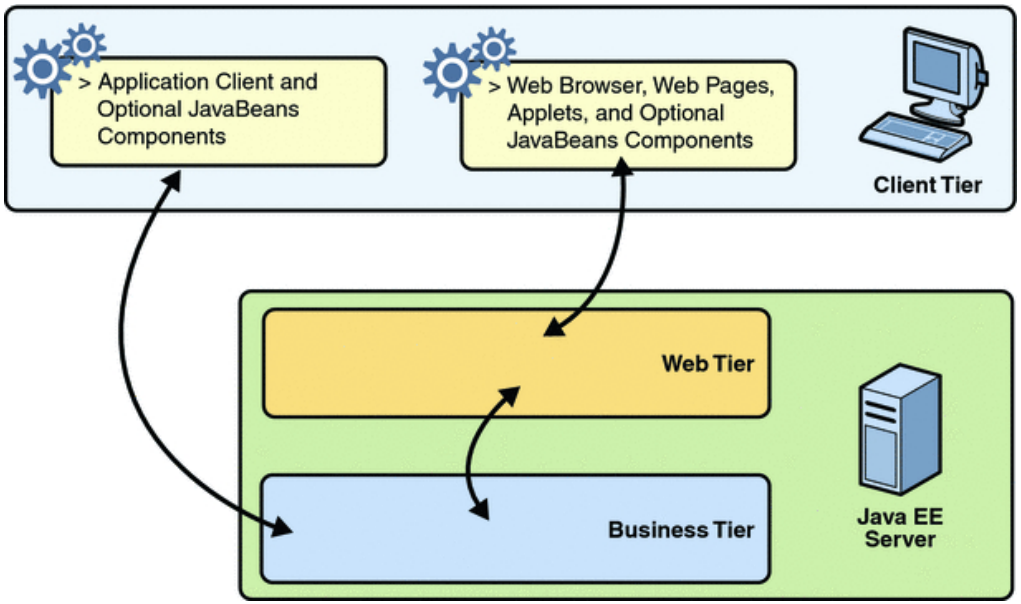
- Client
  - Riche : application Java
  - Léger : navigateur Web
- Serveur d'applications : héberge des applications à base de
  - Composants EJB : classes Java conformes au modèle EJB
  - Composants Web : servlet ou JSP
- SGBD : fournit un support de stockage pour les données de l'application
  - 80% SGBDR (Oracle, SQL Server, PostGreSQL, ...)
  - 20% autres applications de stockage

Les applications JEE sont constituées de composants JEE :
unité logicielle indépendante contenant classes et fichiers

- applications clientes, applets :
  composants qui tournent sur le client.
- composants Web :
  tournent sur le serveur
  JavaServlet, JavaServer Faces, JavaServer Pages
- composants Enterprise Java Beans :
  composants métiers situés sur le serveur

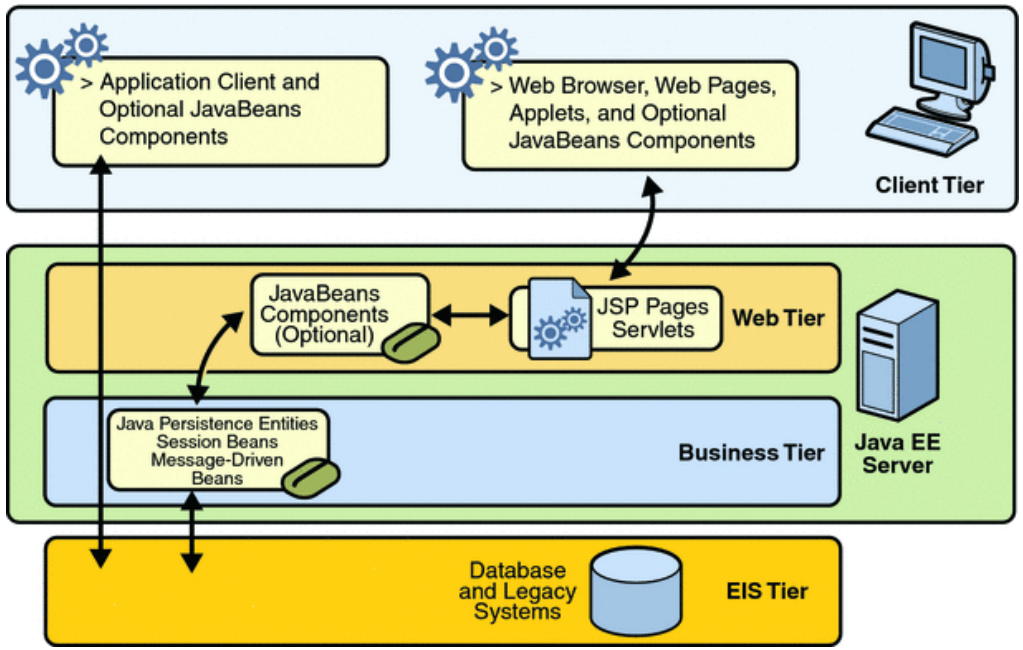
Les composants JEE suivant les spécifications JEE, sont assemblés en application, déployés sur et gérés par le serveur JEE.

- client Web : constitué de pages web dynamiques générées par des composants web situés sur le tiers web et un browser web, on parle souvent de client léger
- applets : petite application cliente en Java s'exécutant dans la JVM du browser web (le client doit posséder le bon plug-in Java et aussi une politique de sécurité)
- applications clientes : tournent sur la machine cliente. Elles possèdent une interface riche, peuvent effectuer des traitements plus importants. Elles accèdent directement au tiers métier.



- servlets : classes Java qui traitent dynamiquement des requêtes et construit les réponses
- pages JSP : documents texte qui s'exécutent comme des servlets mais qui permettent plus naturellement de créer du contenu statique
- JavaServer Faces : un framework pour construire des interfaces utilisateurs web basé sur les servlets et les JSP

- composants métiers :
  - Le code métier résout les besoins d'un domaine métier particulier comme la finance ou le stock.
  - Il est pris en charge par les enterprise beans qui tournent soit sur le tier métier soit sur le tier web.
- tier système d'information : il prend en charge le software du SI et inclue ERP, le système de transactions, les bases de données ...



- Constat sur les applications réparties orientées objet (CORBA-like)
  - mélange code fonctionnel – code non fonctionnel
  - code métier "noyé" dans un ensemble d'appels à des services techniques sécurité, transaction, persistance des données, annuaire, ...
  - difficile à concevoir, comprendre, réutiliser

- Besoins par rapport aux intergiciels basés objet ("CORBA-like")
  - configuration
  - déploiement
  - empaquetage (packaging)
  - assemblage
  - dynamicité
  - gestion des interactions et des dépendances

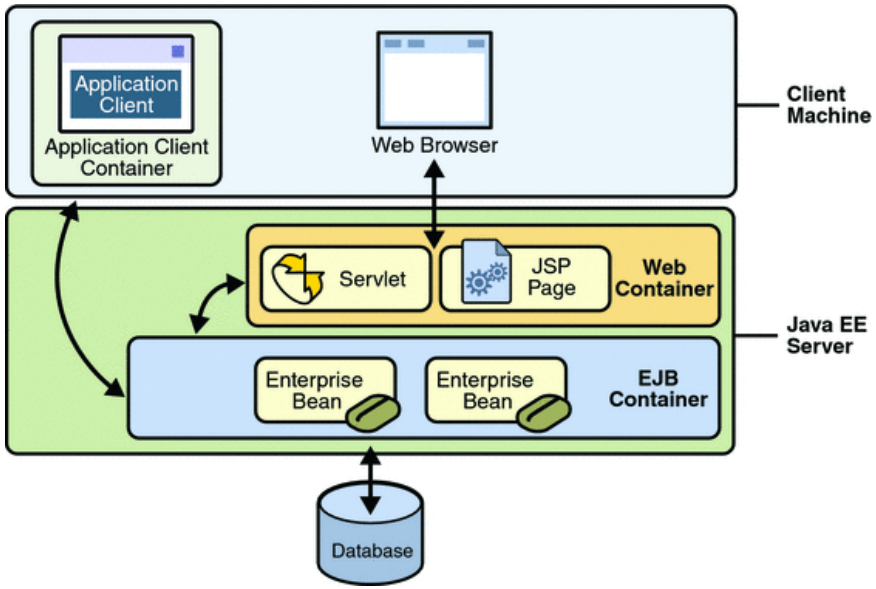
3 notions importantes des intergiciels basés composant

- séparation des préoccupations
  - développer le métier indépendamment des préoccupations non fonctionnelles
  - composants plus facilement réutilisables
- inversion du contrôle
  - container prend en charge l'exécution du code métier (composant)
  - container assure lien avec la partie technique
  - configurer plutôt que programmer
  - approche framework vs librairie
- injection de dépendances
  - vers d'autres composants, vers des services techniques
  - retirer du code métier la gestion des liens vers les autres composants métiers
  - faire gérer l'architecture applicative par le container

- Container : interface entre un composant et les fonctionalités de bas niveau spécifiques à la plate-forme qui l'abrite. Avant l'exécution d'un composant, il doit être assemblé dans un module JEE et déployé dans son container.
- L'assemblage suppose une configuration spécifique du container pour chaque composant : configuration des services fournis par le serveur JEE sous-jacent
  sécurité, transaction, nommage, ...
- le container gère également des services non configurables comme le cycle de vie des enterprise beans, des servlets, le pool de connexions aux bases de données, la persistance, et l'accès à l'api de la plate-forme JEE.
- le serveur JEE : la partie runtime fournit les containers EJB et web

Types de containers

- container EJB : gère l'exécution des EJB. Les EJB et leur container s'exécutent sur le serveur JEE.
- container web : gère l'exécution des pages web, servlets et certains EJB. Les composants web et container s'exécutent sur le serveur JEE.
- container d'application : gère l'exécution de l'application cliente. L'application et son container s'exécutent sur le poste client.
- container d'applet : gère l'exécution des applets, un browser web et un plug-in Java sur le poste client.



4 services fournis par le serveur au container EJB :

- persistance (JPA)
- transaction (JTA)
- nommage (JNDI)
- sécurité (JASS)
- communication bas niveau (RMI)

$\neq$ middleware style CORBA

ces services sont intégrés dès le départ à la plate forme

Une application JEE est empaquetée en une ou plusieurs unités standard de déploiement utilisables sur toute plate-forme JEE. Chaque unité contient :

- un ou plusieurs composants (EJB, page web, servlet, ...)
- un descripteur de déploiement optionnel qui décrit son contenu

Dès qu'un paquetage JEE est produit, il est prêt à être déployé en utilisant l'outil de déploiement de la plate-forme.

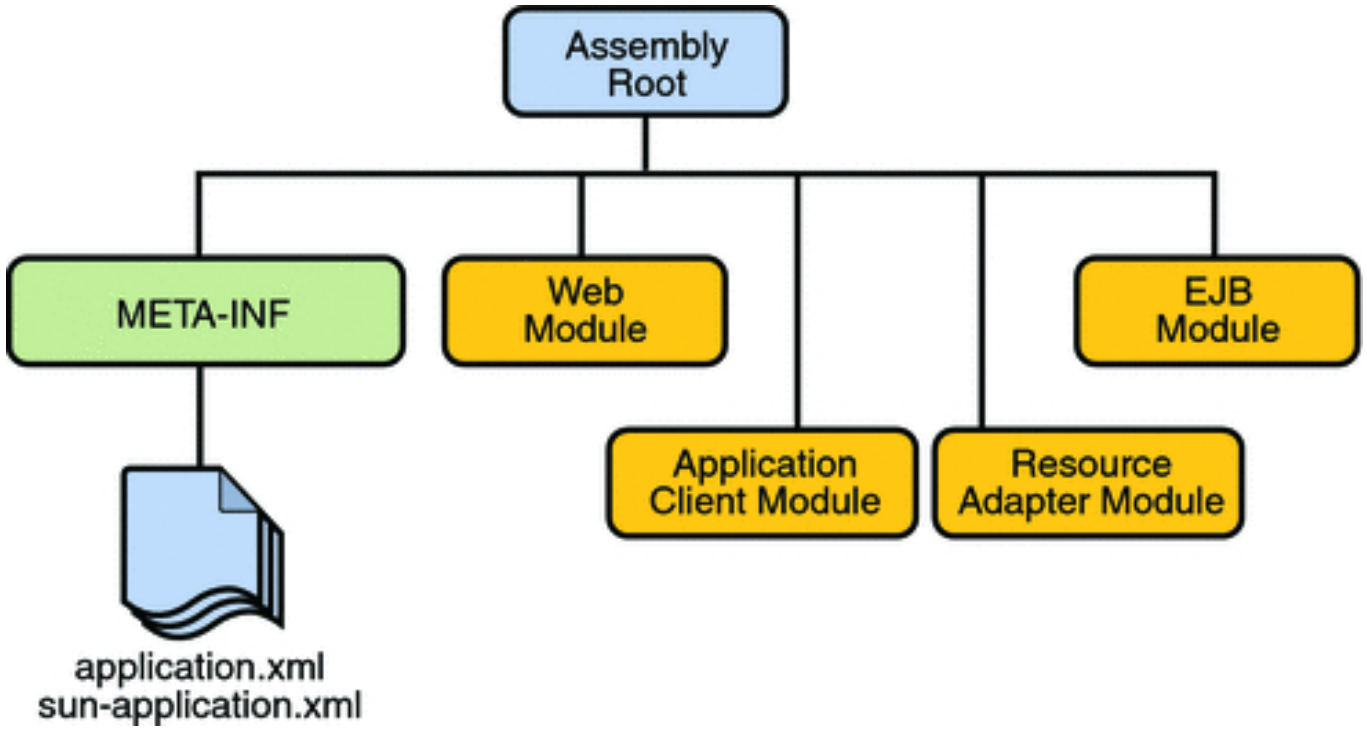
Une application JEE est livrée sous la forme :

- d'une archive Java (un fichier jar)
- d'une archive web (un fichier war)
- d'une archive enterprise (un fichier ear)

un fichier ear ou war est une archive Java standard avec une extrension .ear ou .war

En utilisant ces différentes archives, il est possible d'assembler différentes applications JEE en utilisant certains de ces mêmes composants.

- Un fichier ear contient des modules JEE et des descripteurs de déploiement.
- Un descripteur de déploiement est un fichier XML d'extension .xml qui décrit les propriétés de déploiement d'une application, d'un module ou d'un composant.



deux types de descripteurs de déploiement :

- descripteur JEE : défini par une spécification JEE, il peut être utilisé pour configurer le déploiement sur tout serveur JEE.
- descripteur runtime : configure des paramètres spécifiques à l'implémentation. Il est nommé `sun-moduleType.xml` et se situe dans le même répertoire `META-INF` que le descripteur JEE.

un module JEE est composé d'un ou plusieurs composants du même type de container et d'un descripteur de déploiement de ce type (par exemple, un descripteur de déploiement EJB déclare des propriétés sur les transactions, la sécurité, ... pour un EJB).

Quatre types de modules JEE

- modules EJB : contiennent des fichiers `.class` d'EJB et un descripteur de déploiement EJB. Ils sont empaquetés dans des fichiers `jar` d'extension `.jar`
- modules web : contiennent des fichiers `.class` de servlets, des fichiers web, des fichiers `.class` utilitaires, fichiers images et HTML, et un descripteur de déploiement web. Ils sont empaquetés dans des fichiers `jar` d'extension `.war`
- modules d'application cliente : contiennent des fichiers `.class` et un descripteur de déploiement d'application cliente. Ils sont empaquetés dans un fichier `jar` d'extension `.jar`
- modules d'adaptateurs de ressources : contiennent toutes les interfaces Java, classes, librairies et un descripteur de déploiement d'adaptateur de ressources. Il sont empaquetés dans un fichier `jar` d'extension `.rar`

L'existence de modules réutilisables permet de diviser le développement et le déploiement de l'application en rôles distincts pouvant être affectés à différentes personnes ou entreprises.

- Le fournisseur du produit JEE : l'entreprise qui conçoit et propose une plate-forme JEE, un fournisseur de serveur d'applications
- Le fournisseur d'outils : l'entreprise qui crée les outils de développement, d'assemblage, et de packaging.
- Le fournisseur de composants de l'application : crée les composants web, les composants métiers, ...

Le fournisseur de composants de l'application

- le développeur EJB : fournit un fichier `jar` contenant un ou plusieurs EJB
  - écrit et compile le code source
  - définit le descripteur de déploiement
  - package le tout dans un fichier `jar` EJB
- le développeur de composants web : il fournit un fichier `war` contenant un ou plusieurs composants web
  - écrit et compile le code source des servlet
  - écrit les fichiers JSF , JSP et HTML
  - définit le descripteur de déploiement
  - package le tout dans un fichier `war`
- le développeur de l'application cliente : il fournit un fichier `jar` contenant l'application cliente
  - écrit et compile le code source
  - définit le descripteur de déploiement pour le client
  - package le tout dans un fichier `jar`

L'assembleur et le déployeur

- L'assembleur d'applications : il utilise les modules applicatifs développés et les assemble en une application JEE sous forme de fichier `ear`.
  - assemble des fichiers `jar` EJB et `war` en un fichier `ear`
  - définit le descripteur de déploiement pour l'application
  - vérifie que le contenu du fichier `ear` suit les spécifications JEE

- Le déployeur et administrateur de l'application :
  - configure l'application JEE pour l'environnement opérationnel
  - vérifie que le contenu du fichier `ear` suit les spécifications JEE
  - déploie le fichier `ear` dans le serveur d'application JEE

Sommaire

1 Introduction

2 Composants EJB
- Généralités
- Session bean
- Message-driven bean (MDB)
- Cycles de vie
- Entity bean
- Fonctionnalités avancées

3 Design Patterns 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 container
- la logique de présentation est du ressort du client

Types d'EJB

- Session : performs a task for a client
- Message-Driven : listener processing messages asynchronously
- Entity : represents a business entity object that exists in persistent storage (remplacés depuis JEE 5 par Java Persistence API)

Plusieurs versions : actuellement EJB 3.2

avant EJB 3 succès Java EE en général mais :

- trop compliqué, lourd, contraignant
- concepts objets (héritage, typage, polymorphisme, ...) difficilement exploitables
- mapping objet/relationnel limité
- trop de fichiers XML fastidieux à écrire, maintenir, comprendre

passage EJB 2 vers EJB 3

- utilisation des annotations et de la généricité Java 5
- pour simplifier l'écriture des beans
- éviter autant que faire se peut l'écriture de fichiers XML

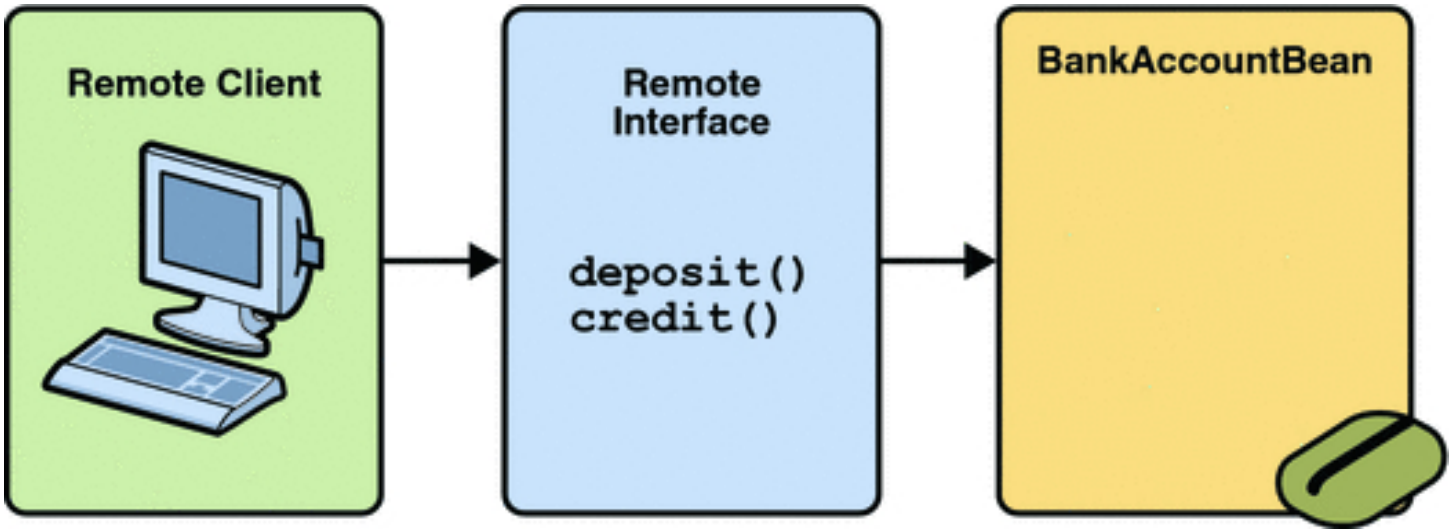
Table 1: Summary of Findings

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

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 quelconques d'un tel bean sont équivalentes
   - 1 instance par invocation
2. Stateful session bean
   - avec un état (en mémoire)
   - similaire session servlet/JSP
   - même instance pendant toute la durée d'une session avec un client
3. Singleton session bean
   - comme stateless mais
   - une instance par application au lieu d'un pool d'instances

- Chaque EJ Bean peut fournir 1 interface d'accès distant
- les services (méthodes) offerts par le bean à ses client accessibles par RMI/IIOP



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

Interface : annotations `@javax.ejb.Local` ou `@javax.ejb.Remote`

```
import javax.ejb.Remote;

@Remote
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);
}
```

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")`
- par défaut, le nom de la classe

Client local

- typiquement une servlet ou une JSP colocalisée sur le même serveur que le bean
- mécanisme dit "injection de dépendance"
- attribut du type de l'interface
- annoté `@EJB` éventuellement `@EJB(name="foobar")`

```
public class ClientServlet extends HttpServlet {
  @EJB(name="foobar")
  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>");
  }
}
```

Client distant

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

```
import javax.naming.Context;

public class Client {

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

Stateful Session Bean

- instance du bean reste en mémoire tant que le client est présent
- expiration au bout d'un délai d'inactivité similaire à la 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'issue de l'exécution de cette méthode

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() { ... }
}
```

Singleton bean

annotation : `javax.ejb.Singleton`

```
@Singleton
public class StatusBean { private String status;...}
```

Le container est responsable du moment de l'initialisation du bean sauf si l'annotation startup est utilisée, auquel cas le bean est initialisé au démarrage de l'application :

```
@Startup
@Singleton
public class StatusBean {
  private String status;

  @PostConstruct
  void init { status = "Ready";}
  ...
}
```

Singleton bean

Plusieurs singleton session bean peuvent être utilisés pour initialiser les données d'une application, ils doivent être initialisés dans un certain ordre :

```
@Singleton
public class PrimaryBean { ... }
```

`SecondaryBean` dépends de `PrimaryBean` :

```
@Singleton
@DependsOn("PrimaryBean")
public class SecondaryBean { ... }
```

Le container EJB initialisera `PrimaryBean` avant `SecondaryBean`. Le singleton session bean `TertiaryBean` dépends de `PrimaryBean` et `SecondaryBean` :

```
@Singleton
@DependsOn("PrimaryBean", "SecondaryBean")
public class TertiaryBean { ... }
```

Gestion des accès concurrents dans un Singleton

- deux moyens de contrôler la concurrence d'accès aux méthodes du bean : `container-managed concurrency` et `bean-managed concurrency` en précisant l'annotation `javax.ejb.ConcurrencyManagement`, par défaut gérée par le container.
- `@ConcurrencyManagement` possède un attribut type qui doit être positionné à `javax.ejb.ConcurrencyManagementType.CONTAINER` ou `javax.ejb.ConcurrencyManagementType.BEAN`.
- en `bean-managed concurrency`, le container autorise un accès concurrent complet au bean. Le développeur est responsable de la gestion de la concurrence.

Singleton bean Container-Managed Concurrency

- Le container contrôle les accès clients aux méthodes du singleton. L'annotation `javax.ejb.Lock` et le type `javax.ejb.LockType` spécifient le mode d'accès.
- On annote avec `@Lock(READ)` si la méthode peut être exécutée concurremment et avec `@Lock(WRITE)` si la méthode doit être exécutée en exclusion mutuelle, par défaut `@Lock(WRITE)`

```
@ConcurrencyManagement(CONTAINER)
@Singleton
public class ExampleSingletonBean
     {

  private String state;

  @Lock(READ)
  public String getState() {
    return state;
  }

  @Lock(WRITE)
  public void setState(String n)
       {
    state = n;
  }
}
```

Singleton bean Container-Managed Concurrency

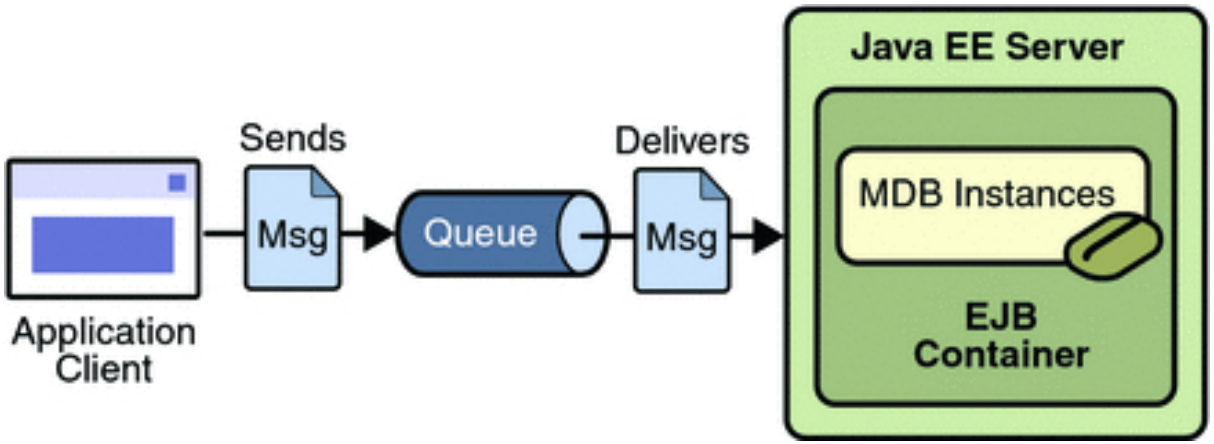
si une méthode est de type `WRITE`, tous les accès clients sont bloqués jusqu'à ce que le client courant termine son appel ou jusqu'au timeout auquel cas le container lève `javax.ejb.ConcurrentAccessTimeoutException` . L'annotation `javax.ejb.AccessTimeout` permet de spécifier en milisecondes le délai du time out.

```
@Singleton
@AccessTimeout(value=120000)
public class StatusSingletonBean {

  private String status;

  @Lock(WRITE)
  public void setStatus(String n) { status = n; }

  @Lock(WRITE)
  @AccessTimeout(value=360000)
  public void doTediousOperation { ... }
}
```

Quel session bean utiliser ?

- stateful session bean :
  - l'état du composant représente l'interaction avec un client précis
  - le composant doit conserver des informations sur le client entre différentes invocations de méthodes
  - le composant sert d'intermédiaire entre le client et d'autres composants de l'application, présentant une vue simplifiée pour le client
- stateless session bean :
  - l'état du composant n'a aucune donnée spécifique à un client
  - une invocation de méthode exécute une tâche générique pour tous les clients
  - il implémente un service web
- singleton session bean :
  - l'état doit être partagé par toute l'application
  - le bean doit être accédé par plusieurs threads concurremment
  - l'application a besoin d'exécuter du code au début et/ou à la fin

- Interaction par envoi message asynchrone
  (MOM : Message-Oriented Middleware)
- 2 modes : n vers 1 (queue), n vers m (topic)
- Caractéristiques
  - consomme des messages asynchrones
  - pas d'état ($\equiv$ stateless session bean)
  - toutes les instances d'une même classe de MDB sont équivalentes
  - peut traiter les messages de clients $\neq$
- Quand utiliser un MDB
  - éviter appels bloquants
  - découpler clients et serveurs
  - besoin de fiabilité : protection crash serveurs



- MDB basé sur Java Messaging Service (JMS)
- `ConnectionFactory` pour créer des connexions vers une queue/topic
- `Connection` une connexion vers une queue/topic
- `Session` période de temps pour l'envoi de messages dans 1 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 connexion)
  3. Création d'un message
  4. Envoi du message
  5. Fermeture session
  6. Fermeture connexion

Producteur

```
public class MyProducerBean {

  @Resource(name="jms/QueueConnectionFactory") //l'id de la factory
  private ConnectionFactory connectionFactory;

  @Resource(name="jms/Queue") // l'id de la queue
  private Queue destination;

  public void produce() {
    Connection connection = connectionFactory.createConnection();
    Session session = connection.createSession(true,Session.
        AUTO_ACKNOWLEDGE);
    MessageProducer producer = session.createProducer(destination);
    TextMessage message = session.createTextMessage();
    message.setText("Hello World!");
    producer.send(message);
    session.close();
    connection.close();
  }
}
```

Consommateur

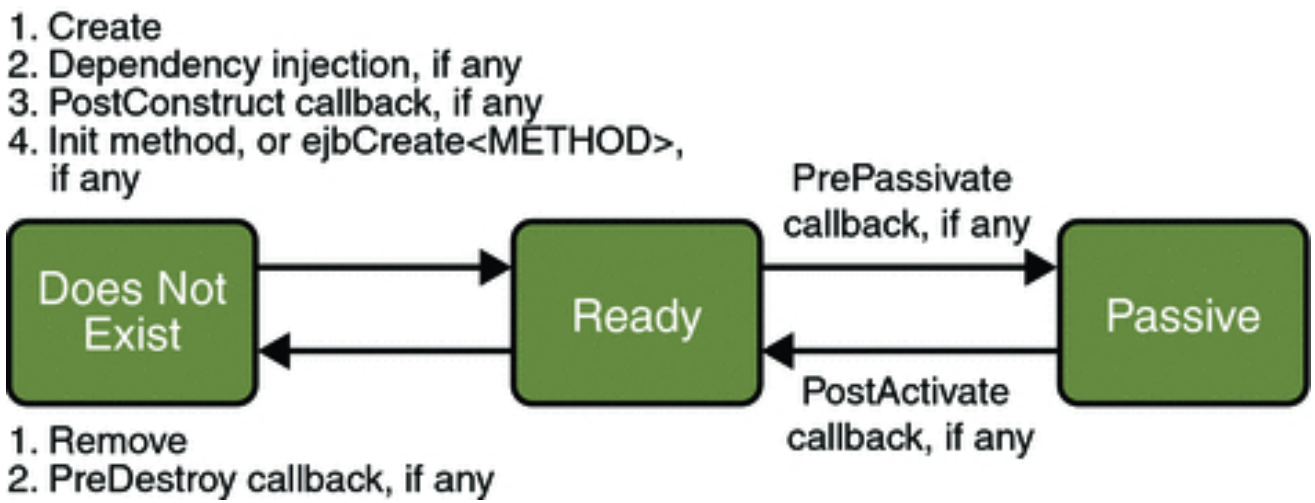
- MDB = classe
  - annotée `@MessageDriven`
  - implantant l'interface `MessageListener`
  - méthode `void onMessage(Message)`

```
@MessageDriven(name="jms/Queue")
public class MyConsumerBean implements MessageListener{

  public void onMessage( Message m ) {
    TextMessage message = (TextMessage) m;
    ...
  }
}
```

Un composant change d'état tout au long de son existence, on parle de cycle de vie. Chaque type de composant a son propre cycle de vie.
Cycle de vie d'un Stateful Session Bean

- Le client démarre le cycle de vie en récupérant une référence sur un stateful session bean.
- Le container exécute les injections de dépendance s'il y a lieu.
- Il invoque ensuite la méthode annotée avec `@PostConstruct` si elle existe.
- Le composant est alors prêt à exécuter les invocations du client.
- Dans l'état prêt, le container peut décider de désactiver, de rendre passif le composant en le plaçant en mémoire secondaire (politique least recently used). Il appelle la méthode annotée par `@Prepassivate` juste avant.

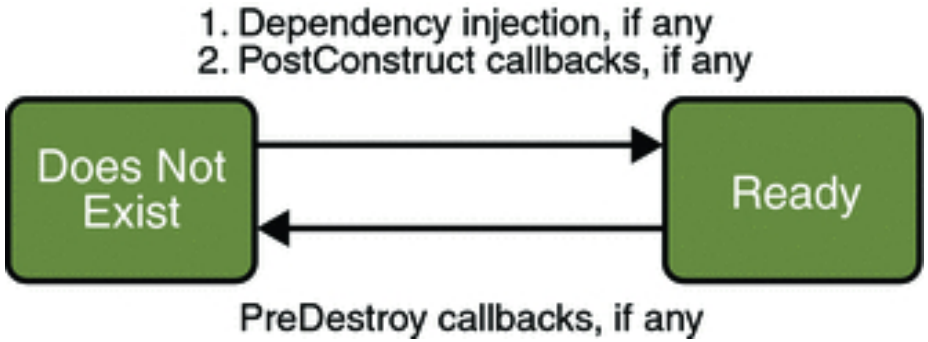
- Si un client invoque une méthode alors que le composant est dans l'état passif, le container l'active, appelle la méthode annotée par `@PostActivate` si elle existe et fait passer le composant à l'état prêt.
- A la fin, le client invoque une méthode annotée par `@Remove`, et le container appelle la méthode annotée par `@PreDestroy`, si elle existe. L'instance peut être alors récupérée par le garbage collector.
- Le code client contrôle l'invocation d'une seule méthode du cycle de vie, celle annotée par `@Remove`. Toutes les autres sont appelées par le container.



Cycle de vie d'un Stateless Session Bean

Comme un stateless session bean ne devient jamais passif, son cycle de vie a seulement deux états : non-existant et prêt.

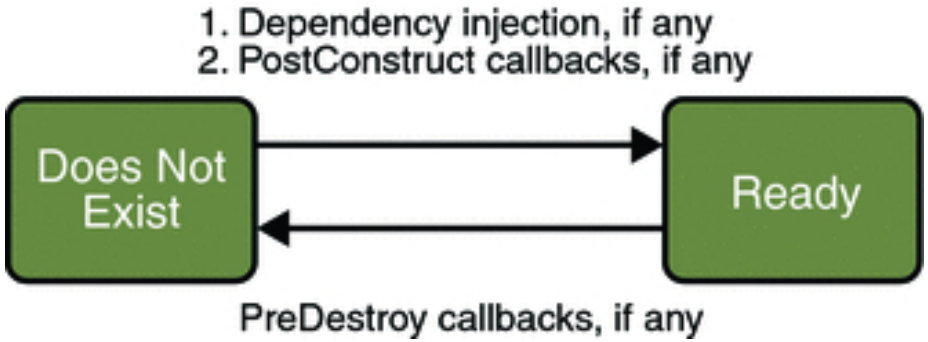
- Généralement, le container crée et gère un pool de stateless session beans, démarrant ainsi son cycle de vie.
- Le container exécute les injections de dépendance s'il y a lieu.
- Il invoque ensuite la méthode annotée par `@PostConstruct` si elle existe.
- Le composant est alors prêt à exécuter les invocations du client.
- A la fin du cycle de vie, le container appelle la méthode annotée par `@PreDestroy`, si elle existe. L'instance peut être alors récupérée par le garbage collector.



Cycle de vie d'un Singleton Session Bean

Comme un stateless session bean, il ne devient jamais passif, son cycle de vie a seulement deux états : non-existant et prêt.

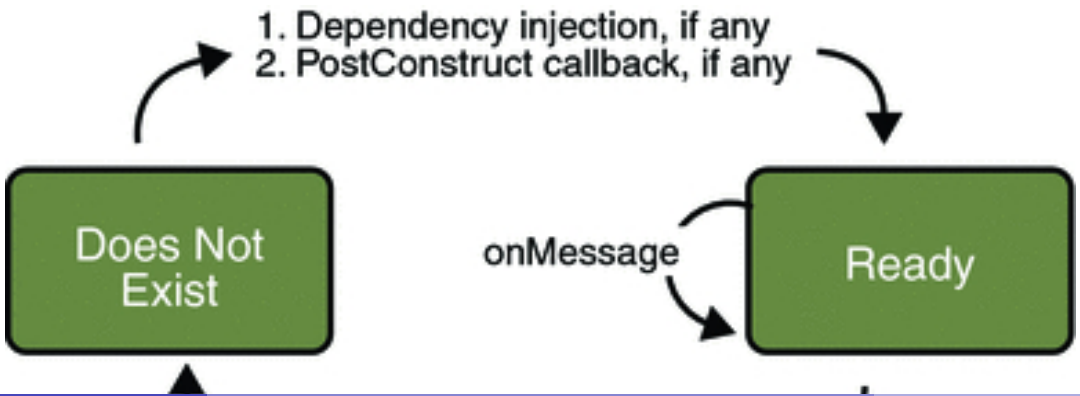
- Le container instancie le composant, démarrant ainsi son cycle de vie. Il le fait au déploiement de l'application si le bean est annoté par `@Startup`.
- Le container exécute les injections de dépendance s'il y a lieu.
- Il invoque ensuite la méthode `@PostConstruct` si elle existe.
- Le composant est alors prêt à exécuter les invocations du client.
- A la fin du cycle de vie, le container appelle la méthode annotée par `@PreDestroy`, si elle existe. L'instance peut être alors récupérée par le garbage collector.



Cycle de vie d'un Message Driven Bean

Comme un stateless session bean, il ne devient jamais passif, son cycle de vie a seulement deux états : non-existant et prêt.

- Généralement, le container crée et gère un pool de message driven beans, démarrant ainsi son cycle de vie.
- Le container exécute les injections de dépendance s'il y a lieu.
- Il invoque ensuite la méthode annotée par `@PostConstruct` si elle existe.
- Le composant est alors prêt à exécuter les invocations du client.
- A la fin du cycle de vie, le container appelle la méthode annotée par `@PreDestroy`, si elle existe. L'instance peut être alors récupérée par le garbage collector.



Représentation d'une donnée manipulée par l'application

- donnée typiquement stockée dans un SGBD (ou tout autre support accessible en JDBC)
  - correspondance objet–tuple relationnel (mapping O/R)
  - possibilité de définir des clés, des relations, des recherches
  - avantage : manipulation d'objets Java plutôt que de requêtes SQL
- mise en oeuvre à l'aide
  - d'annotations Java 5
  - de la généricité Java 5
  - de l'API JPA (Java Persistence API)

annotation @Entity : déclare une classe correspondant à un entity bean (EB)

- chaque classe de EB est mise en correspondance avec une table
  - par défaut table avec même nom que la classe
  - sauf si annotation `@Table(name="...")`
- 2 modes (exclusif) de définition des colonnes des tables
  - property-based access : on annote les méthodes getter
  - field-based access : on annote les attributs
  - par défaut colonne avec même nom que field/property
  - sauf si annotation `@Column(name="...")`
  - annotation `@Id` : définit une clé primaire
  - types supportés
  - primitifs (et leurs types Class correspondants), String, Date

```java
@Entity
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
  public long getId() { return id; }

  public void setId(long id) { this.id = 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; }
}
```

Possibilité de définir des champs auto-incrémentés

- annotation `@GeneratedValue`

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

- `GenerationType.AUTO` : les numéros de séquence sont choisis automatiquement
- `GenerationType.SEQUENCE` : un générateur de numéros de séquence est à fournir

Entity Manager

- assure la correspondance entre les objets Java et les tables relationnelles
- point d'entrée principal dans le service de persistance

- permet d'ajouter des enregistrements
- permet d'exécuter des requêtes

- accessible via une injection de dépendance
  - attribut de type `javax.persistence.EntityManager`
  - annoté par `@PersistenceContext`

Création de trois enregistrements dans la table des livres

```
@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);
  }
}
```

de façon similaire `em.remove(b2)` retire l'enregistrement de la table

Recherche par clé primaire

- méthode `find` du gestionnaire d'entités

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

- retourne `null` si la clé n'existe pas dans la table

- `IllegalArgumentException`
  - si 1er paramètre n'est pas une classe d'EB
  - si 2ème paramètre ne correspond pas au type de la clé primaire

Recherche par requête

- requêtes `SELECT` dans une syntaxe dite EJB-QL étendue
- mot clé `OBJECT` désigne un résultat à retourner sous la forme d'un objet
- paramètres nommés (préfixés par :) pour configurer la requête

```
Query q = em.createQuery("select OBJECT(b) from Book b where b.
    author = :au");
String nom = "Honore de Balzac";
q.setParameter("au",nom);
List<Book> list = (List<Book>) q.getResultList();
```

- méthode `getSingleResult()` pour récupérer un résultat unique
- `NonUniqueResultException` en cas de non unicité

Recherche par requête pré-compilée

- création d'une requête nommée attachée à l'EB

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

- paramètres peuvent être spécifiés
- plusieurs requêtes nommées peuvent être définies

```
@Entity
@NamedQueries(
value={ @NamedQuery("q1","..."), @NamedQuery("q2","...") }
public class Book { ... }
```

Relation 1-n

```
@Entity
public class Author {
  private long id;
  private String name;
  private Collection<Book> books;

  public Author() { books = new ArrayList<Book>(); }

  public Author(String name) { this.name = name; }

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

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

  ...
}
```

Relation 1-n

```
@Entity
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; }

  @ManyToOne
  @JoinColumn(name="Author_id") //nom de la colonne de jointure
  public Author getAuthor() { return author; }

  public void setAuthor(Authro author) { this.author = author;}

  public String getTitle() { return title; }

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

  ...
}
```

Relation n-n : notion de table de jointure

```
@Entity
public 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
public class Item {

  @Id
  @Column(name="ITEM_ID")
  protected long itemId;

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

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;
```

Autres annotations

- `@SecondaryTable` : mapping d'un EB sur plusieurs tables

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

- `@Embeddable`, `@Embedded` : 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;
}
```

Autres annotations

- `@IdClass` : clé composée

```
@Entity
@IdClass(PersonnePK.class)
public class Personne {
  @Id public String getName() { return name; }
  @Id public String getFirstname() { return firstname; }
}

public class PersonnePK implements Serializable {
  private String name;
  private String firstname;
  public PersonnePK( String n, String f ) { ... }
  public boolean equals(Object other) { ... }
  public int hash() { ... }
}
```

Timer bean

- Déclenchement d'actions périodiques
- `@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 initialDuration, long period, Serializable info )`

```
public class EnchereBean {
  @Resource TimerService ts;
  public void ajouterEnchere( EnchereInfo e ) {
    ts.createTimer(1000,25000,e);
  }

  @Timeout
  public void monitorerEnchere( Timer timer ) {
    EnchereInfo e = (EnchereInfo) timer.getInfo();
    ...  }}
```

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 méthode

```
Object <methodname>(InvocationContext ctx) throws Exception
```

- `javax.interceptor.InvocationContext`
  - permet d'obtenir des informations (introspection) sur les méthodes interceptées
  - fournit une méthode `proceed()` pour exécuter la méthode interceptée

Intercepteurs

Plusieurs méthodes dans des classes $\neq$ peuvent être associées à MyInterceptor

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

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();
    //on peut modifier paramètres avec ic.setParameters(...)
    Object ret = ic.proceed(); // Appel du bean (facultatif)
    // ... code après ...
    return ret;
  }
```

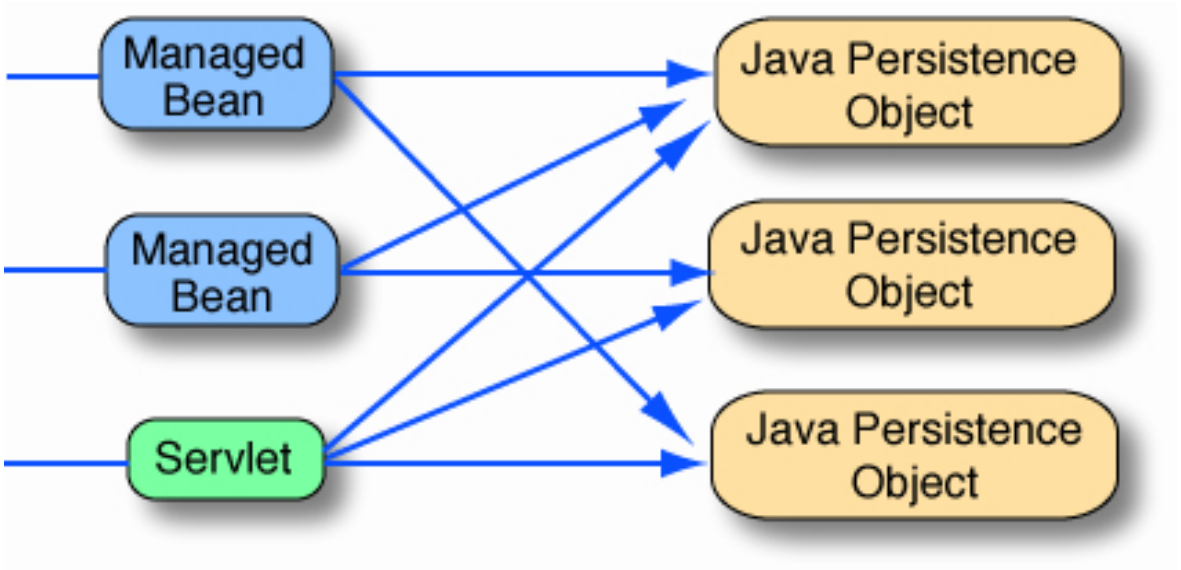
Sommaire

1 Introduction

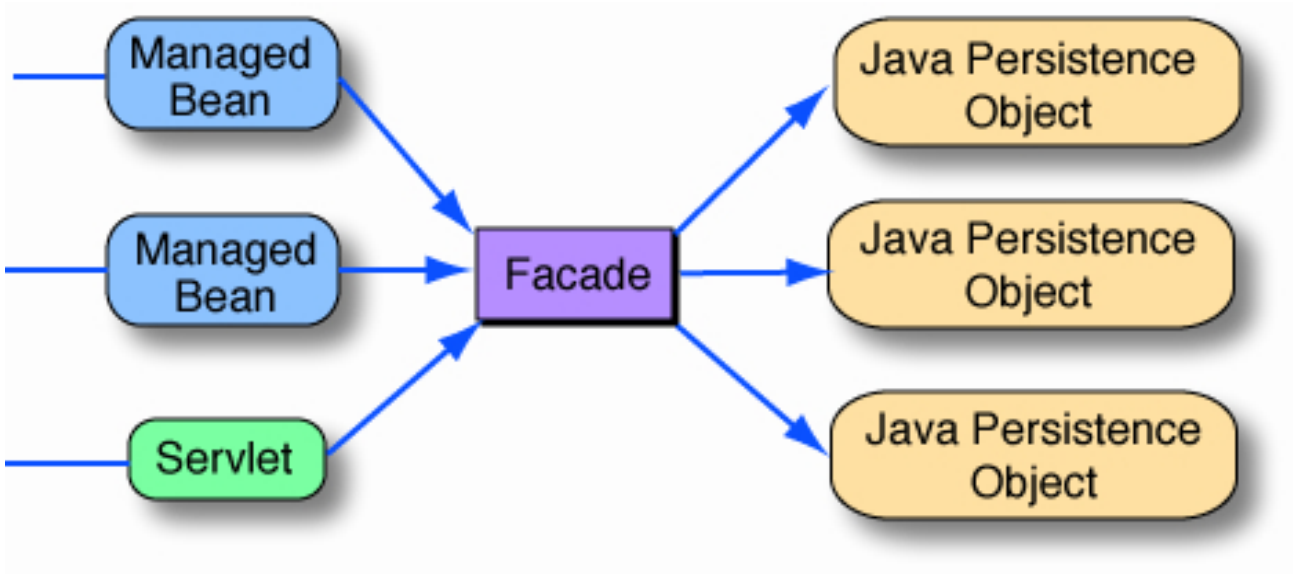
2 Composants EJB

3 Design Patterns EJB
- Session Facade
- Data Transfert Object

- En concevant le code qui accède aux entités persistantes dans le tier modèle, le code client (Servlet, JSP, JSF) accèdera aux API des entités.
- Avec JPA, le client devra se préoccuper de l'API sur les transactions, sur l'EntityManager et d'autres encore nécessaires pour accéder aux entités dans le modèle.
- Le code client doit interagir avec différentes entités chacune d'API différente et peut être ayant des besoins différents quant aux transactions et à l'EntityManager : difficile à gérer car le code client sera fortement couplé au modèle des entités persistantes.



- le DP Facade définit une classe de plus haut niveau qui encapsule et centralise les interactions entre les clients et le modèle d'entités persitantes et les opérations JPA. Il fournit une seule interface pour les opérations d'un ensemble d'entités.
- L'utilisation du DP Facade rend le code du client plus simple, moins couplé au tier modèle et de maintenance plus facile.



Web Component Facade : architecture uniquement web ou on ne veut pas utiliser un container EJB

```
public class CatalogFacade implements ServletContextListener
    {
  @PersistenceUnit(unitName="CatalogPu")
  private EntityManagerFactory emf;

  @Resource UserTransaction utx;
  public CatalogFacade(){}

  public void contextDestroyed(ServletContextEvent sce){
    if(emf.isOpen())
      emf.close();
  }

  public void contextInitialized(ServletContextEvent sce) {
    ServletContext context = sce.getServletContext();
    context.setAttribute("CatalogFacade", this);
  }
```

```
  public void addItem(Item item) throws InvalidItemException {
    EntityManager em = emf.createEntityManager();
    if(item.getName().length() == 0)
      throw new InvalidItemException("The item" + " name cannot be empty.");
    try {
      utx.begin(); em.joinTransaction(); em.persist(item); utx.commit();
    } catch(Exception exe){
      System.err.println("Error persisting item: "+exe);
      try {
        utx.rollback();
      } catch (Exception e) {
        throw new RuntimeException("Error persisting item: "+ e.getMessage(), e);
      }
      throw new RuntimeException("Error persisting item: "+ exe.getMessage(), exe);
    } finally { em.close(); }
  }

  public Item getItem(int itemID){
    EntityManager em = emf.createEntityManager();
    Item item= em.find(Item.class,itemID); em.close(); return item;
  }

  public List getAllItems(){
    EntityManager em = emf.createEntityManager();
    List items = em.createQuery("SELECT OBJECT(i) FROM Item i").getResultList();
    em.close();
    return items;
  }
}
```

```
public class CatalogServlet extends HttpServlet {
  private ServletContext context;
  private CatalogFacade cf;

  public void init(ServletConfig config)throws ServletException {
    context = config.getServletContext();
    cf = (CatalogFacade)context.getAttribute("CatalogFacade");
    initPathMapping();
  }

  public void doGet(HttpServletRequest request, HttpServletResponse response)
      throws ServletException, IOException {
    ...
    try {
      if (selectedURL.equals("additem.do")) {
        String desc = request.getParameter("item_desc");
        String name = request.getParameter("item_name");
        ...
        Item item = new Item(); item.setName(name); item.setDescription(desc);
        ...
        cf.addItem(item); //now call facade
        ...
    }
```

Session Bean Facade : le bean profite des services du container comme par exemple les transactions, le code n'a pas besoin de gérer les transactions en utilisant l'API JTA.

interface du session bean : un bean est par défaut local

```
import java.util.List;

public interface CatalogFacade {
  public void addItem(Item item)throws InvalidItemException;
  public Item getItem(int itemID);
  public List<Item> getAllItems();
}
```

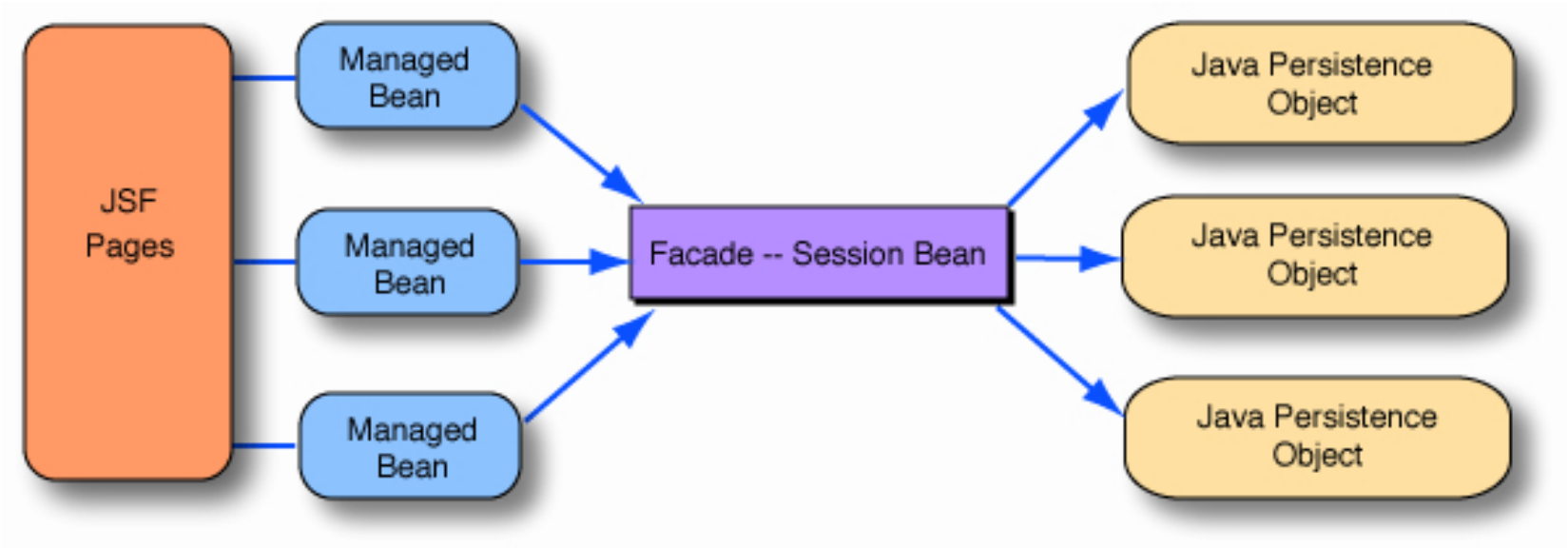
Facade Bean

```
@Stateless
public class CatalogFacadeBean implements CatalogFacade {
  @PersistenceContext(unitName="CatalogPu")
  private EntityManager em;

  public void addItem(Item item) throws InvalidItemException {
    if(item.getName().length() == 0)
      throw new InvalidItemException("The item name cannot be empty.");
    em.persist(item);
  }

  public Item getItem(int itemID) {
    Item item = em.find(Item.class,itemID);
    return item;
  }

  public List<Item> getAllItems() {
    List<Item> items;
    items = em.createQuery("SELECT OBJECT(i) FROM Item i").getResultList();
    return items;
  }
}
```

Composant Web utilisant le bean facade

```
public class CatalogServlet extends HttpServlet {

  @EJB
  private CatalogFacade cf;
  ...

  public void doGet(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException {
    ...
    if ("additem.do".equals(selectedURL)) {
      //get values from request and place them into new Item
      String desc = request.getParameter("item_desc");
      String name = request.getParameter("item_name");
      ...
      Item item = new Item();
      item.setName(name);
      item.setDescription(desc);
      ...
      //use facade to add new Item to database
      cf.addItem(item);
      ...
  }

  ...
}
```

On effectuerait de même avec des JSF : chaque page JSF accède à un JSF managed bean et celui ci accède au session bean facade qui prend en charge tous les accès aux entités persistantes du tier modèle. Le bean facade peut retourner les objets persistants comme des POJO.



Pattern défini à cause des limitations des EJB2, encore pertinent pour les accès distants (mais detached entity)