Cycle Ingénieur de la Spécialité Informatique - 3ème Année

CD/CI

Adeel Ahmad
adeel.ahmad@univ-littoral.fr
Maître de Conférences - section 27,
Directeur des études et Président de Jury Licence 3 Informatique
Responsable Compétences Numériques - site Calais

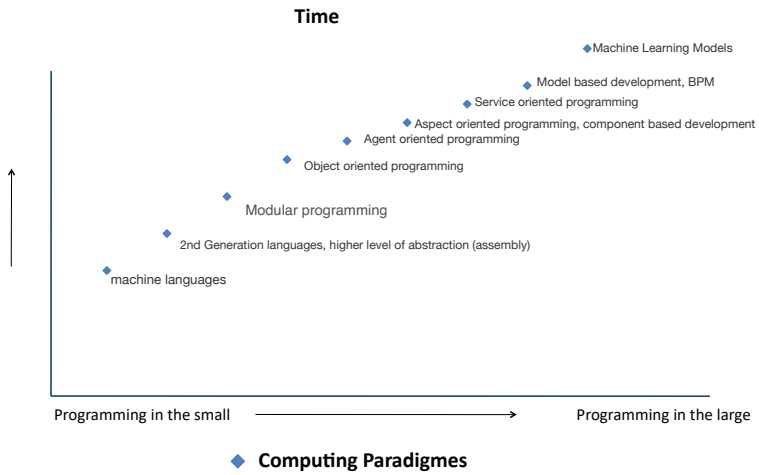
Université du Littoral Côte d'Opale
École d'ingénieurs du Littoral Côte d'Opale
Laboratoire d'Informatique Signal et Images de la Côte d'Opale (LISIC)
Maison de la Recherche Blaise Pascal
50, rue Ferdinand Buisson - BP 719
62228 CALAIS Cedex
FRANCE

la programmation Informatique

- Un programme informatique typique est une succession d'instructions (exécutable par l'ordinateur)
- Un regroupement d'activités
- exécution des opérations étape par étape

procédure, fonction, méthode

Les langages de programmation



Pourquoi l'évolution de langages de programmation?

structured programming,
imperative programming (allows side effects),
functional programming (disallows side effects),
procedural programming (groups code into functions and subroutines),
object-oriented programming (OOP) (groups code together with the
data on which the code works).

- la compréhension?
- la gestion de mémoire?
- le temps d'exécutions?
- le temps de développement des programmes?
- gestion de la logique métiers et la qualité des programmes [documentation, l'utilisabilité (ergonomique), la portabilité, qualité des programmes, normes (tests, pratiques (gestion de versions, interopérabilité, optimisation du code)), etc.]?
- la séquence des instructions?

Programmation Classique

- ensemble des instructions / modules..
  - sequential statements
  - conditional statements
  - iterative statements
  - Modules
  - Programmation procédurale est un meilleur choix qu'une programmation séquentielle à cause de la possibilité de réutiliser le même code, etc..

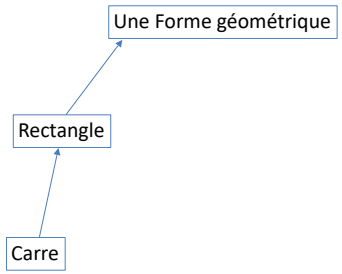
programmation Orientée-Objet

- La programmation objet permet une mutualisation et une unicité des traitements/ procédures/méthodes/opération, etc...

Les caractéristiques de la programmation Orientée-Objet

- l'encapsulation
  - les attributs
  - les méthodes
- l'héritage
  - réutilisation
  - spécialisation/généralization
- polymorphisme / overloading
- l'abstraction
  - Les langages de programmation C#, VB.NET, Objective C, Python, Ruby, C++, Ada, PHP, Smalltalk, Java, etc...

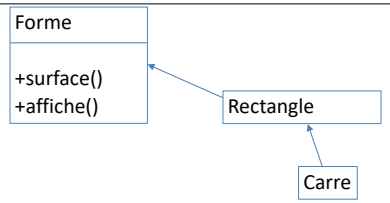
POO (Le concept d'héritage)



exercice

- Spécialité / Généricité
- Définir l'interface qui peut être implémenté par des autres classes (séparer l'implémentation de la spécification)
- Définir une classe qui peut être héritée par sous classes
- Gérer des exceptions

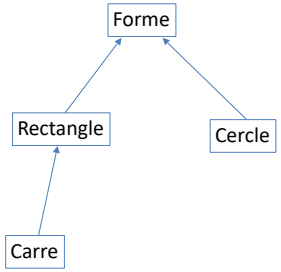

```
public interface Forme{
public int surface();
public void affiche();
}
```



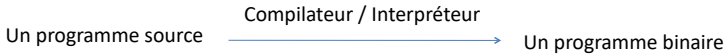

```
public class Rectangle implements Forme {
    private int largeur, longueur;
    public Rectangle(int x, int y) {
        this.largeur = x;
        this.longueur = y;
    }
    public int surface() {
        return this.longueur * this.largeur;
    }
    public void affiche() {
        System.out.println("rectangle " +
        longueur + "x" + largeur);
    }
}
```

```
public class Carre extends Rectangle {
    public Carre(int cote) {
        super(cote, cote);
    }

    public int surface() {
        return super.surface();
    }

    public void affiche() {
        System.out.println("carré " +
        this.getLongueur());
    }
}
```


POO (Le concept d'héritage)

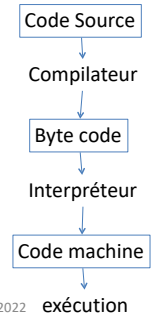


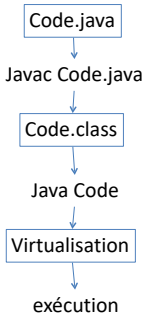
la programmation Informatique

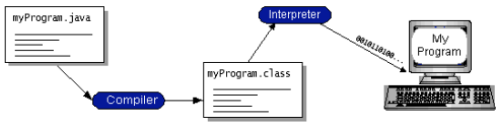


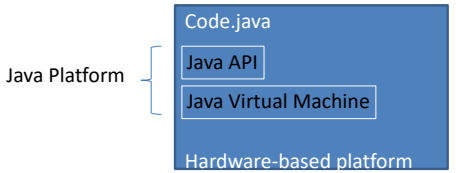
L'environnement de programmation (Java)

- Compilateur
- Interpréteur
- programmation orientée objet en Java

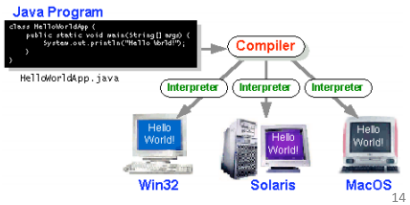


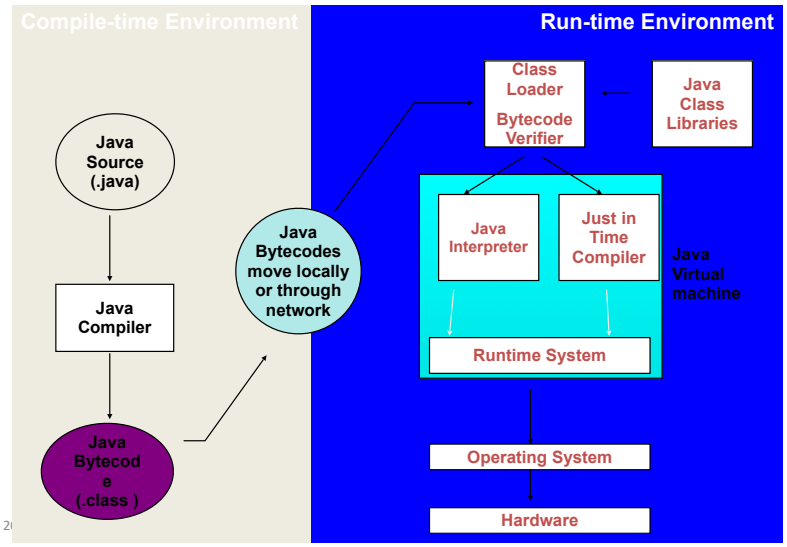






Write Once, Run Anywhere





Architecture d'une application (1)

* Une architecture monolithique ?
    * Application unique
    * Déploiement unique
    * Base de code unifiée
    * Dépendances internes

Avantage :

+ Simplicité de développement
+ Débogage et tests faciles
+ Déploiement rapide

Inconvénients

- Manque de modularité
- Problèmes de scalabilité
- Temps de déploiement et mise à jour
- Limitations technologiques

Architecture d'une application (2)

- Architectures microservices
  - un style de conception de logiciels où une application est décomposée en un ensemble de petits services indépendants
  - chaque fonctionnalité est découplée et développée en tant que service indépendant, ce qui permet une plus grande flexibilité, modularité et scalabilité.
  - Chaque **microservice** correspond à une unité fonctionnelle spécifique de l'application, est autonome, et communique avec les autres microservices par des API (souvent via HTTP ou des messages asynchrones)

Caractéristiques principales des microservices

* Services indépendants
* Responsabilité unique
* Communications via des API
* Autonomie technologique
* Base de données par service
* Scalabilité indépendante

Avantage :

+ **Scalabilité**
+ **Modularité**
+ **Développement rapide**
+ **Résilience**
+ **Facilité de déploiement continu (CI/CD)**

Inconvénients

- Complexité accrue
- Communication interservices
- Gestion des données distribuées
- Surveillance et gestion

Architecture microservices VS architecture monolithique

- Modularité : Contrairement à un monolithe où toutes les fonctions sont regroupées, les microservices sont modulaires, ce qui les rend plus faciles à modifier et à maintenir.

- Scalabilité : Alors qu'un monolithe nécessite de scaler toute l'application, les microservices permettent de faire évoluer indépendamment les services qui en ont besoin.

- Déploiement : Un monolithe nécessite un redéploiement complet de l'application à chaque mise à jour, tandis que les microservices permettent des déploiements indépendants.

Communication entre microservices

- **Pattern REST** : utilisation de requêtes HTTP (GET, POST, PUT, DELETE).
- **Pattern Event-Driven** : utilisation d'événements pour la communication asynchrone.
- **Message Brokers** : RabbitMQ, Apache Kafka pour la communication interservice.
- **Exemple de communication** entre services (exemple de gestion de commande) :
  - Le service "Commande" envoie un événement au service "Stock" pour vérifier la disponibilité.
  - Le service "Paiement" traite la commande une fois le stock validé.

Pattern REST (i)

- Méthodes HTTP dans REST :
  - GET : Récupérer une ressource ou une collection de ressources.
  - POST : Créer une nouvelle ressource.
  - PUT : Mettre à jour une ressource existante ou en créer une nouvelle si elle n'existe pas.
  - DELETE : Supprimer une ressource existante.
  - PATCH : Mettre à jour partiellement une ressource.

Pattern REST (ii)

- Avantages du pattern REST :
  - Simplicité : REST repose sur des protocoles web standard comme HTTP
  - Flexibilité : REST n'impose pas de formats spécifiques pour les données échangées, bien que JSON soit souvent utilisé.
  - Scalabilité : L'approche stateless permet au serveur de traiter de nombreuses requêtes en parallèle sans dépendance entre elles
  - Interopérabilité : REST est compatible avec une grande variété de clients (navigateurs web, applications mobiles, autres services) et fonctionne bien dans des environnements distribués.
  - Cacheabilité : Les réponses cacheables permettent d'optimiser les performances du système en réduisant la charge du serveur.
- Limites du pattern REST :
  - Stateless : Bien que l'absence d'état simplifie l'architecture, cela peut entraîner des charges de communication plus lourdes, car chaque requête doit contenir toutes les informations nécessaires.
  - Pas de support natif pour les transactions : REST ne gère pas directement les transactions complexes qui nécessitent une cohérence entre plusieurs services.
  - Absence de standard pour la gestion des erreurs : Bien que des codes de statut HTTP standard existent, les conventions de gestion des erreurs varient d'une API à l'autre.

Pattern Event-Driven (i)

- Composants principaux de l'architecture event-driven :
  - Émetteur d'événements (Producer) :
    - déclenche un événement lorsqu'une action particulière se produit.
    - Cela peut être un changement d'état, une action utilisateur, un traitement réussi, etc.
  - Événement (Event) :
    - une notification qui signale qu'une action ou un changement d'état s'est produit.
    - Il contient des informations sur ce qui a changé ou ce qui s'est passé. Un événement est souvent représenté sous forme de message.
  - Consommateur d'événements (Consumer) :
    - réagit à un ou plusieurs événements.
    - Il s'abonne aux types d'événements qui l'intéressent et exécute une logique lorsqu'il reçoit un événement.
  - Bus d'événements ou Broker (Event Broker) :
    - Dans un système distribué, les événements sont souvent transférés via un intermédiaire appelé bus d'événements ou message broker (ex. : Apache Kafka, RabbitMQ, AWS SNS).
    - Ce composant gère la diffusion des événements des émetteurs vers les consommateurs.
    - Il permet aux producteurs et consommateurs d'événements de ne pas être directement couplés, facilitant ainsi la communication asynchrone.

Pattern Event-Driven (ii)

- Fonctionnement d'une architecture event-driven :
  - Production de l'événement : Lorsqu'un composant (par exemple, le service de gestion des commandes) détecte qu'une commande est passée, il émet un événement, tel que "Commande créée", qui est envoyé au bus d'événements ou directement à un consommateur.
  - Diffusion de l'événement : Le bus d'événements (ou broker) reçoit l'événement et le transmet à tous les composants abonnés qui sont intéressés par cet événement particulier.
  - Consommation de l'événement : Les composants consommateurs (par exemple, le service de gestion des stocks ou le service de facturation) reçoivent cet événement et déclenchent leur propre logique. Par exemple, le service de gestion des stocks met à jour les niveaux de stock, tandis que le service de facturation génère la facture.
  - Réactions multiples : Un même événement peut déclencher plusieurs actions différentes dans plusieurs services. Par exemple, l'événement "Commande créée" peut également déclencher une notification à l'utilisateur, une mise à jour de l'historique des ventes, etc.

Pattern Event-Driven (iii)

Avantages de l'architecture event-driven :

1. Découplage : Les producteurs et consommateurs d'événements sont faiblement couplés. Le producteur ne connaît pas les détails de mise en œuvre des consommateurs, et vice versa. Cela permet une plus grande flexibilité et évolutivité, car de nouveaux consommateurs peuvent être ajoutés sans modifier le producteur.
2. Scalabilité : Le modèle asynchrone permet aux composants de fonctionner de manière indépendante, ce qui facilite la montée en charge. Les événements peuvent être traités en parallèle, et différents composants peuvent être mis à l'échelle indépendamment en fonction de la charge.
3. Réactivité : Les systèmes event-driven sont très réactifs, car chaque composant réagit immédiatement aux événements, améliorant ainsi la performance globale du système. Cela est particulièrement utile dans les systèmes temps réel.
4. Flexibilité : L'ajout de nouvelles fonctionnalités est plus simple. Par exemple, si vous souhaitez ajouter un nouveau service qui réagit à un événement (par exemple, un service de reporting), vous pouvez le faire sans impacter les autres services.
5. Robustesse : Un composant peut tomber en panne sans affecter tout le système, car les événements sont souvent stockés dans un bus d'événements et peuvent être traités lorsque le consommateur est de nouveau disponible.

Pattern Event-Driven (iv)

Inconvénients de l'architecture event-driven :

1. Complexité accrue : Bien que le découplage apporte de nombreux avantages, il augmente aussi la complexité du système. La gestion des événements asynchrones, des files d'attente et des erreurs est plus complexe qu'avec une communication directe ou synchrone.
2. Difficulté de débogage : Suivre le flux des événements à travers un système distribué peut être compliqué, car les événements peuvent être consommés de manière asynchrone par plusieurs services. Cela peut rendre plus difficile le débogage et la traçabilité des erreurs.
3. Gestion des données distribuées : Dans les architectures complexes, il peut être difficile de gérer la cohérence des données. Des événements peuvent arriver dans un ordre imprévisible, et il faut souvent recourir à des mécanismes pour garantir la cohérence.
4. Dépendance au broker : Le bon fonctionnement du système repose sur le bus d'événements ou le message broker. Si celui-ci connaît des défaillances, l'ensemble du système peut en être impacté.

Pattern Event-Driven (v)

Scénarios où l'architecture event-driven est utile :

1. Systèmes distribués : Les architectures event-driven sont souvent utilisées dans des systèmes distribués, où les composants sont indépendants et doivent échanger des informations de manière asynchrone.
2. Applications nécessitant une réactivité en temps réel : Les systèmes comme les plateformes de trading, les réseaux sociaux, ou les applications de monitoring utilisent souvent cette architecture pour réagir immédiatement à des événements externes.
3. Systèmes évolutifs : Les microservices, les systèmes de traitement des commandes (e-commerce), et les infrastructures de traitement de données en flux bénéficient de la capacité des architectures event-driven à évoluer en fonction de la demande.

Outils et technologies populaires pour les microservices

- **Langages** : Java (Spring Boot), Node.js, Go, Python (Django, Flask).
- **Conteneurisation** : Docker pour l'isolation des services.
- **Orchestration** : Kubernetes pour le déploiement et la gestion des conteneurs.
- **API Management** : Kong, Apigee.
- **Monitoring et observabilité** :
  - Prometheus et Grafana pour la surveillance.
  - Jaeger ou Zipkin pour le traçage distribué.
  - ELK (ElasticSearch, Logstash, Kibana) pour la gestion des logs.

Programmation en Java

- Applications
  - Applications classiques (sur consoles), Applications de réseau (sockets etc.)
  - Applications avec l'interface graphiques
- Scripts et web
  - Applets
  - JSP
  - Servlets
  - JSTL (Java pages Standard Tag Library)
  - JSF, Struts, Spring …
    - Richfaces
    - Primefaces .. Etc.
- J2EE
  - Beans
  - EJBs
  - Persistence, … etc.

Application Graphique en java

```
import javax.swing.*;

public class UnSwing {
        private static void laFrame() {
                JFrame.setDefaultLookAndFeelDecorated(true);

                //Creation de la Frame
                 JFrame frame = new JFrame(« Demo de Swing");
                 frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);

                //Afficher un message
                 JLabel label = new JLabel("Bienvenue en ING3...");
                 frame.getContentPane().add(label);

                //Afficher la fenêtre
                 frame.pack();
                 frame.setVisible(true);
                 }

         public static void main(String[] args) {
                 laFrame();
         }
```


- Importation de packages
- Définition d'un conteneur top-level JFrame, implémenté comme instance de la classe JFrame
- Création/format de ce conteneur
- Définition d'un composant JLabel, implémenté comme instance de JLabel
- Ajout du composant JLabel dans la JFrame
- Définition du comportement de la JFrame si sur un click du bouton de fermeture
- Une méthode main qui crée la JFrame

excercice

- Créer un application en utilisant Jframe qui affiche l'heure et la date du système

J2EE (i)

- Applications d'entreprise à résoudre des problèmes d'entreprises
  - Stockage sécurisé des informations ainsi que leur manipulation et leur traitement
- Un ensemble de spécifications d'API
- Une méthode de packaging et de déploiement des composants
- Une architecture distribuée
- Gestion de communications entre systèmes distants
- Synchronisation de données de plusieurs sources
- S'assurer que le système respecte en permanence les règles de l'activité de l'entreprise (appelées règles « métier »)

J2EE (ii)

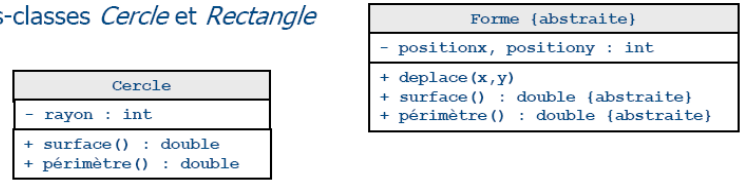
J2EE est une collection de composants, de conteneurs et de services permettant de créer et de déployer des applications distribuées au sein d'une architecture standardisée.

Programmation orientée composant

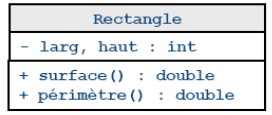
- Une approche modulaire
- Le composant est un ensemble des modules et/ou protocols
- Il sert comme une unité de calcul ou de stockage
- l'interaction entre les composants logiciels
  - API
  - Web-services

Classes abstraites : exemple Forme

- Exemple : la classe *Forme*
  - Les méthodes *surface()* et *périmètre()* sont abstraites
  - Ces méthodes n'ont de « sens » que pour les sous-classes *Cercle* et *Rectangle*




```
public abstract class Forme {
    private int positionx, positiony;

    public void deplacer(double dx, double dy){
        x += dx; y += dy;
    }

    public abstract double périmètre();
    public abstract double surface();
}
```



Pas d'implémentation !!

Exemple de classe abstraite

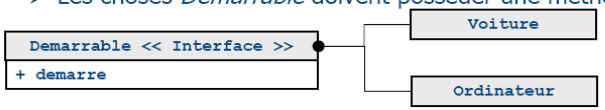
```
class abstract Shape {
    public abstract double perimeter();
}

class Circle extends Shape {
    ...
    public double perimeter() { return 2 * Math.PI * r ; }
}

class Rectangle extends Shape {
    ...
    public double perimeter() { return 2 * (height + width); }
}

...

Shape[] shapes = {new Circle(2), new Rectangle(2,3), new Circle(5)};

double sum_of_perimeters = 0;
for(int i=0; i<shapes.length; i++)
    sum_of_perimeters = shapes[i].perimeter();
```

Les classes abstraites

- Une classe abstraite est une classe ayant au moins une méthode abstraite.
- Une méthode abstraite ne possède pas de définition.
- Une classe abstraite ne peut pas être instanciée.
- Une classe dérivée d'une classe abstraite ne redéfinissant pas toutes les méthodes abstraites est elle-même abstraite.

Les interfaces

- Une interface est un modèle pour une classe
  - Quand toutes les méthodes d'une classe sont abstraites et qu'il n'y a aucun attribut nous aboutissons à la notion d'interface
  - Elle définit la signature des méthodes qui doivent être implémentées dans les classes qui respectent ce modèle
  - Toute classe qui implémente l'interface doit implémenter toutes les méthodes définies par l'interface
  - Tout objet instance d'une classe qui implémente l'interface peut être déclaré comme étant du type de cette interface
  - Les interfaces pourront se dériver
- Exemple
  - Les choses *Demarrable* doivent posséder une méthode *demarre()*



Notion d'interface et Java

- Mise en œuvre d'une interface
  - La définition d'une interface se présente comme celle d'une classe. Le mot clé **interface** est utilisé à la place de **class**

```
public interface NomInterface {
    ...
}
```

Interface : ne pas confondre avec les interfaces graphiques

  - Lorsqu'on définit une classe, on peut préciser qu'elle implémente une ou plusieurs interface(s) donnée(s) en utilisant une fois le mot clé **implements**

```
public class NomClasse implements Interface1,Interface3, ... {
    ...
}
```

  - Si une classe hérite d'une autre classe elle peut également implémenter une ou plusieurs interfaces

```
public class NomClasse extends SuperClasse implements Inter1, ... {
    ...
}
```

Exemple d'interface

```
interface Shape {
    public double perimeter();
}

class Circle implements Shape {
    ...
    public double perimeter() { return 2 * Math.PI * r ; }
}

class Rectangle implements Shape {
    ...
    public double perimeter() { return 2 * (height + width); }
}

...

Shape[] shapes = {new Circle(2), new Rectangle(2,3), new Circle(5)};

double sum_of_perimeters = 0;
for(int i=0; i<shapes.length; i++)
    sum_of_perimeters = shapes[i].perimeter();
```

Notion d'interface et Java

- Mise en œuvre d'une interface
  - Une interface ne possède pas d'attribut
  - Une interface peut posséder des constantes

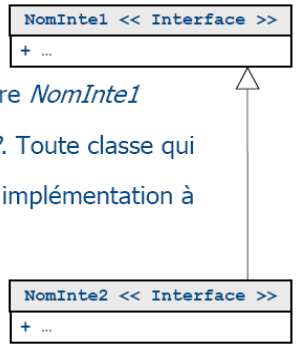
```
public interface NomInterface {
   public static final int CONST = 2;
}
```

  - Une interface ne possède pas de mot clé **abstract**
  - Les interfaces ne sont pas instanciables (Même raisonnement avec les classes abstraites)

```
NomInterface jeTente = new NomInterface(); // Erreur!!
```

Notion d'interface et Java

- Les interfaces pourront se dériver
  - Une interface peut hériter d'une autre interface : « extends »
- Conséquences
  - La définition de méthodes de l'interface mère *NomInte1* sont reprises dans l'interface fille *NomInte2*. Toute classe qui implémente l'interface fille doit donner une implémentation à toutes les méthodes mêmes celle héritées
- Utilisation
  - Lorsqu'un modèle peut se définir en plusieurs sous-modèles complémentaires

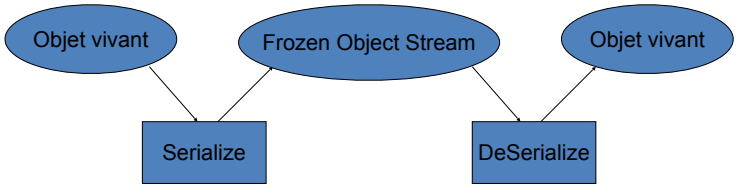


Les interfaces

- Une interface correspond à une classe où toutes les méthodes sont abstraites.
- Une classe peut implémenter (`implements`) une ou plusieurs interfaces tout en héritant (`extends`) d'une classe.
- Une interface peut hériter (`extends`) de plusieurs interfaces.

Serialization (l'interface java.io.Serializable)

- Pour représenter un objet dans un format codé en octets qui peuvent être stockées et transmises à un stream
- Il est reconstruit en cas de besoin



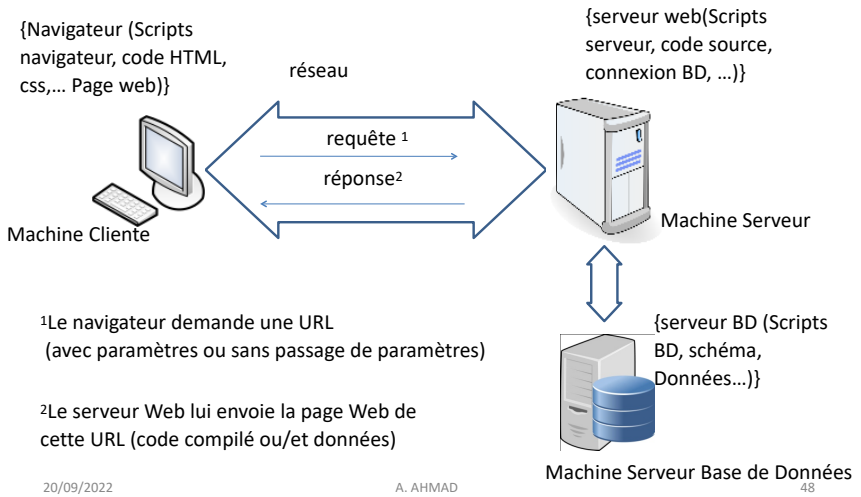
Classes abstraites versus interfaces

- Les classes
  - Elles sont complètement implémentées
  - Une autre classe peut en hériter
- Les classes abstraites
  - Sont partiellement implémentées
  - Une autre classe non abstraite peut en hériter mais doit donner une implémentation aux méthodes abstraites
  - Une autre classe abstraite peut en hériter sans forcément donner une implémentation à toutes les méthodes abstraites
  - Ne peuvent pas être instanciées mais peuvent fournir un constructeur
- Les interfaces
  - Elles ne sont pas implémentées
  - Toute classe qui implémente une ou plusieurs interfaces doit implémenter toutes leurs méthodes (abstraites)

Programmation Web

- L'architecture Client-Server
  - Page Statique (html, des scripts côté client, …)
  - Page Dynamique (des scripts côté serveur, …)
- JSP vs Servlet
  - Code Java
  - Balises JSP
  - Programmation XML, Enterprise Java Beans, connexion BD
- patrons de développement (modèle)

Architecture WEB



[1]Le navigateur demande une URL (avec paramètres ou sans passage de paramètres)

[2]Le serveur Web lui envoie la page Web de cette URL (code compilé ou/et données)

date_time.js

```
function date_time(id)
{
    date = new Date;

    h = date.getHours();

    result = h+':'+m+':'+s;
    document.getElementById(id).innerHTML = result;
    setTimeout('date_time("'+id+'");','1000');
    return true;
}
```

date_time.html

```
<html>
  <head>
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8" />
    <title>Display Date and Time in Javascript</title>
    <script type="text/javascript" src="date_time.js"></script>
  </head>
  <body>
      <span id="date_time"></span>
      <script type="text/javascript">window.onload = date_time('date_time');</script>
  </body>
</html>
```


Les Applications Web J2EE :

- **les servlets**
- **Les JSP**

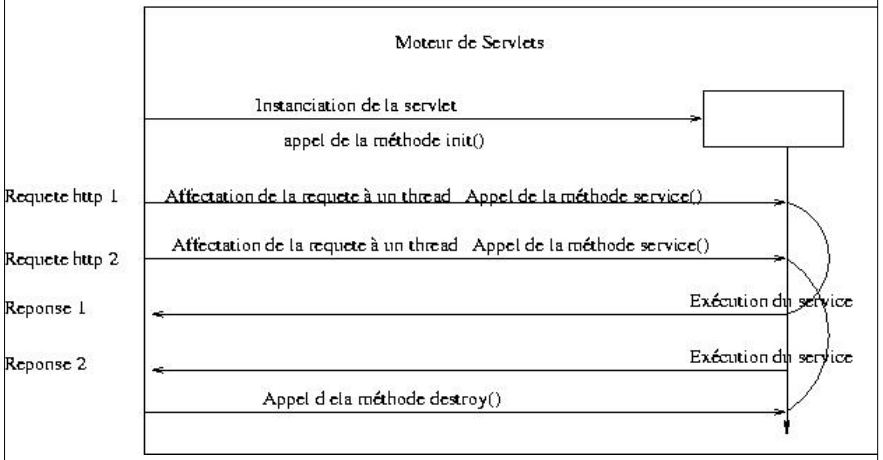
Hello_worldServlet.java

```
Import java.io.*;
Import javax.servlet.*;
Import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {
        public void doGet(HttpServletRequest request, HttpServletResponse response)
 throws IOException, ServletException {

                response.setContentType("text/html");
                PrintWriter out = response.getWriter();
                out.println("<html>");
                out.println("<head>");
                out.println("<title>Hello World!</title>");
                out.println("</head>");
                out.println("<body>");
                out.println("<h1>Hello World!</h1>");
                out.println("</body>");
                out.println("</html>");
        }
}
```


Réponse Java aux CGI

- Exécution côté serveur
- Dynamicité
- Portabilité (serveur Apache)
- Multithreads
- Gratuité (Apache/Tomcat)

Gestion des servlets



Squelette de programme

```
public class Servlet1 extends HttpServlet{

public void doPost( HttpServletRequest req,
      HttpServletResponse res)
            throws ServletException, IOException{

    res.setContentType("text/html");
     PrinterWriter out=res.getWriter();

        out.println("<html>");
       ........
}
```


- Nécessité des 3 paquetages :
  - Import java.io.* ;
  - Import javax.servlet.* ;
  - Import javax.servlet.http.* ;
- Une servlet doit implémenter l'interface **javax.servlet.http**
- La classe à construire implémente HttpServlet

  [qui contient les méthodes init(), service(), doGet(), doPost(), destroy(), etc.]

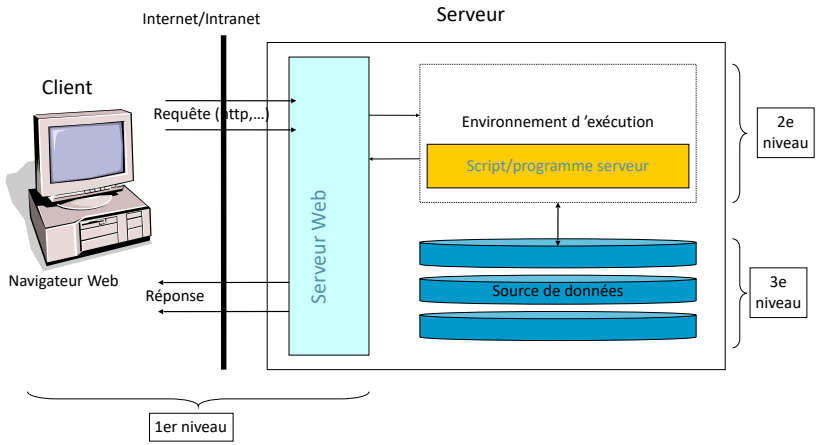
La méthode doPost (ou doGet) doit être implémentée :

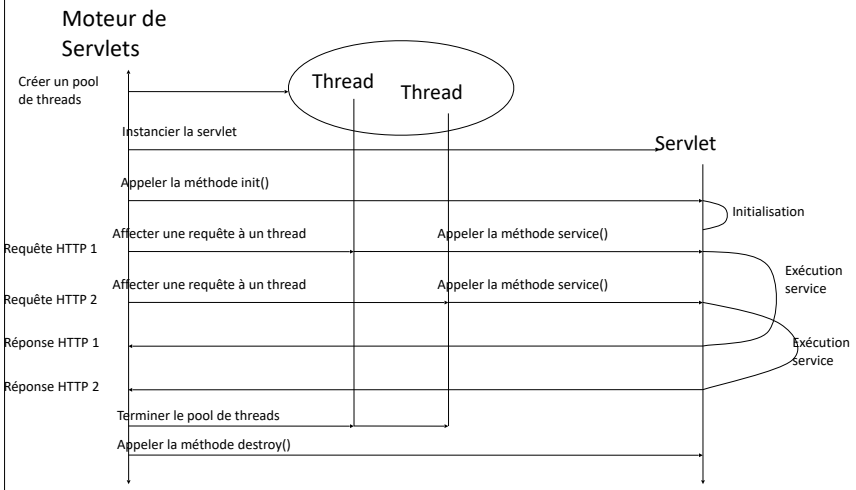
```
public void doPost( HttpServletRequest req,
                    HttpServletResponse res)
    throws ServletException, IOException{
```

doGet() pour les requêtes http de type GET
doPost() pour les requêtes http de type POST

Pour imposer le type du contenu, il faut utiliser une méthode de **HttpServletResponse**, avant d'émettre la réponse par invocation de la méthode de **PrintWriter**

```
res.setContentType( "text/html" );

PrinterWriter out=res.getWriter();

out.println("<HTML><HEAD>...</HTML> ")
```



- Les *servlets* sont le pendant des *applets* du côté serveur
  - mais sans interface graphique utilisateur …
    - elle est limitée à la puissance du langage HTML …
  - par contre, elles ne sont pas astreintes aux mêmes règles de sécurité que les *applets*
    - peuvent établir une connexion avec d'autres clients (RMI, …)
    - peuvent faire des appels système (utilisation pont JDBC-ODBC)
    - ou manipuler des ressources locales (sur le serveur), ...

Définition d'une Servlet

- Servlets
  - Les **servlets** sont des programmes côté serveur permettant d'embarquer la logique applicative dans le processus de réponse aux requêtes HTTP
  - Elles permettent d'étendre les fonctionnalités du serveur Web afin d'intégrer du contenu dynamique dans HTML, XML et autres langages Web
  - Avec J2EE, la spécification relative aux servlets a atteint la version 2.4

Modèle de programmation

- Une servlet doit implémenter l'interface **`javax.servlet.Servlet`**
  - soit directement,
  - soit en dérivant d'une classe implémentant déjà cette interface comme (**GenericServlet ou HttpServlet**)

- Cette interface possède les méthodes pour :
  - initialiser la servlet : **init()**
  - recevoir et répondre aux requêtes des clients : **service()**
  - détruire la servlet et ses ressources : **destroy()**

Structure de base d'une servlet

```
import javax.servlet.*;
public class first implements Servlet {

   public void init(ServletConfig config)
       throws ServletException {…}

   public void service(      ServletRequest req,
                             ServletResponse rep)
       throws ServletException, IOException {…}

   public void destroy() {…}
}
```


Le cycle de vie

1. la *servlet* est créée puis initialisée (**`init()`**)
   - cette méthode n'est appelée par le serveur qu'une seule fois lors du chargement en mémoire par le moteur de servlet
2. le service du client est implémenté (**`service()`**)
   - cette méthode est appelée automatiquement par le serveur à chaque requête de client
3. la *servlet* est détruite (**`destroy()`** **)**
   - cette méthode n'est appelée par le serveur qu'une seule fois à la fin
   - permet de libérer des ressources (allouées par `init()`)

Une servlet Web : `HttpServlet`

- Pour faciliter le traitement particulier des serveurs Web, la classe `Servlet` est affinée en `javax.servlet.http.HttpServlet`
  - 2 méthodes remplacent **service()** de la classe mère :
    - **doGet()** : pour les requêtes Http de type GET
    - **doPost()** : pour les requêtes Http de type POST
  - la classe *servlet* doit obligatoirement contenir l'une ou l'autre de ces 2 méthodes redéfinie, choisie selon le mode d'envoi du formulaire HTML qui l'exécute
  - **service()** de **HttpServlet** appelle automatiquement la bonne méthode en fonction du type de la requête HTTP

Squelette d 'une servlet Http (GET)

```
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleServlet extends HttpServlet {

   public void init(HttpServletConfig c)
        throws ServletException  {…}

   public void doGet(HttpServletRequest req,
                     HttpServletResponse res)
        throws ServletException, IOException {…}

   public void destroy() {…}

   public String getServletInfo() {…}
}
```


Les méthodes doGet() et doPost()

- Utiliser les objets `HttpServletRequest` et `HttpServletResponse` passés en paramètres de ces méthodes pour implémenter le service :
  - `HttpServletRequest` contient les renseignements sur le formulaire HTML initial (utile pour `doPost()`) :
    - la méthode **getParameter()** récupère les paramètres d'entrée
  - `HttpServletResponse` contient le flux de sortie pour la génération de la page HTML résultat (`getWriter()`)

Manipuler les servlets Web

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SomeServlet extends HttpServlet {
  public void doGet(HttpServletRequest  request,
                    HttpServletResponse response)
      throws ServletException, IOException {

    // utiliser "request" pour lire les paramètres et cookies
   request.getParameter("nom du paramètre");
   ...
    // utiliser " response" pour spécifier la réponse HTTP
    // (i.e. spécifier le content type, les cookies).

     PrintWriter out = response.getWriter();
    // utiliser l'objet "out" pour envoyer du contenu au browser
  }
```

Un exemple

```
package hall;

import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

  public void doGet(HttpServletRequest request,
                    HttpServletResponse response)
      throws ServletException, IOException {

    PrintWriter out = response.getWriter();
    out.println("Hello World");
    out.flush();
  }
}
```

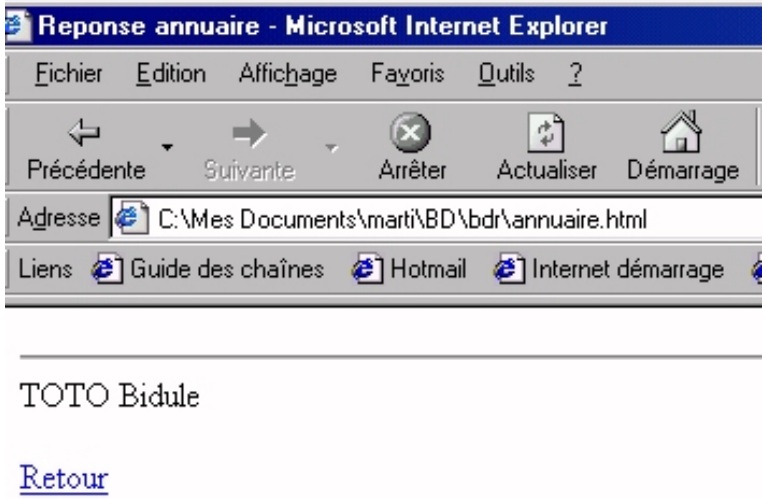

Exemple : annuaire DESS

```
<HTML>
<HEAD><TITLE> ANNUAIRE TIIR </TITLE></HEAD>
<BODY BGCOLOR="#FFFFFF"<CENTER>
<CENTER><H1>ANNUAIRE DU DESS TIIR </H1></CENTER>
<HR><CENTER>
<H2>Recherche de coordonnées </H2></CENTER>
<P> Tapez le début du nom de la personne recherchée:
<P><FORM METHOD=POST
ACTION=http://localhost:8080/examples/servlets/annuaire
   method=post>
<INPUT TYPE=TEXT NAME="nom" SIZE=20  MAXLENGTH=30 VALUE="">
<P><INPUT TYPE=SUBMIT NAME="go" VALUE="RECHERCHER">
<INPUT TYPE=RESET NAME="reset" VALUE="ANNULER">
</FORM>
</BODY></HTML>
```


annuaire DESS (Servlet)

```
import java.io.*;
import java.sql.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class Annuaire extends HttpServlet{
 public void doPost( HttpServletRequest req, HttpServletResponse res)
  throws ServletException, IOException{
   res.setContType("text/html");
   PrintWriter out=res.getWriter();
   out.println("<HEAD><TITLE>Réponse annuaire </TITLE></HEAD><BODY>"
   out.println("<HR>");
   try{
     Class.forName("sun.jdbc.odbc.JdbcOdbcDriver").newInstance();
     String url ="jdbc:odbc:mabase";
     java.sql.Connection c=DriverManager.getConnection(url,"","");
```

```
java.sql.Statement st = c.createStatement();
    java.sql.ResultSet rs =
      st.executeQuery("Select * from matable where nom like '"
        +req.getParameter("nom"+"'*'"));
    rs.next();
    out.println(rs.getString("prenom")+" "+rs.getString("nom") );
   }
   catch (SQLException e){
     out.println("Cette personne n'existe pas");
   }
   out.println("<P><A href = annuaire.html> Retour</A>"</P>);
   out.println("</BODY>");
   out.close();
  }
  public String getServletInfo(){
   return "Servlet Annuaire";
  }
```

Sessions

Garder la mémoire des informations d'une page à l'autre :

- Utiliser les cookies (sécurité!)
- Syntaxe CGI : paramètres dans l'URL
- Champs ``HIDDEN'' de formulaires

<INPUT TYPE=''HIDDEN'' NAME=''PARAM1''VALUE=''VAL1''>

- Objet HttpSession

HttpSession session=request.getSession(true);

Classe objet = (Classe) session.getValue(''param1'');

Méthodes : **getValue(), putValue(), removeValue()**

L'objet session

- Principe :
  - Un objet "session" peut être associé *avec chaque requête*
  - Il va servir de "container" pour des informations persistantes
  - Durée de vie limitée et réglable

Modèle basique

```
HttpSession session = request.getSession(true);
Caddy caddy = (Caddy) session.getValue("caddy");

if(caddy != null) {
   // le caddy n'est pas vide !
  afficheLeContenuDuCaddy(caddy);
} else {
   caddy = new Caddy();
   ...
   caddy.ajouterUnAchat(request.getParameter("NoArticle"));
   session.putValue("caddy", caddy);
}....
```


Méthodes de la classe HttpSession

- **getID()**
- **isNew()**
- **getCreationTime() / getLastAccessedTime()**
- **getMaxInactiveInterval()**
- …
- **getValue(), removeValue(), putValue()**
- …
- **invalidate()**

HttpSession session = req.getSession(true);

If (session.isNew()){

session.putValue(**''toto'', new int[] {0});**

}

Int[] toto = (int[]) session.getValue(''toto'');

Cookies

- l'enregistrement/création d'un cookie avec servlet:
  i. Cookie cookie = new Cookie("key", "value");
     - N'utilisez pas les caractères spéciaux
     - E.g. [] () = , " / ? @ : ;
  ii. Méthode cookie.setMaxAge(sécondes);
     - E.g. cookie.setMaxAge(86400);
       - cookie.setMaxAge(24*60*60);
  iii. response.addCookie(cookie)

Cookies (ii)

- La lecture d'un cookie
  - Méthode request.getcookies() renvoie un tableau d'objets Cookie (dans l'en-tête de la requêt http)
  - Il faut parcourir le tableau et rechercher le cookie à partir de son nom (méthode getName() de l'objet Cookie)

```
Cookie[] cookies = request.getCookies();
String valeur = "";
for(int i=0;i<cookies.length;i++) {
    if(cookies[i].getName().equals("nom")) {
    valeur=cookies[i].getValue();
    }
}
```

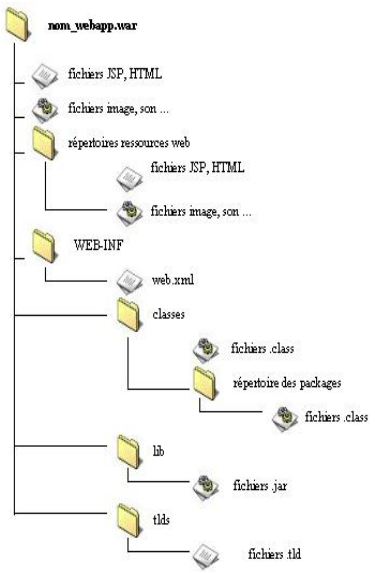

Exemple de Servlet

```
public class essai1 extends HttpServlet{
 public void doPost(HttpServletRequest req, HttpServletResponse res)
        throws ServletException, IOException{
   res.setContentType("text/html");
   PrinterWriter out=res.getWriter();
   out.println("<html>");
   out.println("<head><title>servlet essai</title></head><body><center>");
   out.println("<h1>Test de la SERVLET</h1></center>");
   out.println("<P>retour vers la ");
   out.println("<A HREF="http://localhost:8080/essai.html">page web</A>");
   out.println("</body></html>");r
   RequestDispacter dispatch = req.getRequestDispatcher
    ("essai.html");
 dispatch.forward(req, res);
// req.getRequestDispatcher ("essai.html").forward(req, res);
 }
```


Le descripteur de déploiement : web.xml

" se situe dans le répertoire `WEB-INF/ de l'application Web`

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.2//EN"
    "http://java.sun.com/j2ee/dtds/web-app_2.2.dtd">

<web-app>
    <servlet>
      <servlet-name>HelloWorld</servlet-name>
      <servlet-class>fr.ulco.HelloWorld</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>HelloWorld</servlet-name>
        <url-pattern>/hello</url-pattern>
    </servlet-mapping>

</web-app>
```

Les Applications Web J2EE :

- **les servlets**
- **Les JSP**

Hello_world.jsp

```
<html>
<head><title>Hello World!</title></head>
<body>
<center><h1>Une page JSP </h1>
<h2><%="Hello World!"%></h2><br>
</body>
</html>
```


Java Server Pages

- Du java dans du HTML, entre les balises **<%** et **%>**
- Autant d'éléments de script que l'on souhaite dans une page HTML
- Le script JSP est converti en *servlet* au niveau du serveur

Principes de base et architecture JSP

- But des JSP : simplifier la création et l'administration des pages Web dynamiques, en séparant le contenu de la présentation
- Les pages JSP ne sont que des fichiers contenant du code HTML ou XML standard et d nouvelles balises de script
- Converties en servlet lors de leur toute première invocation

- **L'objectif principal consiste à simplifier la couche de présentation dynamique via un architecture multiniveaux**

- Quatre types de balises:

- Les directives : **<%@ directive ... %>**
- Les déclarations : **<%! ... %>** qui sont séparées par des points-virgules.
- Les expressions évaluables : **<%= ... \%>** sans point-virgule
- Les *scriptlets* : **<% ... \%>** qui regroupent déclarations et instructions

Directives

- <%@ page langage=''java'' %>
- <%@ page import=''java.util.*'' %>
- <%@ page contentType=''text/plain'' %>
- <%@ page errorpage="error.jsp" %>
- <%@ page script="javascript" %>
- <%@ page info="exemple" %>

- include

<%@ include file=''../page.html'' %>

Déclarations

```
<%!
  int i = 0; double c;
  Circle c = new Circle(2.0);
%>
```


Instructions

```
<%= a+b+c %>

<%= nom[i] %>
```

Scriptlets

```
<%
 String nom = “dolores”;
 if request.getParameter("nom") == null)
  {  % >
 Vous avez oublié de donner votre nom.
 Veuillez le saisir pour continuer.
<% } %>
```


JSP

- Du java dans une page WWW !

```
http://www.norsys.fr:8080/jsp/Test.jsp?titre=Les+JSP
...
<I> <%= request.getParameter("titre") %> </I>
...
```

  - Entre les balises JSP <% … %>
- On peut mettre des pages `.jsp` partout où l’on met des pages HTML
- Elles sont converties "au vol" en servlet par le moteur de JSP

Exemple de page JSP

```
<HTML>
<BODY>
 <H1> TEST JSP SIMPLE </H1>
 <BR>
<%
for (int i = 0; i<5; i++)
   out.println("<li>" +i);
%>

</BODY>
</HTML>
```


Un exemple simple

```
<html><head><title>Un exemple de page JSP</title></head><body>
<!-- définit les informations globales a la page -->
<%@page language="java" %>

<!-- Déclare la variable c -->
<%! String c = ""; %>

<!-- Scriplet (code java) %>
<%
   for(int i = 0; i < 26; i++){
        for(int j = 0; j < 26; j++){
          c = "valeur : " + i*j + "<BR>";
%>
<%= c %>
<%       }
%>
<br>
<% }
%>
</body></html>
```

Exemple de page JSP

```
<HTML>
<BODY>
 <H1> TEST JSP SIMPLE </H1>
 <BR>
<%
for (int i = 0; i<5; i++)  { %>

   <li>  <%=i%> </li>
<%}  %>

</BODY>
</HTML>
```


TEST JSP SIMPLE

- 0
- 1
- 2
- 3
- 4

date_time.jsp

```
 <% //programme Java affichant l'heure %>
 <%@ page import="java.util.*" %>

<%// code JAVA pour calculer l'heure
Calendar calendrier=Calendar.getInstance();
int heures=calendrier.get(Calendar.HOUR_OF_DAY);
int minutes=calendrier.get(Calendar.MINUTE);
int secondes=calendrier.get(Calendar.SECOND);
// heures, minutes, secondes sont des variables globales
// qui pourront être utilisées dans le code HTML %>

<% // code HTML %>
<html>
<head><title>Page JSP affichant l'heure</title></head>
<body>
<center><h1>Une page JSP </h1>
<h2>Il est <%=heures%>:<%=minutes%>:<%=secondes%></h2><br>
</body>
</html>
```


Les Applications Web J2EE :

- **les servlets**
- **Les JSP**
- **Les Java Beans**

Les Enterprise JavaBeans

- "J2EE defines the standard for developing component-based multitier enterprise applications. J2EE simplifies building enterprise applications that are portable, scalable, and that integrate easily with legacy applications and data."

[http://java.sun.com/j2ee/]

Le modèle EJB

- Le modèle Enterprise JavaBeans est basé sur le concept *Write Once, Run Everywhere* pour les serveurs.
- Le modèle EJB repose sur l'architecture en couches suivante :
  - L'EJB Server contient l'EJB Container et lui fournit les services de bas niveau.
  - L'EJB Container est l'environnement d'exécution des composants Enterprise JavaBeans (interface entre le bean et l'extérieur).
  - Les clients ne se connectent pas directement au bean, mais à une représentation fournie par le conteneur. Celui-ci route les requêtes vers le bean.

Les règles métier

- Les applications à base d'EJB organisent les règles métiers en composants :
  - une entité métier représente les informations conservées par l'Entreprise
  - un processus métier définit les interactions d'un utilisateur avec des entités métier.
- Les règles business peuvent être extraites et placées dans un moteur de règles (système expert, etc.), puis manipulées *via* un EJB : nouvelles tendances.

L'entité métier

- Elle possède un état, conservé en permanence (SGBD), modifié généralement par les processus métier.
- Exemple:
  - Une entité Commande encapsulera les données des commandes d'un client avec ses règles métiers (i.e. formatage du N° de commande, vérification du compte du client, etc.)

Le processus métier

- Il modifie l'état des entités métier et possède son propre état, souvent provisoire.
- Un processus métier est dédié à un acteur (utilisateur ou programme) qui engage une conversation avec le système :

  processus métier conversationnel
- Exemple :

  Une personne qui retire de l'argent à un distributeur communique avec le DAB au travers de plusieurs écrans. Ceux-ci guident le processus visant à valider la transaction et à distribuer l'argent.

Les types de beans Enterprise

- Les entités et processus métier sont implémentés au choix par trois types de beans :
  - les beans entité : créés pour vivre longtemps et être partagés par plusieurs clients, souvent utilisés pour la gestion des données,
  - les beans session : créés en réponse aux requêtes d'un seul ou de plusieurs clients, souvent utilisés dans les processus.
  - les beans orientés messages pour gérer la communication inter-composants
- Les EJBs orientés messages sont une classe à part des EJBs et sont définis en relation avec l'API Java Message Service

Les beans et les transactions

- Dans une application J2EE, on utilise des transactions pour :
  - combiner l'envoi et la réception de messages (JMS),
  - Effectuer des mises à jours de bases de données et
  - Réaliser d'autres opérations de gestion de ressources (EAI).
- Ces ressources peuvent être accédées à partir de multiples composants d'application à l'intérieur d'une même transaction.
- Par exemple:

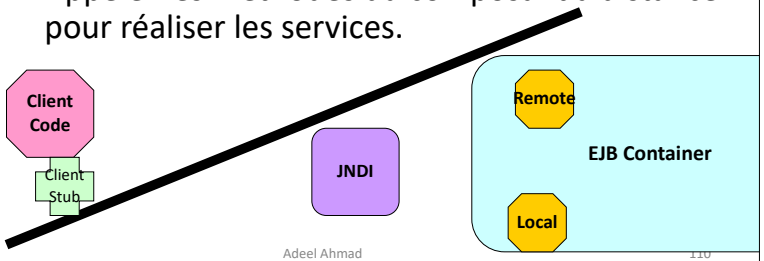
  une servlet peut démarrer une transaction pour accéder à de multiples bases de données, invoquer un enterprise bean qui envoi un message JMS, invoquer un autre enterprise bean pour modifier un ERP en utilisant l'architecture J2EE Connector, et finalement faire un commit de la transaction.

Les beans et les transactions

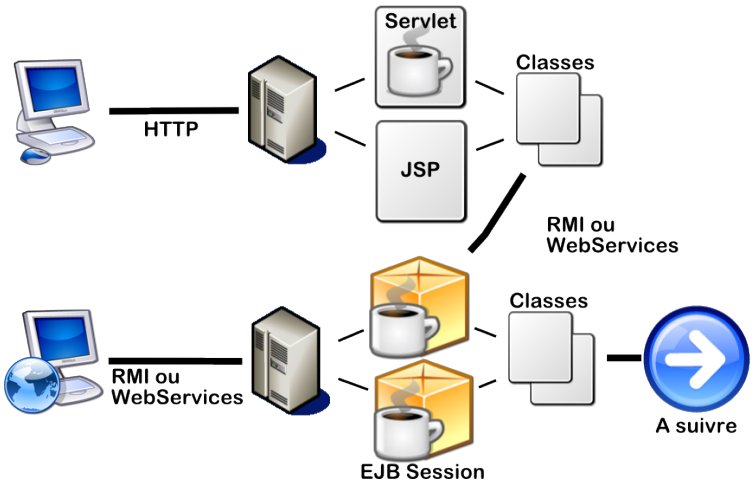
- Les transactions distribuées peuvent être de deux types :
  - **Container-managed transactions.** Le conteneur EJB contrôle l'intégrité de vos transactions sans que vous deviez effectuer un commit ou un rollback.
    - Les CMT sont recommandées pour les applications J2EE qui utilisent JMS.
    - Vous pouvez spécifier des attributs de transaction pour les méthodes des beans.
    - Utiliser l'attribut Required pour s'assurer qu'une méthode fait partie d'une transaction.
    - Lorsqu'une transaction est en cours et qu'une méthode est appelée, celle-ci sera incluse dans la transaction; si aucune transaction n'est en cours, alors une nouvelle transaction sera démarrée avant l'appel de la méthode et sera validée (commit) lorsque la méthode sera terminée.
  - **Bean-managed transactions.** Elles permettent au bean de contrôler finement les transactions via l'interface *javax.transaction.UserTransaction*, permettant d'utiliser ses propres méthodes de commit et de rollback afin de délimiter les frontières des transactions.

Les clients vis-à-vis du conteneur EJB

- Utiliser JNDI pour trouver un composant : Local
  - Java Naming and Directory Interface
  - Abstraction des services de nommage (CosNaming, etc.) et des services d'accès aux structures arborescentes (répertoires, annuaire LDAP, etc.)
- Accès aux services du composant : Remote
- Appeler les méthodes du composant à distance pour réaliser les services.



Architecture EJB 3.x



Bean

- Modélisation de données (serializable)
- Bean Session
- Bean Entité
- Message-Driven Bean
- …

Définition du Bean de Session

- Un bean session représente un simple client J2EE, celui-ci invoque les méthodes du bean pour accéder aux services de l'application J2EE.
- Pour accéder à une application déployée sur un serveur, le client invoque les méthodes du bean session.
- Il réalise le travail du client, en lui cachant la complexité applicative en exécutant des opérations métiers à l'intérieur du serveur.

Type de beans Session?

- Les beans ne sont pas persistants et ne sont pas partagés par les clients.
- Il existe deux types de bean session :
  - Un bean session est utilisé pour mettre en place une communication avec état (stateful) entre un client et un serveur.
  - Un bean session sans état (stateless) ne conserve pas l'état de la communication avec le client.

La session stateful

- ***L'état d'un objet consiste en les valeurs de ses variables d'instance.***
- Dans un bean stateful, les variables d'instance représentent :
  - l'état d'un unique client, car le client interagit ("parle") avec son bean,
  - cet état est appelé l'état *conversationnel*.
- L'état est maintenu pendant la durée de la session du bean client. Si le client supprime le bean (remove) ou quitte la connexion, la session se termine et l'état disparaît.

Quand utiliser les beans session?

- En règle générale, on utilise un bean session dans les circonstances suivantes :
  - A n'importe quel moment, un unique client a accès à l'instance d'un bean.
  - L'état d'un bean n'est pas persistant, car il n'existe que pour une courte période de temps (de l'ordre de quelques heures).

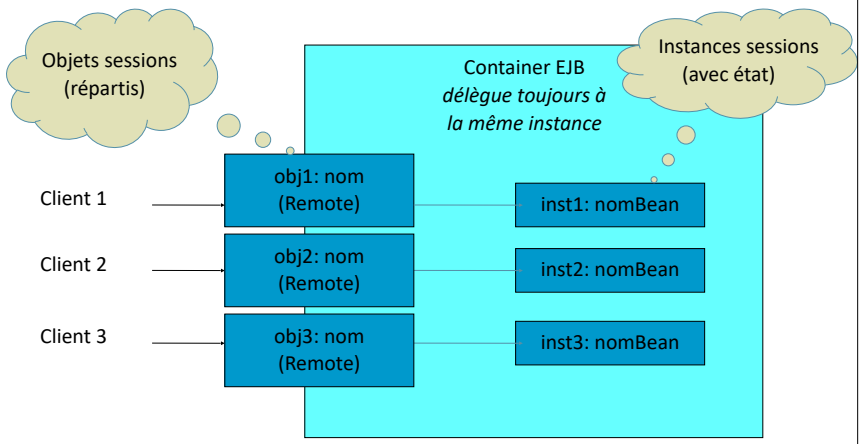
Quand utiliser un bean Session?

- Pour implanter la logique session d'applications Web.
  Par exemple : saisir les renseignements de livraison et de facturation concernant un client *via des pages JSP*.
- Pour implanter la logique session d'applications classiques.
  Par exemple : saisir les renseignements de livraison et de facturation concernant un client, *utilisant JFC/ Swing* (application cliente).

Quand utiliser les beans session Stateful?

- Les beans session stateful sont appropriés dans les conditions suivantes:
  - L'état du bean représente l'interaction entre le bean et un client spécifique.
  - Le bean a besoin de conserver des informations concernant le client à travers les différentes invocations de méthodes.
  - Le bean est un médiateur entre le client et les autres composants de l'application, présentant ainsi une vue simplifiée au client.
  - Derrière la scène, le bean gère le workflow de plusieurs beans enterprises.

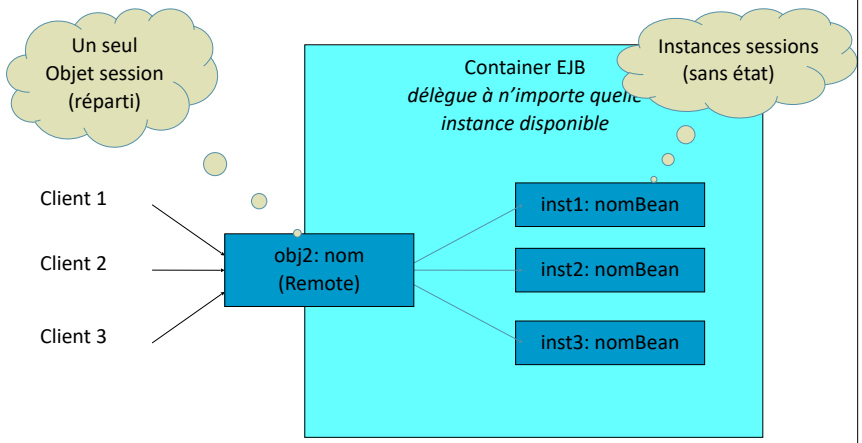
Etat des Beans Sessions stateful



Quand utiliser les beans session Stateless?

- Pour améliorer la performance, on peut choisir un bean stateless pour les raisons suivantes :
  - L'état du bean ne possède pas de données relatives à un client en particulier.
  - Lors d'une simple invocation de méthode, le bean réalise une tâche générique pour tous les clients.
    - *exemple, on peut utiliser une bean stateless pour envoyer un e-mail qui confirme une commande en ligne.*
  - Le bean extrait d'une base de données un ensemble de données en lecture seule, qui est souvent utilisé par les clients (cache).
    - *Un tel bean peut récupérer les lignes d'une table qui représentent les produits qui ont été vendus ce mois.*

Etat des Beans Sessions stateless



La session stateless

- Comme les beans stateless peuvent supporter de multiples clients, ils permettent d'offrir une meilleure extensibilité des applications J2EE.
- En fait, une application J2EE ne nécessite que très peu de beans stateless par rapport aux beans stateful pour supporter la même charge de clients.
- Le conteneur EJB n'écrira jamais de beans stateless en mémoire secondaire. Ainsi, ils peuvent offrir de meilleures performances que les beans stateful.

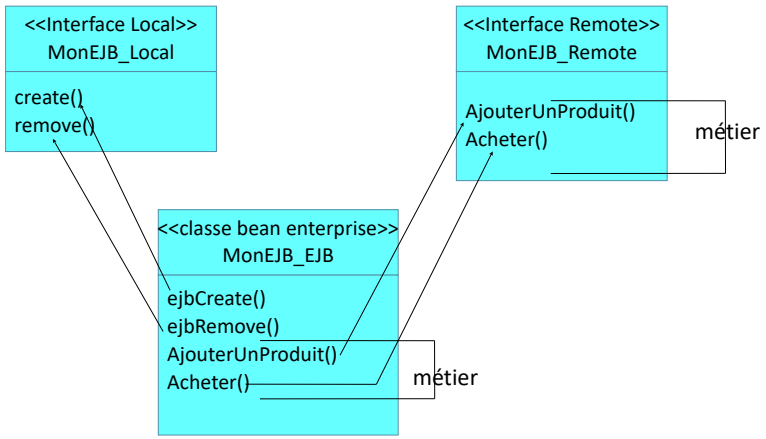
Cycle de vie d'un EJB

- Définir le type de l'EJB :
  - Un EJB session,
  - Un EJB Entité
  - Un EJB message
- Développer le bean
  - Ecrire l'interface Local et l'interface Remote
  - Implémenter les services du bean dans une classe
- Déployer le bean sur un serveur d'applications
  - Créer une description du déploiement (souvent en XML)
  - Nommer l'EJB (souvent un JNDI name)
  - Assembler l'EJB dans un fichier jar (+ librairies, + classes utilitaires)
  - Utiliser l'outil de déploiement du serveur d'applications
- Attendre que l'EJB soit sollicité par une requête.

Structure des Beans Enterprise

- La classe bean enterprise est une classe Java:
  - les méthodes métiers et
  - les méthodes du cycle de vie d 'un objet,
  - nom de la classe : *nomEJB*
- L 'API client-vue :
  - interface Local : create(), remove(), find() (uniquement Entité) et
    - nom de la classe : *nomLocal*
  - interface remote:
    - nom de la classe : *nomRemote* (Sun préconise *nom*)

Structure d'un Bean Enterprise Session



Interface Bean Local

- Elle contrôle le cycle de vie des objets EJBs.
- Les méthodes permettent de créer, de localiser, et de supprimer des EJBs.
- Elles sont héritées des interfaces

javax.ejb.EJBLocal

et

java.rmi.Remote

L'Interface Local

- Associée à un "Objet Local" qui est automatiquement généré par le conteneur EJB
  - Il sera nécessaire de définir les implémentations des méthodes dans la classe de l'EJB:
    - create => ejbCreate
  - La création effective de l 'EJB est réalisée par le constructeur de MonEJB (initialisation de l'état par défaut de l'EJB).
  - La méthode create ne remplace pas le constructeur de l'objet, elle sert uniquement à initialiser et charger les ressources de l'EJB, etc.

Interface Bean Local

- Il peut y avoir plusieurs méthodes *create*() avec un nombre de paramètres variables.
- Un client récupère l'interface Remote du bean créé.
- Les méthodes create retournent soit l'interface Remote, soit une collection de telles interfaces.

Exemple de bean Local

```
import java.rmi.*;
import javax.ejb.*;


public interface MonEJB_Local extends javax.ejb.EJBLocal
{
    public MonEJB_Remote create(String nomclient)
        throws RemoteException, CreateException;
    public void remove(Handle handleEJB)
        throws RemoteException, RemoveException;
}
```


Interface Bean Remote

- Elle définit les méthodes métiers qu'un client peut appeler sur chaque objet bean enterprise.
- Le Bean hérite des interfaces

javax.ejb.EJBObject

et

java.rmi.Remote

L' Interface Remote

- Elle définit les méthodes qui peuvent être appelées par les clients
  - Les méthodes sont appelées sur des "Objets distants" qui sont automatiquement générés par le conteneur EJB
- Pas de conventions de nommage
- Il est obligatoire d'implémenter dans la classe MonEJB les méthodes déclarées dans cette spécification du métier de l'EJB.

Interface Bean Remote

```
import java.rmi.*;
import javax.ejb.*;

public interface MonEJB_Remote extends javax.ejb.EJBObject
{
    boolean AjouterUnProduit(int numeroProduit)
        throws java.rmi.RemoteException;
    boolean Acheter()
       throws java.rmi.RemoteException;
}
```

Classe Bean Enterprise (session)

- Elle spécifie des méthodes à implémenter pour la création, la suppression des objets métiers et les méthodes métiers.
- Les méthodes métiers accèdent aux classes utilitaires (métiers).
- La classe du bean session doit :
  - être déclarée public,
  - contenir un constructeur publique sans paramètres,
  - ne pas être final ou abstract
  - ne pas définir de destructeurs (finalize)
- La classe du bean session implémente l'interface *javax.ejb.SessionBean*

Interface Bean Enterprise

```
import java.rmi.*;
import javax.ejb.*;
public class MonEJB_EJB implements SessionBean
{ public MonEJB_EJB(){
        super();
   }
  public boolean AjouterUnProduit(int numeroItem)
   {    // le code pour ajouter des items au panier
        // peut se connecter via JDBC. }
   public boolean Acheter()
   { // le code pour acheter.}
   public void ejbCreate(String nomClient)
   { // code d 'initialisation de l 'objet   }
   //… plus les méthodes héritées de SessionBean
}
```


Connexion à un bean session

- Le client d'un EJB session devra récupérer l'interface Local de l'EJB *via* JNDI

  InitialContext ic = new InitialContext();

  MonEJB_Local objLocal = (MonEJB_Local) ic.lookup(" nom jndi de ejb ");
- Il devra impérativement effectuer un appel à la méthode create() pour récupérer une référence sur l'EJB (remote).

  MonEJB_Remote objremote = objLocal.create();
- Ensuite il sera possible d'invoquer les méthodes métiers de l'EJB.

  objremote.Acheter();

Les EJB 3.x

- Le développement d'EJB a longtemps été fastidieux, nécessitant même des outils de génération automatique de code basés :
  - sur des modèles UML ou
  - sur du code Java agrémenté de commentaires spécifiques (cf. XDoclet).
- Cette nouvelle mouture élimine de nombreuses redondances tant au niveau du code des composants qu'au niveau de leur configuration.
- Le développement des EJB 3.0 s'exprime au sein du code source par le biais d'annotations (ou "méta-données").
- Le code client a été simplifié au passage, rendant caduque l'utilisation de la notion de **Local** une Abstract Factory de composants qui était implémentée par le conteneur jusqu'à la version EJB 2.1.

Exemple EJB 3.x

```
package service.implementation;

    import java.util.List;
    import javax.ejb.Remote;
    import javax.ejb.Stateless;
    import javax.ejb.TransactionAttribute;
    import javax.ejb.TransactionAttributeType;
    import javax.jws.WebMethod;
    import javax.jws.WebService;
    import javax.jws.soap.SOAPBinding;
    import javax.persistence.EntityManager;
    import javax.persistence.PersistenceContext;
    import business.Signet;
    import business.Categorie;

    @Remote

    @Stateless // Sous entendu : session stateless

    @SOAPBinding
       (style = SOAPBinding.Style.DOCUMENT,
       use = SOAPBinding.Use.LITERAL,
       parameterStyle = SOAPBinding.ParameterStyle.WRAPPED)

    @WebService
       (name = "EndpointInterface",
       targetNamespace = "http://www.deruelle.fr/GestionSignets",
       serviceName = "MoteurRecherche")
```


Exemple EJB 3.x

```
public class MoteurRechercheBean implements Catalog {

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    @WebMethod
    public List<Signet> rechercherTousSignets(){
       // ...
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    @WebMethod
    public String rechercherURL(Signet s){
       // ...
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    @WebMethod
    public List<Signet> rechercherSignets(String categorie){
       // ...
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    @WebMethod
    public List<Signet> rechercherSignetsParTitre(String partieTitre){
       // ...
    }

    @TransactionAttribute(TransactionAttributeType.SUPPORTS)
    @WebMethod
    public List<Signet> rechercherSignetsParURL(String partieURL){
       // ...
    }

}
```


Les annotations pour EJB 3.x

- L'annotation @Remote exprime le fait que notre composant est invocable par RMI
- @Stateless permettra au conteneur de composants d'optimiser le recyclage des instances de composants.
- @SoapBinding et @WebService gèrent le paramétrage protocolaire et l'encodage XML à employer lorsque le composant est sollicité par SOAP
- @WebMethod permet d'indiquer quelles méthodes peuvent être invoquées à distance par le protocole SOAP. Dans notre exemple, nous avons exposé toutes les méthodes; l'interface WSDL sera déduite de ce code par le conteneur d'EJB et reflétera ce choix. **N.B:** Dans la pratique, on n'expose pas systématiquement les méthodes de services dont les types sont complexes et qui nécessitent une adaptation; dans ce cas, il est possible d'ajouter des méthodes dédiées aux invocations via WebServices mais qui prennent en paramètre ou renvoient des types simplifiés (tableaux, objets restreints, types primitifs). Ces méthodes procèdent à l'adaptation de type et invoquent à leur tour les méthodes naturelles du service.
- Enfin @TransactionAttribute permet de régler finement la démarcation transactionnelle des composants : quelle méthode nécessite/requiert/ne supporte pas d'être invoquée dans une transaction. Cette exigence sera interprétée par le conteneur de composants comme la nécessité d'initier une nouvelle transaction, d'enrôler le service dans une transaction existante ou au contraire de lever une exception (ou de suspendre la transaction) si une transaction est en cours mais que le service ne peut pas s'y intégrer sans risque.

La persistance avec les EJB 3.x

- Les techniques pour établir une connexion entre nos services (EJB Session) et une base de données relationnelle :
  - Utiliser JDBC directement.
  - Déclencher depuis les EJB Session une bibliothèque technique appelée "framework de persistance" telle que Hibernate, IBatis, une implémentation de JDO, etc...
  - Enfin, les EJB Session peuvent s'appuyer sur des **EJB Entité**.
- les deux dernières techniques d'accès à la base sont quasi-identiques !

Les EJB Entités 3.x

- Le mapping des Objets persistants et des accès SQL à la base de données sont gérés automatiquement par le conteneur EJB.
- Les annotations guide le conteneur dans le mapping :
  - @Entity spécifie que l'infrastructure d'accès aux données doit rendre cette classe persistante
  - @Id identifie l'attribut qui sera mappé sur la clé primaire (ici, une clé primaire technique)
  - @GeneratedValue délègue au framework la génération et l'affection d'une clé primaire lors du premier enregistrement d'un objet en base. La génération de la clé se fait généralement par le biais d'une séquence en base, d'un attribut entier auto-incrémenté, ou encore par la production d'un GUID.
  - @ManyToOne exprime une partie de l'association entre Catégorie et Signet qui est de type 1-n. Dans la classe Catégorie, on trouve logiquement un autre attribut de type List<Signet> qualifié de @OneToMany.

Exemple EJB Entité 3.0

```
package business;

import java.io.Serializable;
import javax.persistence.Entity;
import javax.persistence.GeneratedValue;
import javax.persistence.Id;
import javax.persistence.ManyToOne;

@Entity
public class Signet implements Serializable {
   @Id
   @GeneratedValue
   private long id;
   public long getId {
      return id;
   }
   public void setId(long value){
      id = value;
   }

   private String titre;
   public String getTitre {
      return titre;
   }
   public void setTitre(String value){
      titre = value;
   }
```


Exemple EJB Entité 3.0

```
   private String url;
   public String getUrl {
      return url;
   }
   public void setUrl(String value){
      url = value;
   }

   @ManyToOne
   private Categorie categorie;
   public Categorie getCategorie {
      return categorie;
   }
   public void setCategorie(Categorie value){
      categorie = value;
   }
}
```


Interface Ejb Session et Entité

- Il suffit de déclarer un attribut de type EntityManager (annoté @PersistenceContext) dans l'EJB Session pour pouvoir interagir avec le gestionnaire de persistance.
- L'annotation permet au conteneur de composant d'injecter automatiquement la dépendance (la référence) vers le bon objet sans que nous ayons à nous en soucier.
- Le langage utilisé n'est pas tout à fait du SQL mais un langage de plus haut niveau appelé Java Persistance Query Language.
- L'avantage notable est :
  - évite toute dépendance vis-à-vis des dialectes SQL, des multiples bases relationnelles,
  - offre une syntaxe cohérente pour interroger des graphes d'objets pouvant tirer parti de relations d'héritage complexes.
- L'inconvénient découle du premier avantage :
  - le développeur a moins de maîtrise du code SQL qui sera finalement envoyé à la base de données,
  - ce qui peut avoir un impact sur les performances, la robustesse et plus tard la maintenance de l'application.

Interface Ejb Session et Entité

```
public class MoteurRechercheBean implements Catalog {
   @PersistenceContext
   private EntityManager mgr;

   @TransactionAttribute(TransactionAttributeType.SUPPORTS)
   @WebMethod
   public List<Signet> rechercherTousSignets(){
      return mgr
         .createQuery("select s from Signet s")
         .getResultList();
   }

   @TransactionAttribute(TransactionAttributeType.SUPPORTS)
   @WebMethod
   public String rechercherURL(Signet s){
      return mgr
         .createQuery("select s.url from Signet s where s.id = :id")
         .setParameter("id", s.getId())
         .getUniqueResult();
   }
```


Connexion aux sources de données

- Les Enterprise JavaBeans et les applications Web ont nécessairement besoin d'accéder à des systèmes de gestion de bases de données relationnelles.
- Deux possibilités:
  - Accès via JDBC à la base de données, ce qui implique de charger les drivers JDBC, de coder l'URL de la base de données et de transmettre le login/pasword.
  - Accès via DataSource, permet de décharger l'application de la logique de connexion à la base de données. Le serveur d'applications est paramétré pour définir un DataSource qui contient le driver JDBC, l'URL, le login et password.

DataSource

- Une **DataSource** (package *javax.sql*) est une interface représentant une « source de données ».
- Cette « source de données » est en fait une simple fabrique de connexions vers la source de données physique.
- Ce mécanisme, apparu avec **JDBC 3.0**, est désormais préféré au **DriverManager**.
- En général, une DataSource est utilisée (appelée) via **JNDI** (Java Naming and Directory Interface). Mais, ce n'est pas une obligation, c'est simplement ce qui est le plus couramment rencontré.

Exemple de connexion via JDBC

```
Connection connection = null;
String driver = "com.mysql.jdbc.Driver";
String connexionURL="jdbc:mysql://localhost:3306/une_base_de_donnees";
String User = ”ahmad";
String password = "ascffef67";
try {
    Class.forName(driver);
    connection = DriverManager.getConnection(connexionURL, user,
                                                    password);
}catch (SQLException ex1)
{
    System.out.println("SQL Error in setting datasource : " + ex1);
      ex1.printStackTrace();
}
```

Connexion JBoss/Mysql via DataSource

- Le DataSource est basé sur :
  - Le driver JDBC
  - L'URL de connexion
- Extraire l'archive mysql-connector-java-[version]-stable-bin.jar pour installer le driver JDBC
- Placer le JAR dans %JBOSS_DIST%/server/[nom_du_serveur]/lib/.
- Dans %JBOSS_DIST%/server/[nom_du_serveur]/deploy/, il faut créer un fichier mysql-ds.xml (le nom se termine par -ds.xml)
- mysql-ds.xml peut contenir autant de connexions que requis.

Exemple de fichier datasource

```
<?xml version="1.0" encoding="UTF-8"?>
    <!-- Datasource config for MySQL using 2.0.11 driver -->
<datasources>
    <local-tx-datasource>
            <jndi-name>premiere_source_de_donnees</jndi-name>
            <connection-url>jdbc:mysql://localhost:3306/une_base_de_donnees
            </connection-url>
            <driver-class>com.mysql.jdbc.Driver
            </driver-class>
            <user-name>nom
            </user-name>
            <password>mot_de_passe
            </password>
    </local-tx-datasource>

    <local-tx-datasource>
            <jndi-name>deuxieme_source</jndi-name>
            …
    </local-tx-datasource>
</datasources>
```


Exemple de localisation d'une source de données

```
DataSource ds = null;
Connection connection = null;
String dataSourceName = "premiere_source_de_donnees";
      try
      {
         InitialContext it = new InitialContext();
         ds = (DataSource) it.lookup(dataSourceName);
         connection = ds.getConnection();
      }
      catch (SQLException ex1)
      {
         System.out.println("SQL Error in setting datasource : " + ex1);
         ex1.printStackTrace();

      }
      catch (NamingException ex2)
      {
         System.out.println("Error in setting datasource : " + ex2);
         ex2.printStackTrace();
      }
```


persistence.xml

```
<persistence>
  <persistence-unit name="myapp">
    <provider>org.hibernate.ejb.HibernatePersistence</provider>
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      property name="hibernate.dialect"
            value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

persistence.xml

```
<?xml version="1.0" encoding="UTF-8"?>

<persistence xmlns="http://java.sun.com/xml/ns/persistence"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xsi:schemaLocation="http://java.sun.com/xml/persistence/persistence
                        http://java.sun.com/xml/persistence/persistence_2_0.xsd"
        version="2.0">


        <persistence-unit name="managerCategorie"
                transaction-type="JTA">
                <jta-data-source>java:/HelloDS</jta-data-source>
                <class>fr.ulco.model.EntityBeanClass</class>
                <properties>
                        <property name="hibernate.dialect" value="org.hibernate.dialect.MySQLDialect" />
                        <property name="hibernate.hbm2ddl.auto" value="update" />
                        <!-- <property name="hibernate.hbm2ddl.auto" value="create" /> -->
                        <property name="hibernate.show_sql" value="true" />
                </properties>
        </persistence-unit>

</persistence>
```


Entity Manager

```
@PersistenceContext
EntityManager em;

Customer customer = new Customer ();
// populate data in customer

// Save the newly created customer object to DB
em.persist (customer);

// Increase age by 1 and auto save to database
customer.setAge (customer.getAge() + 1);

// delete the customer and its related objects from the DB
em.remove (customer);
//em.merge(...),

// Get all customer records with age > 30 from the DB
List <Customer> customers = em.query (
    "select c from Customer where c.age > 30");

//// CriteraBuiler ...
```


ORM – JPA 2.2

```
@Entity
public class MyEntity {

   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   @Column(name = "id", updatable = false, nullable = false)
   private Long id;

   @Column
   private LocalDate date;

   @Column
   private LocalDateTime dateTime;

   ...
}
```


Présentation de J2EE

- Java 2 Platform, Enterprise Edition est une spécification parue officiellement à la conférence JavaOne en 1998 en même temps que les Enterprise JavaBeans.
- Il est né de besoins des entreprises pour développer des applications complexes distribuées et ouvertes sur l'Internet, exploitables dans un Intranet.
- Il possède une infrastructure d' exécution appelée serveurs d'applications

Présentation de J2EE

- Java 2 Platform, Enterprise Edition inclut les API fondamentales Java suivantes:
  - Les Enterprise JavaBeans (EJB) : composants métiers
  - Les JavaServer Pages (JSP) et les Servlets : composants Web
  - Java DataBase Connectivity (JDBC) pour l'accès aux bases de données relationnelles.

Présentation de J2EE

- Java Transaction Service (JTS) permet d'accéder à un service de transactions réparties.
- L'API Java Transaction (JTA) fournit une démarcation des transactions dans une application.
- Java Message Service (JMS) : pour accéder à divers services de messageries intra-applications et inter-applications.
- Java Naming and Directory Interface (JNDI) fournit un accès aux services de dénomination, DNS, LDAP.
- Remote Method Invocation (RMI) sur Internet Inter-ORB Protocol (IIOP) permet une invocation de méthodes à distance au-dessus du protocole IIOP de CORBA.

La plate-forme J2EE

- Serveurs d'applications J2EE :
  - *Un **serveur d'applications** est un serveur sur lequel sont installées les applications utilisées par les usagers.*
  - *Ces applications sont chargées sur le serveur d'applications et accédées à distance, souvent par réseau.*
  - *Les IHM (Interfaces Hommes-Machines) sont distribuées sur les postes clients* ou via un client léger
  - Dans une infrastructure N-tiers régulière, on peut déployer plusieurs serveurs d'applications:
    - répartir la charge lorsque le nombre élevé de postes clients est une exigence critique
    - redonder lorsque leur disponibilité est une exigence critique

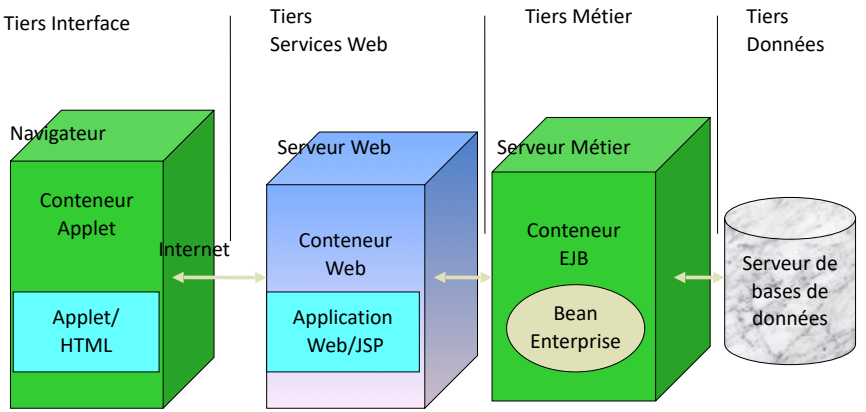
La plate-forme J2EE

- *Les serveurs d'applications sont des logiciels occupant la couche centrale dans une **architecture multicouche:***
  - *Une architecture classique 3-tiers (postes clients, serveur de données, serveur d'applications)*
  - *Une architecture étendue (n-tiers) lorsqu'elle intègre des serveurs d'acquisition (données de terrain, données de process, de back-office, de gateways, de systèmes coopérants externes, etc.).*

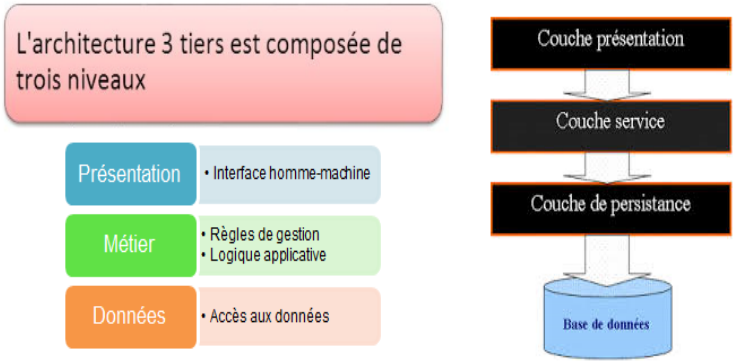
La plate-forme J2EE

- Environnement d'exécution de J2EE
  - J2EE regroupe un certain nombre d'API, mais il présente également la caractéristique remarquable de faire abstraction de l'**infrastructure d'exécution**
    - => Juste une spécification, ensuite implantée par les éditeurs logiciels qui mettent au point les serveurs d'applications
  - Informatique distribuée "traditionnelle" = souvent problèmes liés non pas à la logique propre à l'application mais à la mise en œuvre de services complexes (threading, transactions, sécurité…)
  - J2EE introduit la notion de **conteneur**, et via les **API J2EE**, il élabore un **contrat** entre le conteneur et les applications
  - C'est le vendeur du conteneur qui se charge de mettre en œuvre les services pour les développeurs d'applications J2EE, dans le respect des **standards**

J2EE dans les architectures multi-tiers

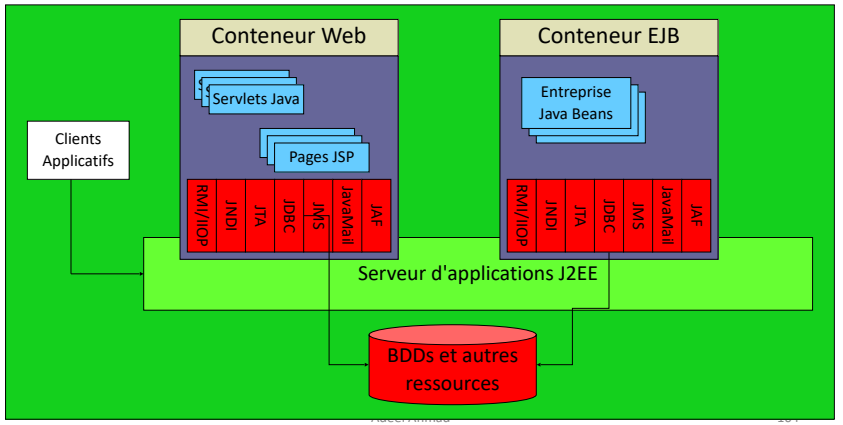


Architecture 3 tiers du schéma technique de l'application



Architecture J2EE - Conteneurs

- Un conteneur J2EE est un environnement d'exécution chargé de gérer des composants applicatifs et de donner un accès aux API J2EE



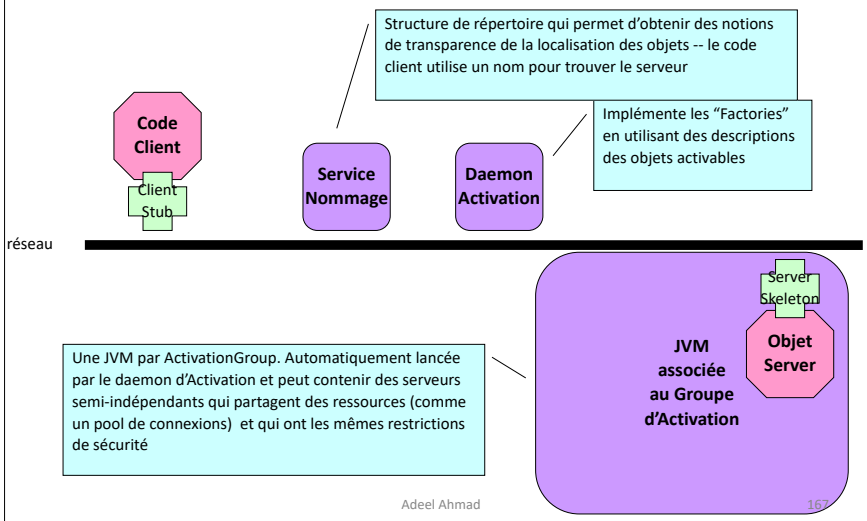
Le container Web

- Il fournit un environnement pour le développement, le déploiement et la gestion de l'exécution des *Servlets* et des *JSP*.
- Les Servlets et les JSP sont regroupés dans des unités de déploiement baptisées *applications Web (webapp)*.
- Les WebApp implémentent la logique de présentation d'une application.

Les services du container EJB

- L'EJB Container est responsable de la fourniture de services aux beans, quelque soit leurs implémentations :
  - Le support du mode transactionnel : spécifié lors de la déclaration du bean sans ajout de codes. La granularité pouvant descendre au niveau de la méthode.
  - La gestion des multiples instances : les EJB sont développés de façon mono-client et exécutée en mode multi-clients:
    - gestion de pool d'instances,
    - gestion de cache,
    - optimisation des accès ressources et données, *etc.*
  - La persistance (obligatoire dans la spécification EJB 2.0).
  - La sécurité par les ACL (Access Control List) au niveau du bean ou pour chaque méthode.
  - Gestion de versions et administration des EJBs.

Les conteneurs *via* Java RMI



Architecture J2EE - Conteneurs

- Quelques services des conteneurs
  - Gestion de la durée de vie des composants applicatifs
    - Cette gestion implique la création de nouvelles instances de composants applicatifs ainsi que le pooling et la destruction de ces composants lorsque les instances ne sont plus nécessaires
  - Pooling de ressources
    - Les conteneurs peuvent à l'occasion mettre en œuvre le rassemblement des ressources, sous la forme, par exemple, de pooling d'objets ou de pooling de connections
  - Peuplement de l'espace de noms JNDI avec les objets nécessaires à l'utilisation des API de services des conteneurs
  - Clustering sur plusieurs machines
    - Répartition de charge ou "Load Balancing"
  - Sécurité
  - ...

Architecture MVC

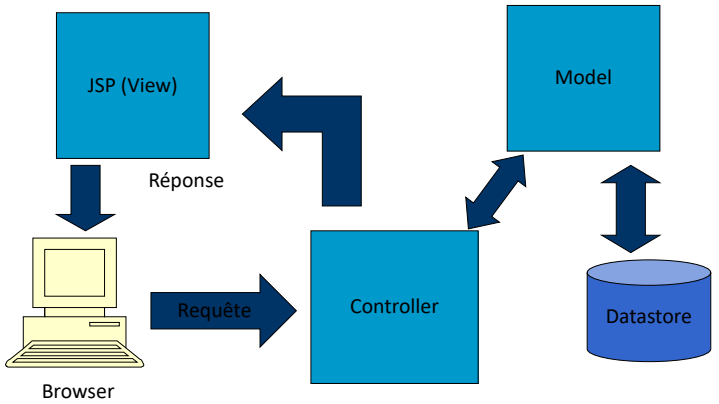
Architecture MVC

- L'architecture modèle 1: la logique métier, la logique d'affichage et la manipulation des requêtes sont mélangées dans un même composant
- L'architecture modèle 2 ou MVC: sépare la logique métier de l'affichage
  - Un composant est chargé de recevoir les requêtes
  - Une autre de traiter les données
  - Et un troisième de préparer l'affichage

MVC

- Modèle – Le modèle englobe à la fois la logique métier et les données sur lesquelles il opère.
- Vue – Une fois la requête traitée, le contrôleur détermine quel composant doit être employé pour afficher les données
- Contrôleur – Les composants de cette catégorie reçoivent les requêtes des clients, les traitent et les transmettent aux composants chargés de traiter les données. Il des dirigent ensuite vers le composants responsables de la vue.

Le pattern Model-View-Controller (MVC 2)

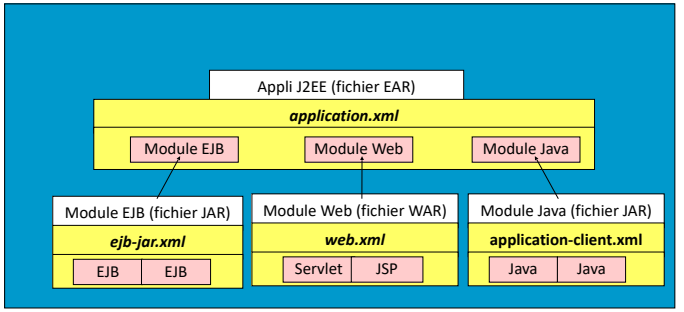


Développer des applications J2EE

- Constitution de composants applicatifs en modules
  - Un module sert a empaqueter un ou plusieurs composants du même type
  - L'application J2EE est une archive **EAR** contenant le descripteur de déploiement (**application.xml**), les modules Web et EJB
  - Les modules Web incluent :
    - Servlets, JSP, TagLibs, JARs, HTML, XML, Images...
    - Empaquetés dans un fichier d'archive web, **WAR**
    - Un WAR s'apparente à un JAR avec en plus un répertoire WEB-INF contenant le descripteur de déploiement **web.xml**
  - Les modules EJB incluent :
    - Les EJB (codes compilés) et leurs descripteurs de déploiement (**ejb-jar.xml**)
    - Empaquetés dans une archive **JAR**
  - Les modules Java du client :
    - pour les clients Java, également une archive **JAR** avec le descripteur de déploiement **application-client.xml**

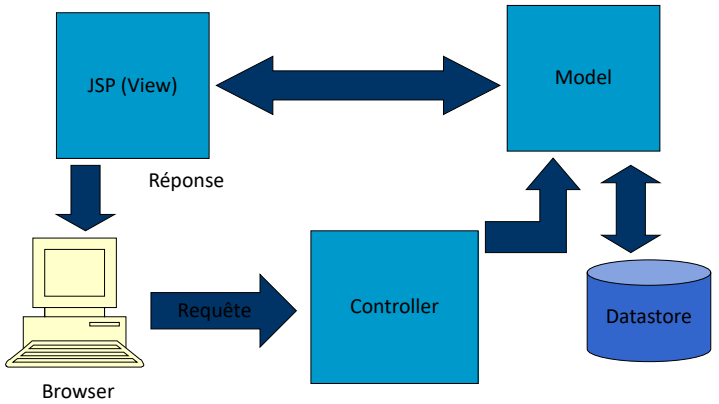
Développer des applications J2EE

- Constitution de modules en application
  - Niveau le plus accompli : celui des applications
  - Appli J2EE = ensemble de modules placés dans un fichier EAR (Entreprise Archive)



Développer des applications J2EE

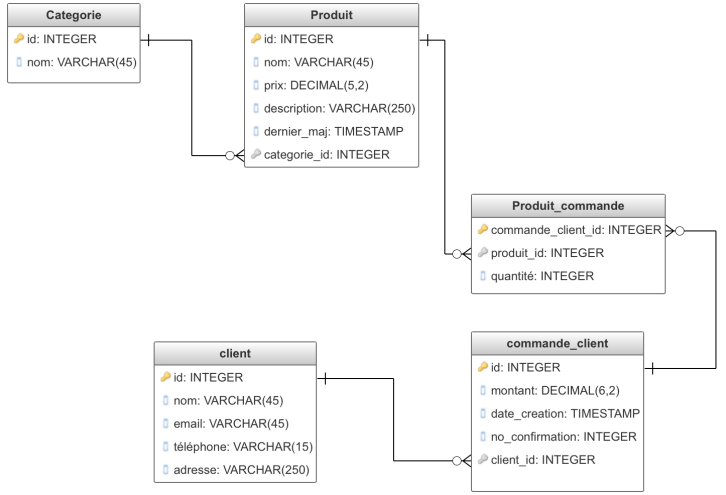
- Déploiement d'applications
  - Le déploiement consiste à installer et à personnaliser des modules empaquetés sur une plate-forme J2EE
  - Deux étapes :
    - Préparation de l'application (recopie des fichiers JAR, WAR, EAR..., génération des classes au moyen du conteneur, puis installation sur le serveur
    - Configuration de l'application en utilisant les informations spécifiques au serveur d'applications
      - Création de sources de données, fabriques de connexion...



Travail Pratique

- Développement d'un module de gestion du catalogue en ligne :
  - Affichage du catalogue (client), ajout et suppression de produits (administrateur)
  - Choix d'un article à ajouter au panier (client)
  - Visualisation du panier (client)
  - Saisie données du client (login/password => coordonnées client)
  - Validation du panier et génération de la commande (client)

Le schéma de la base de données



Décomposition d'une application de e-commerce en microservices (i)

Identification des **domaines fonctionnels** dans une application e-commerce.

- Gestion des produits
- Gestion des utilisateurs et authentification
- Gestion des commandes
- Paiements
- Gestion des stocks
- Notifications (e-mails, SMS)
- Recommandations de produits

Décomposition d'une application de e-commerce en microservices (ii)

Décomposition en **microservices indépendants** :

- Service de catalogue produit
- Service d'authentification
- Service de panier
- Service de gestion de commande
- Service de paiement
- Service de gestion de stock

Décomposition d'une application de e-commerce en microservices (iii)

Communication entre microservices

- **Pattern REST** : utilisation de requêtes HTTP (GET, POST, PUT, DELETE).
- **Pattern Event-Driven** : utilisation d'événements pour la communication asynchrone.
- **Message Brokers** : RabbitMQ, Apache Kafka pour la communication interservice.
- **Exemple de communication** entre services (exemple de gestion de commande) :
  - Le service "Commande" envoie un événement au service "Stock" pour vérifier la disponibilité.
  - Le service "Paiement" traite la commande une fois le stock validé.

Décomposition d'une application de e-commerce en microservices (iv)

Gestion des données et transactions dans les microservices

- **Data Isolation** : chaque service a sa propre base de données.
- **Pattern SAGA** pour la gestion des transactions distribuées.
- **Exemple d'un flux de transaction** dans une application e-commerce :
  - Commande -> Paiement -> Stock -> Expédition
  - Comment gérer les annulations et échecs à chaque étape (rollback via SAGA).

Déploiement et gestion des microservices

- **CI/CD** (Intégration Continue / Déploiement Continu) : pipelines CI/CD pour automatiser les tests et les déploiements.
- **Blue-Green Deployment** et **Canary Releases** pour les déploiements sans interruption.
- **Rollbacks** et stratégies de récupération en cas d'échec.
- Mise en place du repo **Git** + pipeline CI basique (compilation + tests).
- CI/CD : GitHub Actions, GitLab CI, ou Jenkins.
- **Dockerisation et build automatisé d'images**.
- **Déploiement continu avec Docker Compose**.
- **Orchestration avec Kubernetes (initiation)**.
- **Utilisation de Kubernetes** pour la gestion des services
- **Monitoring et logs dans le pipeline CI/CD**.

Code du Service Catalogue Produit (simplifié)

```
// src/main/java/com/ecommerce/catalog/Product.java
package com.ecommerce.catalog;

import jakarta.persistence.*;
import lombok.*;

@Entity
@Data
@NoArgsConstructor
@AllArgsConstructor
@Builder
public class Product {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String description;
    private Double price;
    private Date lastUpdate;
}
```

```java
// src/main/java/com/ecommerce/catalog/ProductRepository.java
package com.ecommerce.catalog;
import org.springframework.data.jpa.repository.JpaRepository;

public interface ProductRepository extends JpaRepository<Product, Long> {
}
```



```java
// src/main/java/com/ecommerce/catalog/ProductController.java
package com.ecommerce.catalog;

import org.springframework.web.bind.annotation.*;
import java.util.List;

@RestController
@RequestMapping("/products")
public class ProductController {

    private final ProductRepository repo;

    public ProductController(ProductRepository repo) {
        this.repo = repo;
    }
    @GetMapping
    public List<Product> getAll() {
        return repo.findAll();
    }

    @PostMapping
    public Product add(@RequestBody Product p) {
        return repo.save(p);
    }
}
```


Structure du projet

ecomm	erce-catalog/
├──	pom.xml
├──	src/
│	├── main/java/com/ecommerce/catalog/...
│	├── main/resources/application.properties
│	└── test/java/com/ecommerce/catalog/...
├──	Dockerfile
└──	.github/workflows/ci.yml

pom.xml

```xml
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>

    <groupId>com.ecommerce</groupId>
    <artifactId>catalog</artifactId>
    <version>0.0.1-SNAPSHOT</version>
    <packaging>jar</packaging>

    <properties>
        <java.version>17</java.version>
        <spring.boot.version>3.3.4</spring.boot.version>
    </properties>

    <dependencies>
        <!-- Spring Boot -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>

        <!-- H2 Database (dev/test) -->
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
            <scope>runtime</scope>
        </dependency>

        <!-- Lombok -->
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>

        <!-- Tests -->
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>
    </dependencies>

    <build>
        <plugins>
            <!-- Spring Boot plugin -->
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
</project>
```

Dockerfile

```
# Étape 1 : Build avec Maven intégré
FROM maven:3.9.6-eclipse-temurin-17 AS build
WORKDIR /workspace

# Copier uniquement pom.xml et télécharger dépendances pour optimiser cache
COPY pom.xml .
RUN mvn dependency:go-offline

# Copier le code source
COPY src ./src

# Construire l'application (JAR Spring Boot)
RUN mvn clean package -DskipTests

# Étape 2 : Image finale légère
FROM eclipse-temurin:17-jdk-alpine
WORKDIR /app

# Copier le jar généré
COPY --from=build /workspace/target/catalog-0.0.1-SNAPSHOT.jar app.jar

# Exposer le port
EXPOSE 8080

# Lancer l'application
ENTRYPOINT ["java", "-jar", "app.jar"]
```


Pipeline CI (GitHub Actions)
.github/workflows/ci.yml

```
name: CI - Catalog Service

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]

jobs:
  build:
    runs-on: ubuntu-latest

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '17'

      - name: Build and test with Maven
        run: mvn clean verify

  docker:
    runs-on: ubuntu-latest
    needs: build

    steps:
      - name: Checkout
        uses: actions/checkout@v4

      - name: Log in to DockerHub
        uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_HUB_USERNAME }}
          password: ${{ secrets.DOCKER_HUB_TOKEN }}

      - name: Build and push Docker image
        uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ secrets.DOCKER_HUB_USERNAME }}/catalog-service:latest
```


Pipeline CD (déploiement continu) (i)
Option A – Déploiement avec Docker Compose (sur un serveur VPS)
.github/workflows/cd.y

```
name: CD - Deploy to VPS

on:
  push:
    branches: [ "main" ]

jobs:
  deploy:
    runs-on: ubuntu-latest

    steps:
    - name: Deploy to VPS via SSH
      uses: appleboy/ssh-action@v1.2.0
      with:
        host: ${{ secrets.VPS_HOST }}
        username: ${{ secrets.VPS_USER }}
        key: ${{ secrets.VPS_SSH_KEY }}
        script: |
          docker pull ${{ secrets.DOCKER_HUB_USERNAME }}/catalog-service:latest
          docker stop catalog || true
          docker rm catalog || true
          docker run -d --name catalog -p 8080:8080 ${{ secrets.DOCKER_HUB_USERNAME }}/catalog-service:latest
```


Pipeline CD (déploiement continu)(ii)
Option B – Déploiement avec Kubernetes
k8s/catalog-deployment.yml

```
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
spec:
  replicas: 2
  selector:
    matchLabels:
      app: catalog
  template:
    metadata:
      labels:
        app: catalog
    spec:
      containers:
      - name: catalog
        image: adeelhub/catalog-service:latest
        ports:
        - containerPort: 8080
---
apiVersion: v1
kind: Service
metadata:
  name: catalog-service
spec:
  type: LoadBalancer
  selector:
    app: catalog
  ports:
    - port: 80
      targetPort: 8080
```

IntelliJ intégration Docker

Dans IntelliJ IDEA Ultimate :

1. File > Settings > Plugins > Docker → installer.
2. Settings > Build, Execution, Deployment > Docker :
   - Ajouter un serveur Docker (Unix Socket ou TCP).
3. Clic droit sur le Dockerfile → Run → IntelliJ va construire et lancer ton conteneur.

Workflow dev → CI/CD avec IntelliJ

- **Étape 1** : Dev du code → mvn clean install depuis IntelliJ.
- **Étape 2** : Test local → lancer Spring Boot depuis IntelliJ ou Docker.
- **Étape 3** : Commit + Push → GitHub déclenche le pipeline CI.
- **Étape 4** : Image Docker construite → envoyée sur Docker Hub.
- **Étape 5 (optionnel)** : Déploiement automatique (CD) sur ton VPS/Kubernetes.

docker-compose.yml

```
version: "3.9"

services:
 catalog-db:
  image: mysql:8.0
  container_name: catalog-db
  restart: always
  environment:
   MYSQL_ROOT_PASSWORD: root
   MYSQL_DATABASE: catalogdb
   MYSQL_USER: cataloguser
   MYSQL_PASSWORD: catalogpass
  ports:
   - "3306:3306"
  volumes:
   - catalog_data:/var/lib/mysql

 catalog-service:
  image: adeelhub/catalog-service:latest # image DockerHub
  container_name: catalog-service
  depends_on:
   - catalog-db
  ports:
   - "8080:8080"
  environment:
   SPRING_DATASOURCE_URL: jdbc:mysql://catalog-db:3306/catalogdb
   SPRING_DATASOURCE_USERNAME: cataloguser
   SPRING_DATASOURCE_PASSWORD: catalogpass
   SPRING_JPA_HIBERNATE_DDL_AUTO: update
   SPRING_JPA_SHOW_SQL: "true"

volumes:
 catalog_data:
```


Lancer avec IntelliJ

1. Dans **IntelliJ**, installe le plugin **Docker**.
2. Ouvre docker-compose.yml.
3. Clic droit → **Run 'docker-compose up'**.
4. IntelliJ va lancer la DB + microservice.

http://localhost:8080/products

Exemple de application.properties (au cas où, lancer sans Docker)

```
spring.datasource.url=jdbc:mysql://localhost:3306/catalogdb
spring.datasource.username=cataloguser
spring.datasource.password=catalogpass
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true
```


pipeline CI/CD GitHub Actions

- Tester non seulement le code du microservice,
- mais aussi sa connexion à une vraie base de données avant le déploiement.

Préparer les tests d'intégration
src/test/java/com/ecommerce/catalog/ProductIntegrationTest.java

```
package com.ecommerce.catalog;

import org.junit.jupiter.api.Test;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.boot.test.context.SpringBootTest;
import org.springframework.test.annotation.DirtiesContext;
import org.springframework.test.context.ActiveProfiles;
import static org.assertj.core.api.Assertions.assertThat;

@SpringBootTest
@DirtiesContext(classMode = DirtiesContext.ClassMode.AFTER_EACH_TEST_METHOD)
@ActiveProfiles("test")
class ProductIntegrationTest {

    @Autowired
    private ProductRepository repo;

    @Test
    void testAddProduct() {
        Product p = Product.builder()
                .name("Laptop")
                .description("Gaming Laptop")
                .price(1200.0)
                .stock(10)
                .build();

        Product saved = repo.save(p);

        assertThat(saved.getId()).isNotNull();
        assertThat(repo.findAll()).hasSize(1);
    }
}
```


CI/CD avec docker-compose .github/workflows/ci-integration.yml

```
name: CI - Integration Tests with Docker Compose

on:
  push:
    branches: ["main"]
  pull_request:
    branches: ["main"]

jobs:
  integration-tests:
    runs-on: ubuntu-latest

    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_DB: catalogdb
          POSTGRES_USER: cataloguser
          POSTGRES_PASSWORD: catalogpass
        ports:
          - 5432:5432
        options: >-
          --health-cmd "pg_isready -U cataloguser -d catalogdb"
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5

    steps:
      - name: Checkout repository
        uses: actions/checkout@v4

      - name: Set up JDK 17
        uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '17'

      - name: Wait for DB to be ready
        run: |
          echo "Waiting for PostgreSQL..."
          for i in {1..10}; do
            nc -z localhost 5432 && echo "Postgres is up!" && break
            echo "Waiting..."
            sleep 5
          done

      - name: Run integration tests
        run: mvn clean verify -Dspring.profiles.active=test
```

Pipeline complet en 3 jobs (CI/CD) (i)

- **build-test** → compile + tests unitaires
- **integration-tests** → lance Postgres + microservice → tests d'intégration
- **docker-build-push** → construit & push l'image Docker

Pipeline complet en 3 jobs (CI/CD) (ii)
.github/workflows/ci-cd.yml

```yaml
name: CI/CD - Catalog Service

on:
 push:
  branches: ["main"]

jobs:
 build-test:
  runs-on: ubuntu-latest
  steps:
   - uses: actions/checkout@v4
   - uses: actions/setup-java@v4
     with:
       distribution: 'temurin'
       java-version: '17'
   - run: mvn clean test

 integration-tests:
  runs-on: ubuntu-latest
  needs: build-test
  services:
   postgres:
     image: postgres:16
     env:
       POSTGRES_DB: catalogdb
       POSTGRES_USER: cataloguser
       POSTGRES_PASSWORD: catalogpass
     ports:
       - 5432:5432
     options: >-
       --health-cmd "pg_isready -U cataloguser -d catalogdb"
       --health-interval 10s
       --health-timeout 5s
       --health-retries 5
  steps:
   - uses: actions/checkout@v4
   - uses: actions/setup-java@v4
     with:
       distribution: 'temurin'
       java-version: '17'
   - run: mvn verify -Dspring.profiles.active=test

 docker-build-push:
  runs-on: ubuntu-latest
  needs: integration-tests
  steps:
   - uses: actions/checkout@v4
   - uses: docker/login-action@v3
     with:
       username: ${{ secrets.DOCKER_HUB_USERNAME }}
       password: ${{ secrets.DOCKER_HUB_TOKEN }}
   - uses: docker/build-push-action@v6
     with:
       context: .
       push: true
       tags: ${{ secrets.DOCKER_HUB_USERNAME }}/catalog-service:latest
```


Déploiement Continu (CD) sur Kubernetes
Préparer les manifests Kubernetes
k8s/catalog-deployment.yml

```yaml
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-service
  labels:
    app: catalog
spec:
  replicas: 2
  selector:
    matchLabels:
      app: catalog
  template:
    metadata:
      labels:
        app: catalog
    spec:
      containers:
        - name: catalog
          image: adeelhub/catalog-service:latest # DockerHub
          ports:
            - containerPort: 8080
          env:
            - name: SPRING_DATASOURCE_URL
              value: jdbc:postgresql://catalog-db:5432/catalogdb
            - name: SPRING_DATASOURCE_USERNAME
              value: cataloguser
            - name: SPRING_DATASOURCE_PASSWORD
              value: catalogpass
---
apiVersion: v1
kind: Service
metadata:
  name: catalog-service
spec:
  selector:
    app: catalog
  type: LoadBalancer
  ports:
    - port: 80
      targetPort: 8080
```


Déploiement Continu (CD) sur Kubernetes
Préparer les manifests Kubernetes
k8s/catalog-db.yml (PostgreSQL en cluster)

```yaml
apiVersion: v1
kind: PersistentVolumeClaim
metadata:
  name: postgres-pvc
spec:
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 1Gi
---
apiVersion: apps/v1
kind: Deployment
metadata:
  name: catalog-db
spec:
  replicas: 1
  selector:
    matchLabels:
      app: catalog-db
  template:
    metadata:
      labels:
        app: catalog-db
    spec:
      containers:
        - name: postgres
          image: postgres:16
          ports:
            - containerPort: 5432
          env:
            - name: POSTGRES_DB
              value: catalogdb
            - name: POSTGRES_USER
              value: cataloguser
            - name: POSTGRES_PASSWORD
              value: catalogpass
          volumeMounts:
            - mountPath: /var/lib/postgresql/data
              name: postgres-storage
      volumes:
        - name: postgres-storage
          persistentVolumeClaim:
            claimName: postgres-pvc
---
apiVersion: v1
kind: Service
metadata:
  name: catalog-db
spec:
  selector:
    app: catalog-db
  ports:
    - port: 5432
      targetPort: 5432
```

Étendre pipeline CI/CD .github/workflows/ci-cd.yml

```yaml
name: CI/CD - Catalog Service

on:
  push:
    branches: ["main"]

jobs:
  build-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '17'
      - run: mvn clean test

  integration-tests:
    runs-on: ubuntu-latest
    needs: build-test
    services:
      postgres:
        image: postgres:16
        env:
          POSTGRES_DB: catalogdb
          POSTGRES_USER: cataloguser
          POSTGRES_PASSWORD: catalogpass
        ports:
          - 5432:5432
        options: >-
          --health-cmd "pg_isready -U cataloguser -d catalogdb"
          --health-interval 10s
          --health-timeout 5s
          --health-retries 5
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-java@v4
        with:
          distribution: 'temurin'
          java-version: '17'
      - run: mvn verify -Dspring.profiles.active=test

  docker-build-push:
    runs-on: ubuntu-latest
    needs: integration-tests
    steps:
      - uses: actions/checkout@v4
      - uses: docker/login-action@v3
        with:
          username: ${{ secrets.DOCKER_HUB_USERNAME }}
          password: ${{ secrets.DOCKER_HUB_TOKEN }}
      - uses: docker/build-push-action@v6
        with:
          context: .
          push: true
          tags: ${{ secrets.DOCKER_HUB_USERNAME }}/catalog-service:latest

  deploy-k8s:
    runs-on: ubuntu-latest
    needs: docker-build-push
    steps:
      - uses: actions/checkout@v4

      - name: Set up kubectl
        uses: azure/setup-kubectl@v4
        with:
          version: 'v1.30.0'

      - name: Configure kubeconfig
        run: |
          echo "${{ secrets.KUBE_CONFIG_DATA }}" | base64 --decode > $HOME/.kube/config

      - name: Deploy to Kubernetes
        run: |
          kubectl apply -f k8s/catalog-db.yml
          kubectl apply -f k8s/catalog-deployment.yml
          kubectl rollout status deployment/catalog-service
```


Code du Service Catalogue Produit (simplifié)

- **API Gateway** : un point d'entrée unique pour tous les microservices.
- **Circuit Breaker** : gère la défaillance des services pour éviter la surcharge.
- **Service Discovery** : permet aux services de se découvrir dynamiquement (Eureka, Consul).
- **Centralized Logging** : gestion des logs centralisée (ELK Stack : Elasticsearch, Logstash, Kibana).
- **Health Checks** : suivi de la santé des services.

Sécurité dans les architectures microservices

- **Authentification et autorisation** : OAuth2, OpenID Connect.
- **Sécurité des API** : gestion des tokens JWT, certificats SSL.
- **Séparation des rôles** (RBAC) et politiques d'accès granulaires.
- **Sécurisation des communications interservices** : TLS, VPNs.