



Frameworks et architectures Java



Plan

- Introduction
- Environnements de développement
- Java EE et les serveurs d'applications
- Architecture MVC
- Services web SOAP et REST
- Architectures micro-services
- Message brokers



Introduction



Introduction

- Cette présentation est un séminaire sur les frameworks et architectures java
- Il n'a pas vocation à former des développeurs sur les sujets évoqués dans cette présentation, mais à couvrir les technologies actuelles en matière de frameworks et d'architectures Java



Environnements de développement



Environnements de développement

- Différences entre librairie et framework
- Le pattern IoC, son application dans les frameworks
- Services techniques, gestion du cycle de vie des objets, Injection de dépendances
- Framework pour les UI, binding de composants graphiques
- Les librairies OpenSource Java, les dépôts, exemple de Maven
- Apport d'un outil de build



Librairie vs Framework

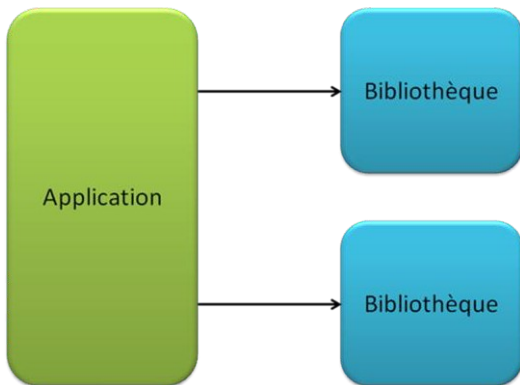


Librairie

- la conception objet peut avantageusement bénéficier de librairies (ou bibliothèques) de classes:
 - spécialisées ou non
 - standard ou non
- les bénéfices attendus sont importants:
 - réduction de la quantité de code à développer et à tester
 - amélioration de la qualité globale
- points à surveiller:
 - la librairie de classes est-elle maintenue?
 - par qui?
 - les sources sont-ils disponibles?

Librairie

- Les classes d'une bibliothèque ont (en principe) été testées de façon approfondie



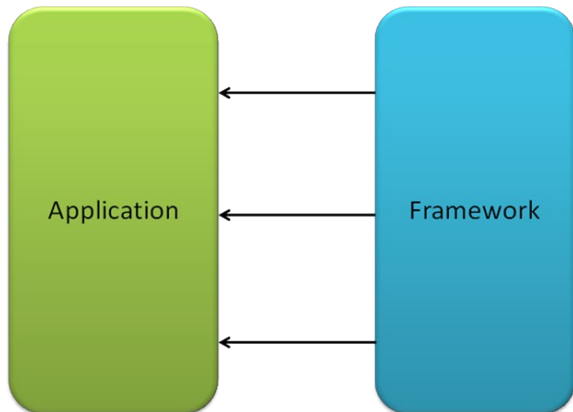


Framework

- à la différence d'une bibliothèque de classes, qui est passive car utilisée par une application, un framework contrôle lui-même les objets de l'application:
 - leur cycle de vie
 - l'exécution de leurs méthodes
- les objets de l'application doivent respecter une interface spécifique liée au framework

Framework

- un framework fournit également des classes appropriées, comme le fait une bibliothèque





Framework

- les bénéfices attendus sont importants:
 - réduction importante de la quantité de code à développer et à tester
 - amélioration de la qualité globale
 - exécution dirigée sous contrôle du framework
- de nombreux frameworks Java, notamment Open Source, sont apparus ces 20 dernières années (Spring, Struts, Hibernate, JUnit,...)



Le pattern IOC



Inversion de contrôle

- l'inversion de contrôle (IoC) permet aux classes d'être faiblement couplées, les dépendances étant décrites en XML ou dans des annotations
- un conteneur IoC a pour but de mettre en œuvre l'injection de dépendances pour la mise en relation des composants
- le conteneur IoC gère le cycle de vie des objets et permet de câbler les objets par injections
- Ce mécanisme est mis en œuvre, entre autres, dans le framework Spring et dans les serveurs d'applications Java EE



IoC: exemple de Spring

- Spring est un framework open source destiné à faciliter la conception d'applications Java SE ou Java EE
 - il fournit des moyens de gérer les objets métiers et leurs dépendances
 - il s'agit d'un conteneur léger (lightweight container), par opposition aux conteneurs lourds comme les serveurs d'applications Java EE
- Spring s'appuie sur la programmation par contrat et l'inversion de contrôle (IoC)
 - la programmation par contrat utilise les interfaces plutôt que les classes pour la mise en relation des objets



programmation par contrat

- la programmation par contrat consiste à distinguer la spécification d'un service de son implémentation
 - la spécification conduit à la conception d'une interface (contrat)
 - l'implémentation conduit à la conception d'une ou plusieurs classes (réalisation) qui implémentent cette interface
- le développement de l'interface et celui des classes peut être effectué par des personnes différentes
- le choix entre ces implémentations est décidé à l'intégration



programmation par contrat

- objectifs

- réduire les dépendances
 - les classes d'implémentation peuvent facilement être remplacées par d'autres sans impact sur l'application
- faciliter les tests
 - chaque couche logicielle ayant une spécification claire, il est facile de lui associer un jeu de tests utilisable indépendamment de son implémentation
- simplifier le code
 - la programmation par contrat permet de mieux découper les couches logicielles et de simplifier le code
- organiser le développement
 - lorsque la spécification est établie, le développement des classes d'implémentation est facilité



programmation par contrat

- un composant peut être couplé à un autre de plusieurs manières:
 - par association directe:

```
public class Login {  
    private LDAP_Authentication auth;  
  
    public Login(LDAP_Authentication auth){  
        this.auth=auth  
    }  
    public authentifier(User user){  
        . . .  
        boolean ok=auth.authentifier(user);  
    }  
}
```

- la classe **LDAP_Authentication** doit être développée en même temps que la classe Login



programmation par contrat

- par association d'interface:

```
public class Login {  
    private Authentication auth;  
  
    public Login(Authentication auth)  
    { this.auth=auth;  
    }  
    public authenticier(User user){  
        . . .  
        boolean ok=auth.authenticier(user);  
    }  
}  
  
public interface Authentication { public  
    boolean authenticier(User user);  
}
```

- l'interface **Authentication** peut désigner une autre classe que **LDAP_Authentication**
 - le code de la classe **Login** n'aura pas à subir de modification en cas de remplacement de la classe **LDAP_Authentication** par une autre



le design pattern IoC

- le principe d'IoC consiste à inverser le contrôle avec l'aide d'une classe supplémentaire chargée d'initialiser les dépendances entre les différents composants:

```
Authentication auth=new LDAP_Authentication ();  
Login lg=new Login(auth);
```

- la classe d'implémentation (LDAP_Authentication) est "injectée" dans le constructeur de Login
- le terme ***dependency injection*** a été inventé par Martin Fowler pour exprimer ce type de couplage entre composants



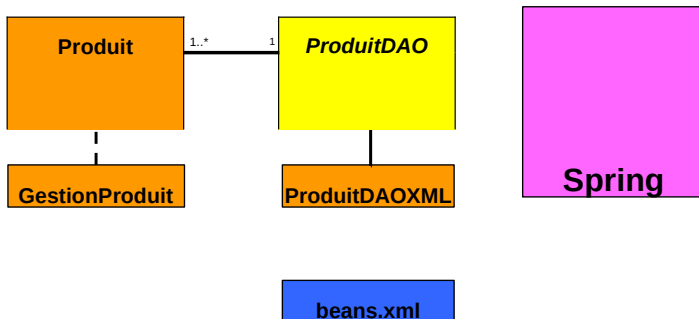
mise en œuvre de Spring

- les objets constituant l'application sont de simples POJOs (beans) qui implémentent ou non une interface
- les relations entre beans sont mises en œuvre par le biais de ces interfaces
- les beans et leurs relations sont décrits dans des annotations ou dans un ou plusieurs fichiers XML de configuration, appelés fichiers de contexte
- le conteneur IoC de Spring exploite le ou les fichiers de contexte et gère le cycle de vie des beans

mise en œuvre de Spring

- exemple:

- classes `Produit`, `ProduitDAOXML`, `GestionProduit`
- interface `ProduitDAO`
- fichier de contexte `beans.xml`





mise en œuvre de Spring

■ classe Produit:

```
package produit;
import produit.dao.ProduitDAO;
public class Produit{ private
    ProduitDAO dao; private
    long id; private String
    nom;
private String description; public
    Produit(){
public Produit(ProduitDAO dao)
    {this.dao = dao;}
public void setProduitDAO(ProduitDAO dao){this.dao = dao;} public long
    getId(){return id;}

public void setId(long id){this.id=id;} public
    String getNom(){return nom;}
public void setNom(String nom){this.nom=nom;}
public String getDescription(){return
    description;}
public void setDescription(String desc){this.description=desc;} public void
    save() throws DAOException{ dao.persist(this); }
}
```



mise en œuvre de Spring

- l'interface **ProduitDAO**:

```
package produit.dao; import  
produit.Produit;
```

```
public interface ProduitDAO {  
    public void  
    persist(Produit produit)  
    throws DAOException;  
}
```

- la classe
ProduitDAOXML:

```
package produit.dao; import  
produit.Produit; import  
java.io.FileWriter; import  
java.io.PrintWriter; import  
java.io.IOException;  
public class ProduitDAOXML implements ProduitDAO { private  
    String fichier;  
    public ProduitDAOXML(){}  
    public String getFichier(){return fichier;}  
    public void setFichier(String fichier){this.fichier=fichier;}
```




mise en œuvre de Spring

■ la classe ProduitDAOXML (suite):

```
public synchronized void persist(Produit produit) throws
    DAOException{
    PrintWriter pw=null; try {
    pw=new PrintWriter(new FileWriter(fichier,true),true);
    pw.println("<produit>"); pw.println("\t<id>"+produit.getId()
    +"</id>"); pw.println("\t<nom>"+produit.getNom()+"</nom>");
    pw.println("\t<description>"+produit.getDescription()+
    "</description>");
    pw.println("</produit>");
    }catch(IOException e)
    {throw new DAOException();
    }finally{
    if(pw!=null)
    pw.close();
    }
    }
}
```

}



mise en œuvre de Spring

- la classe GestionProduit:

```
package gestion;
import produit.Produit;
import
org.springframework.context.ApplicationContext;
import org.springframework.context.support.
    ClassPathXmlApplicationContext;
public class GestionProduit{
    public static void main(String[] args) throws Exception{
        ApplicationContext context=
            new ClassPathXmlApplicationContext("beans.xml");
        Produit p1=(Produit)context.getBean("produit");
        p1.setId(567543);
        p1.setNom("assiette");
        p1.setDescription("assiette plate ronde");
        p1.save();
        context.close();
    }
}
```



mise en œuvre de Spring

- le fichier de configuration beans.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://www.springframework.org/schema/beans"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.springframework.org/schema/beans
http://www.springframework.org/schema/beans/spring-beans-4.3.xsd
">

<bean id="gestionProduit" class="gestion.GestionProduit"/>
<bean id="produitDAO" class="produit.dao.ProduitDAOXML">
  <property name="fichier" value="produit.xml" />
</bean>
<bean id="produit" class="produit.Produit" scope="prototype">
  <property name="produitDAO" ref="produitDAO" />
</bean>
</beans>
```



injections par annotations

- l'annotation java **@Resource** permet l'injection sur les champs ou les propriétés
- Spring cherche un bean dont l'id ou le nom correspond à celui de l'attribut **name** de l'annotation **@Resource**

```
public class
    GestionContrats{ private
        PersonneDao personneDao;

    @Resource(name="personneDao")
        public void setPersonneDao (PersonneDao dao) {
            this.personneDao = dao;
        }
    }
```



injections par annotations

- si l'attribut **name** n'est pas précisé, Spring cherche un bean dont l'id ou le nom correspond à celui du champ ou de la propriété, ou par son type en dernier ressort



injections par annotations

- L'annotation Spring **@Autowired** est plus large d'utilisation et s'applique aux propriétés, aux méthodes ayant plusieurs arguments, aux constructeurs, et même aux champs

```
public class GestionContrats{  
    private PersonneDao personneDao;  
  
    @Autowired  
    public void setPersonneDao (PersonneDao dao) {  
        this.personneDao = dao;  
    }  
}
```



injections par annotations

- l'annotation **@Value** sur un attribut, une méthode ou un argument de méthode permet l'injection d'une valeur ou d'une propriété provenant d'un fichier

```
@Component
```

```
public class Vehicule{
```

```
    @Value("ch")
```

```
    private String unite_puissance;
```

```
    @Value("${puissance.din}")
```

```
    private int puissance;
```

```
    @Value("${boite: manuelle}") // manuelle par défaut
```

```
    private String boite_vitesse;
```

```
//...
```




injections par annotations

- l'annotation **@PropertySource** permet de préciser un fichier de propriétés
- exemple:

```
@Configuration
```

```
@PropertySource("classpath:postgresql.properties")
```

```
public class PostgreSQLConfig {  
    @Value("${jdbc.driver}") private String jdbcDriver;  
    @Value("${jdbc.url}") private String jdbcUrl;  
    @Value("${jdbc.username}") private String username;  
    @Value("${jdbc.password}") private String  
password; @Bean public DataSource  
getPostgreSQLDataSource() {  
    BasicDataSource dataSource = new BasicDataSource();  
    dataSource.setDriverClassName(jdbcDriver);  
    dataSource.setUrl(jdbcUrl); dataSource.setUsername(username);  
    dataSource.setPassword(password); return dataSource;  
}  
}
```

Context and Dependency Injection (CDI)



introduction

- CDI est une spécification (JSR 299) introduite dans java EE 6
- Norme inspirée de SEAM, Google Guice, Spring
- Utilise les annotations
- Permet le couplage lâche des composants
- Offre des contextes de cycle de vie extensibles auxquels les composants sont liés
- Intégration avec Unified Expression Language (EL)



mise en oeuvre de CDI

- suivant un ensemble de règles simples, tout objet Java (POJO) peut être traité comme un bean CDI
 - CDI supporte l'injection par champ, par propriété, par constructeur
 - la mise en oeuvre de CDI nécessite parfois la présence d'un fichier nommé **beans.xml** dans le répertoire **WEB-INF** (application web) ou **META-INF** (application EJB) du fichier de déploiement

```
<?xml version="1.0" encoding="UTF-8"?>
<beans xmlns="http://xmlns.jcp.org/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee http://xmlns.jcp.org/xml/ns/javaee/beans\_1\_1.xsd"
  version="1.1"
  bean-discovery-mode="all">
  ...
</beans>
```



l'injection par champ

- pour injecter un bean, il suffit d'annoter un constructeur, une propriété, ou un champ avec l'annotation **@javax.inject.Inject**
 - cette annotation définit un point d'injection
- exemple (field injection):

```
public class ShoppingCart implements Serializable {  
    @Inject private User customer;  
    ...  
}
```

- le conteneur instancie la classe **ShoppingCart** en faisant appel au constructeur sans argument
- il recherche ensuite une classe du type **User** ou une classe qui implémente l'interface **User** et l'instancie pour initialiser le champ **customer**



l'injection par méthode

- exemple (method injection):

```
public class ShoppingCart implements Serializable {  
    private User customer;
```

```
@Inject
```

```
    public void setCustomer(User customer){  
        this.customer=customer;  
    }  
    ...
```

```
}
```

- le conteneur instancie la classe **ShoppingCart** en faisant appel au constructeur sans argument
- il recherche ensuite une classe du type **User** ou une classe qui implémente l'interface **User** et l'instancie pour transmettre l'instance à la méthode **setCustomer**



l'injection par constructeur

- exemple (constructor injection)

```
public class ShoppingCart implements Serializable {  
    private User customer;
```

```
public ShoppingCart() {}
```

```
    @Inject public ShoppingCart(User customer) {  
        this.customer = customer;  
    }  
    ...  
}
```

- un seul constructeur peut être annoté avec **@Inject**



portée (scope)

- chaque bean CDI a une portée (scope) et est associé à un cycle de vie
 - **@Dependent**
 - scope par défaut. L'instance est créée chaque fois qu'elle est injectée et détruite lorsque la cible est détruite
 - **@ApplicationScoped**
 - une seule instance est créée pour toute l'application et détruite lorsque l'application se termine
 - **@RequestScoped**
 - une instance est créée pour toute requête HTTP et détruite lorsque la requête se termine



portée (scope)

- **@SessionScoped**

- une instance est créée pour toute la durée de la session et détruite lorsque la session prend fin

- **@ConversationScoped**

- une conversation correspond à une sorte de session dont le début et la fin sont délimités par l'application de façon programmée



portée (scope)

- exemple:

@RequestScoped

```
public class Customer implements User{  
    private String    login;  
private String password;  
    private String name;  
  
public Customer () {}  
  
...  
}
```



identification d'un bean

- lorsque plusieurs classes sont candidates au point d'injection, il faut un moyen d'identifier le type du bean à injecter

```
public class PaiementVisa implements Paiement {  
    .    .    .  
}  
public class PaiementCheque implements Paiement {  
    .    .    .  
}  
public class Commande implements Serializable {  
    @Inject private Paiement paiement;  
    ...  
}
```

- le conteneur ne peut déterminer s'il doit injecter une instance de PaiementCheque ou de PaiementVisa



identification d'un bean: `@Qualifier`

- solution 1

- les beans sont identifiés via un qualificateur
- il faut tout d'abord définir des annotations personnalisées avec **`@Qualifier`**

```
@Qualifier
```

```
@Retention (RUNTIME)
```

```
@Target({TYPE, METHOD, FIELD, PARAMETER})
```

```
public @interface Cheque {}
```

```
@Qualifier
```

```
@Retention (RUNTIME)
```

```
@Target({TYPE, METHOD, FIELD, PARAMETER})
```

```
public @interface Visa {}
```



identification d'un bean: @Qualifier

- ces annotations personnalisées sont ensuite ajoutées aux classes d'implémentation

@Visa

```
public class PaiementVisa implements Paiement {  
    . . .  
}
```

@Cheque

```
public class PaiementCheque implements Paiement {  
    . . .  
}
```

- field injection

```
public class Commande implements Serializable {  
    @Inject @Visa private Paiement paiement;  
    ...  
}
```



identification d'un bean: @Qualifier

- method injection

```
public class Commande implements Serializable {  
    private Paiement paiement;  
    @Inject public void setPaiement(@Visa Paiement paiement){  
        this.paiement=paiement;  
    }  
    ...  
}
```

- constructor injection

```
public class Commande implements Serializable {  
    private Paiement paiement;  
    @Inject public Commande (@Visa Paiement paiement){  
        this.paiement=paiement;  
    }  
    ...  
}
```



identification d'un bean: @Named

■ solution 2

- un bean peut être identifié via l'annotation **@Named**
- ce bean devient alors accessible depuis JSF

```
@Named("cheque")
```

```
public class PaiementCheque implements Paiement {  
    . . .  
}
```

```
@Named("visa")
```

```
public class PaiementVisa implements Paiement {  
}
```

- le point d'injection précise la classe de l'objet à injecter au moyen du nom associé

```
public class Commande implements Serializable {  
    @Inject @Named("visa") private Paiement paiement;  
    . . .  
}
```



identification d'un bean: **@Named**

- en l'absence de valeur explicite dans l'annotation **@Named**, le nom est celui de la classe commençant par une lettre minuscule

@Named

```
public class PaiementVisa implements Paiement {  
    . . .  
}
```

- le point d'injection précise la classe de l'objet à injecter au moyen du nom associé

```
public class Commande implements Serializable {  
    @Inject @Named("paiementVisa") private Paiement  
    paiement;  
    ...  
}
```





Maven

- **maven** facilite et automatise la gestion de certaines tâches liées à la gestion d'un projet Java
 - compilation, tests unitaires et déploiement des applications qui composent le projet
 - dépendances vis à vis des bibliothèques nécessaires au projet
 - documentations concernant le projet
- dans le but de simplifier la vie de l'utilisateur **maven** applique le paradigme "Convention over Configuration"



Maven

- **maven** est un outil de gestion de projet qui comprend:
 - un modèle objet pour définir un projet
 - un ensemble de standards
 - un cycle de vie
 - un système de gestion des dépendances



terminologie

- un projet **maven** est configuré par un POM (Project Object Model)
 - le POM est un fichier **pom.xml** situé à la racine du projet qu'il décrit (nom du projet, numéro de version, dépendances vers d'autres projets, bibliothèques nécessaires à la compilation...)
- une **phase** est une des étapes du "cycle de vie du build"
 - le cycle de vie du build est une suite ordonnée de phases aboutissant à la construction d'un projet
- **maven** comporte la logique nécessaire à l'exécution d'actions pour des phases bien définies du cycle de vie, par le biais de plugins



terminologie

- un **plugin** est constitué d'un ou plusieurs goals
 - le plugin **archetype** permet de générer des projets à partir de modèles
 - le plugin **jar** permet de créer des fichiers jar
 - le plugin **compile** permet de compiler des fichiers
- un **goal** est une tâche unitaire d'un **plugin**
 - `archetype:generate`
 - `archetype:create`
- un **dépôt** est un dossier particulier dans lequel **maven** stocke toutes les ressources dont il a besoin



le fichier pom.xml

- les fichiers **pom.xml** comportent un schéma
 - <http://maven.apache.org/POM/4.0.0>
- structure d'un fichier **pom.xml** minimaliste:

```
<project ...">
<modelVersion>4.0.0</modelVersion>
<groupId>org.sonatype.mavenbook.simple</groupId>
<artifactId>simple</artifactId>
<packaging>jar</packaging>
<version>1.0-SNAPSHOT</version>
<name>simple</name>
<url>http://maven.apache.org</url>
<dependencies>
    .
    .
    .
</dependencies>
</project>
```



coordonnées
maven



le fichier pom.xml

- l'élément racine de **pom.xml** est **project**:
 - **modelVersion**: la version du pom utilisé pour la création du projet
 - **groupId**: équivalent à un package java
 - **artifactId**: (arborescence) identifiant du projet
 - **packaging**: format du packaging du projet
 - **version**: version du projet
 - **name**: nom du projet pour affichage
 - **url**: emplacement du projet
 - **dependencies**: dépendances associées au projet
- le triplet **groupId:artifactId:version** identifie de manière unique un projet
- les **goals** exploitent le fichier **pom.xml** pour effectuer leurs actions



numéros de version

- forme

`<major>.<mini>[.<micro>][-<qualifier>[-<buildnumber>]]`

- Incrément

- major : changement majeur
 - pas de retro-compatibilité (descendante) garantie
 - mini : ajouts fonctionnels
 - retro-compatibilité garantie
 - micro : maintenance corrective (*bug fix*)



numéros de version

■ Qualificateurs

- SNAPSHOT (Maven) : version en évolution
- alpha1 : version alpha (très instable et incomplète)
- beta1, b1, b2 : version beta (instable)
- rc1, rc2 : release candidate
- m1, m2 : milestone
- ea : early access
- 20081014123459001 : date du build
- jdk5 : dépendance avec une arch, un os, un langage



numéros de version

- ordre sur les versions
 - différent de l'ordre lexicographique
 - $1.1.1 < 1.1.2 < 1.2.2$
 - $1.1.1\text{-SNAPSHOT} < 1.1.1$
 - $1.1.1\text{-alpha1} < 1.1.1\text{-alpha2} < 1.1.1\text{-b1} < 1.1.1\text{-rc1} < 1.1.1\text{-rc2} < 1.1.1$
- remarque (parfois)
 - `<mini>` pair : release stable
 - `<mini>` impair : release instable



la structure d'un projet

- un projet **maven** suit en général une arborescence type:

/src :	les sources du projet
/src/main :	code source et fichiers source
/src/main/java	principaux code source
/src/main/resources :	fichiers de ressources (images, fichiers annexes etc.)
/src/main/webapp :	webapp du projet fichiers de
/src/test :	test
/src/test/java :	code source de test
/src/test/resources :	fichiers de ressources de test
/src/site :	informations sur le projet et/ou les rapports générés suite aux traitements effectués
/target :	fichiers résultat, les binaires, les archives générées et les résultats des tests



la structure d'un projet

- de façon à faciliter le développement, un projet peut être découpés en modules (sous-projets)
 - le POM des modules héritent du POM de leur parent
 - dans le POM du projet parent:

```
<modules>  
  <module>simple-weather</module>  
  <module>simple-webapp</module>  
</modules>
```

- dans le POM d'un projet fils:

```
<parent>  
  <groupId>org.sonatype.mavenbook.multi</groupId>  
  <artifactId>simple-parent</artifactId>  
  <version>1.0</version>  
</parent>
```



le cycle de vie

- le cycle de vie du build est une suite ordonnée de phases aboutissant à la construction d'un projet
- **maven** peut supporter différents cycles de vie
 - le plus utilisé est le cycle de vie par défaut de maven, qui commence par une phase de validation de l'intégrité du projet et se termine par une phase qui déploie le projet en production
- les phases du cycle de vie sont laissées vagues intentionnellement, définies comme validation, test, ou déploiement elles peuvent avoir un sens différent selon le contexte des projets
 - pour un projet qui produit une archive Java, la phase package va construire un JAR ; pour une application web, elle produira un WAR



le cycle de vie

- ce cycle de vie permet à un développeur de passer d'un projet à l'autre sans connaître tous les détails du build de chacun d'entre eux
- pendant que **maven** parcourt les phases du cycle de vie, il exécute les goals rattachés à chacune d'entre elles
 - on peut rattacher de zéro à plusieurs goals à chaque phase



le cycle de vie

- exemple de cycle de vie avec ses phases:
 - validate: vérifie que le POM est correct et complet
 - compile: compile les sources du projet qui se trouvent dans *src/main/java* et place les fichiers compilés (.class) dans *target*
 - test: compile les tests qui se trouvent dans *src/test/java* et lance l'intégralité des tests, qui sont placés dans *target*
 - package: génère une archive de l'application dans *target/artifactId-version.jar*
 - verify: contrôle l'archive
 - install: utilise l'archive générée à la phase package et l'ajoute au dépôt local. Elle peut dès lors être utilisée comme dépendance pour un autre projet.
 - deploy: déploie l'application (varie selon la configuration)



le cycle de vie

- à chaque phase est associé un ou plusieurs goals d'un ou de plusieurs **plugins**
- remarque
 - `mvn resources:resources compiler:compile resources:testResources compiler:testCompile surefire:test jar:jar`
 - est équivalent à **mvn package**
 - d'autre cycles de vie ont été définis
 - `clean = pre-clean->clean->post-clean`
 - `site = pre-site->site->post-site->site-deploy`



les dépôts

- un dépôt **maven** est un ensemble d'artefacts de projet rangés selon une structure de répertoires correspondant aux coordonnées **maven**
- lors de la première installation, **maven** extrait ce dont il a besoin à partir d'un dépôt central
<http://repo1.maven.org/maven2/>
 - et les enregistre dans le dépôt **maven** local de votre machine généralement situé à l'emplacement:
`~/.m2/repository`
 - les projets installés (`mvn install`) sont également enregistrés dans le dépôt local
- d'autres dépôts publics existent (et peuvent être enregistrés dans `~/.m2/settings.xml`)



la gestion des dépendances

- l'une des principales forces de **maven** est sa gestion des dépendances
 - au lieu de chercher et lister explicitement toutes ces dépendances, il suffit de déclarer dans **pom.xml** la dépendance à la bibliothèque nécessaire au projet

```
<dependencies>
  <dependency>
    <groupId>commons-lang</groupId>
    <artifactId>commons-lang</artifactId>
    <version>2.5</version>
  </dependency>
</dependencies>
```
 - **maven** se charge d'ajouter ses dépendances implicitement au projet
 - **maven** gère les conflits de dépendances



la gestion des dépendances

- **maven** fait appel aux dépôts, locaux ou distants, pour trouver les bibliothèques nécessaires à la gestion des dépendances
 - elles sont d'abord téléchargées depuis un dépôt global, elles sont ensuite mises en cache dans un dépôt local



la gestion des dépendances

- une dépendance a une portée (*scope*), c'est à dire qu'elle est utilisée pour certaines tâches
 - **compile** : c'est le scope par défaut. La dépendance est active pour compiler le code et l'exécuter
 - **test**: cette dépendance est nécessaire seulement pour la compilation et l'exécution des tests
 - **runtime**: la dépendance n'est active qu'à l'exécution de l'application (driver JDBC)
 - **provided**: la dépendance n'est utile que pour la compilation et les tests
 - **system**: similaire à **provided** mais elle n'est pas issue d'un dépôt



profils

- motivation: améliorer la portabilité des projets par rapport aux environnements
 - différents JVM, versions de Java, serveurs, SGBD, développement versus production
- moyens: créer des variations (=profils) de projets
 - élément **<profile>** du build
 - contient les variations de plugins et entre les plugins
 - activation du profil
 - profil par défaut
 - en fonction des propriétés (systèmes, version JDK, ...)
 - par son identifiant

```
mvn --activate-profiles felix,equinox clean install
```



les plugins

- **maven** dispose de nombreux plugins (plus de 50), parmi les principaux:
 - **archetype**: génère le squelette de la structure d'un projet à partir d'un modèle
 - **javadoc**: produit la documentation Java du projet
 - **jar**: génère un fichier
 - **war**: jar génère un
 - **ear**: fichier war génère
 - **surefire**: exécute les tests unitaires JUnit
 - **clean**: nettoie après construction
 - **compiler**: compile les sources Java
 - **deploy**: déploie le build vers le dépôt distant
 - **failsafe**: exécute les tests d'intégration JUnit
 - **site**: création d'un site pour le projet
- pour plus d'informations:
<http://maven.apache.org/plugins/>



configuration des plugins

- passage de paramètres autre que ceux définis par défaut

```
<build>
  <plugins>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-compiler-plugin</artifactId>
      <configuration>
        <source>1.5</source>
        <target>1.5</target>
      </configuration>
    </plugin>
    <plugin>
      <groupId>org.apache.maven.plugins</groupId>
      <artifactId>maven-jar-plugin</artifactId>
      <configuration>
        <archive>
          <manifest>
            <mainClass>${artifactId}.Main</mainClass>
            <addClasspath>true</addClasspath>
          </manifest>
        </archive>
      </configuration>
    </plugin>
  </plugins>
</build>
```



plugin archetype

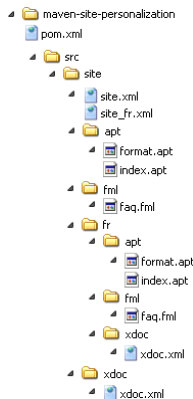
- le plugin **archetype** permet la construction initiale d'un projet **maven**
- exemple en mode interactif:

```
mvn archetype:create
```

 - DgroupId=demo.maven
 - DartifactId=hello
 - Dversion=0.1.0-SNAPSHOT
 - DarchetypeGroupId=org.apache.maven.archetypes
 - DarchetypeArtifactId=maven-archetype-quickstart

plugin site

- maven permet de générer une documentation accessible par un navigateur web plugging **site**
- exemple en mode interactif:
mvn site
 - le résultat apparaît par défaut dans **target/site**
- 3 formats sont possibles par défaut
 - xdoc (XML)
 - apt (almost plain text)
 - fml (pour les FAQ)



Les outils de développement





Présentation

- de nombreux outils de développement Java existent aujourd'hui, notamment en Open Source:
 - Eclipse (Open Source)
 - NetBeans (Open Source)
 - RAD (IBM)
 - JDeveloper (Oracle)
 - IntelliJ (JetBrains)



principaux IDE (Integrated Development Environnement)

- **Eclipse** est une plateforme de développement universelle créée par IBM puis fournie en Open Source
- Eclipse permet à la base de développer en Java, mais tout langage peut être supporté par l'introduction de plug-ins
- les plug-ins utiles au développement Java EE sont désormais intégrés
- les dernières versions d'Eclipse permettent le développement de tout composant Java EE, rendant ainsi caduques l'emploi de plug-ins payants



principaux IDE (Integrated Development Environnement)

- **NetBeans** est le concurrent direct d'Eclipse dans le développement Java en Open Source
- bien connu pour ses capacités de développement d'applications graphiques Swing, NetBeans intègre désormais des outils pour le développement Java EE
- il est particulièrement adapté au développement Java EE pour la plateforme d'Oracle
- la dernière version est la 8.0



principaux IDE (Integrated Development Environnement)

- IBM fournit un IDE basé sur Eclipse: **RAD** (Rational Application Developer)
 - il s'intègre parfaitement aux produits WebSphere et facilite le développement d'applications pour cet environnement
- JetBrains fournit un IDE commercial **IntelliJ IDEA** très puissant pour le développement Java
 - il supporte le développement Spring Struts, Hibernate, Android

Java EE et les serveurs d'applications



Java EE et les serveurs d'applications

- Le modèle Java EE, multi-tiers, composants web, métier, persistance
- La spécification JavaEE et les apports d'un serveur d'applications
- Le modèle de déploiement
- Le modèle concurrentiel
- Technologies : Servlets, JSF, EJB et JPA



L'architecture n-tiers



Rappel sur les besoins

- tout système d'information nécessite la réalisation de trois groupes de fonctions:
 - le **stockage** des données
 - la **logique** applicative et
 - la **présentation** des données



La couche de persistance

- Stockage et accès aux données :
 - le système de stockage des données a pour but de conserver une quantité plus ou moins importantes de données de façon structurée
 - on peut l'utiliser pour cette partie des systèmes (très variés) qui peuvent être :
 - des systèmes de fichiers
 - des bases de données relationnelles
 - etc...



La couche métier

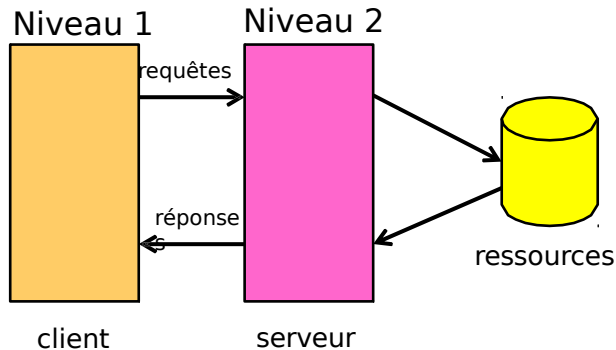
- Logique applicative :
 - la logique applicative est la réalisation informatique du mode de fonctionnement de l'entreprise
 - cette logique constitue les traitements nécessaires sur l'information afin de la rendre exploitable par chaque utilisateur



La couche de présentation

- Présentation :
 - la présentation est la partie la plus immédiatement visible pour l'utilisateur
 - elle a donc une importance primordiale pour rendre attrayante l'utilisation de systèmes informatiques
 - son évolution a été très importante depuis les débuts de l'informatique et des systèmes d'information

Architecture 2-tiers





Architecture à deux niveaux

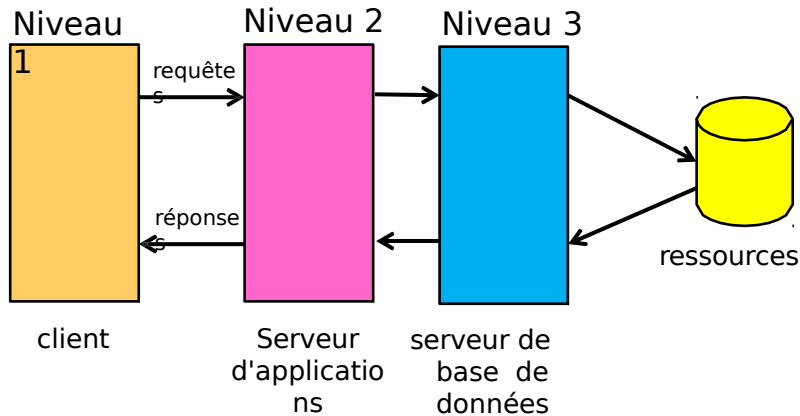
- aussi appelée *architecture 2-tier*, « tier » signifiant *rangée* en anglais
- elle caractérise les systèmes clients/serveurs pour lesquels :
 - le client demande une ressource (GET)
 - le serveur la lui fournit directement (OUT), en utilisant ses propres ressources
- cela signifie que le serveur ne fait pas appel à une autre application afin de fournir une partie du service



Architecture à deux niveaux

- l'architecture à deux niveaux est donc une architecture client/serveur dans laquelle :
 - le serveur est **polyvalent**, c'est-à-dire qu'il est capable de fournir directement **l'ensemble** des ressources demandées par le client

Architecture 3-tiers





Architecture à trois niveaux

- dans l'architecture à 3 niveaux, appelée *architecture 3-tiers*, il existe un niveau intermédiaire, c'est une architecture partagée entre :
 - un client, c'est-à-dire l'ordinateur demandeur de ressources, équipée d'une interface utilisateur (généralement **un navigateur web**)
 - le serveur d'application (ou **middleware**), chargé de fournir les ressources mais faisant appel à un autre serveur
 - le serveur de données (ou **database server**) qui va fournir au serveur d'application les données dont il a besoin



Architecture à trois niveaux

- dans l'architecture à trois niveaux, les applications au **niveau serveur** sont **délocalisées**, c'est-à-dire que chaque serveur est spécialisé dans une tâche :
 - serveur web
 - serveur de base de données
- afin d'apporter:
 - une plus grande flexibilité/souplesse
 - une sécurité accrue car elle peut être définie à chaque niveau
 - de meilleures performances, étant donné le partage des tâches entre les différents serveurs



Architecture multi-niveaux

- constat : dans l'architecture à 3 niveaux, chaque serveur (niveaux 2 et 3) effectue une tâche (un service) spécialisée :
 - un serveur peut donc utiliser les services d'un ou plusieurs autres serveurs afin de fournir son propre service
 - par conséquent, l'architecture à trois niveaux est potentiellement une architecture à N niveaux...



Architecture n-tiers

Présentation traitements

IHM

Contrôle

Commu-
nication

Serveur de métier

Logique
Applicative

Logique
Métier

Logique
d'accès
aux
données

Serveur de données



Client lourd

- le terme «**client lourd**», en anglais «*fat client*» ou «*heavy client*» par opposition au «client léger», désigne une application cliente graphique exécutée sur le système d'exploitation de l'utilisateur
- un client lourd possède généralement des capacités de traitement évoluées et peut posséder une interface graphique sophistiquée.
- néanmoins, ceci demande un effort de développement et tend à mêler la **logique de présentation** (l'interface graphique) avec la **logique applicative** (les traitements)



Client léger

- le terme «**client léger**», parfois «client pauvre», en anglais «*thin client*» par opposition au «client lourd» désigne une application accessible via une interface web (en HTML) consultable à l'aide d'un navigateur web, où la totalité de la logique métier est traitée du côté du serveur
- pour ces raisons, le navigateur est parfois appelé **client universel**
 - l'origine du terme provient de la limitation du langage HTML, qui ne permet de faire des interfaces relativement pauvres en interactivité, si ce n'est pas le biais du langage javascript, DHTML, XHTML, etc...



Client léger & langage HTML

- apparu en 1991 au CERN, à Genève, HTML est l'abréviation de HyperText Markup Langage (langage de balises pour l'hypertexte...)
- en 1994 apparaît la version 1.0, sous l'égide du W3C, qui est chargé de normaliser le langage
- HTML n'est pas un langage de programmation (C, Java, Cobol)
- HTML permet essentiellement de présenter des informations et de naviguer en hypertexte au sein de multiples documents HTML, situés sur la même machine ou sur des machines différentes



Client léger & langage HTML

- sur la forme:
 - un fichier HTML est un fichier texte contenant des balises
 - l'extension du nom d'un tel fichier est .html ou .htm
 - il faut un logiciel adéquat pour l'exploiter: un navigateur internet
- sur le fond:
 - à l'origine, les balises n'avaient pas pour fonction d'indiquer un format de présentation
 - la présentation était dépendante du navigateur utilisé
 - l'objectif principal était la navigation par les hyperliens
 - progressivement, de nouvelles balises destinées à la seule présentation sont apparues



Client léger & langage HTML

- HTML est souvent utilisé conjointement avec
 - des langages de programmation (JavaScript)
 - des formats de présentation (CSS)
- la version actuelle est 5.0, finalisée en octobre 2014



Client léger & langage HTML

- exemple:

```
<!DOCTYPE html>
<html>
  <head>
    <title> Exemple de HTML </title>
  </head>
  <body> bla,bla,bla...
    <a href="cible.html">hyperlien</a>
    <p> paragraphe </p>
  </body>
</html>
```



Client léger: rôle du navigateur web

- les pages web nécessitent un navigateur pour jouer leur rôle: assurer la présentation et surtout faciliter la navigation entre les pages web des ordinateurs en réseau (local ou internet)
- bien que généralement précisée aujourd'hui par des balises spécifiques, la présentation d'un même document HTML peut varier d'un navigateur à l'autre
- un navigateur web est un client HTTP



Client léger: rôle du navigateur web

- quelques navigateurs connus:
 - Internet Explorer (Microsoft)
 - Google Chrome
 - Opera (Opera Software)
 - Safari (Apple)
 - Mozilla FireFox (Open Source)



Client riche

- un «**client riche**» est un compromis entre le client léger et le client lourd
 - l'objectif du client riche est donc de :
 - proposer une interface graphique, décrite avec une grammaire de description basée sur la syntaxe XML
 - obtenir des fonctionnalités similaires à celles d'un client lourd (glisser déposer, onglets, multi-fenêtrage, menus déroulants, ...)



Client riche

- il existe des standards permettant de définir une application riche :
- **XAML** (*eXtensible Application Markup Language*), prononcer « zammel », un standard XML proposé par Microsoft
- **XUL**, prononcez « zoul », un standard XML proposé par la fondation Mozilla ;
- **Flex**, un standard XML proposé par la société Macromedia
- **JavaFX**, nouvelle API apparue en 2014 avec Java 1.8



Quelles technologies ?

- XML comme format de données standard
- JavaEE, .Net, PHP comme environnement de dév.
- Linux/Windows comme système d'exploitation
- Apache comme serveur web
- MySQL, Oracle comme serveur de base de données
- HTML, JavaScript, Ajax comme langage de scripts
- SOAP, CORBA, DCOM, comme langage de communication
- Multi-threading comme technique de répartition de charge entre serveurs



La spécification Java EE



la spécification Java EE

- Java EE 7 (java Enterprise Edition) apparue en mai 2013 désigne les API permettant la manipulation de composants serveurs:
 - les servlets (3.1)
 - les Java Server Pages (JSP 2.3)
 - les Java Server Faces (JSF 2.2)
 - les Enterprise Java Beans (EJB 3.2)
 - les services web (JAX-WS 2.2 et JAX-RS 2.0)
- Java EE 8 est sortie en septembre 2017:
 - les servlets (4.0)
 - les Java Server Pages (JSP 2.3)
 - les Java Server Faces (JSF 2.3)
 - les Enterprise Java Beans (EJB 3.2)
 - les services web (JAX-WS 2.2 et JAX-RS 2.1)

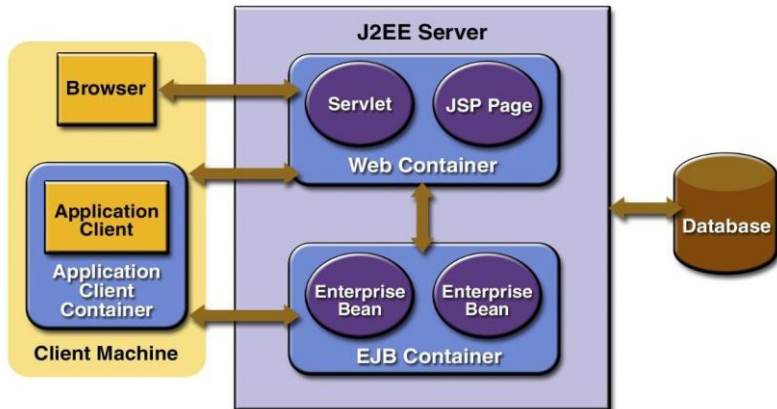


notion de conteneur

- l'infrastructure de Java EE est découpée en domaines logiques appelés conteneurs
 - chacun d'eux joue un rôle spécifique
 - supporte un ensemble d'API
 - offre des services à ses composants (sécurité, accès aux données, gestion des transactions, injection de ressources, etc.)
- les conteneurs masquent les aspects techniques et améliorent la portabilité
 - en fonction du type d'application à construire, il est nécessaire de comprendre les possibilités et les contraintes de chaque conteneur
 - une couche de présentation web est déployée dans un conteneur web, non dans un conteneur EJB
 - Une application web qui appelle une couche métier est déployée à la fois dans un conteneur web et un conteneur EJB

notion de conteneur

- conteneurs Java EE





le conteneur d'applications client

- situé côté client, le conteneur d'applications client (ACC, *application client container*) contient un ensemble de classes et de bibliothèques Java ainsi que d'autres fichiers afin d'ajouter l'injection, la gestion de la sécurité et le service de nommage aux applications Java SE
 - applications Swing
 - traitements non interactifs
 - ou simplement une classe avec une méthode main
- ACC communique avec le conteneur EJB en utilisant le protocole RMI-IIOP et avec le conteneur web *via* le protocole HTTP (pour les services web, notamment)



le conteneur Web

- situé côté serveur, le conteneur web (ou conteneur de servlets) fournit les services sous-jacents permettant de gérer et d'exécuter les composants web:
 - servlets
 - JSP
 - filtres
 - écouteurs
 - pages JSF
 - services web
- il est responsable de l'instanciation, de l'initialisation et de l'appel des servlets et du support des protocoles HTTP et HTTPS
- c'est lui qu'on utilise pour servir les pages web aux navigateur des clients



le conteneur EJB

- situé côté serveur, le conteneur EJB est responsable de la gestion de l'exécution des beans entreprise
- contenant la couche métier d'une application Java EE, il a en charge:
 - la création de nouvelles instances d'EJB
 - la gestion de leur cycle de vie
 - fournit des services comme:
 - les transactions
 - la sécurité
 - la concurrence
 - la distribution
 - le nommage
 - les appels asynchrones



services

- les conteneurs fournissent à leurs composants de nombreux services sous-jacents
 - le développeur peut alors se concentrer sur l'implémentation de la logique métier au lieu de résoudre les problèmes techniques auxquels sont exposées les applications d'entreprise
 - en effet, en plus des API directement liées aux composants serveurs, d'autres API sont souvent nécessaires:
 - Java DataBase Connectivity (JDBC)
 - Java Persistence API (JPA)
 - Java Transaction API (JTA)
 - Java Message Service (JMS)
 - Java API for XML Web Services (JAX-WS)
 - Java API for XML Processing (JAXP)
 - Java Architecture for XML Binding (JAXB)
 - Java Authentication and Authorization Service (JAAS)
 - JavaBeans Activation Framework (JAF)



principales API Java EE

- JDBC (Java Database Connectivity)
 - ces API permettent l'invocation de requêtes SQL depuis Java
 - elles peuvent être utilisées aussi bien dans un composant du conteneur web que dans un EJB
 - cette API comporte deux parties:
 - une interface applicative utilisée par les composants Java pour accéder à la base de données
 - une interface pour le fournisseur du SGBD permettant d'accrocher un driver JDBC à la plateforme Java



principales API Java EE

- JPA (Java Persistence API)
 - il s'agit d'une solution standard java pour la persistance d'objets en base de données relationnelle
 - ces API peuvent également être utilisées dans une application Java SE
- la technologie de persistance Java inclut:
 - les API JPA
 - le langage de requête JPQL
 - les méta-données de correspondance objet/relationnel (Object/relational mapping metadata)



principales API Java EE

- JTA (Java Transaction API)
 - il s'agit d'une interface standard pour la gestion programmée des transactions
 - l'architecture Java EE fournit un mode par défaut appelé auto-commit pour gérer les opération de commit et de rollback des transactions
 - si l'application a besoin d'une gestion plus élaborée, le développeur peut faire appel aux API JTA afin de délimiter le début et la fin des transactions: begin, commit et rollback



principales API Java EE

- JNDI (Java Naming and Directory Interface API)
 - ces API permettent aux applications d'accéder à des services de nommage variés comme LDAP, DNS, et NIS
 - elles fournissent des moyens d'écrire et de lire dans tout service de nommage, notamment pour stocker et récupérer des objets de tout type, comme des pools de connexions aux bases de données, des fabriques de messages JMS, des proxies sur EJB
- JMS (Java Message Service)
 - ces API permettent aux composants Java de créer, envoyer, recevoir, et lire des messages
 - elles permettent notamment des communications distribuées fiables asynchrones avec un couplage faible



principales API Java EE

- **JAX-WS (Java API for XML Web Services)**
 - ces API permettent le développement de services web s'appuyant sur le binding JAXB
 - ces services web peuvent être développés à partir de servlets ou d'EJB
 - cette spécification décrit également le support des gestionnaires de messages pour traiter les requêtes et réponses
- **JAX-RS (Java API for RESTful Web Services)**
 - ces API permettent le développement de services web en architecture REST (Representational State Transfer)
 - une application JAX-RS est une application web qui comporte au moins une servlet, packagé dans un fichier WAR



principales API Java EE

■ Managed Beans

- les Managed Beans sont de simples objets (POJOs) comportant un faible jeu de fonctionnalités: injection de ressources, callback liés au cycle de vie, intercepteurs
- il s'agit d'une généralisation des Managed Beans de JSF (Java Server Faces), de sorte qu'ils peuvent être utilisés dans tout module d'une application, pas uniquement dans un module web
- il s'agit d'une API introduite récemment dans Java EE



principales API Java EE

- CDI (Contexts and Dependency Injection for the Java EE Platform - JSR 299)
 - ces API définissent des services qui permettent au développeur l'utilisation d' EJB avec des JSF
 - conçus pour des objets avec état (stateful objects), CDI apporte une grande flexibilité pour l'intégration de différents types de composants
 - il s'agit d'une API introduite récemment dans Java EE
- DI (Dependency Injection for Java - JSR 330)
 - l'injection de dépendances définit un jeu d'annotations (et une interface) pour faciliter l'injection d'objets dans des classes



principales API Java EE

- Bean Validation

- cette spécification définit un modèle de méta-données et des API permettant la validation des données dans les Java Beans
- plutôt que de répartir la validation sur plusieurs couches de l'application, comme dans le navigateur web et dans le serveur, les contraintes de validation peuvent être localisées à un seul endroit et partagées par plusieurs composants de l'application



principales API Java EE

- JCA (Java EE Connector Architecture)
 - l'architecture JCA est utilisée par des outils propriétaires et des intégrateurs pour créer des adaptateurs de ressources (resource adapters) afin de donner accès au système d'information de l'entreprise
 - un adaptateur de ressource est un composant logiciel qui permet à des composants Java EE d'accéder et d'interagir avec le gestionnaire de ressources du SIE
 - l'architecture JCA fournit également une intégration poussée des services web Java EE avec des services web du système d'information de l'entreprise, en mode synchrone ou asynchrone
 - les architectures JAX-WS et JCA sont donc complémentaires pour la mise en oeuvre de l'EAI (Enterprise Application Integration)



principales API Java EE

■ JavaMail

- les applications Java EE utilisent ces API pour l'envoi de mails
- elle comporte deux parties:
 - une interface applicative utilisée par les composants Java EE pour envoyer des emails
 - une interface pour le fournisseur de service

■ JAF (JavaBeans Activation Framework)

- les API sont utilisées par les API JavaMail
- JAF fournit des services permettant de déterminer le type d'une donnée, d'encapsuler son accès, de découvrir les opérations possibles et enfin de créer les Java Beans capable d'effectuer ces opérations



principales API Java EE

- JASPIC (Java Authentication Service Provider Interface for Containers)
 - cette spécification définit une interface pour fournisseur de services (SPI) par laquelle les fournisseurs d'authentification qui implémentent les mécanismes d'authentification par message peuvent intégrer les conteneurs de gestion des messages
- JACC (Java Authorization Contract for Containers)
 - cette spécification définit un contrat entre un serveur d'applications Java EE et un fournisseur de services d'autorisations (authorization policy provider)



principales API Java EE

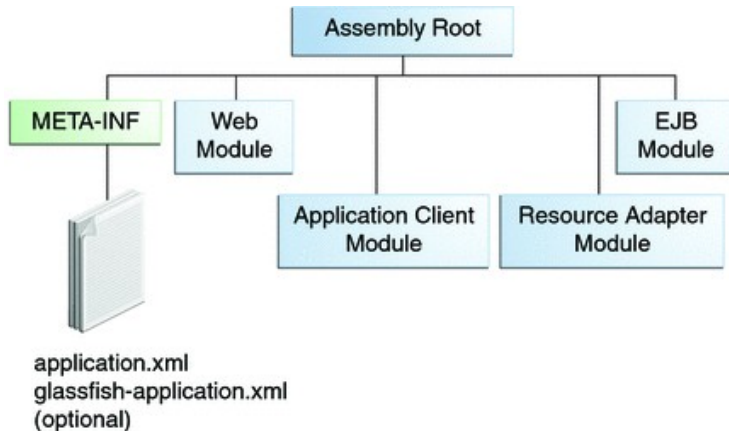
- JAXP (Java API for XML Processing)
 - ces API font partie de Java SE et permettent le traitement de documents XML en utilisant DOM (Document Object Model), SAX (Simple API for XML), et XSLT (Extensible Stylesheet Language Transformations)
- JAXB (Java Architecture for XML Binding)
 - ces API fournissent un moyen commode d'établir une correspondance (binding) entre un schéma XML et une représentation en Java
 - JAXB peut être utilisé avec ou indépendamment de JAX-WS
- JAAS (Java Authentication and Authorization Service)
 - ces API permettent aux applications Java EE d'assurer l'authentification d'un utilisateur et d'assurer la gestion des autorisations



packaging d'applications

- une application Java EE est livrée selon le cas:
 - dans un fichier JAR (Java Archive)
 - dans un fichier WAR (Web Archive)
 - dans un fichier EAR (Enterprise Archive)
- un fichier WAR ou EAR est un fichier JAR standard (extension .jar) d'extension .war ou .ear
- l'utilisation de fichiers JAR, WAR, EAR et de modules permet de créer des applications de rôles distincts qui utilisent des composants communs
- un fichier WAR ou EAR contient des modules Java EE et un descripteur de déploiement facultatif
 - un descripteur de déploiement (deployment descriptor) est un fichier XML qui contient les informations liées au déploiement de l'application, d'un module, d'un composant

packaging d'applications

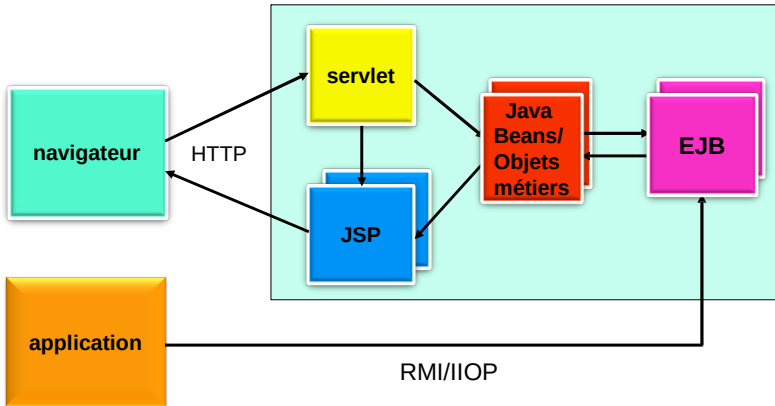




relations entre servlets, JSP, EJB

- les servlets et les JSP d'une application web s'exécutent au sein d'un conteneur Web, tandis que les EJB s'exécutent au sein d'un conteneur EJB
- dans une architecture MVC, les pages HTML et les JSP sont chargés de la présentation (V), les servlets du contrôleur (C), les EJB du modèle (M)
- ceci permet de séparer logiquement et physiquement les entités qui composent une application d'entreprise

relations entre servlets, JSP, EJB





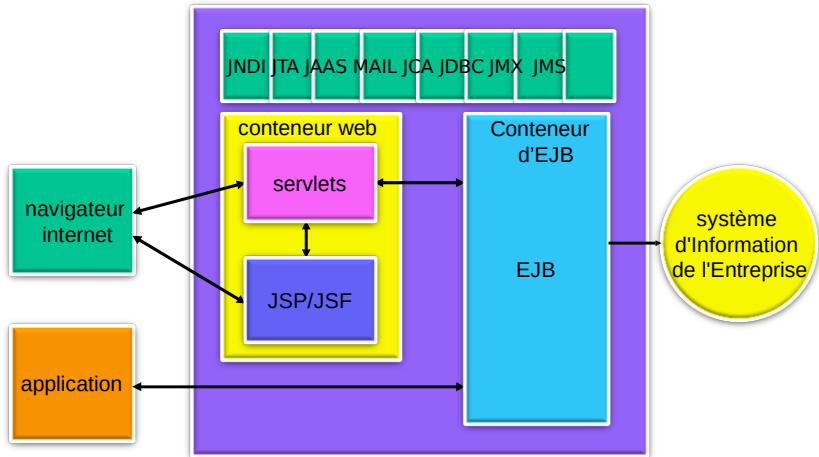
Les serveurs d'applications Java EE



présentation

- un serveur d'applications Java est une implémentation de la spécification Java EE
- son rôle consiste à permettre l'exécution des servlets, JSP et EJB et à fournir tous les services annexes définis dans la spécification Java EE
- il comporte un conteneur web et un conteneur EJB, ce dernier pouvant être absent dans certains serveurs "allégés" comme le serveur Tomcat

constitution





fonctionnalités

- les fonctionnalités principales d'un serveur d'applications :
 - l'accès aux objets serveurs
 - la gestion du cycle de vie des objets serveurs
 - la gestion transactionnelle
 - la répartition de la charge
- tout serveur d'applications dispose d'une console d'administration permettant de contrôler l'exécution des applications
 - la gestion des versions
 - le déploiement d'applications
 - la gestion de la sécurité
 - la gestions des performances
 - la gestion des erreurs d'exploitation



principaux serveurs d'applications

- principaux serveurs d'applications java EE:
 - serveurs commerciaux:
 - TMAX JEUS 8 de TmaxSoft (Java EE 7)
 - WebLogic Server 12 d'Oracle (Java EE 7)
 - WebSphere Application Server 8.5 d'IBM (Java EE 7)
 - WebSphere Application Server 9.x d'IBM (Java EE 8)
 - serveurs open sources:
 - GlassFish 4.x (Java EE 7)
 - GlassFish 5.x (Java EE 8)
 - Wildfly 14 (Java EE 8)
 - Wildfly 8 (Java EE 7)
 - Geronimo 3 du groupe Apache (Java EE 6)



Les servlets



plan

- principe des servlets
- architecture
- descripteur de déploiement
- déploiement
- mise en oeuvre
- sauvegarde des informations utilisateur
- redirection
- filtres de servlet



principe des servlets

- les servlets sont des composants Java destinés principalement à remplacer les scripts CGI (Common Gateway Interface) des serveurs
 - comme les scripts CGI, les servlets ont pour rôle de prendre en charge les requêtes des clients et de leur fournir les réponses
- les servlets nécessitent un conteneur web compatible pour être exécutées
 - étant gérés en multi-threading par le serveur, les servlets sont plus rapides à l'exécution que les scripts CGI
 - ils sont aussi plus simples à écrire et entièrement portables



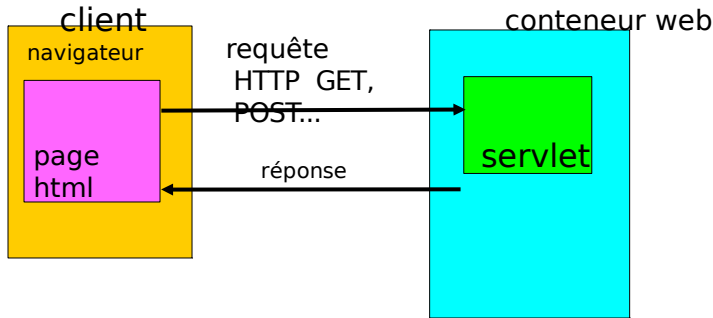
principe des servlets

- l'échange d'informations entre clients et serveurs Web est géré respectivement par les navigateurs et par les serveurs via le protocole HTTP
 - le protocole HTTP permet d'échanger des données sous la forme de caractères sur 8 bits, et autorise l'échange de tous types de documents comme des pages HTML, des images, des programmes

principe des servlets

un échange en HTTP s'effectue en 4 étapes:

1. ouverture de la connexion
 2. envoi de la requête vers le serveur
 3. envoi de la réponse vers le client
 4. fermeture de la connexion
- HTTP ne mémorisant pas l'état d'un échange, le serveur ferme la connexion avec le client après l'envoi de sa réponse





mise en oeuvre

- une requête peut être émise vers une servlet selon 4 moyens:
 - 1 - par saisie d'une URL dans la barre d'adresses du navigateur
 - 2 - par activation d'un lien hypertexte
 - 3 - par validation d'un formulaire
 - 4 - par inclusion ou redirection
- les moyens 1 et 2 émettent une requête de type GET
- les moyens 3 et 4 émettent soit une requête de type GET soit de type POST



mise en oeuvre

- requêtes de type GET

```
<HTML>
  <HEAD>
    <TITLE>acces a la servlet Bonjour! </TITLE>
  </HEAD>
  <BODY>
    <P>GET par lien hypertexte</P>
    <A HREF="Bonjour">Cliquer ici</A>
    <P>GET par formulaire</P>
    <FORM ACTION="Bonjour">
      <INPUT TYPE="text" NAME="nom">
      <INPUT TYPE="submit" VALUE="envoi">
    </FORM>
  </BODY>
</HTML>
```



mise en oeuvre

- les paramètres de la requête peuvent être récupérés au moyen de la méthode **getParameter** sur l'objet **request**

```
public class Bonjour extends HttpServlet {  
    public void doGet (HttpServletRequest request,  
        HttpServletResponse response) throws ServletException, IOException {  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        String nom=request.getParameter("nom");  
        if(nom==null) nom="";  
        out.println("<HTML><HEAD><TITLE>");  
        out.println("reponse de la servlet");  
        out.println("</TITLE></HEAD><BODY>");  
        out.println("<H1> Bonjour! "+nom+ "<br></H1>");  
        out.println("</BODY></HTML>");  
    }  
}
```



mise en oeuvre

- un formulaire utilisera avantageusement des requêtes de type POST
 - afin de ne pas faire apparaître les paramètres de la requête dans l'URL

```
<HTML>
  <HEAD>
    <TITLE>acces a la servlet Bonjour! </TITLE>
  </HEAD>
  <BODY>
    <P>POST par formulaire</P>
    <FORM ACTION="Bonjour" METHOD="POST" >
      <INPUT TYPE="text" NAME="nom">
      <INPUT TYPE="submit" VALUE="envoi">

    </FORM>
  </BODY>
</HTML>
```



mise en oeuvre

- comme pour une requête GET, les paramètres de la requête POST peuvent être récupérés au moyen de la méthode **getParameter** sur l'objet **request**

```
public class Bonjour extends HttpServlet {  
    public void doPost (HttpServletRequest request,  
                        HttpServletResponse response) throws  
ServletException, IOException {  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        String nom=request.getParameter("nom");  
        if(nom==null) nom="";  
        out.println("<HTML><HEAD><TITLE>");  
        out.println("reponse de la servlet");  
        out.println("</TITLE></HEAD><BODY>");  
        out.println("<H1> Bonjour! "+nom+ "<br></H1>")  
        out.println("</BODY></HTML>");  
    }  
}
```



sauvegarde des informations utilisateurs

- le protocole HTTP ne mémorise pas l'état des échanges entre clients et serveur durant une session (dialogue d'un client avec un serveur)
- un dialogue client/serveur étant constitué d'une suite d'échanges, il n'est pas possible de relier un échange effectué par un client à l'échange suivant effectué par ce même client
- de plus, les informations utilisateurs ne doivent pas être stockés en attributs d'une servlet car celle-ci est partagée par les threads utilisateurs



sauvegarde des informations utilisateurs

- trois mécanismes principaux peuvent être mis en oeuvre par les servlets pour maintenir l'état d'une conversation:
 1. champs cachés
 2. cookies
 3. session



champs cachés

- le mécanisme des champs cachés (hidden fields) possède l'avantage de ne pas pouvoir être invalidé, mais les données véhiculées peuvent être espionnées facilement
 - il s'agit de transmettre des chaînes de caractères en valeur de paramètres à l'intérieur de champs d'un formulaire qui ne seront pas visibles du client
 - lors de la requête du client sur le formulaire comportant ce(s) champ(s), les paramètres et leurs valeurs seront transmises comme le sont habituellement ceux d'un formulaire



cookies

- les cookies sont un procédé par lequel un serveur émet des informations sous la forme de chaînes de caractères vers le client qu'il fera mémoriser chez ce dernier et qu'il récupèrera ultérieurement
 - les servlets envoient des cookies vers les clients en insérant des champs dans les entêtes des réponses HTTP
 - les navigateurs renvoient automatiquement les cookies en insérant des champs dans les entêtes des requêtes HTTP
 - dans Java, un cookie est une instance de la classe **Cookie**



sessions

- afin de rendre les informations utilisateurs plus confidentielles, il est préférable de faire appel aux objets "session"
 - ces objets sont créés par le serveur, sur demande d'une servlet, afin d'y stocker des informations sous la forme d'objets
 - un objet session est systématiquement associé par le serveur à un identifiant unique (chaîne de caractères) stocké dans un cookie nommé **jsessionid**
 - c'est ce cookie qui permet ensuite au serveur de retrouver l'objet session associé à l'utilisateur



sessions

- ce mécanisme permet à une servlet de mémoriser des informations dans un objet situé sur le serveur afin de pouvoir les relire ultérieurement



ré-écriture d'URL

- lorsque le navigateur du client refuse les cookies ou lorsqu'il est incapable de les gérer, on peut utiliser le mécanisme de ré-écriture d'URL (URL rewriting) pour véhiculer l'identifiant de session
 - l'identifiant de session est automatiquement rajouté aux URL qui figurent dans les pages web renvoyées en réponse au client
 - liens hypertextes
 - Formulaires
 - les requêtes ultérieures sur ces URL permettront au serveur de récupérer l'identifiant de session



redirections

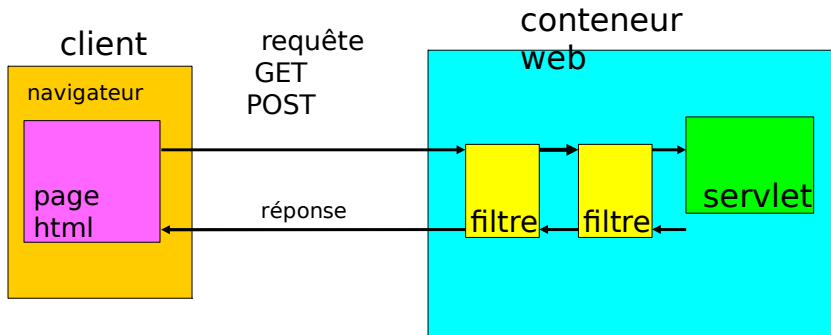
- une servlet peut rediriger la requête vers une page HTML, JSP ou servlet, au moyen d'une redirection interne ou externe
- redirection externe:
 - la réponse HTTP comporte l'URL vers laquelle le navigateur doit émettre une requête GET
 - avantage: l'URL de la page cible apparaît dans le navigateur
- redirection interne:
 - la servlet propage la requête vers une autre servlet ou JSP, sans faire appel au navigateur du client



filtres de servlets

- les filtres de servlets sont des objets destinés à traiter:
 - les requêtes avant qu'elles parviennent aux servlets
 - les réponses avant qu'elles parviennent au navigateur
- un filtre peut notamment modifier ou intercepter les requêtes ou les réponses
- les filtres peuvent être chaînés les uns aux autres

filtres de servlets





filtres de servlets

- quelques applications des filtres:
 - authentification des utilisateurs
 - traçage de requêtes
 - conversion d'échelle pour images
 - compression de données
 - adaptation automatique à la langue utilisée



Les Websockets



WebSockets

- les WebSockets sont une nouveautés de Java EE 7
- elles permettent d'établir une connexion bidirectionnelle entre un serveur et un client
 - le serveur peut spontanément émettre des messages vers le client
 - la plupart des navigateurs supportent aujourd'hui le protocole WebSocket

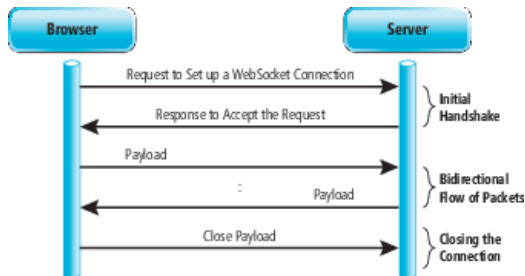


WebSockets

- les WebSockets sont plus efficaces et sont plus performantes que les autres solutions :
 - elles requièrent moins de bande passante car elles ne requièrent pas d'en-tête dans chaque message
 - la latence est réduite
 - elles permettent de mettre en place des solutions quasi temps réel pour recevoir des données
- les cas d'utilisation des WebSockets sont nombreux: ils sont utilisables dès que des données doivent être envoyées du serveur vers le ou les clients

WebSockets

- la mise en oeuvre des WebSockets nécessite plusieurs étapes :
 - établir une connexion
 - envoyer des messages côté client et serveur (bi-directionnel) indépendamment les uns des autres (full duplex)
 - fin de la connexion





WebSockets

- une connexion WebSocket est initialisée via le protocole HTTP:
 - chaque connexion à une WebSocket débute par une requête HTTP qui utilise l'option upgrade dans son en-tête
 - cette option permet de préciser que le client souhaite que la connexion utilise un autre protocole, en l'occurrence le protocole WebSocket

```
GET /path/to/websocket/endpoint HTTP/1.1
Host: localhost
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: xqBt3ImNzJbYqRINxEFlkg==
Origin: http://localhost
Sec-WebSocket-Version: 13
```



WebSockets

- lorsque le serveur répond, la connexion est établie et le client et le serveur peuvent envoyer et recevoir des messages

HTTP/1.1 101 Switching Protocols

Upgrade: websocket

Connection: Upgrade

Sec-WebSocket-

Accept:

K7DJLdLooIwIG/M0pvW

FB3y3FE8=



WebSockets

- une fois la connexion établie, le protocole HTTP n'est plus utilisé au profit du protocole WebSocket
 - c'est toujours le client qui initie une demande de connexion
 - le serveur ne peut pas initier de connexions mais il est à l'écoute des clients qui le contacte pour créer une connexion
 - la fermeture de la connexion peut être à l'initiative du endpoint client ou serveur pour permettre de passer la WebSocket à l'état déconnecté
- une WebSocket est identifiée par une URI particulière dont la syntaxe générale est :
`ws(s)://host[:port]path[?param]`



Les Java Server Pages (JSP)



présentation

- Java EE 7 considère obsolète les Java Server Pages (JSP) comme technologie de présentation, au profit des facelets utilisées dans Java Server Faces (JSF)
- néanmoins, étant très répandue, il paraît utile de présenter cette technologie

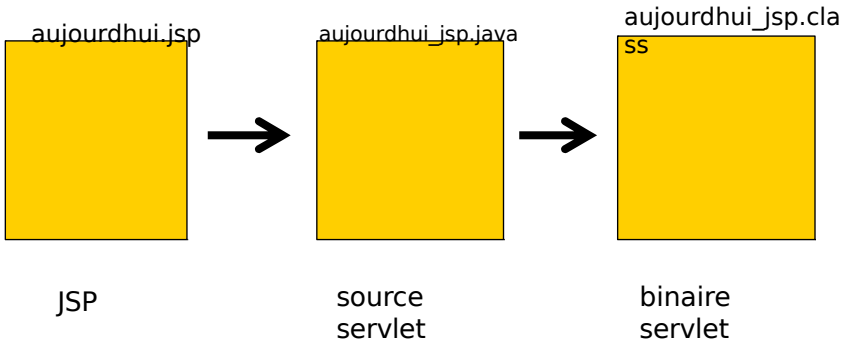


principe des JSP

- les Java Server Pages (JSP) ont l'apparence de pages HTML incorporant à certains endroits des balises particulières, à l'intérieur desquelles se trouvent des instructions Java
- le serveur, qui doit être compatible, génère automatiquement une servlet à partir de la JSP dès qu'il reçoit une première requête sur cette JSP
- la servlet ainsi générée émet vers le client les balises HTML standards qui se trouvent dans la JSP et exécute de plus les instructions Java qu'il y rencontre

principe des JSP

- le but des JSP est de faciliter l'écriture de pages HTML au contenu dynamique, l'utilisation de servlets seules étant souvent fastidieuse





principe des JSP

- soit la JSP **aujourd'hui.jsp**:

```
<html>
  <head>
    <title>date du jour</title>
  </head>
  <body>
    <h2>Aujourd'hui, nous sommes le:
    <%= new java.util.Date() %>
    </h2>
  </body>
</html>
```



principe des JSP

- code source de la servlet générée par le serveur Tomcat à partir de la JSP:

```
public class aujourd'hui_jsp extends HttpJspBase {  
    private static java.util.Vector _jspx_includes;  
    public java.util.List getIncludes() {  
        return _jspx_includes;  
    }  
    public void _jspService(HttpServletRequest request,  
        HttpServletResponse response) throws java.io.IOException,  
        ServletException {  
        JspFactory _jspxFactory = null;  
        javax.servlet.jsp.PageContext pageContext = null;  
        HttpSession session = null;  
        ServletContext application = null;  
        ServletConfig config = null;  
        JspWriter out = null;  
        Object page = this;  
        JspWriter _jspx_out = null;
```



principe des JSP

- code source de la servlet (suite):

```
try {  
    _jspxFactory = JspFactory.getDefaultFactory();  
    response.setContentType("text/html;charset=ISO-8859-1");  
    pageContext = _jspxFactory.getPageContext(this,  
        request,  
        response,null, true, 8192, true);  
    application =  
        pageContext.getServletContext(); config =  
        pageContext.getServletConfig(); session =  
        pageContext.getSession();  
    out = pageContext.getOut();  
    _jspx_out = out;  
    out.write("<html>\r\n");  
    out.write("<head>");  
    out.write("<title>date du  
jour"); out.write("</title>");  
    out.write("</head>\r\n");  
}
```



principe des JSP

- code source de la servlet (suite):

```
out.write("<body>\r\n");
    out.write("<h2>Aujourd'hui, nous sommes le:\r\n");
    out.print( new java.util.Date() );
    out.write("\r\n");
out.write("</h2>\r\n");
    out.write("</body>\r\n");
out.write("</html> \r\n");
} catch (Throwable t)
{
    out = _jspx_out;
    if (out != null && out.getBufferSize() != 0) out.clearBuffer();
    if (pageContext != null) pageContext.handlePageException(t);
} finally { if (_jspxFactory != null)
_jspFactory.releasePageContext(pageContext);
}
}
}
```




principe des JSP

- la servlet générée par le serveur à partir de la JSP comporte une méthode **_jspService** qui est exécutée lors d'une requête sur la JSP
- cette méthode est exécutée quelle que soit le type de la requête HTTP (GET, POST, PUT etc...) reçue
- la méthode **_jspService** contient du code java reproduisant le code HTML de la JSP, et inclut de plus le code java présent dans la JSP
 - par exemple, la balise `</body>` de la JSP est traduite en:
`out.write("</body>\r\n");`



déploiement des JSP

- les JSP sont déployées comme le sont les pages HTML
- il suffit donc de les placer dans des répertoires appropriés sur le serveur HTTP, comme on le ferait pour des pages HTML
- elles sont en général invoquées depuis une page HTML
- les classes Java qu'elles utilisent (JavaBean) doivent se trouver dans le répertoire **classes** de l'application ou dans une librairie, elle-même située dans le répertoire **lib** de l'application web



intérêt des JSP

- les servlets ont la capacité d'émettre des réponses aux requêtes HTTP, notamment des pages HTML construites dynamiquement à l'aide de l'instruction `out.println("....");`
- le code HTML inclus dans les instructions `println` n'est pas accessible par les outils de développement de pages web, ce qui rend difficile la mise-au-point et la maintenance des pages web élaborées de cette manière
- les JSP facilitent donc la conception de pages dynamiques et complètent particulièrement bien les servlets



Java Server Page Standard Tag Library (JSTL)



JSTL

- la JSTL est un ensemble de balises personnalisées destinées à éviter au développeur de JSP l'usage du code Java dans ses JSP

fonctionnalités	TLD	URI
fonctions de base	c.tld	http://java.sun.com/jsp/jstl/core
traitements XML	x.tld	http://java.sun.com/jsp/jstl/xml
internationalisation	fmt.tld	http://java.sun.com/jsp/jstl/fmt
fonctions	fn.tld	http://java.sun.com/jsp/jstl/functions
traitements SQL	sql.tld	http://java.sun.com/jsp/jstl/sql



mise en oeuvre

- exemple de JSP:

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core"
    prefix="c" %>
<html>
<head>
    <title>JSTL example</title>
</head>
<body>
<%
    int valeur=101;
    pageContext.setAttribute("valeur",new Integer(valeur));
    %>
    <c:out value="valeur =${pageScope.valeur}" />
    <br/>
</body>
</html>
```



le langage EL

- EL (Expression language) est un langage destiné à faciliter la manipulation d'objets situés dans les différents espaces de stockage d'une application web
- syntaxe de base :
 - `${xxxx}`
 - où xxxx désigne le nom d'une variable ou d'un objet dans un espace de stockage donné
- EL permet d'éviter l'utilisation de code Java
 - `${sessionScope.personne.nom}`
 - est équivalent à:
`<%=session.getAttribute("personne").getNom()%>`



Architecture MVC



Architecture MVC

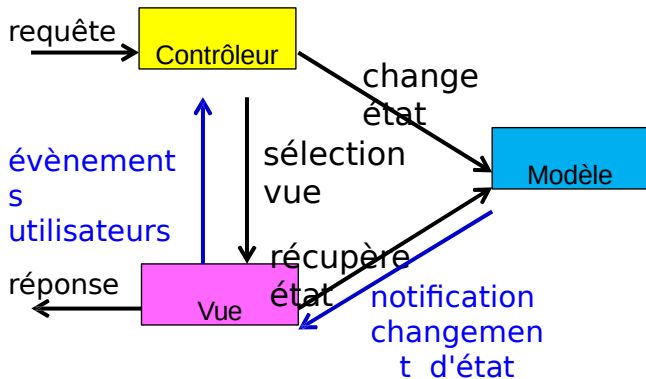
- Le pattern MVC
- JSF, Validation, Modèle de navigation
- Le support pour Ajax
- Spring MVC



pattern MVC

- le pattern Modèle Vue Contrôleur (MVC) sépare l'application en trois couches distinctes:
 - Le Modèle
 - contient les données à visualiser et les traitement métiers
 - La Vue
 - contient les composants destinés à l'affichage des données
 - Le Contrôleur
 - assure la gestion d'évènements issus de la vue, l'alimentation des composants graphiques avec les données du modèle
- le couplage entre ces couches doit être le plus faible possible
 - les frameworks facilitent sa mise en oeuvre

pattern MVC





Java Server Faces



introduction

- Java Server Faces (JSF) désigne un Framework pour le développement d'applications web en architecture MVC
 - JSF est un standard faisant partie de Java EE et s'appuie sur les technologies: Servlet, JSP/JSTL, Facelets
- principaux concurrents:
 - Struts 2
 - Spring MVC



introduction

- historique:
 - JSF 2.1 (10/2010)
 - version actuelle. Changements mineurs par rapport à la version 2.0
 - JSF 2.0 (06/2009)
 - version majeure comportant de nombreuses améliorations
 - intégré à Java EE 6
 - JSF 1.2 (05/2006-05)
 - nombreuses améliorations, intégré à Java EE 5
 - JSF 1.1 (05/2004)
 - correction de bogues, sans amélioration
 - JSF 1.0 (03/2004)
 - première version



objectifs

- le modèle de programmation et les bibliothèques de balises ont pour objectif de faciliter la construction et la maintenance de la couche de présentation des applications web
 - avec JSF, il est possible de :
 - ajouter des composants d'interface à une page (Drag and Drop)
 - lier les interactions utilisateur (événements) à du code côté serveur
 - lier l'état des composants à des données serveurs
 - réutiliser et étendre les composants existants
 - sauvegarder et restaurer l'état d'une page entre différentes requêtes



services rendus par JSF

- architecture MVC pour séparer l'interface utilisateur, la couche de persistance et les processus métier, utilisant la notion d'événement
- conversion des données et validation des données
- automatisation de l'affichage des messages d'erreur en cas de problèmes de conversion ou de validation
- internationalisation
- support d'Ajex sans programmation javascript (communication en arrière-plan et mise à jour partielle de l'interface utilisateur)



services rendus par JSF

- fournit des composants standards simples pour l'interface utilisateur
- possibilité d'ajouter ses propres composants
- adaptable à d'autres langages de balise que HTML (WML par exemple pour les téléphones portables)



pattern MVC

- JSF respecte le pattern MVC (Modèle Vue Contrôleur)
 - Modèle: représenté par des classes Java, les backing beans, sur le serveur
 - managed beans « données » (dont les propriétés sont liées aux *UIComponents*)
 - services métier des couches plus basse
 - Vue: les vues sont représentées par des composants Java sur le serveur, transformées en pages HTML sur le client
 - arbre de Composants d'IHM (UIComponents), balises JSF standard ou provenant de librairies JSF
 - indépendants de la technologie d'affichage (HTML en standard et notion de Render Kits)



pattern MVC

- Contrôleur: le contrôleur est une servlet qui intercepte les requêtes HTTP liées aux applications JSF et qui organise les traitements JSF
 - servlet JSF + managed beans « contrôleur »
 - gestionnaires d'évènements sur les composants
 - gestion des règles de navigation entre les vues
 - validation du contenu, conversion des données



implémentations

- implémentations de base:
 - Mojarra de Sun/Oracle
 - MyFaces d'Apache
- librairie de composants:
 - Richfaces (by Exadel/Jboss)
 - IceFaces
 - PrimeFaces



architecture d'une application JSF

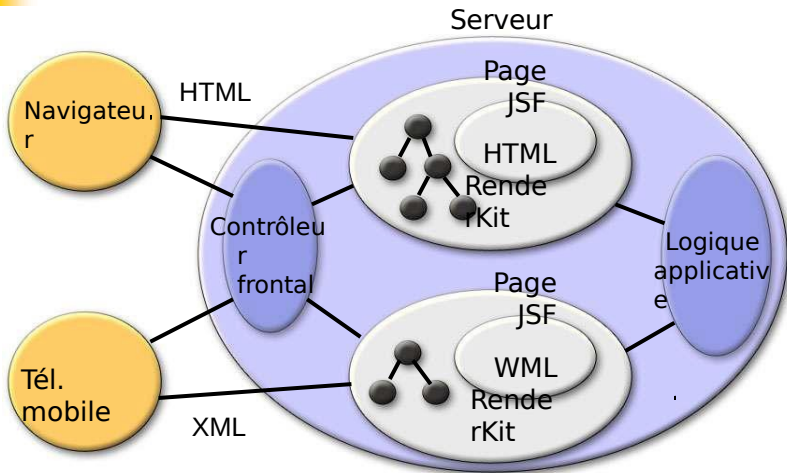
- une application JSF est une application web Java EE ayant les particularités suivantes:
 - une configuration dans **web.xml** qui précise la servlet contrôleur **FacesServlet** et son **url-pattern**
 - une collection de fichiers de facelets d'extension **.xhtml**
 - des managed beans, déclarés dans **faces-config.xml** ou par des annotations



architecture d'une application JSF

- le **contrôleur** est une servlet **FacesServlet** de JSF
 - elle doit être déclarée dans le fichier **web.xml** (servlet 2.5)
 - toute requête est dirigée vers cette servlet
 - en l'absence de **web.xml** (servlet 3.0), la servlet **FacesServlet** est automatiquement mappée vers les url: `/faces/*`, `*.jsf`, `*.faces`
 - elle exploite un fichier **faces-config.xml** ou des annotations
 - déclaration, initialisation durée de vie des beans
 - logique de navigation (cinématique)
 - ressources (messages internationalisés)

architecture d'une application JSF





architecture d'une application JSF

- la **Vue** JSF correspond à un ensemble de fichiers d'extension **.xhtml** appelés **facelets** comportant des balises HTML et JSF
 - les balises sont associées à des composants d'IHM structurés en arbre sur le serveur
 - les composants d'IHM sont associés à :
 - des gestionnaires d'évènements
 - des validateurs et des convertisseurs de données
- le **Modèle** correspond à des beans standards ou des beans gérés par JSF (managed beans)



principe de JSF

- JSF utilise des composants côté serveur pour construire la page Web
 - par exemple, un composant java `UIInputText` du serveur sera représenté par une balise `<INPUT>` dans la page XHTML
- une page Web sera représentée par une vue, `UIViewRoot`, hiérarchie de composants JSF qui reflète la hiérarchie de balises HTML
- JSF synchronise les composants Java de JSF avec les composants graphiques, principalement HTML



premiers pas

- le développement d'une application JSF simple suppose:
 - d'effectuer le mapping de la servlet **FacesServlet** dans **web.xml**
 - de développer des managed beans
 - de créer des pages web comportant des balises JSF



premiers pas

- mapping de la servlet **FacesServlet** dans **web.xml**

```
<servlet>
  <servlet-name>Faces Servlet</servlet-name>
  <servlet-class>
    javax.faces.webapp.FacesServlet
  </servlet-class>
  <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
  <servlet-name>Faces Servlet</servlet-name>
  <url-pattern>/faces/*</url-pattern>
</servlet-mapping>
```



premiers pas

- ❖ développement d'un managed bean avec annotations

- classe **Bonjour.java**

```
package pp;
import javax.faces.bean.ManagedBean;

@ManagedBean
public class Bonjour{
    final String message= "Bonjour à tous!";

    public String getMessage() { return
        message;
    }
}
```

- une instance de cette classe a pour nom `bonjour` par défaut
- cette instance comporte une propriété nommée `message`



premiers pas

- création d'une page web JSF
 - fichier **bonjour.xhtml**

```
<html lang="en" xmlns="http://www.w3.org/1999/xhtml"  
      xmlns:h="http://java.sun.com/jsf/html">
```

```
<h:head>
```

```
    <title>Facelet Bonjour</title>
```

```
</h:head>
```

```
<h:body>
```

```
    #{bonjour.message}
```

```
</h:body>
```

```
</html>
```



présentation

- une vue JSF peut être définie comme:
 - page JSP contenant une balise f:view (premières versions de JSF)

```
<%@ taglib uri="http://java.sun.com/jsf/core"
  prefix="f"
%>

<%@ taglib uri="http://java.sun.com/jsf/html"
  prefix="h"
%>

<f:view>
...
<BODY>
. . .
</BODY></HTML>
</f:view>
```



présentation

- une vue JSF peut être définie comme:
 - facelet (la balise f:view n'est plus nécessaire)

```
<?xml version='1.0' encoding='UTF-8' ?>
<html xmlns="http://www.w3.org/1999/xhtml"
      xmlns:h="http://java.sun.com/jsf/html"
      xmlns:f="http://java.sun.com/jsf/core"
      xmlns:ui="http://java.sun.com/jsf/facelets">
<h:head>
    .
    .
</h:head>
<h:body>
    .
    .
    .
</h:body>
</html>
```



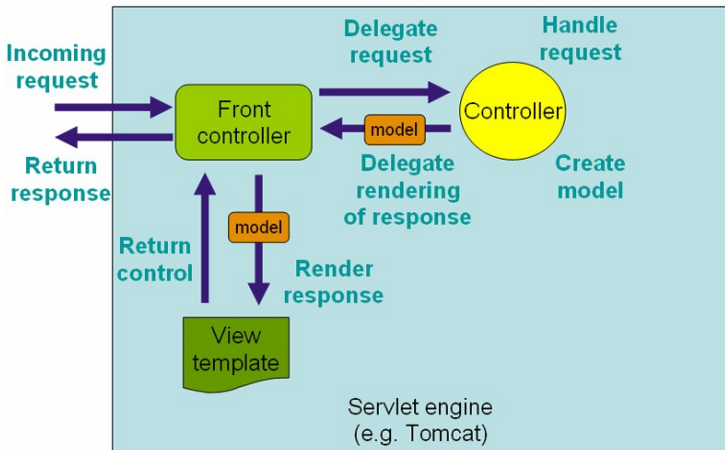
Spring MVC



introduction

- Spring MVC est un module de Spring qui permet de développer des applications web en architecture MVC, tout en bénéficiant des fonctionnalités propres à Spring (injection, AOP, transactions, etc...)
 - M comme **Modèle** :
 - les données (et traitements) de l'application
 - V comme **Vue** :
 - les interfaces graphiques qui présentent ces données
 - C comme **Contrôleur** :
 - la logique qui gère l'interaction entre Vue et Modèle

principe de Spring MVC





composants de Spring MVC

- composants fondamentaux de Spring MVC
 - DispatcherServlet
 - la servlet de String qui reçoit les requêtes
 - Controller
 - prend en charge une requête
 - HandlerMapping
 - associe une requête à un controleur
 - ViewResolver
 - associe un nom à une vue
 - ModelAndView
 - encapsule une vue et le(s) modèle(s) associé(s)



principe de Spring MVC

- la servlet **DispatcherServlet** reçoit la requête
 - elle consulte le **HandlerMapping** et invoque le **Controller** associé à la requête
 - le contrôleur traite la requête en faisant appel à la méthode appropriée et retourne un objet **ModelAndView** à la servlet **DispatcherServlet**
 - l'objet **ModelAndView** contient les données du modèle et le nom de la vue
 - la servlet **DispatcherServlet** envoie le nom de la vue à un **ViewResolver** pour trouver la vue à invoquer
 - la servlet **DispatcherServlet** transmet l'objet modèle à la vue pour afficher le résultat



Enterprise Java Beans (EJB)



EJB: présentation

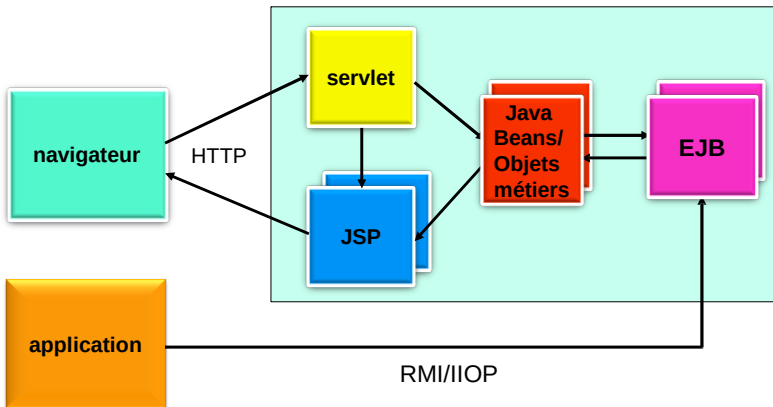
- les Enterprise Java Beans (EJB) sont des composants serveurs distribués développés en Java, destinés à être exécuté dans un conteneur EJB, le plus souvent intégré à un serveur d'applications
 - le fait qu'ils soient distribués apporte beaucoup de souplesse dans leur utilisation, puisque leur localisation physique importe peu
 - l'architecture EJB (Enterprise Java Beans) se veut être la convergence des architectures RMI et CORBA tout en simplifiant la mise-en-œuvre des composants distribués
 - l'intérêt majeur est la réduction considérable du temps de développement de ces composants serveurs dits EJB



EJB: présentation

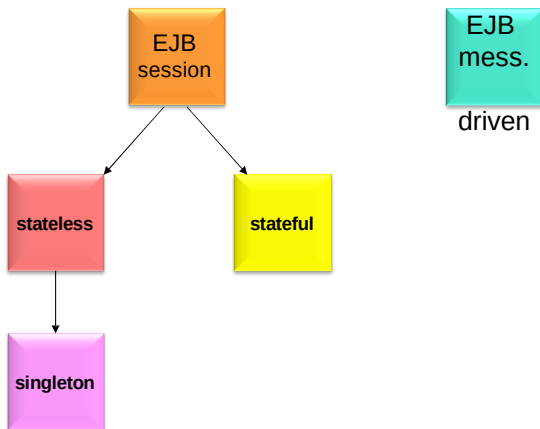
- le conteneur EJB est lui-même intégré à un serveur d'applications, qui fournit de nombreux services
- clients privilégiés des EJB :
 - les composants d'une application web (servlets, JSP, JavaBeans)
 - d'autres EJB
 - les applications autonomes en Java, en C++ ou C#
 - les applications clientes d'un web service

relations entre servlets, JSP, EJB



taxonomie des EJB 3

- l'architecture EJB 3 propose deux types d'EJB:





EJB session et message-driven

- les EJB session sont des composants distribués destinés à fournir des traitements "métier", par exemple des calculs de tarifs, des transactions bancaires, de la compression vidéo, des opérations en base de données
 - un EJB session est associé à un client unique à un instant donné, et ne peut être partagé
 - ce type d'EJB n'est pas persistant
 - sa durée de vie est celle de la session du client: de l'obtention de sa référence jusqu'à la fin de son utilisation.
 - il disparaît également lorsque le serveur s'arrête de fonctionner
 - un EJB message driven est destiné à recevoir des messages asynchrones JMS, autorisant un couplage faible entre émetteur et récepteur de messages



EJB session



session stateless ou stateful ?

- un EJB session stateless ne maintient pas l'état conversationnel du client entre deux appels de méthodes
 - en pratique, le client doit lui fournir les données nécessaires par le biais d'arguments transmis aux méthodes de l'EJB, ou par le biais d'une ressource extérieure comme une base de données
 - si un EJB session stateless comporte des attributs, ceux-ci ne doivent pas représenter des données propres à un utilisateur, mais communes à ces derniers
 - un EJB session stateless singleton est instancié une seule fois dans une application



session stateless ou stateful ?

- un EJB session stateful maintient l'état conversationnel du client entre les appels de méthodes et les transactions
 - l'état conversationnel du client apparaît dans la valeur des attributs de l'EJB stateful (ainsi que dans celle de ses intercepteurs)
 - dans certains cas, l'état conversationnel peut être associé à des ressources extérieures, comme des connexions réseau ou des bases de données

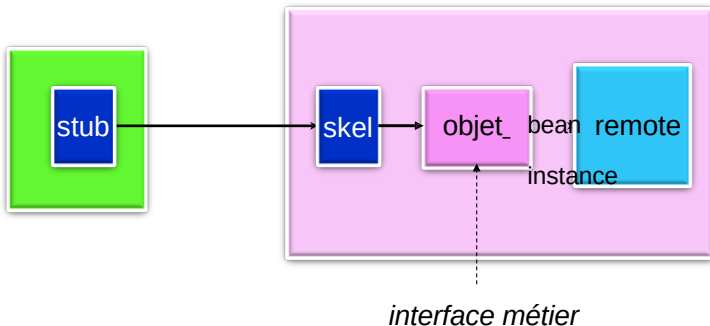


Constitution

- le cœur d'un EJB est un objet, dit "*Bean instance*", dont les méthodes sont invoquées par un client
 - le client n'invoque jamais directement les méthodes de la *Bean instance*, mais au travers d'un objet dit "*interface métier*", qui relaie les appels du client vers la *Bean instance*
 - assurant l'interface entre le client et la *Bean instance*, l'*interface métier* expose les méthodes que le client peut invoquer sur la *Bean instance*
 - l'*interface métier* permet au conteneur de gérer tous les aspects délicats des services serveur, comme l'accès réseau, la gestion des transactions, la gestion de la sécurité, la gestion d'un pool de Bean instances

EJB session: les interfaces distantes

en accès distant, les arguments et les résultats des méthodes sont transmis par valeur (donc sérialisés)

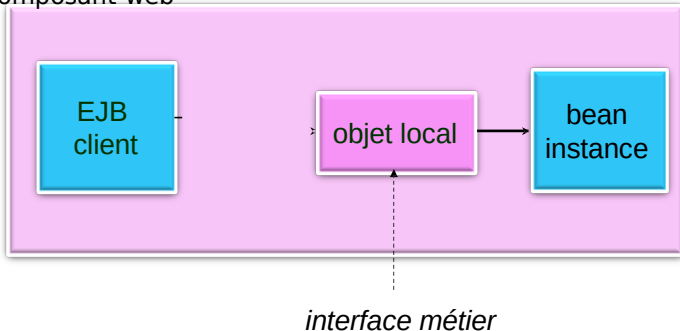


EJB session: les interfaces locales

en accès local, les arguments et les résultats des méthodes sont transmis de la même façon qu'en Java standard:

- par valeur pour les types primitifs
- par référence pour les objets

l'accès local à un EJB n'est possible que si le client se situe dans la même JVM que le conteneur de l'EJB: ce peut être un autre EJB ou un composant web





EJB session: accès local ou distant

- choix entre un accès local ou distant à un EJB:
 - l'accès distant permet de s'affranchir de la localisation physique de l'EJB
 - en accès distant, les objets transmis aux méthodes de l'EJB et retournés par elles doivent être sérialisables
 - l'accès distant est pénalisant en terme de performance, à cause notamment de la couche réseau, de la sérialisation des données et de leur transmission par valeur
 - l'accès local impose que le client et l'EJB soient exécutés dans la même machine virtuelle, ce qui restreint l'intérêt de la distribution d'objets
- l'accès local favorise les EJB à fine granularité



EJB session stateless: exemple

- classe Bean de l'EJB:

@Stateless

```
public class CalculatorBean implements Calculator{  
    public int add(int x, int y){return x + y;}  
    public int subtract(int x, int y){return x - y;}  
}
```

- interface métier distante de l'EJB:

@Remote

```
public interface Calculator{  
    public int add(int x, int y);  
    public int subtract(int x, int y);  
}
```



EJB session singleton

- un singleton est un type d'EJB session annoté **@Singleton** dont une seule instance existe dans le conteneur
- les applications sont nombreuses et concernent souvent la mise en cache et la concurrence d'accès



EJB session stateful: exemple

- classe Bean de l'EJB:

```
import javax.ejb.Stateful;
import java.io.Serializable;
@Stateful(name="CaddieEJB")
@Local(Caddie.class)
public class CaddieBean implements Caddie,
    Serializable{
    private Prix prix;
    private Vector<Produit> panier; public
    void ajoutProduit(Produit p){...}
    public boolean retireProduit(Produit p, int
        quantite){...}
    public void reset(){...}
    @Remove public void valide(){...}
}
```



EJB session stateful: exemple

- Interface métier locale de l'EJB:

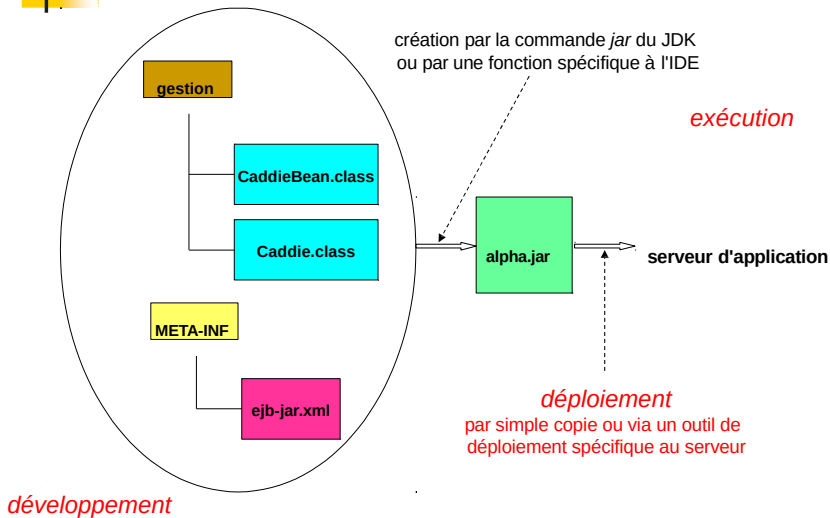
```
public interface Caddie{  
    public void ajoutProduit(Produit p);  
    public boolean retireProduit(Produit p, int  
        quantite);  
    public void reset();  
}
```



déploiement d'EJB

- le déploiement simplifié d'EJB consiste tout d'abord à placer les fichiers .class constituant l'EJB dans un fichier .jar
- un descripteur de déploiement **ejb-jar.xml** peut figurer de façon optionnelle dans un répertoire **META-INF**
- lorsque le fichier .jar est déployé dans le serveur, ce dernier extrait le contenu du fichier .jar et détecte par introspection le rôle des fichiers fournis grâce aux annotations

déploiement d'EJB





déploiement d'EJB

- un déploiement au sein d'une application d'entreprise consiste à créer un fichier jar d'extension **.ear** et à le déployer ensuite dans le serveur
- c'est la solution à retenir lorsque l'application est constituée de composants web et d'EJB
- le fichier **ear** doit contenir:
 - le fichier **war** contenant les composants web
 - le(s) fichier(s) jar contenant les EJB
 - un descripteur **application.xml** dans un répertoire **META-INF**

déploiement d'EJB

■ exemple: **application.xml**

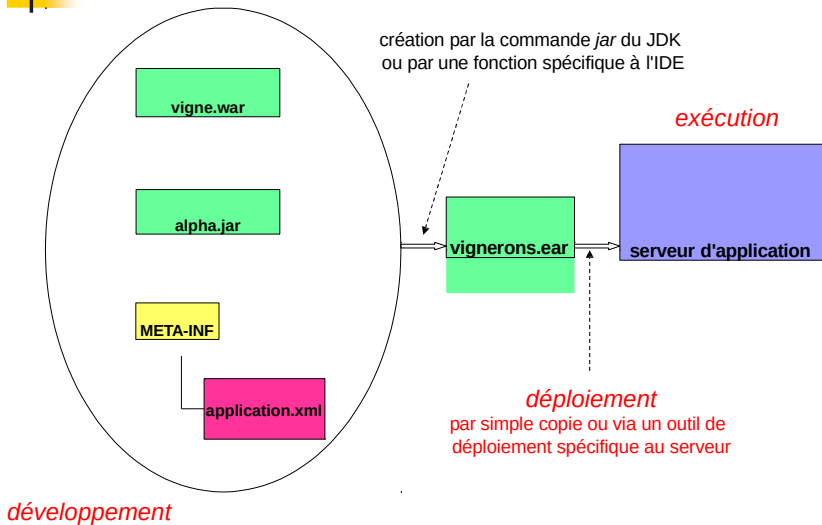
```
<?xml version="1.0" encoding="UTF-8"?>
<application version="5" ...>
  <display-name>vignerons</display-name>
  <module>
    <web>
      <web-uri>vigne.war</web-uri>
      <context-root>vignerons</context-root>
    </web>
  </module>
  <module>
    <ejb>alpha.jar</ejb>
  </module>
</application>
```

nom du fichier war
contenant les servlets/JSP

nom de l'application

nom du fichier jar contenant des EJB

déploiement d'EJB





déploiement d'EJB

- lorsqu'un EJB 3 est déployé dans le serveur, il est enregistré dans le service de nommage du serveur (JNDI: Java Naming and Directory Interface)
- trois espaces de noms JNDI sont disponibles pour l'accès à un EJB
 - l'espace **java:global** permet l'accès distant (depuis un
 - l'espace **java:module** permet l'accès depuis un composant du même module
 - l'espace **java:app** permet l'accès depuis tout module de la même application (même ear)



le service d'annuaire JNDI

- un service de nommage a pour rôle d'associer des noms à des objets ou des ressources, permettant ensuite de localiser un objet ou une ressource en utilisant le nom qui lui a été donné
- un service de nommage offre deux services:
 - un service d'association (binding) d'une ressource ou d'un objet à un nom
 - un service de consultation (lookup) permettant de localiser la ressource ou l'objet à partir de son nom



le service d'annuaire JNDI

- les services d'annuaires (directory services) sont des services de nommage avancés qui permettent d'organiser les ressources, les objets de façon arborescente, fournissant également des méta- données sur ces ressources ou ces objets
- tout serveur d'application comporte un service de nommage, accessible via les API JNDI
- L'API JNDI est utilisée (entre autre) pour localiser les EJBs



le client d'un EJB session

- le client d'un EJB peut être une application, un composant web, un autre EJB
- avant d'invoquer les méthodes métiers sur l'EJB, le client doit obtenir une référence sur l'interface métier (locale ou distante) de l'EJB, au choix:
 - par un lookup JNDI
 - seule possibilité depuis une application Java SE
 - par une injection de référence sur l'EJB (plus simple, mais possible uniquement si le client est un EJB ou une application web)



EJB et JPA (Java Persistence API)



présentation

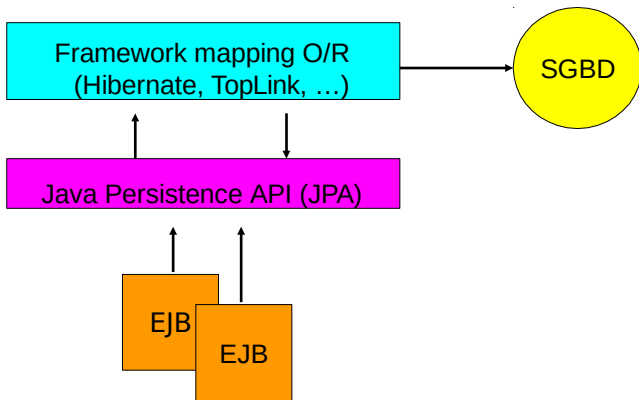
- un bean entité représente nécessairement des données persistantes comme un compte bancaire, un fournisseur, un article
- il comporte des attributs associés à des champs d'une ou plusieurs tables d'une base de données relationnelle (mapping Objet/Relationnel)
- en EJB 3, un bean entité est un simple objet Java (POJO) comportant des annotations liées au mapping O/R



présentation

- la plupart du temps, les beans entités sont gérés par des EJB session
- la persistance est assurée par un service appelé **EntityManager**
- JPA (Java Persistence Api) est le nom donné aux API qui donnent accès à la couche de persistance
 - ces API se trouvent dans le package **javax.persistence**
 - la couche de persistance est généralement implémentée sous la forme d'un framework de mapping O/R:
 - Hibernate (par défaut dans JBoss, Open Source)
 - TopLink (par défaut dans GlassFish, Open Source d'origine Oracle)

présentation





conception

- l'élaboration d'un bean entité s'effectue simplement:
 - créer une classe* comme pour les Java Beans avec une annotation **@Entity** sur la classe et une annotation **@Id** sur un attribut
 - ou bien renseigner le descripteur de déploiement
- par défaut, le bean entité est associé à une table du nom de sa classe
- un bean entité est toujours associé à une clé primaire (primary key) qui permet d'identifier ce bean de façon unique dans la base de données



conception

- la clé primaire doit être un champ ou une propriété du bean entité, annotée avec `@javax.persistence.Id`
- le nom des colonnes est par défaut:
 - celui des champs (Field) du bean si l'annotation `@Id` est placée sur un champ
 - celui des propriétés (Property) du bean si l'annotation `@Id` est placée sur une propriété
- tous les champs sont persistants par défaut
 - l'annotation `@Transient` signale un champ ou une propriété comme non persistante

exemple de bean entité

- dans l'exemple suivant:
 - la classe **Personne** est associée à une table **Personne**
 - la clé primaire est désignée par le champ **id**
 - chacun des champs **id**, **nom**, **prenom** est mappé respectivement vers les colonnes **id**, **nom**, **prenom** de la table **Personne**

@Entity

```
public class Personne implements Serializable{  
    @Id private int id;  
    private String nom;  
    private String prenom;  
  
    public Personne(){}  
    public int getId(){return id;}  
    public void setId(int pk){id=pk;}  
    public String getNom(){return nom;}  
    public void setNom(String n){nom=n;}  
    public String getPrenom(){return prenom;}  
    public void setPrenom(String p){prenom=p;}  
}
```

la classe **Personne** est mappée
sur une table **Personne**

l'attribut **id** désigne la clé
primaire
le mapping est défini sur
les attributs du bean

les attributs **id**, **nom**, **prenom**
sont mappés respectivement
sur
les colonnes **id**, **nom**, **prenom**

exemple de bean entité

■ exemple: classe **Personne**

```
@Entity @Table(name="PERSONNEL")
```

la classe **Personne** est mappée
sur une table **PERSONNEL**

```
public class Personne implements  
    Serializable{ private int id;  
    private String nom;  
    private String prenom;  
    private int age;  
    public Personne(){  
        @Id @Column(name="MATRICULE") public  
        int getId(){return id;} public void  
        setId(int pk) {id = pk;}  
        @Column(name="NOM")  
        public String getNom() {return nom;}  
        public void setNom(String n){nom = n;}  
        @Column(name="PRENOM_1")  
        public String getPrenom() {return prenom;}  
        public void setPrenom(String p){prenom = p;}  
        @Transient  
        public String getAge() {return age;}  
        public void setAge(int a){age = a;}  
    }
```

l'attribut **id** désigne la clé primaire
le mapping est défini sur
les propriétés du bean
la propriété **id** est mappée
sur la colonne
MATRICULE

les propriétés **nom** et **prenom** sont
mappées
respectivement sur les colonnes
NOM et **PRENOM_1**

la propriété **age** n'est pas
persistante

fichier de mapping

- en remplacement ou en compléments d'annotations, le mapping O/R peut être spécifié dans un fichier nommé **orm.xml** déployé dans le répertoire META-INF

```
<entity-mappings version="1.0" ...>
  <entity class="dma.Personne" access="PROPERTY">
    <table name="PERSONNEL"/>
    <attributes>
      <id name="id">
        <column name="MATRICULE"/>
      </id>
      <basic name="nom">
        <column name="NOM"/>
      </basic>
      <basic name="prenom">
        <column name="PRENOM_1"/>
      </basic>
      <transient name="age" />
    </attributes>
  </entity>
</entity-mappings>
```

le mapping concerne la classe entité dma.Personne

le bean entité est mappé sur la table PERSONNEL

le mapping est défini sur les propriétés du bean (FIELD sur les attributs)

la clé primaire est la propriété id mappée sur la colonne MATRICULE

la propriété nom est mappée sur la colonne NOM

la propriété prenom est mappée sur la colonne PRENOM_1

la propriété age n' est pas persistante



rôle de l'EntityManager

- l'instanciation d'une classe annotée avec **@Entity** n'entraîne pas automatiquement la persistance de l'instance
- il faut faire intervenir l'**EntityManager**, service qui gère la persistance des entités en base relationnelle:
 - recherche d'entités
 - insertions d'entités
 - suppression d'entités
 - synchronisation entre instances et base de données
 - requêtes
 - gestion des caches
 - interaction avec les services de gestion des transactions

le fichier persistence.xml

■ exemple:

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

```
<persistence version="1.0" ...>
```

```
  <persistence-unit name="entreprise">
```

```
    <jta-data-source>
```

```
      java:/DefaultDS
```

```
    </jta-data-source>
```

```
    <mapping-file>orm.xml</mapping-file>
```

```
    <properties>
```

```
      <property
```

```
        name="hibernate.hbm2ddl.auto"
```

```
        value="create-drop"/>
```

```
      </property>
```

```
    </persistence-unit>
```

nom donné à l'unité
de persistance

nom JNDI de la
DataSource

nom du fichier de
mapping

(orm.xml par défaut)

propriétés
spécifiques
à l'implémentation
de la couche de
persistance



exemple de mise en oeuvre

@Stateless

```
public class TravelAgentBean implements TravelAgentRemote{
```

```
    @PersistenceContext(unitName="titan")
```

```
    private EntityManager manager;
```

```
    public void createCabin(Cabin cabin){
```

```
        manager.persist(cabin);
```

```
    }
```

```
    public Cabin findCabin(int pKey){
```

```
        return manager.find(Cabin.class,  
            pKey);
```

```
    }
```

```
}
```



exemple de mise en oeuvre

```
@Entity
@Table(name="CABIN")
public class Cabin implements java.io.Serializable {
    private int id;
    private String name;
    private int deckLevel;
    private int shipId;
    private int bedCount;

    @Id @Column(name="ID")
    public int getId() {
        return id;
    }

    public void setId(int pk) {
        id = pk;
    }
}
```



exemple de mise en oeuvre

```
@Column(name="NAME")
public String getName() {
    return name;
}
public void setName(String str) {
    name = str;
}

@Column(name="DECK_LEVEL")
public int getDeckLevel()
{
    return deckLevel;
}
public void setDeckLevel(int level){
    deckLevel = level;
}
```



exemple de mise en oeuvre

```
@Column(name="SHIP_ID")
public int getShipId() {
    return shipId;
}

public void setShipId(int sid) {
    shipId = sid;
}

@Column(name="BED_COUNT")
public int getBedCount()

{
    return bedCount;
}

public void setBedCount(int bed)
    { bedCount = bed;
}

}
```



présentation

- les requêtes en bases de données assurant la persistance avec JPA sont formulées:
 - en JPQL
 - le langage JPQL (Java Persistence Query Language), issu de JPA, est un sous-ensemble de HQL
 - le langage HQL (Hibernate Query Language) est un langage de requêtes orienté objet dont l'intérêt est qu'il permet d'exprimer facilement des requêtes sur les objets persistants, sans se soucier de l'aspect lié aux bases de données
 - en SQL
 - le langage SQL peut dans certains cas être utilisé car il permet d'exploiter au mieux certaines fonctionnalités de la base de données
 - via les API **Criteria**



JPA Query Language (JPQL)

■ examples:

```
select p from Payment p
```

```
select c.nom from Client c
```

```
select c.adresse.ville from  
Client c
```

```
select cl from Client cl  
where cl.cp like '75%'
```

```
select p from Produit p where p.prix between 60.50 and  
100.00
```

```
select p from Produit p where p.prix >= 55.00  select c  
from Customer c where c.name = 'MARTIN'  select o from  
Order o where o.referenceNumber = 123
```

```
select o from Order o where o.total > 5000.00
```

```
select o from Order o where o.total > 5e+3
```

```
select count(*), sum( o.total ), avg( o.total ),  
min( o.total ), max( o.total ) from Order o
```

```
select count( distinct c.name ) from Customer c
```



JPA Query Language (JPQL)

- exemple avec jointures:

```
select p from Person p join p.events where p.id = 4
```

```
select p from Person p left join p.events  
    where p.id = 4
```

```
select c.id, c.name, sum( o.total ) from Customer c left  
    join c.orders o group by c.id, c.name
```

- exemple avec paramètres:

```
Query query=manager.createQuery(  
    "select c from Client c where c.ville= :town");  
query.setParameter("town", "Paris");  
List result=query.getResultList();
```

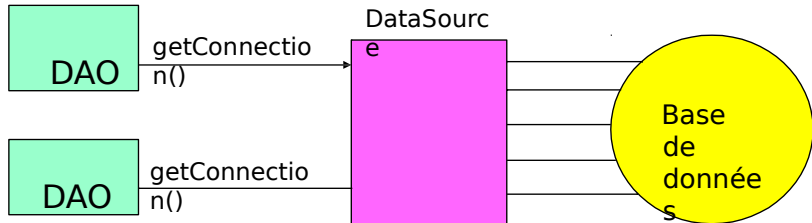



pools de connexions

- l'accès aux bases de données depuis une application web nécessite en général de bonnes performances, compte-tenu du nombre de requêtes potentielles
- afin d'éviter des temps de connexion pénalisants, les serveurs d'applications sont tous pourvus de pools (réserves) de connexions aux bases de données
 - un pool de connexions contient un certain nombre de connexions physiques ouvertes vers la base de données, l'utilisateur puisant dans ce pool en fonction de ses besoins
 - l'obtention d'une connexion (logique) s'effectue en s'adressant au pool, en lieu et place du SGBD, et devient une opération rapide

pools de connexions

- les pools de connexions sont des objets administrés du serveur, qu'il faut configurer via la console d'administration du serveur ou par modifications de fichiers de configuration XML





pools de connexions

- dans tous les cas, la configuration d'un pool de connexions nécessite au minimum les informations suivantes:
 - url de la base de données utilisée (protocole + nom)
 - classe principale du driver utilisé
 - nom JNDI donné au pool de connexions



services web SOAP avec JAX- WS

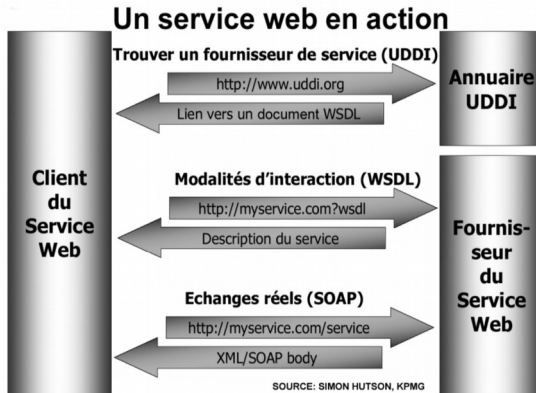
Services Web SOAP

Les API métier peuvent également être consommées par d'autres systèmes via les **services web SOAP** caractérisés par :

- leur grande interopérabilité,
- leur extensibilité
- et leurs descriptions pouvant être traitées automatiquement (WSDL).

Ces fonctionnalités sont apportées par XML

Scénario complet , service Web



Inconvénients de XML

Rigueur : L'utilisation de schéma crée des exigences supplémentaires

Verbosité : Le volume des données à échanger est fortement augmenté par les balises, les références aux schémas, les espaces de noms

Exigences sur le client : Le client doit intégrer un parseur XML

Exemple SOAP

```
<?xml version="1.0" encoding="UTF-8" standalone="no" ?>
<SOAP-ENV:Envelope
  SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/1999/XMLSchema">
  <SOAP-ENV:Body>
    <ns1:doubleAnInteger xmlns:ns1="urn:MySoapServices">
      <param1 xsi:type="xsd:int">123</param1>
    </ns1:doubleAnInteger>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```


WSDL

- **WSDL** est une normalisation regroupant la description des éléments permettant de mettre en place l'accès à un Web Service
- C'est un document XML qui débute par la balise **<definition>** et qui contient les balises suivantes :
 - ♦ **<message>** : définition de données en entrée et sortie du service
 - ♦ **<portType>** : description des méthodes disponibles sur le WS
 - ♦ **<binding>** : protocole de communication (RPC/DOCUMENT)
 - ♦ **<service>** : URL du service

Exemple (1/2)

```
<?xml version="1.0" ?>
<definitions name="WSbibliotheque"
targetNamespace="http://corail1.utt.fr:8080/ws/WSbiblio.wsd1"
xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns="http://schemas.xmlsoap.org/wsdl/" >
<!-- Données d'entrée/sortie -->
<message name="getPrixRequest">
  <part name="livre" type="xsd:string" />
</message>
<message name="getPrixResponse">
  <part name="return" type="xsd:float" />
</message>
<!-- Méthodes disponibles du service web -->
<portType name="WSbiblioPortType">
  <operation name="getPrix">
    <input message="getPrixRequest" name="getPrix"/>
    <output message="getPrixResponse" name="getPrixServeur"/>
  </operation>
</portType>
```

Exemple (2/2)

<!-- Protocole de communication -->

```
<binding name="WSbiblioBinding" type="WSbiblioPortType">
  <soap:binding style="rpc" transport="http://schemas.xmlsoap.org/soap/http"/>
  <operation name="getPrix">
    <soap:operation soapAction="" />
    <input><soap:body use="encoded" namespace="urn:cite_biblio"/></input>
    <output><soap:body use="encoded" namespace="urn:cite_biblio"/></output>
  </operation>
</binding>
```

<!-- URL du service -->

```
<service name = "WSBibliotheque">
  <documentation>Prix d'un livre</documentation>
  <port name = "WSbiblioPortType" binding="WSbiblioBinding">
    <soap:address location="http://corail1.utt.fr:8080/soap"/>
  </port>
</service>
</definitions>
```

Exemple JAX-WS

```
package helloservice;

import javax.xml.ws.WebService;
import javax.xml.ws.WebMethod;

@WebService
public class Hello {
    private final String message = "Hello, ";

    public Hello() {
    }

    @WebMethod
    public String sayHello(String name) {
        return message + name + ".";
    }
}
```

Consommation

```
import helloservice.endpoint.HelloService;
import javax.xml.ws.WebServiceRef;

public class HelloAppClient {
    @WebServiceRef(wsdlLocation =
        "http://localhost:8080/helloservice-war/HelloService?WSDL")
    private static HelloService service;

    private static String sayHello(java.lang.String arg0) {
        helloservice.endpoint.Hello port =
        service.getHelloPort();
        return port.sayHello(arg0);
    }
}
```

JAX-WS Annotations

@WebService : Classe d'implémentation du service Web

@SOAPBinding : Mapping SOAP

@WebMethod : Méthode exposée par le service

@RequestWrapper et **@ResponseWrapper** : Classe Java encapsulant les paramètres SOAP d'entrée ou de sortie

@WebFault : Mapping Exception Java vers *wsdl:fault*

@Oneway : Indique le service ne renvoie pas de réponse

@WebParam : Paramètre d'entrée/sortie du service

@WebResult : Spécifie la partie SOAP concernant la valeur de retour



services web RESTful avec JAX-RS



présentation

- REST signifie Representational State Transfer imaginé en 2000 par Roy Fielding
- un service web RESTful est un service web conforme au principe REST:
 - tout est ressource accessible via une URL
 - les ressources sont accessible par des requêtes HTTP
- aucun format des données n'est imposé pour l'échange des données:
 - XML
 - JSON
 - HTML
 - PDF



caractéristiques

- client-serveur
- sans état
 - cache: permet d'améliorer les performances
- interface unique: l'accès aux ressources s'effectue de manière uniforme (HTTP GET, POST, PUT, DELETE)
- les ressources sont nommées
- composants en couches: des serveurs proxy, passerelles, caches, etc.. peuvent s'interposer entre les clients et les ressources afin d'ajouter des fonctionnalités (performances, sécurité, etc...)



accès aux ressources

- chaque ressource (opération d'un service web) doit être associée à une URI :
<http://apha.com/beta/calcul>
<http://alpha.com/gamma/theta/maj/30>
- le type de requête HTTP détermine également la ressource associée à une URI donnée
 - une même URI peut donner accès à deux ressources distinctes
 - c'est le principe des servlets avec doGet et doPost



accès aux ressources

- en général, les requêtes HTTP utilisées correspondent aux opérations effectuées en base de données:
 - Create, POST
Insert:
 - Read, GET
Select:
 - Update: PUT
 - Delete: DELETE



JAX-RS

- JAX-RS (Java API for RESTful Web Service) est une API permettant de créer des services Web en architecture REST
- cette API fait un usage important d'annotations
- JAX-RS fait partie de Java EE
- cette API est également disponible via des frameworks:
 - Restlet
 - Jersey
 - CXF
 - RestEasy



annotations JAX-RS

Annotation Description

- @Path** The @Path annotation's value is a relative URI path indicating where the Java class will be hosted: for example, /helloworld. You can also embed variables in the URIs to make a URI path template. For example, you could ask for the name of a user and pass it to the application as a variable in the URI: /helloworld/{username}.
- @GET** The @GET annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP GET requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- @POST** The @POST annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP POST requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- @PUT** The @PUT annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP PUT requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
- @DELETE** The @DELETE annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP DELETE requests. The behavior of a resource is determined by the HTTP method to which the resource is



annotations JAX-RS

Annotation	Description
------------	-------------

@HEAD	The @HEAD annotation is a request method designator and corresponds to the similarly named HTTP method. The Java method annotated with this request method designator will process HTTP HEAD requests. The behavior of a resource is determined by the HTTP method to which the resource is responding.
--------------	---

@PathParam	The @PathParam annotation is a type of parameter that you can extract for use in your resource class. URI path parameters are extracted from the request URI, and the parameter names correspond to the URI path template variable names specified in the @Path class-level annotation.
-------------------	---

@QueryParam	The @QueryParam annotation is a type of parameter that you can extract for use in your resource class. Query parameters are extracted from the request URI query parameters.
--------------------	--

@Consumes	The @Consumes annotation is used to specify the MIME media types of representations a resource can consume that were sent by the client.
------------------	--

@Produces	The @Produces annotation is used to specify the MIME media types of representations a resource can produce and send back to the client: for example, "text/plain".
------------------	--

@Provider	The @Provider annotation is used for anything that is of interest to the JAX-RS runtime, such as MessageBodyReader and MessageBodyWriter.
------------------	---



URI statique

```
@Path("/helloWorld")
```

```
public class HelloWorld {
```

```
    @GET
```

```
    @Produces("text/html") // format des données renvoyées
```

```
    public String sayHello() {
```

```
        return "<h1>Hello World</h1>";
```

```
    }
```

```
    @POST
```

```
    @Consumes("text/plain") // format des données consommées
```

```
    public void storeMessage(String message) { ...}
```

```
}
```

```
}
```

- l'accès à l'opération sayHello sera effectué par une requête HTTP de type GET à l'URI:

`http://localhost:8080/Hello/res/helloWorld`



URI dynamique

```
@Path("/helloWorld")
public class HelloWorld {
    @GET
    @Path("/nom/{user}")
    @Produces("text/html") // format des données
    public String sayHello( @PathParam("user") String name) {
        return "<h1>Hello World"+name+"</h1>";
    }
}
```

- l'accès à l'opération sayHello sera effectué par une requête HTTP de type GET à l'URI:
`http://localhost:8080/Hello/res/helloWorld/nom/Martin`



URI dynamique

```
@Path("/helloWorld")
```

```
public class HelloWorld {
```

```
    @GET
```

```
    @Path("/nom")
```

```
    @Produces("text/html") // format des données
```

```
    public String sayHello( @QueryParam("user") String name){  
        return "<h1>Hello World"+name+"</h1>";
```

```
    }
```

```
    .
```

```
    }
```

- l'accès à l'opération sayHello sera effectué par une

requête HTTP de type GET à l'URI:

`http://localhost:8080/Hello/res/helloWorld/nom?user=Martin`



URI statique avec paramètre

- exemple avec paramétrée dans requête POST:

```
@Path("/insee")
```

```
public class WSGestionInsee {
```

```
    @POST
```

```
    @Path("/etatcivil")
```

```
    @Consumes("application/x-www-form-urlencoded")
```

```
    @Produces("text/html")
```

```
    public String inseeToEtatCivil(
```

```
        @FormParam("nir") String nir) {
```

```
        .  
        .  
        .  
    }
```

```
}
```

- l'accès à l'opération `inseeToEtatCivil` sera effectué par une requête HTTP de type POST à l'URI:

`http://localhost:8080/Hello/res/insee`



Json

- le format JSON (JavaScript Object Notation) est largement utilisé dans les échanges de données entre clients et service web RestFul
- un document JSON ne comprend que deux éléments structurels :
 - des ensembles de paires nom / valeur
 - des listes ordonnées de valeurs
- ces éléments représentent 3 types de données :
 - des objets ;
 - des tableaux ;
 - des valeurs génériques de type tableau, objet, booléen, nombre chaîne ou null



exemple Json

- format JSON :

```
{ "menu": {  
  "id": "file",  
  "value": "File",  
  "popup": {  
    "menuitem": [  
      { "value":  
        "New",  
        "onclick":  
        "CreateNewDoc(  
        )" },  
      { "value":  
        "Open",  
        "onclick":  
        "OpenDoc()" },  
      { "value":  
        "Close",  
        "onclick":  
        "CloseDoc()" }  
    ]  
  }  
}
```



exemple Json

- équivalent au format XML:

```
<menu id="file" value="File">
  <popup>
    <menuitem value="New" onclick="CreateNewDoc()" />
    <menuitem value="Open" onclick="OpenDoc()" />
    <menuitem value="Close" onclick="CloseDoc()" />
  </popup>
</menu>
```



Service web avec JSON

```
@Path("/insee")
public class GestionInsee {
    @POST
    @Path("/etatcivil")
    @Consumes("application/x-www-form-urlencoded")
    @Produces("application/json")
    public EtatCivil
    inseeToEtatCivil(@FormParam("nir") String nir)
    {
        Insee insee=Insee.getInstance();
        try { return insee.nirToEtatCivil(nir);
        }catch(Exception e){
            return null;
        }
    }
}
```

- la classe `EtatCivil` doit être annotée avec `@XmlRootElement` pour la sérialisation JSON



avantages et inconvénients

- les services web RESTful sont accessibles de clients multiples:
 - simples navigateurs web
 - JavaScript
 - tout autre langage
- le format des données échangées étant totalement libre, il ne peut être contrôlé (contrairement au protocole SOAP)
- le nombre d'URI nécessaires peut devenir important



Architectures micro-services



Architectures micro-services

- Contexte des architectures micro-services
- Services techniques offert par le framework ou l'infrastructure
- Clients REST, load-balancing, résilience
- Le framework Spring Cloud
- Déploiements micro-service : docker-compose, cloud foundry, Kubernetes



Introduction

- Le terme « ***micro- services*** » décrit un nouveau pattern de développement visant à améliorer la rapidité et l'efficacité du développement et de la gestion de logiciel
- Les méthodes agiles, la culture *Dev Ops* , le *PaaS* , les containers d'application et les environnement d'injection de dépendances permettent d'envisager de construire de grand systèmes orientés services de façon modulaire



Architecture

- Une architecture micro-services implique la décomposition des applications en très petit services
 - faiblement couplés
 - ayant une seule responsabilité
 - Développés par des équipes full-stack indépendantes.
- Le but étant de livrer et maintenir des systèmes complexes avec la rapidité et la qualité demandées par le business digital actuel
- On l'appelle également SOA 2.0 ou *SOA For Hipsters*



Caractéristiques

- Design piloté par le métier
 - La décomposition fonctionnelle est pilotée par le métier (voir *Evans's DDD approach*)
- Principe de la responsabilité unique :
 - Chaque service est responsable d'une seule fonctionnalité et la fait bien !
- Une interface explicitement publiée
 - Un producteur de service publie une interface qui peut être consommée
- DURS (Deploy, Update, Replace, Scale) indépendants
 - Chaque service peut être indépendamment déployé, mis à jour, remplacé, scalé
- Communication légère
 - REST sur HTTP, STOMP sur WebSocket,



Bénéfices

- Scaling indépendant
 - les services les plus sollicités (cadence de requête, mutualisation d'application) peuvent être scalés indépendamment (CPU/mémoire ou sharding),
- Mise à jour indépendantes
 - Les changements locaux à un service peuvent se faire sans coordination avec les autres équipes
- Maintenance facilitée
 - Le code d'un micro-service est limité à une seule fonctionnalité
- Hétérogénéité des langages
 - Utilisation des langages les plus appropriés pour une fonctionnalité donnée



Bénéfices

- Isolation des fautes
 - Un service dysfonctionnant ne pénalise pas obligatoirement le système complet.
- Communication inter-équipe renforcée
 - Full-stack team



Contraintes

- **Réplication**
 - Un micro-service doit être scalable facilement, cela a des impacts sur le design (stateless, etc...)
- **Découverte automatique**
 - Les services sont typiquement distribués dans un environnement PaaS. Le scaling peut être automatisé selon certains métriques. Les points d'accès aux services doivent alors s'enregistrer dans un annuaire afin d'être localisés automatiquement
- **Monitoring**
 - Les services sont surveillés en permanence. Des traces sont générées et éventuellement agrégées



Contraintes

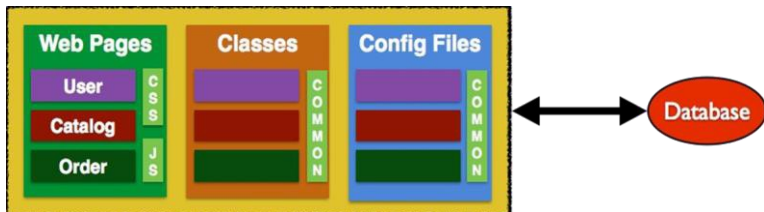
■ Résilience

- Les services peuvent être en erreur.
pouvoir résister aux erreurs.
L'application doit pouvoir résister aux erreurs

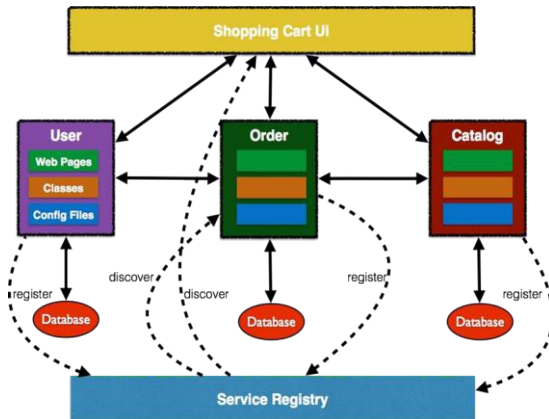
■ DevOps

- L'intégration et le déploiement continu sont indispensables pour le succès.

Architecture monolitique



Version micro-services



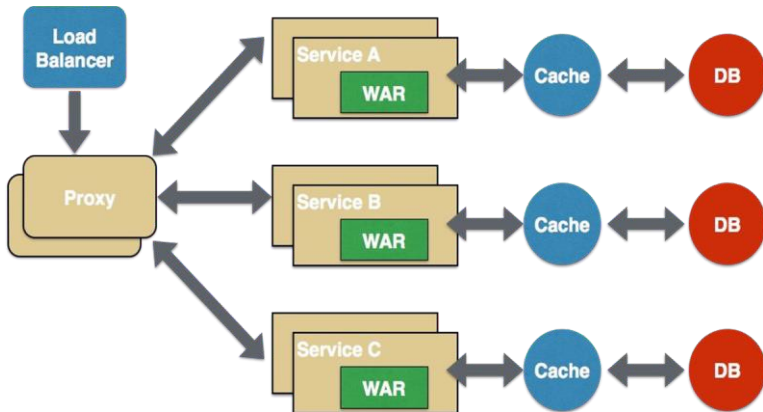


Patterns

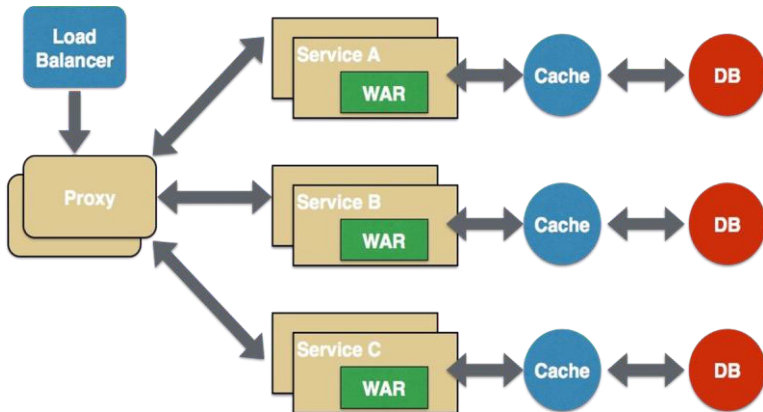
Les micro-services peuvent être combinés afin de fournir un micro-service composite

- Comme design patterns communs, citons :
 - **L'agrégateur**
Agrégation de plusieurs micro- service et fourniture d' une autre API REST
 - **Proxy**
Délégation à un service caché avec éventuellement une transformation
 - **Chaîne**
Réponse consolidée à partir de plusieurs sous-services
 - **Branche**
Idem agrégateur avec le parallélisme

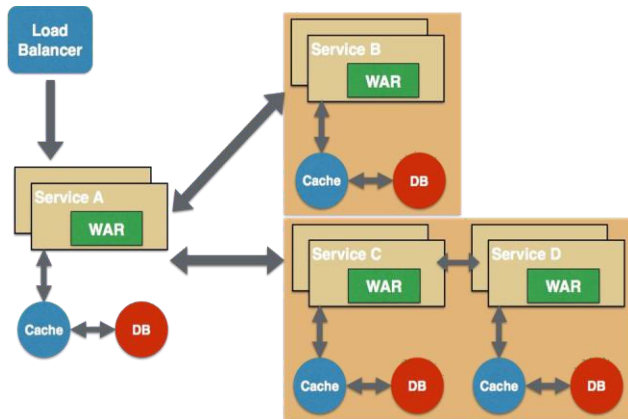
Agrégateur



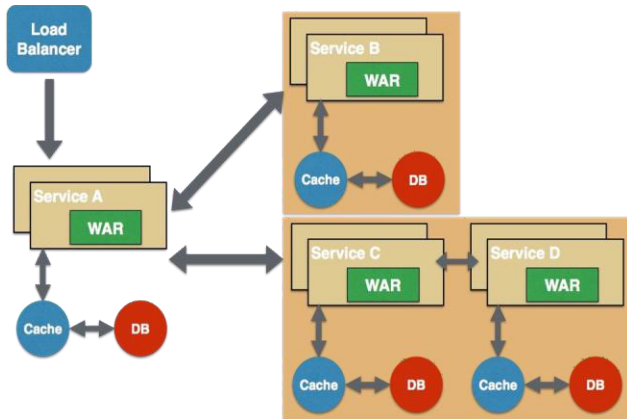
Proxy



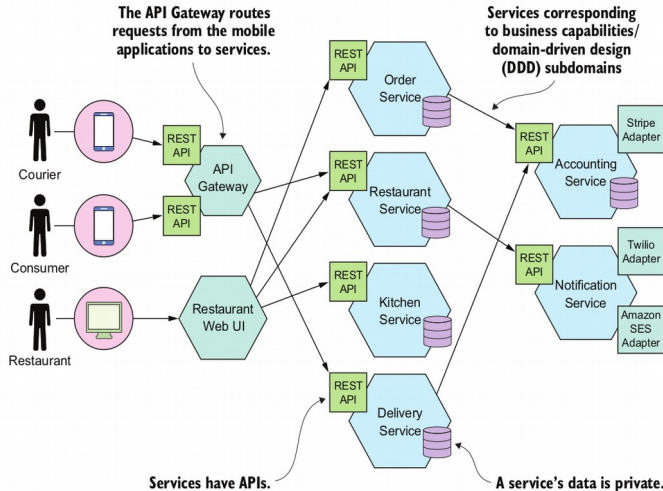
Chaîne



Branche

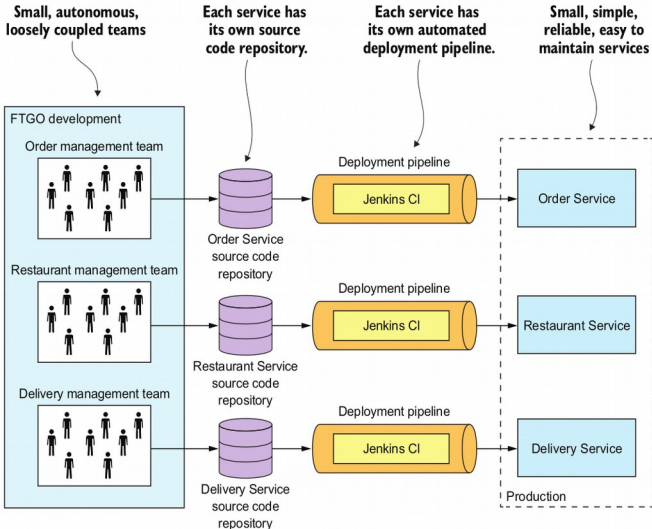


Une architecture micro-service



Microservices Pattern, Chris Richardson, 2017

Organisation DevOps



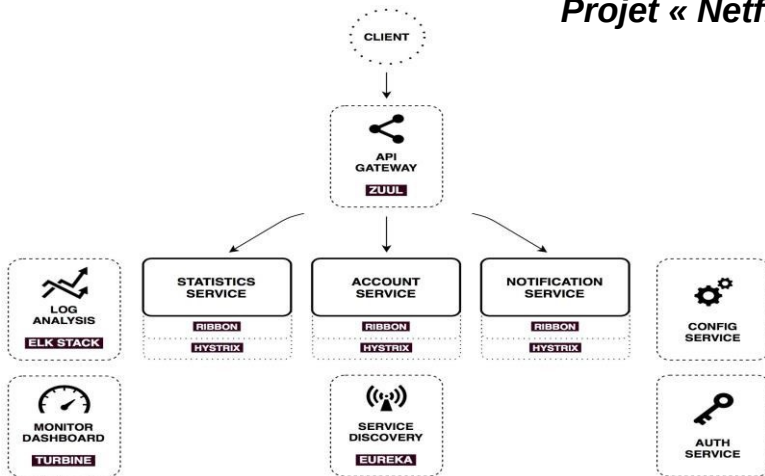
Services techniques

Les architectures micro-services nécessitent des services technique :

- Service **de discovery** permettant à un micro-service de s'enregistrer et de localiser ses micro-services dépendants
- Service de centralisation de **configuration** facilitant la configuration et l'administration des micro-services
- Services **d'authentification** offrant une fonctionnalité de SSO parmi l'ensemble des micro-services
- Service de **monitoring** agrégeant les métriques de surveillance en un point central
- Support pour la répartition de charge, le fail-over, la résilience aux fautes
- Service de session applicative, horloge synchronisée,

Micro-services techniques

Projet « Netflix »





Infrastructure de déploiement

- La nécessité que chaque micro-service soit scalable et que l'on puisse facilement créer de nouvelles instances de n'importe quel service détermine l'infrastructure de déploiement :
 - Virtualisation + outils de gestion de conf : Possible mais Peu adapté
 - Orchestration de conteneur (Kubernetes)
 - Offre Paas (AWS, Google, ...)

Offre Spring Cloud

Spring Cloud a donc vocation d'offrir tous les outils et services techniques nécessaire aux architectures fortement distribuées

Spring Cloud nécessite Spring Boot

Il offre :

- des facilités de déploiement vers des environnements Cloud (Amazon Web Services, Cloud Foundry, Heroku Paas)
- Des facilités pour la mise en place d'architecture Micro-services et plus généralement de systèmes distribués

Offre Spring Cloud

Sprint Cloud basé sur Spring Boot fournit un framework de développement de micro-services qui :

- Apporte des abstractions des micro-services techniques nécessaires : Permettant d'adapter rapidement son code à une implémentation spécifique
- Du support pour les clients REST incluant la répartition de charge et la résilience
- Du support pour l'intégration des micro-services aux middleware de messagerie
- Bénéficie de l'environnement Spring Boot et de l'écosystème Spring (Testabilité, Spring MVC / REST, Spring Data (SQL ou NoSQL), ...)



Message brokers



Message brokers

- Le modèle distribuée asynchrone, ses cas d'usage
- JMS dans Java EE, implémentation de MDB
- Le standard AMQP, ses implémentations dans le monde Java
- Message Broker pour le Big Data, l'avènement de Kafka
- Intégration Kafka/Spring, le projet Spring Cloud Data Stream



principe des MOM

- les MOM (Message Oriented Middleware) sont des serveurs dont le rôle est de veiller au bon acheminement de messages asynchrones entre applications
- le principe est de permettre à une application d'émettre un message à destination d'une autre application, et de continuer à fonctionner sans attendre la réponse du destinataire
 - dans tous les cas, le destinataire doit recevoir tôt ou tard le message qui lui est destiné, même s'il n'est pas disponible à l'instant auquel l'émetteur le lui a envoyé
- un des MOM les plus connus est MQSeries d'IBM



Message Broker

- un message broker qu'on appelle aussi MQ comme Message Queueing permet de gérer des files d'attente de messages
 - il est possible ainsi de publier un message qui sera donc en attente d'être consommé
- Une telle infrastructure, de par son fonctionnement, oriente fortement les applications vers une programmation événementielle (asynchrone)
 - Cela a pour avantage de découpler les applications, ce qui permet non seulement de paralléliser les développements avec un risque minime mais aussi de les rendre plus modulables : plus propices à l'évolution, au changement.



Principe

- Les messages étant stockés le temps que le traitement s'effectue, une interruption du serveur n'entraîne donc pas la perte d'un traitement en cours
- De plus, cela permet aussi une meilleure scalabilité réduisant ainsi les problématiques de performance
- Le plus gros désavantage réside cependant dans un effet secondaire : le découplage
 - en effet un producteur publie des messages en supposant qu'un consommateur les traite par la suite
 - en pratique, rien n'affirme ce prédicat

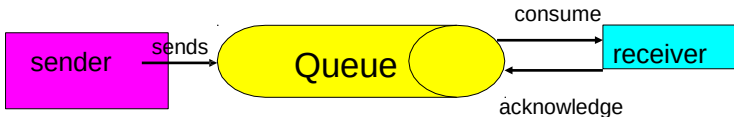


modèles Pub/Sub et PTP

- il existe deux modèles de messages (messaging domains):
 - le modèle PTP (Point To Point)
 - le modèle Pub/Sub (Publish/Subscribe)
- le modèle PTP (Point To Point) s'appuie sur les concepts de file d'attente (queue), d'émetteurs (senders) et de destinataire (receivers)
- chaque message est émis vers une file d'attente spécifique, les destinataires se chargeant d'extraire les messages de la ou des files d'attente qui leur ont été attribuées

modèles PTP

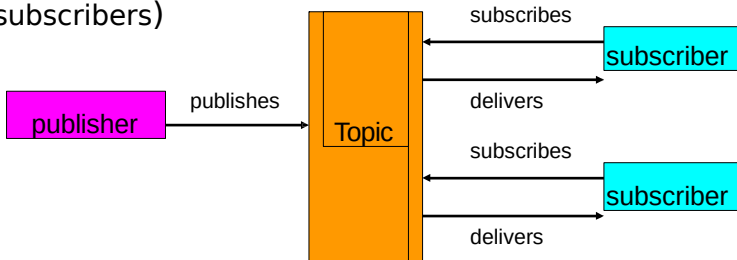
- le modèle PTP (Point To Point) est utilisé lorsque chaque message doit avoir un seul destinataire et lorsque le facteur temps n'a pas d'importance dans la relation émetteur-destinataire



modèle Pub/Sub

- le modèle Pub/Sub (Publish/Subscribe) permet un client d'émettre des messages vers un sujet (topic), les messages étant reçus par les abonnés à ce sujet
 - chaque message peut donc avoir plusieurs destinataires

(subscribers)





Cas d'utilisation

- Les Message brokers sont souvent utilisés dans les cas suivants:
 - Transactions financières
 - Traitement de commandes E-commerce
 - Protection de données sensibles en dépôt ou en transit



Serveur de messages

- Pour pouvoir gérer des files d'attente, il faut un serveur de message
 - Il en existe une multitude (ActiveMQ, Kafka, RabbitMQ, OMS, JMS, Redis, Service Bus, ...)
- *RabbitMQ* supporte le protocole AMQP dont l'objectif est de standardiser les échanges entre serveurs de messages
 - toute interaction avec le serveur se fait au travers de ce protocole



AMQP

- L'Advanced Message Queuing Protocol, développé depuis 2003, a tout d'abord été développé par JPMorgan Chase, une banque américaine
 - AMQP résout plusieurs problèmes en même temps
 - garantit d'une part une transmission des données fiable (à l'aide d'un message broker)
 - permet de stocker des messages dans des files d'attente, permettant ainsi une **communication asynchrone**
 - Le destinataire (consommateur) du message n'est pas obligé d'accepter directement l'information, de la traiter et d'en accuser réception auprès de l'émetteur (producteur). À la place, il va récupérer le message dans la file d'attente lorsqu'il en a la capacité. Le producteur peut ainsi continuer à travailler sans créer de période d'inactivité.



AMQP

- Le succès rencontré par ce protocole relativement récent lui vient également de son interopérabilité
- Dans l'univers d'AMQP, il existe trois acteurs et un objet :
 - le **message** est l'élément central sur lequel repose toute la communication
 - Le producteur (**Producer**) procède à la création et à l'envoi du message
 - Le **message broker** distribue le message aux différentes files d'attente (**Queue**) selon des règles définies
 - Le consommateur (**Consumer**) récupère le message dans la file d'attente à laquelle il a accès et traite le message



JMS

- l'API JMS (Java Message Service) désigne un ensemble de classes et d'interfaces Java permettant de créer, d'envoyer, de recevoir et de lire des messages
- JMS permet des communications:
 - asynchrones
 - avec un faible degré de couplage entre applications
 - fiables



principe des EJB MD

- les EJB message-driven sont des objets transactionnels distribués spécialement conçus pour le traitement de messages asynchrones JMS de type PTP ou Pub/Sub
- le conteneur prend à sa charge l'environnement de l'EJB, comme l'aspect transactionnel, la sécurité, la concurrence et l'acquittement des messages
- l'apport le plus intéressant des EJB MDB est sans doute la gestion concurrente des messages par le conteneur, évitant au programmeur de développement d'applications transactionnelles multi- threadées



principe des EJB MD

- un EJB MDB peut recevoir des centaines de messages JMS provenant d'applications différentes et les traiter simultanément
- le conteneur multiplie en effet les instances d'EJB MDB pour accomplir cette tâche
- le programmeur doit donc simplement définir la classe Bean et le descripteur de déploiement



Exemple d'EJB MD

```
@MessageDriven( name = "BookMessageHandler",
    activationConfig = {
        @ActivationConfigProperty(propertyName = "destinationType",
            propertyValue = "javax.jms.Queue"),
        @ActivationConfigProperty( propertyName = "destination",
            propertyValue = "/queue/BookQueue") } )

public class LibraryMessageBean implements MessageListener {
    @Resource private MessageDrivenContext mdctx;

    @EJB LibraryPersistentBeanRemote libraryBean;

    public LibraryMessageBean() { }

    public void onMessage(Message message) {
        // traitement du message
    }
}
```



Kafka



Présentation

- *Kafka* est un système de messagerie distribué, originellement développé chez LinkedIn, et maintenu au sein de la fondation Apache depuis 2012
 - Son adoption n'a cessé de croître pour en faire un quasi de-facto standard dans les pipelines de traitement de données actuels
- Bien plus qu'un simple concurrent des outils conçus autour des standards JMS ou AMQP, Kafka a pour ambition de devenir la plateforme centralisée de stockage et d'échange de toutes les données émises par une entreprise en temps réel



Présentation

- *Kafka* est un système de messagerie distribué, en mode publish-subscribe, persistant les données qu'il reçoit, conçu pour facilement monter en charge et supporter des débits de données très importants
 - Kafka conserve les données qu'il reçoit dans des **topics**, correspondant à des catégories de données
 - On nomme les systèmes qui publient des données dans des topics Kafka des **Producers**.
 - Les **Consumers**, sont les systèmes qui vont lire (transfert de données de type pull) les données des topics Kafka

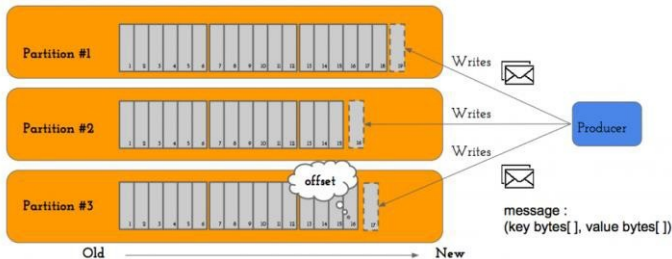


Principe

- Le log est l'abstraction de base au cœur du système *Kafka*
 - Il s'agit d'une structure de données abstraite ayant les caractéristiques suivantes :
 - Il s'agit d'un Array de messages.
 - Dans lequel les données sont ordonnées selon le temps de réception des messages.
 - Chaque nouvelle donnée sera ajoutée à la « fin » du log (append only).
 - Les données sont immuables.
 - Un index unique, ici nommé l'**offset**, est attribué à chaque message.

Principe

- Pour assurer un débit d'écriture et de lecture très important, les topics seront divisés en **partitions** distribuées au sein du cluster *Kafka*
 - Chaque partition est répliquée au sein du cluster pour assurer une tolérance à la panne





Principe

- Les messages, constitués d'une clé optionnelle et d'une valeur, seront envoyés par les producers à un topic *Kafka* sous la forme d'un simple tableau de bytes, ce qui permet de transmettre et de persister des messages dans un format arbitraire
- Les messages possédant une clé commune seront automatiquement envoyés à la même partition d'un topic
 - la partition étant choisie en effectuant un hash sur la clé d'un message modulo le nombre de partitions du topic

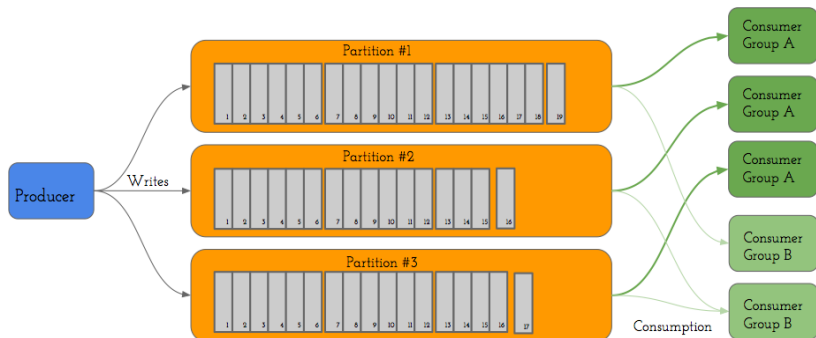


Principe

- Chacune des partitions d'un topic se comporte comme un commit log
 - Contrairement aux autres brokers, Kafka ne maintient pas une liste des consommateurs d'un topic ou de leur état de consommation, mais va persister sur disque tous les messages en leur assignant un offset au sein d'une partition d'un topic
 - La rétention des messages sera configurée selon une durée ou une taille des logs. Charge aux différents consommateurs de maintenir leur état de consommation des différentes partitions d'un topic
- *Kafka* va ainsi pouvoir supporter un nombre très importants de consommateurs, sans impact sur ses performances

Principe

- La consommation des messages d'un topic peut elle même être réalisée par un système distribué (Hadoop, Spark, Akka cluster...). *Kafka* introduit ainsi la notion de groupe de consommation





Spring Cloud Stream



Présentation

- Spring Cloud Stream est un framework basé sur Spring Boot et Spring Integration qui permet de créer des microservices gérés par événements ou par messages
 - La communication entre les points de terminaison est assurée par des partenaires de messagerie tels que *RabbitMQ* ou *Apache Kafka*
 - Les services communiquent en publiant des événements de domaine via ces points de terminaison ou canaux
 - Les messages destinés aux destinations sont livrés selon le modèle de messagerie *Publish-Subscribe*