

JEE Introduction

Professeur Titulaire : Millimono Sory (AI Seacher)
Le 04/06/2023

Plan Du Cours



- **Importance de la qualité des données.**
- **Techniques d'exploration des données: visualisation, statistiques descriptives.**
- **Nettoyage et traitement des données manquantes.**
- **Transformation des données : normalisation, standardisation.**
- **Régression linéaire**

The Four Platforms

Java SE
(standard)

Java ME
(micro)

Java FX
(eff-ects)

Java EE
(enterprise)

The Four Platforms

Java Application

Application Programming Interface (API)

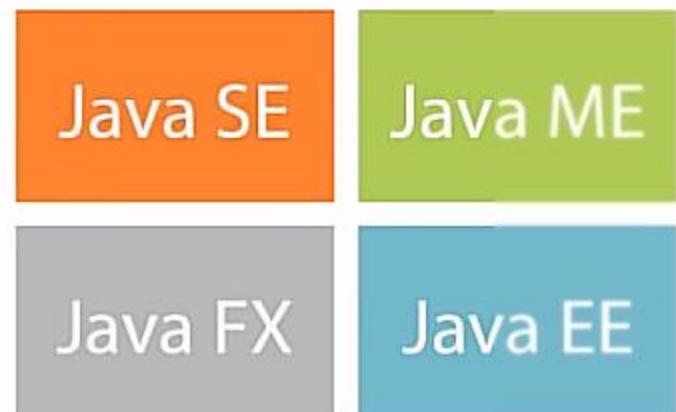
Java Virtual Machine (JVM)

Operating System

Hardware

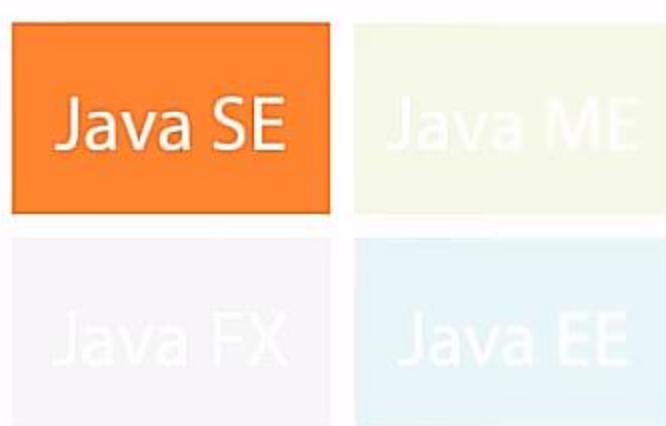
Each Platform

- JVM and API
- Run on any compatible system
- Take advantage of the Java language
- One of the most widely used platforms
- Development of just about any solution
- Enterprise applications



Java SE

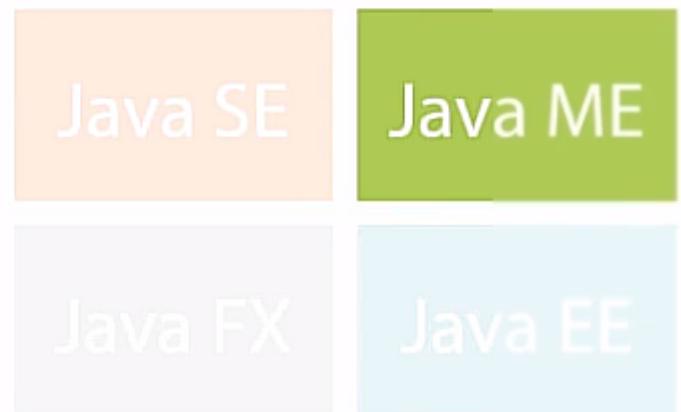
- Java Standard Edition
- Core platform
- Core libraries and APIs
- Basic types and objects to high-level classes
- JVM
- Development tools
- Deployment and monitoring
- ...



JDK (Java Development Toolkit)

Java ME

- Java Micro Edition
- Subset of Java SE
- Mobile devices
- Small-footprint JVM
- Small devices
- Internet of things



Java FX

- Rich internet applications
- User-interface API
- Hardware-accelerated graphics
- High-performance clients
- Modern look-and-feel
- Connect to remote services

Java SE

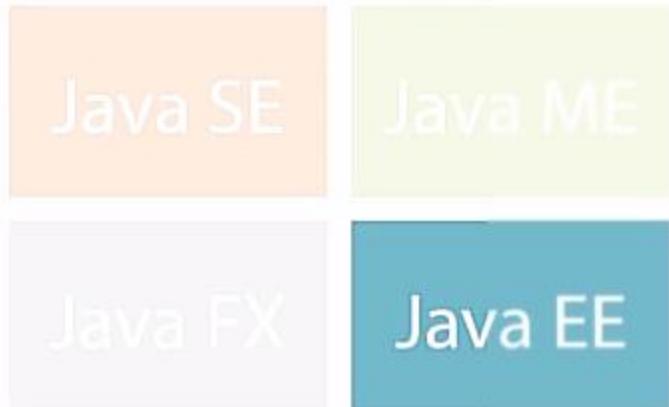
Java ME

Java FX

Java EE

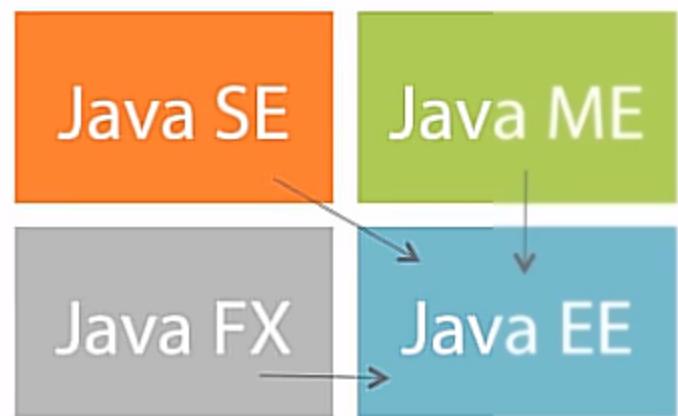
Java EE

- Java Enterprise Edition
- Java EE extends Java SE
- Enterprise software
- Large scale
- Distributed system
- Consider Java EE instead of Java SE



Java EE

- Java Enterprise Edition
- Java EE extends Java SE
- Enterprise software
- Large scale
- Distributed system
- Consider Java EE instead of Java SE



Applications

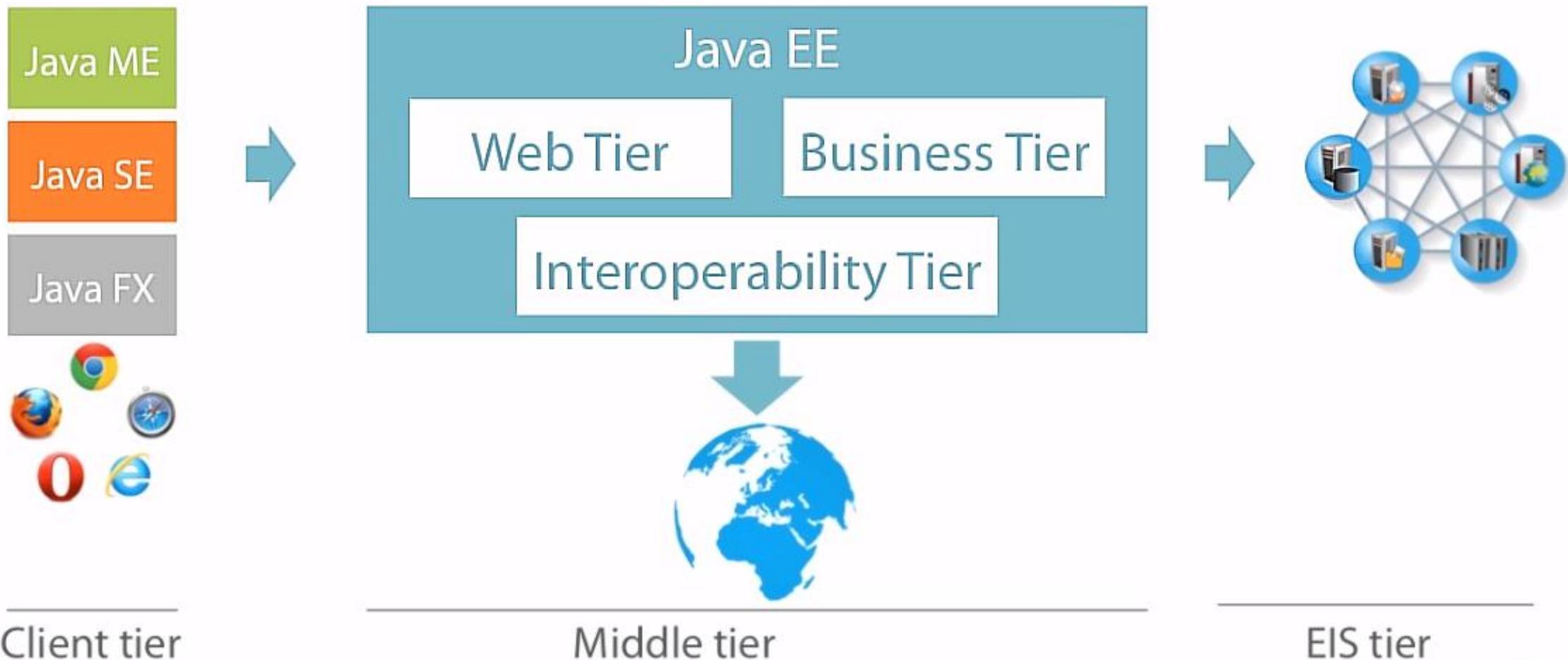
Java SE
Standard
applications

Java ME
Mobile device
applications

Java FX
Rich user interfaces

Java EE
Enterprise
applications

Tiered Application



Java SE vs. Java EE

- Java SE
 - APIs handle collections
 - The JVM is a container
 - Lower-level services
- Java EE
 - APIs handle transactions, messaging, persistence...
 - Code runs in a container
 - Higher-level services



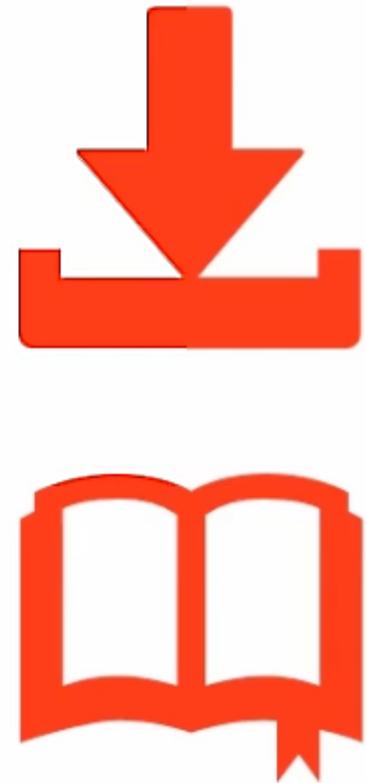
Java EE Reduces Complexity

- Enterprise applications are powerful
- But complex
- Java EE reduces complexity
- Programming model
- APIs
- Runtime environment
- Developers concentrate on business requirements



What's Wrong with Java SE?

- SQL is not Java
- Low-level API (JDBC)
- SQL is not easy to refactor
- JDBC is verbose
- Hard to read
- Hard to maintain



Advantages of Java EE

- No manual mapping
- No SQL statements
- Non intrusive
- Metadata (@Entity, @Id)
- Higher-level of abstraction



Convention over Configuration

```
@Entity
@Table(name = "t_book")
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    @Column(name = "book_title", nullable = false)
    private String title;
    @Column(length = 2000)
    private String description;
    @Column(name = "unit_cost")
    private Float price;
    private String isbn;

    // Constructors, getters & setters
}
```

Java EE Architecture

Les Conteneurs Java EE

Les conteneurs Java EE sont des environnements d'exécution qui gèrent le cycle de vie des composants Java EE, fournissent des services de base, et assurent l'isolation et la sécurité entre les composants. Ils jouent un rôle crucial dans le déploiement et l'exécution des applications Java EE.

différents types de conteneurs Java EE :

Conteneur Web

Conteneur EJB

Conteneur de Messagerie

Conteneur d'Application

Conteneur Client Application

Les Conteneurs Java EE

Les conteneurs Java EE sont des environnements d'exécution qui gèrent le cycle de vie des composants Java EE. fournissent des services de base. et assurent l'isolation et la sécurité entre les composants Java EE.

différents types

Conteneur Virtualisé

- Runtime environment

Conteneur EJB

- Hide technical complexity

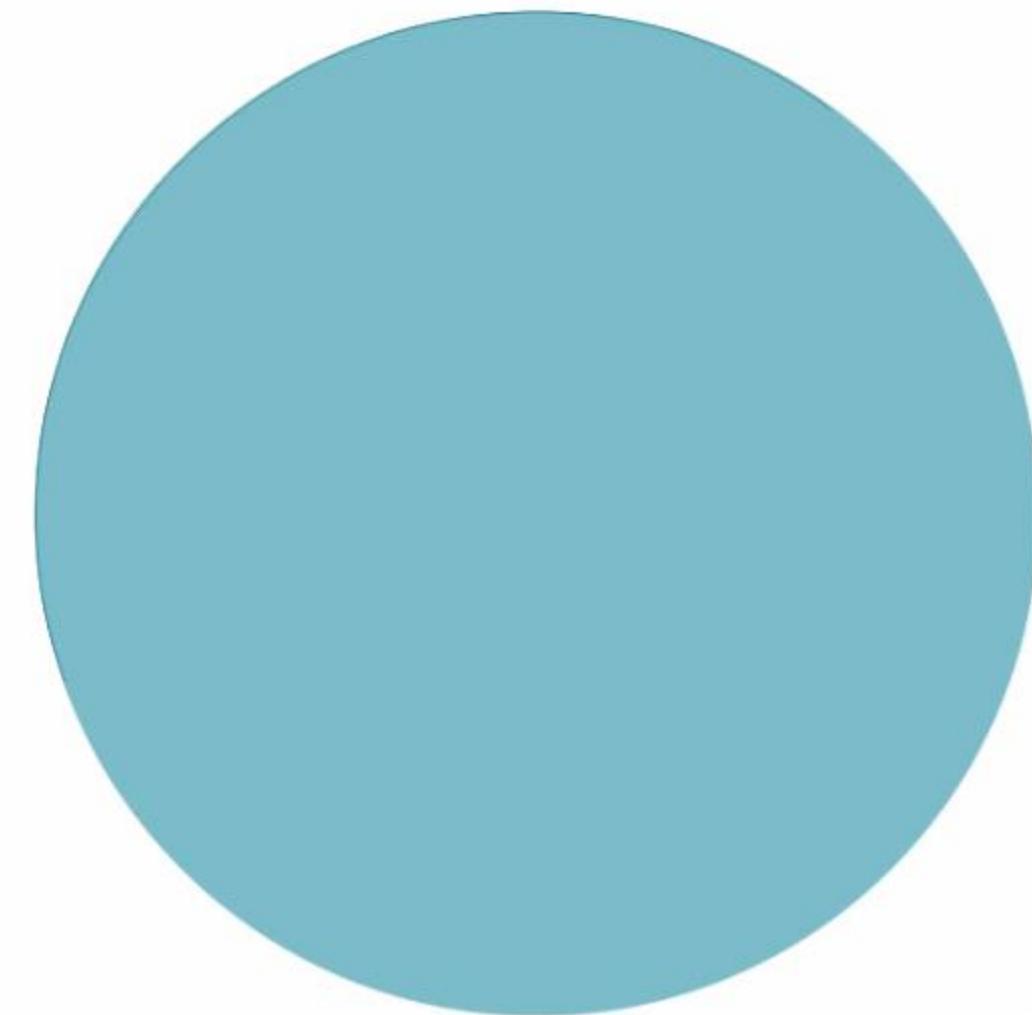
Conteneur CDI

- Enhance portability

Conteneur GlassFish

- Host applications
- Handle complex low-level details
- Administrates the applications

Container



Composants Java EE

Les composants Java EE sont les éléments modulaires qui s'exécutent au sein des conteneurs Java EE. Chaque type de composant a un rôle spécifique dans l'architecture d'une application d'entreprise.

principaux composants Java EE :

Servlets

JavaServer Pages (JSP)

Enterprise JavaBeans (EJB)

Java Message Service (JMS)

Java Persistence API (JPA)

JavaMail

Composants Java EE

Les composants Java EE sont les éléments modulaires qui s'exécutent au sein des conteneurs Java EE. Ch

d'entrepris

principaux

- Static or dynamic Web pages
- Server-side classes
- Handle business code
- Process data
- Access legacy systems

Servlets

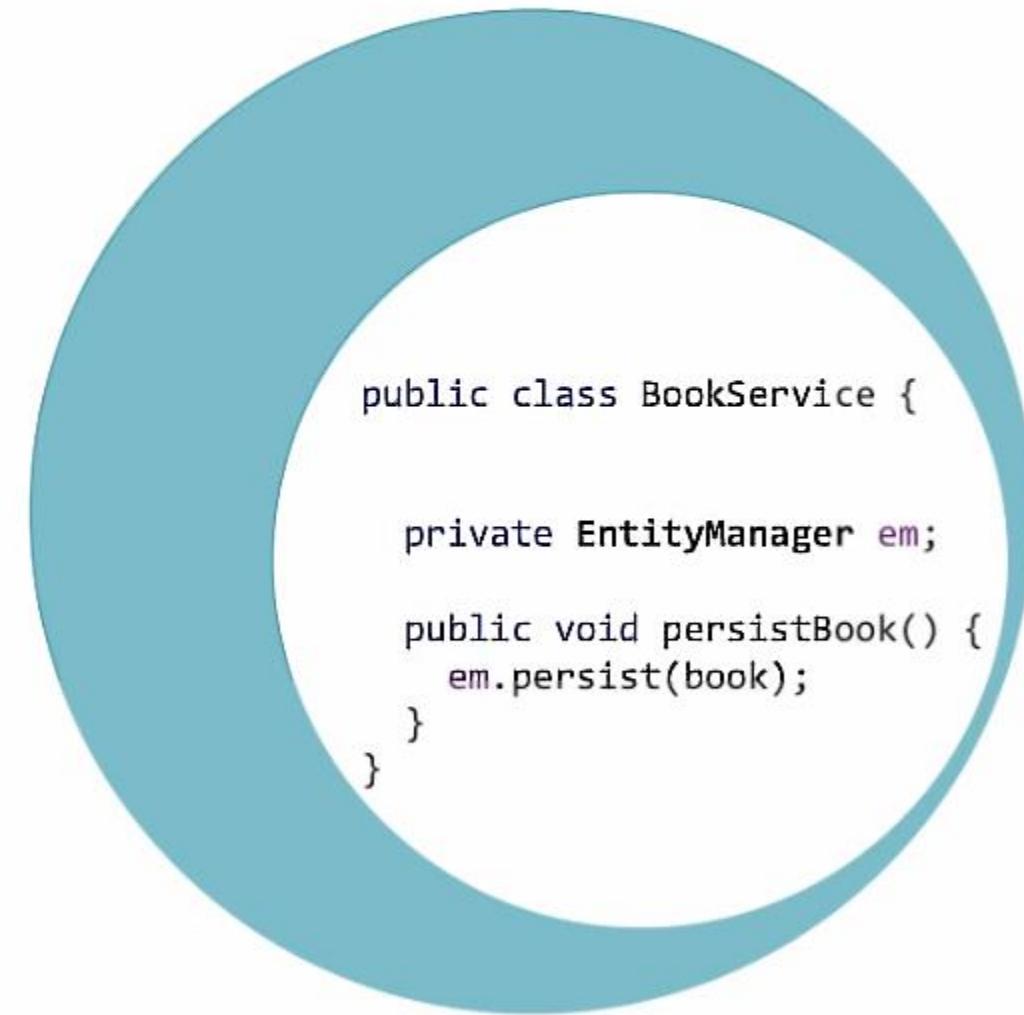
**JavaServer
Enterprise**

**Java Message
Service**

**Java Persistence
API**

JavaMail

Components



Services Java EE

Les services Java EE jouent un rôle crucial en fournissant des fonctionnalités essentielles qui facilitent la gestion des conteneurs et des composants Java EE. Ces services assurent une interaction cohérente et transparente entre les différents composants de l'application, en gérant des aspects tels que la sécurité, les transactions, la persistance, la messagerie, etc.

Les services Java EE fournissent l'infrastructure nécessaire pour que les conteneurs gèrent efficacement les composants de l'application. Ils permettent aux composants de fonctionner de manière sécurisée, transactionnelle, persistante, et scalable, en assurant une interaction fluide et cohérente entre les différentes parties de l'application.

Service de Sécurité

Service de Gestion des Transactions

Service de Persistance

Service de Messagerie

Service de Web Services

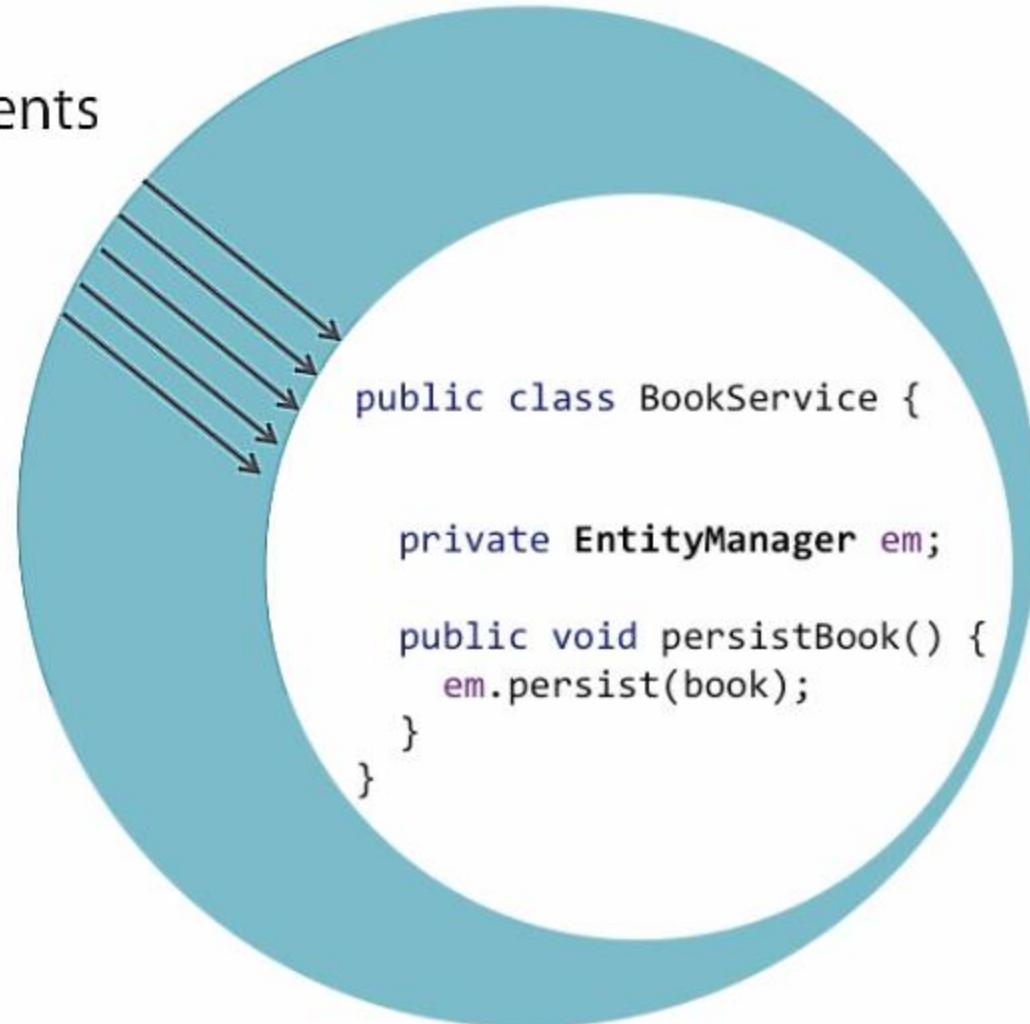
Service de Gestion des Connexions

Service de Concurrence

Service de Minutage

Services

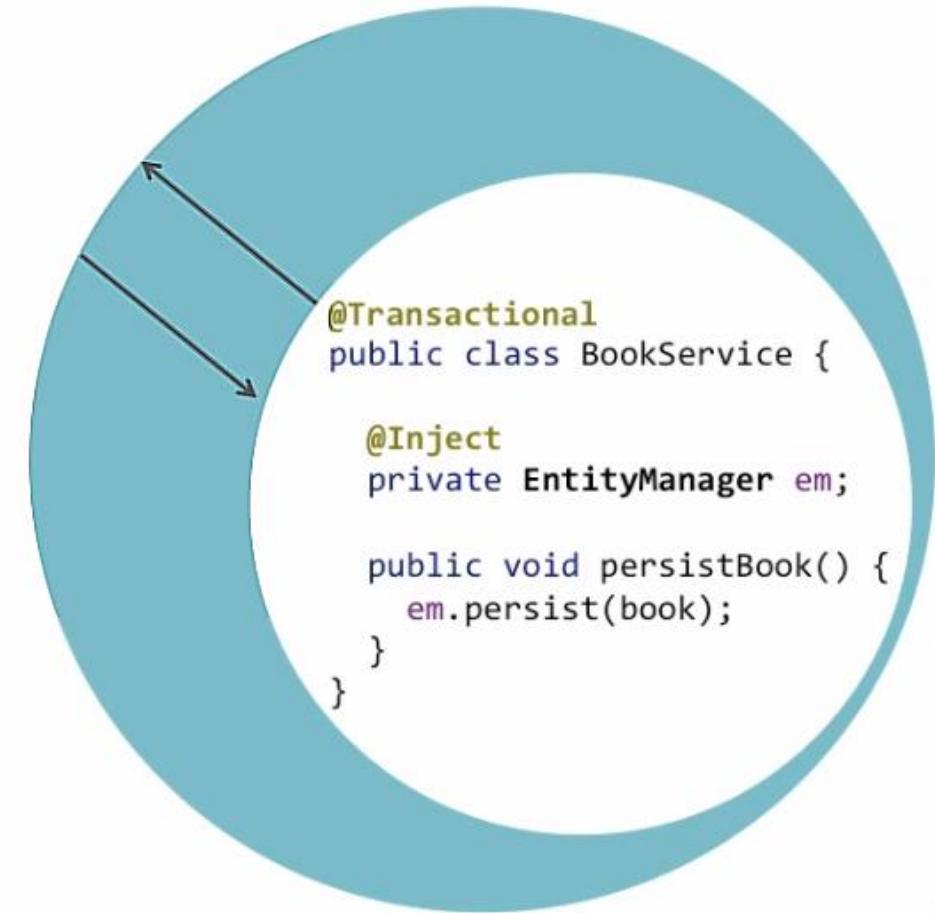
- Container provides services to components
 - Security
 - Transaction management
 - Naming
 - Remote connectivity
 - ...
- Services are configurable
- Configuration is isolated



Les métadonnées Java EE définissent et configurent les composants, spécifient les politiques de sécurité, gèrent les transactions, configurent les ressources, et définissent les points de terminaison des services web pour permettre aux conteneurs de gérer efficacement les applications.

Metadata

- User code
- Metadata
- Annotation
- XML descriptor

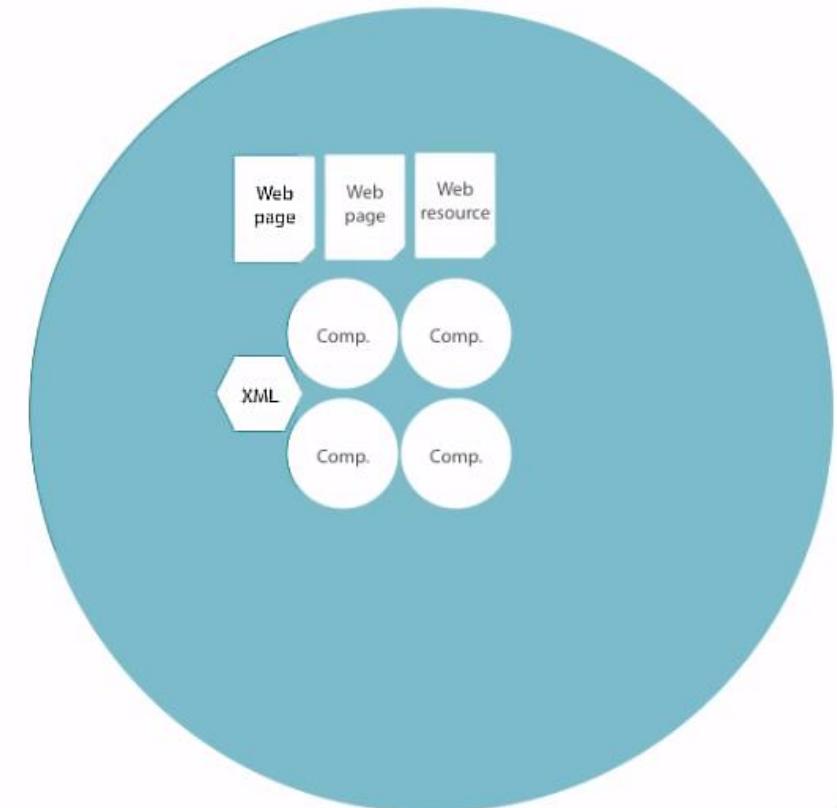


Applications Java EE

Les applications Java EE (Enterprise Edition) sont des applications d'entreprise robustes, scalables et sécurisées. Elles utilisent une combinaison de composants, services et conteneurs Java EE pour répondre aux besoins complexes des entreprises modernes.

Application

- Aggregation of components
- Web pages
- Web resources
- Business components
- Database access components
- Deployment descriptors

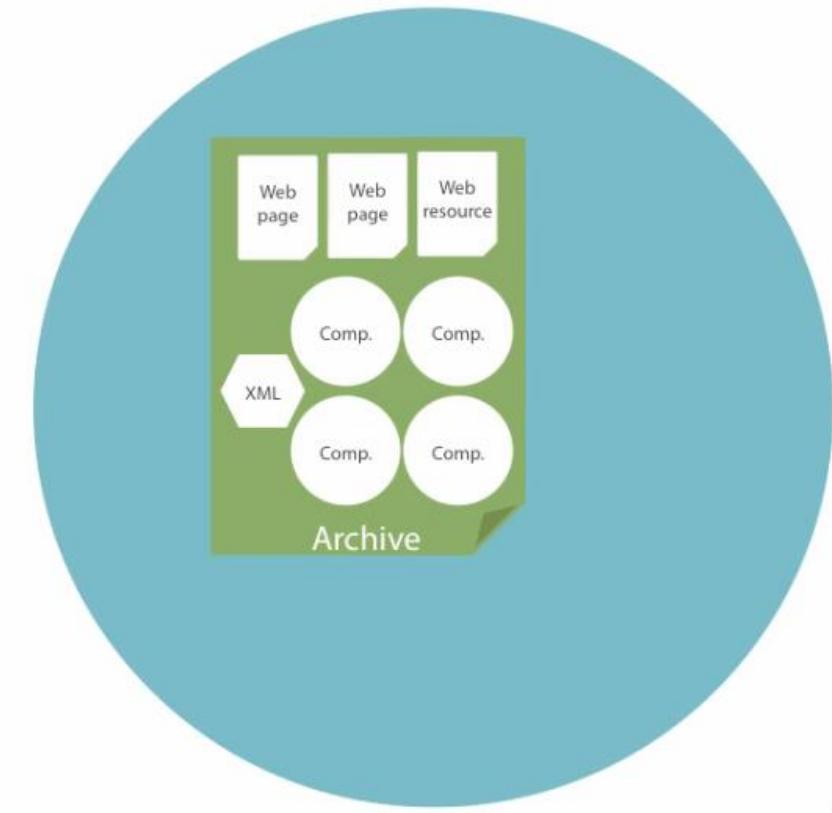


Packaging des Applications Java EE et Jakarta EE

Le packaging des applications Java EE et Jakarta EE consiste à organiser et empaqueter les composants de l'application pour le déploiement sur un serveur d'applications. Les formats de packaging standard sont les archives WAR, EAR et JAR, chacune ayant des usages spécifiques.

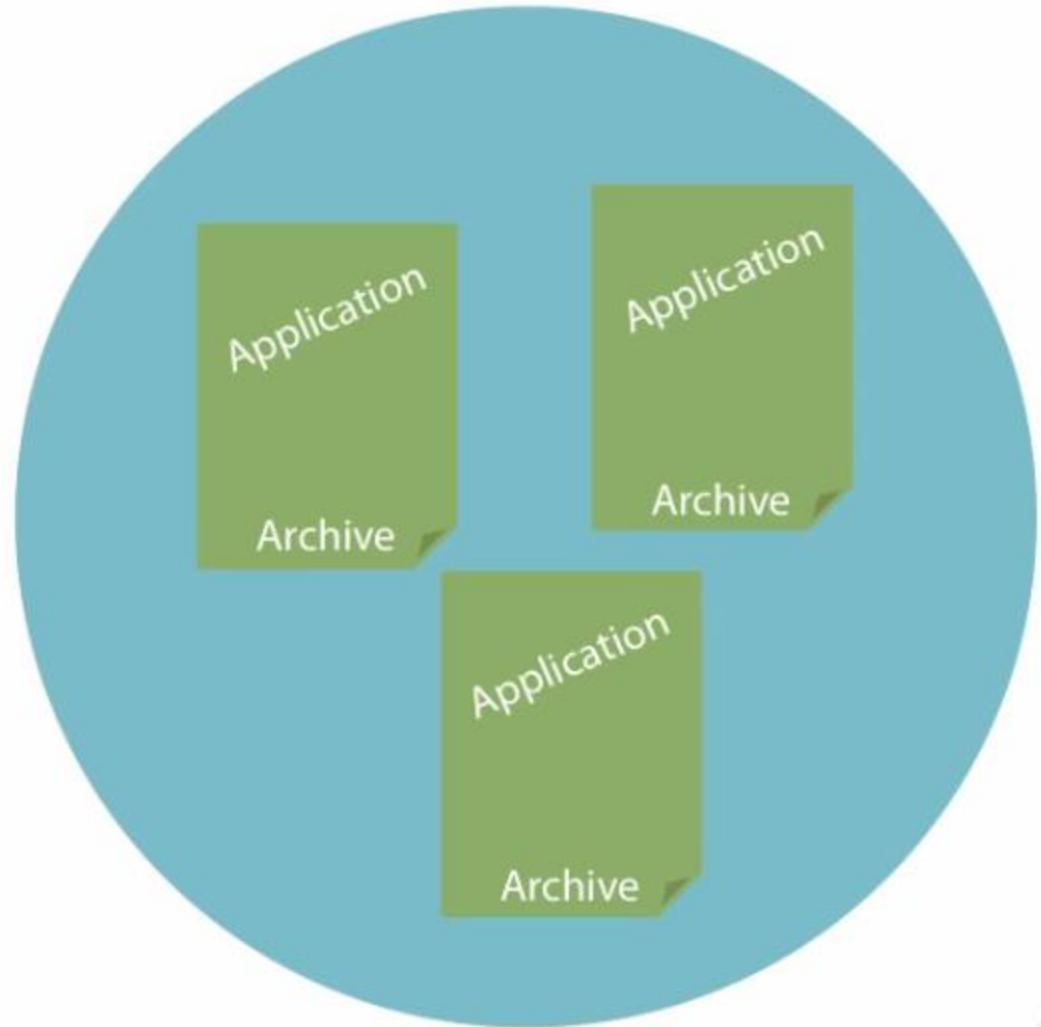
Packaging

- Assemble components together
- Archive
- Deployment
- Standard format



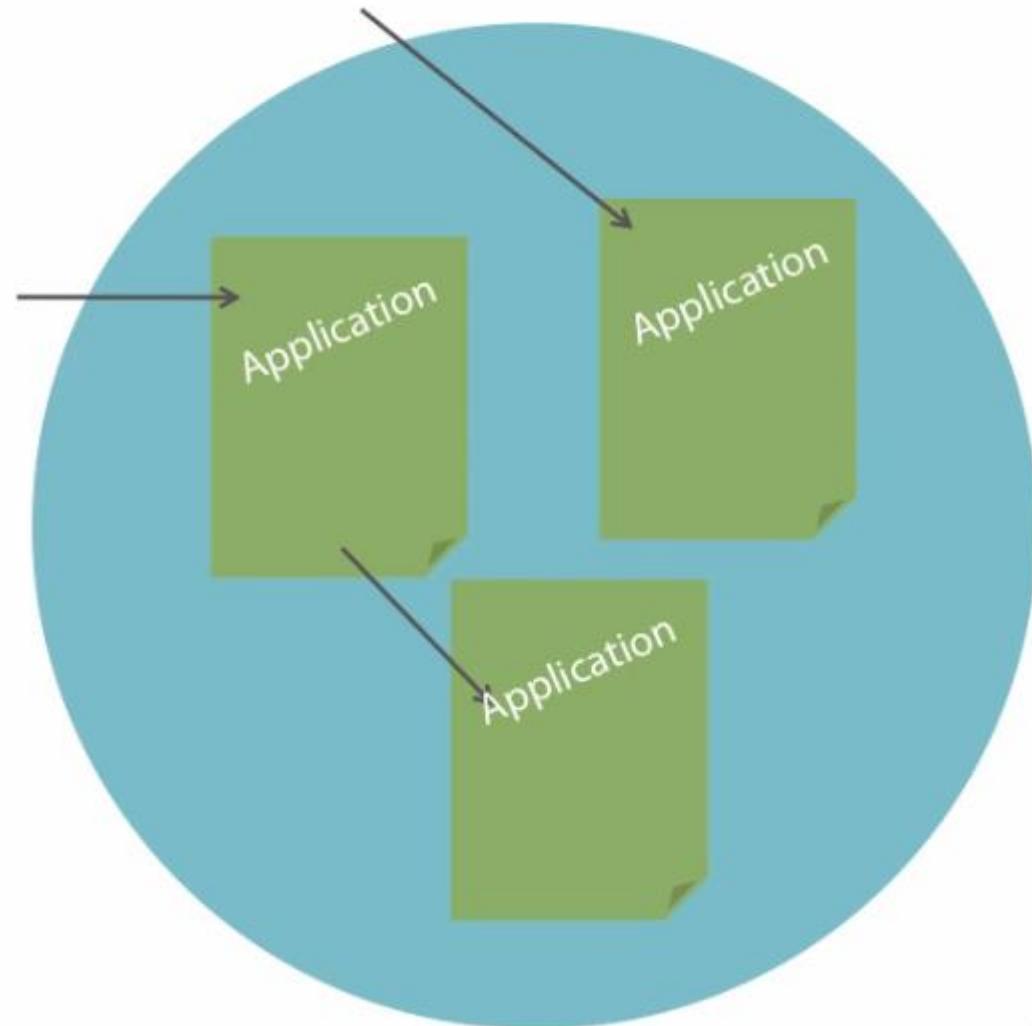
Applications

- Deploy several applications
- Isolation
 - Own resources
 - Own components
 - Own class-loader
- Administration



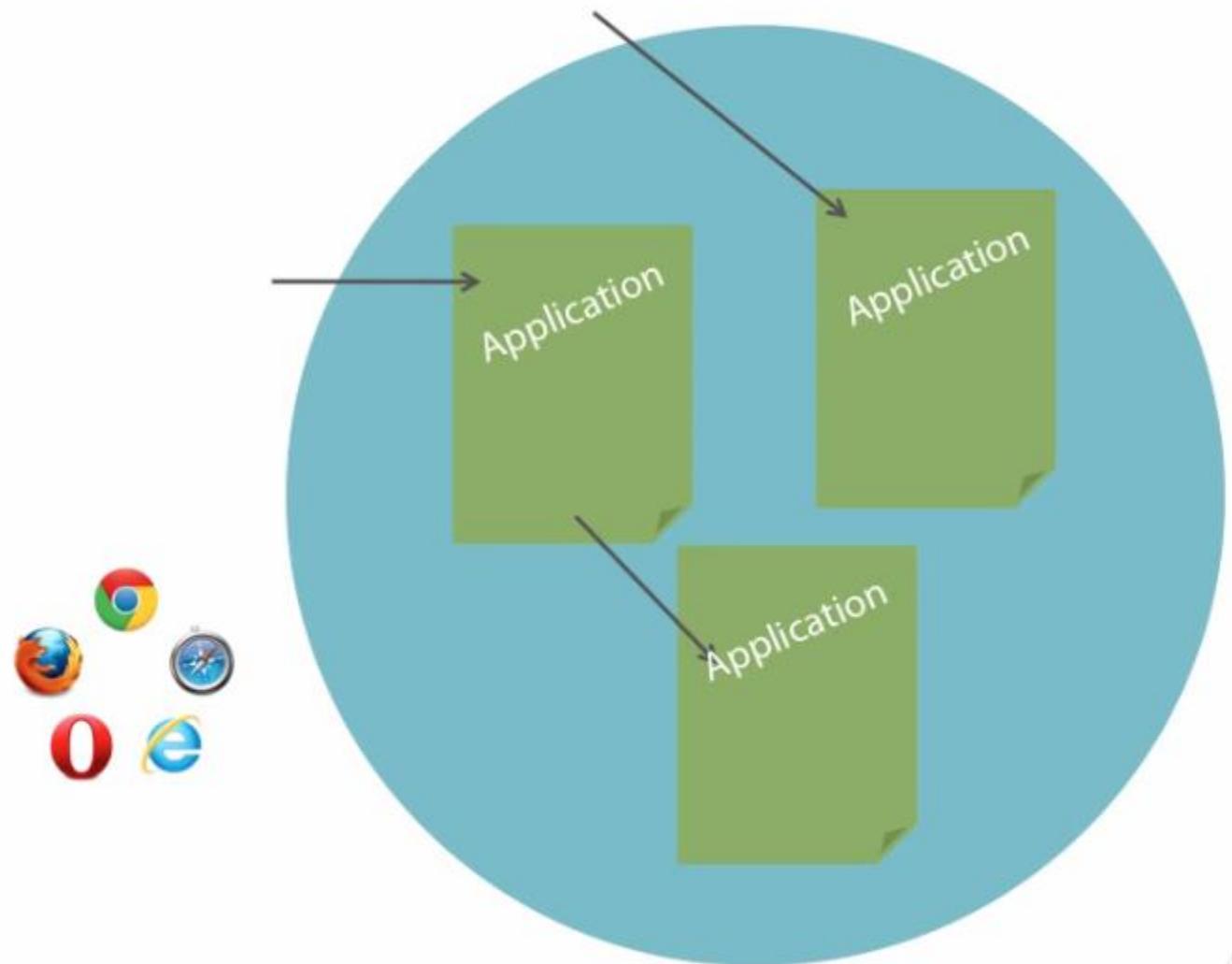
Protocols

- HTTP
- HTTPS (SSL)
- RMI/IOP
 - Remote Method Invocation
 - Internet Inter-ORB Protocol
 - CORBA
 - Ada, C, C++, Cobol...



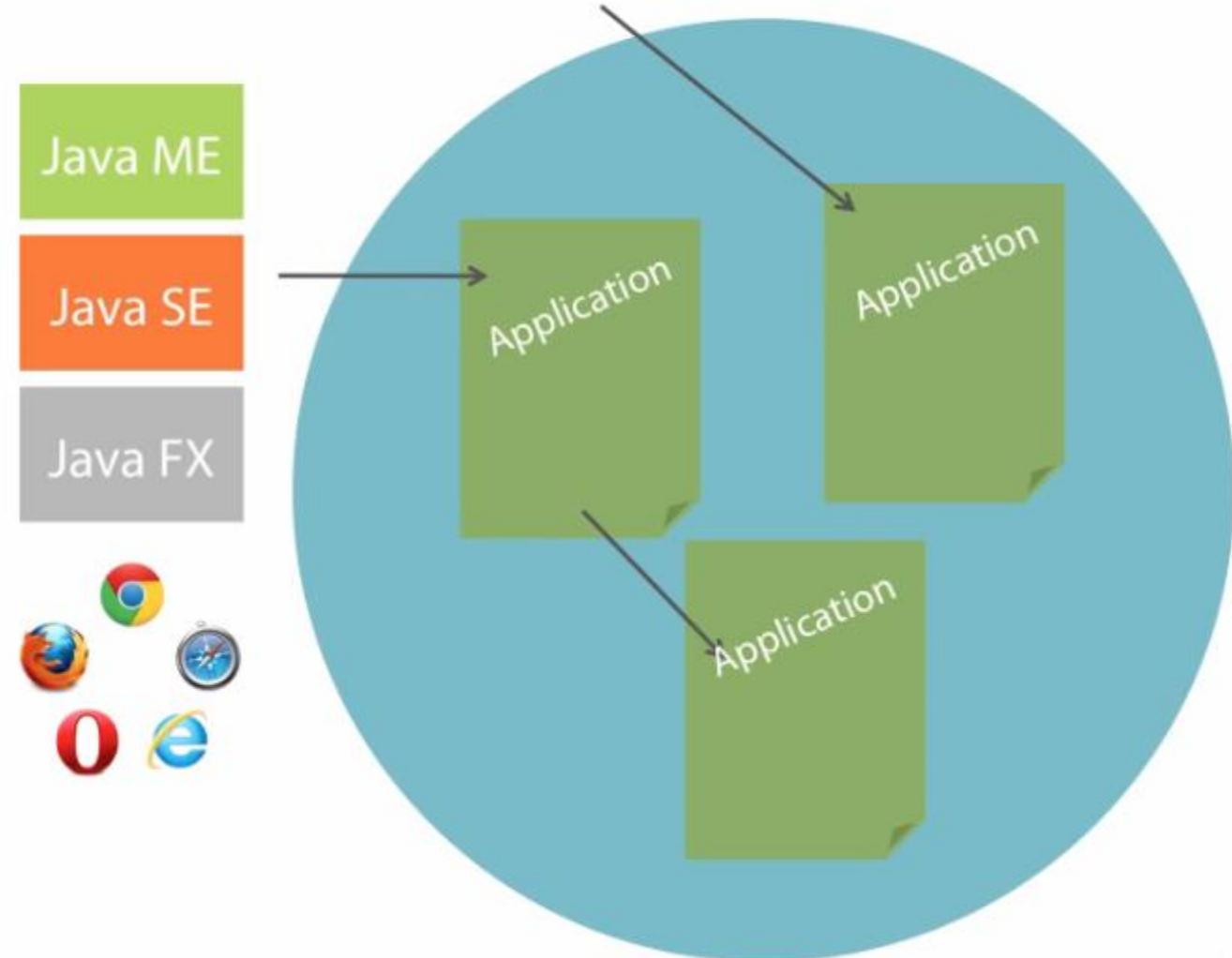
Clients

- Web clients
- Web applications
- REST / SOAP services
- Mobile apps
- B2B



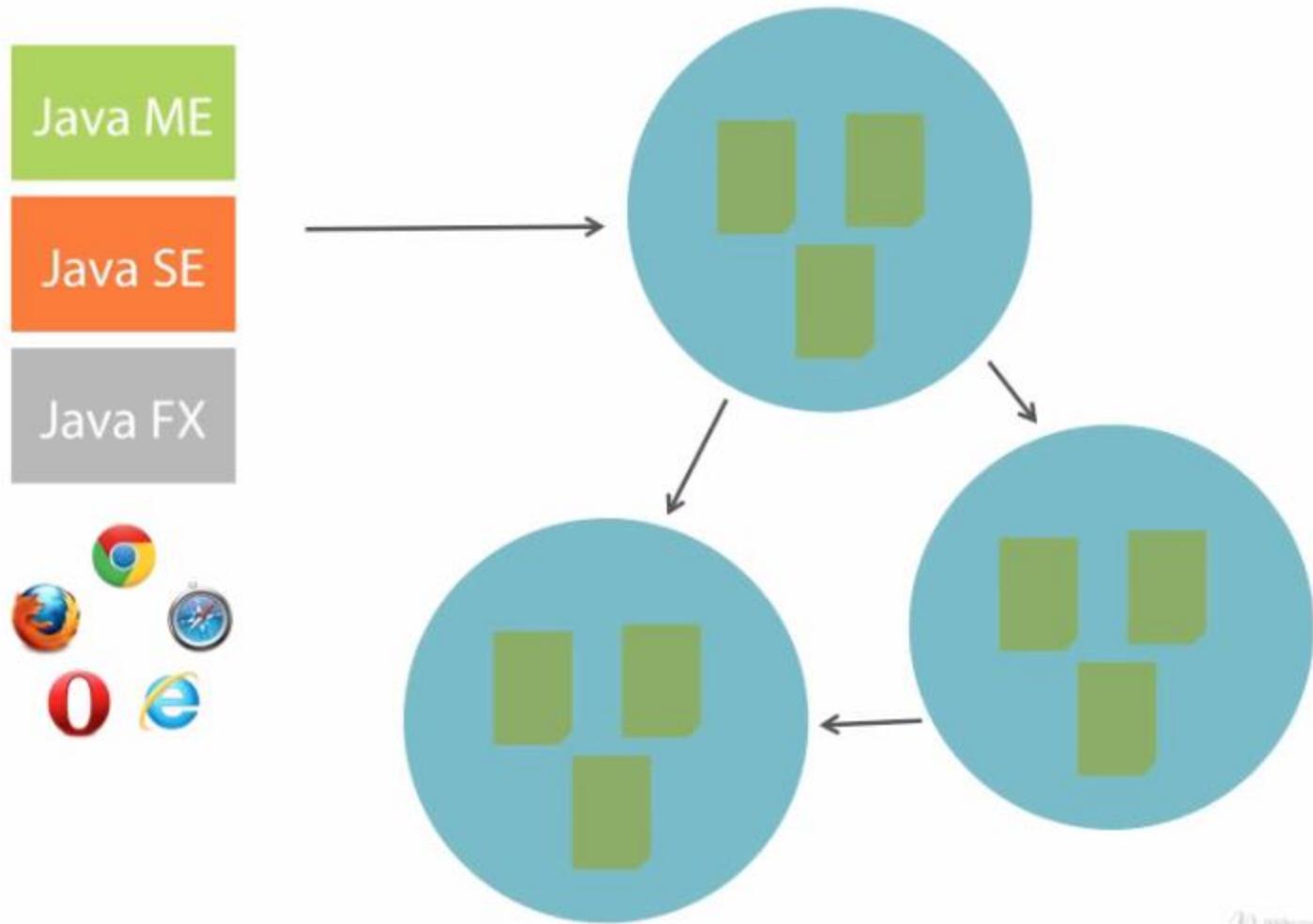
Clients

- Web clients
- Web applications
- REST / SOAP services
- Mobile apps
- B2B

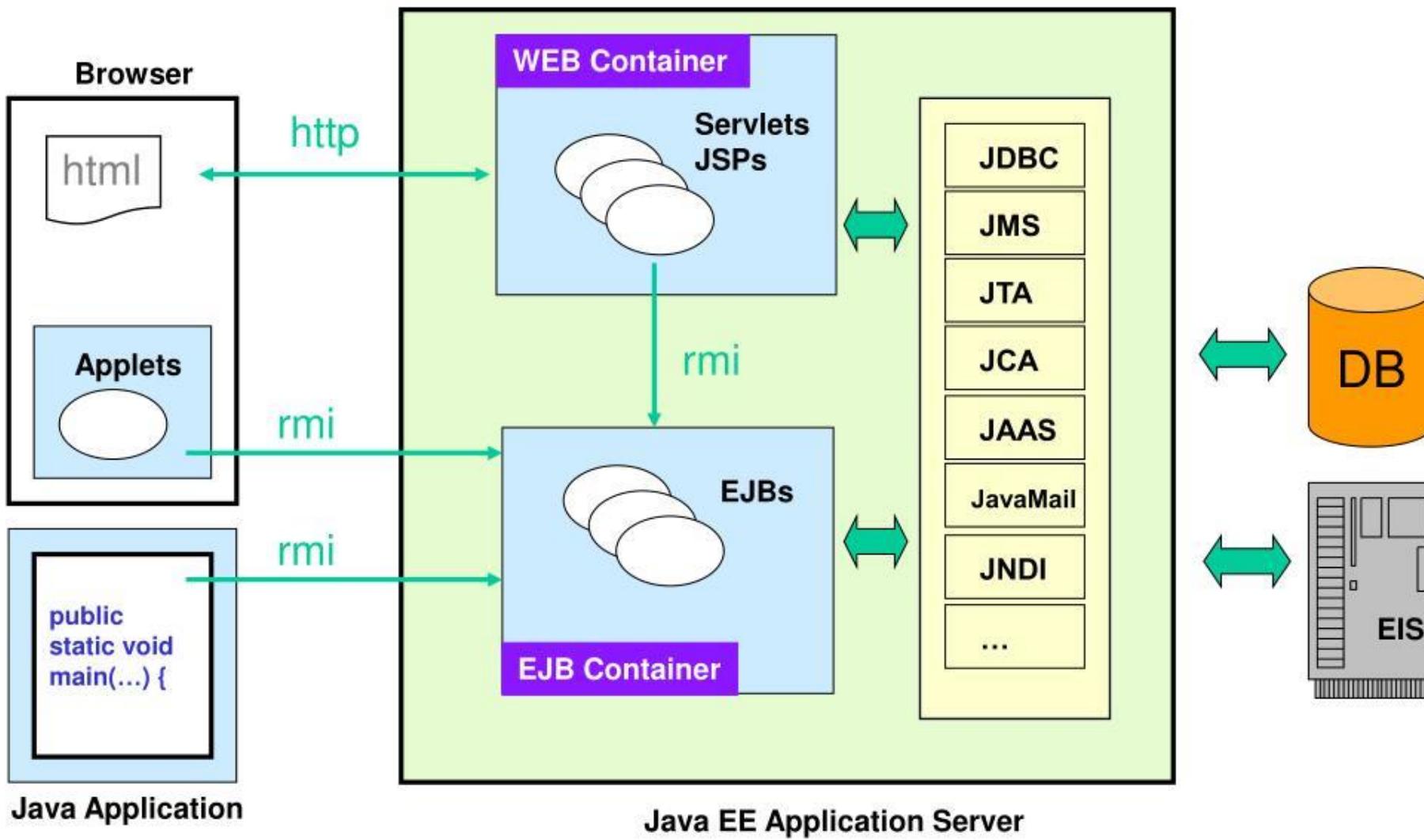


Distribution

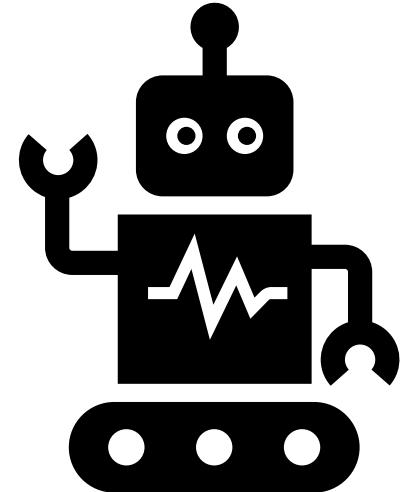
- Distributed application
- Load balancing
- Fail over
- Scalability
- Availability



Java EE - Architecture



Comment ça marche le web
déjà ?

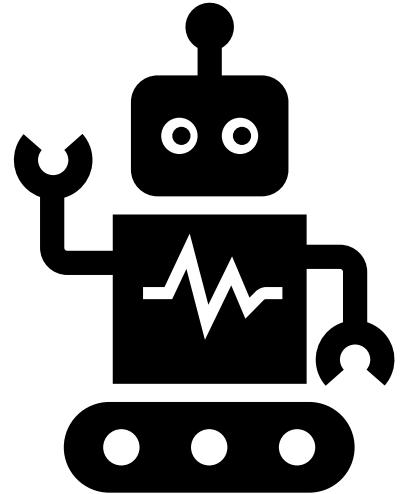


Serveur et client Web

Le serveur Web Apache

Ce qu'il faut savoir du protocole HTTP

Qu'est ce qu'une application
Web Java ?



Un JSR (Java Specification Request) est une proposition formelle pour ajouter une nouvelle fonctionnalité à la plateforme Java. Les JSRs sont soumises au Java Community Process (JCP), un organisme responsable de l'évolution des technologies Java. Chaque JSR suit un processus de révision et d'approbation, qui inclut des contributions et des commentaires de la communauté Java.

Exemples de JSRs

Voici quelques exemples de JSRs bien connus :

JSR 299 (Contexts and Dependency Injection for Java EE) : Ajoute des fonctionnalités pour l'injection de dépendances et la gestion des contextes dans Java EE.

JSR 317 (Java Persistence API) : Définit l'API pour la gestion de la persistance des objets dans les applications Java, remplaçant en grande partie les anciennes technologies de persistance comme EJB CMP.

JSR 340 (Java Servlet 3.1) : Décrit les spécifications pour l'API Servlet, une composante essentielle pour le développement d'applications web en Java.

Les JSRs sont cruciales pour l'évolution continue de la plateforme Java. Elles permettent l'ajout de nouvelles fonctionnalités de manière structurée et coordonnée, tout en impliquant la communauté Java dans le processus de développement. Cela garantit que la plateforme reste pertinente et à jour avec les besoins de l'industrie et de la communauté des développeurs.

Une Implementation de Référence (Reference Implementation ou RI) est une implémentation concrète d'une spécification ou d'un standard qui sert de modèle et de guide pour les autres implémentations

Exemples dans le contexte Java

1. JavaServer Faces (JSF) :

1. Spécification : JSR 372 pour JSF 2.3.
2. Implementation de Référence : Mojarra est l'implémentation de référence de JSF.

2. Java Persistence API (JPA) :

1. Spécification : JSR 338 pour JPA 2.1.
2. Implementation de Référence : EclipseLink est l'implémentation de référence de JPA.

3. Servlets :

1. Spécification : JSR 340 pour Servlet 3.1.
2. Implementation de Référence : Apache Tomcat est l'une des implémentations de référence les plus courantes pour les servlets.

Comment une RI est utilisée

1. Développement et Test : Les développeurs peuvent utiliser la RI pour tester leurs propres implémentations et s'assurer qu'elles sont conformes à la spécification.

2. Documentation : La RI est souvent accompagnée de documentation détaillée et de commentaires dans le code source, aidant ainsi les développeurs à comprendre les choix de conception et les meilleures pratiques.

Java EE (Jakarta EE) est une plateforme robuste pour développer des applications d'entreprise. Elle repose sur des abstractions et des conteneurs pour simplifier la gestion des composants, et elle est guidée par des Java Specification Requests (JSR) et des Implementations de Référence (RI).

Fonctionnement en Java EE

Développement : Utilisation des API standardisées définies par les JSR pour écrire du code.

Déploiement : Applications empaquetées en WAR ou EAR, déployées sur des serveurs d'applications comme WildFly ou GlassFish.

Exécution : Les conteneurs gèrent le cycle de vie, les transactions et la sécurité des composants.

Importance des JSR et RI

Standardisation : Les JSR garantissent que toutes les implémentations suivent les mêmes standards.

Interopérabilité : Les RI montrent comment implémenter correctement les spécifications, assurant ainsi que les différentes implémentations peuvent fonctionner ensemble.

Exemple Concret

JSR 340 (Java Servlet 3.1) : Décrit les spécifications pour les servlets.

RI : Apache Tomcat, une implémentation courante des servlets, montrant comment les utiliser dans des applications web.

Conclusion

Java EE, guidé par les JSR et validé par les RI, fournit un cadre structuré et standardisé pour développer des applications d'entreprise. Les abstractions et les conteneurs simplifient le développement et la gestion des composants, permettant aux développeurs de se concentrer sur la logique métier sans se soucier des détails de l'infrastructure.

Rôle des Conteneurs en Java EE

1. Exécution et Gestion des Composants :

1. Les conteneurs exécutent les composants d'application (comme les Servlets, JSP, EJB) et gèrent leur cycle de vie.
2. Ils créent, initialisent, gèrent et détruisent les instances des composants en fonction des besoins de l'application.

2. Fourniture de Services :

1. Les conteneurs fournissent des services essentiels aux composants, tels que :
 1. Gestion des Transactions : Assure la cohérence et l'intégrité des données en gérant les transactions distribuées.
 2. Sécurité : Gère l'authentification et l'autorisation des utilisateurs.
 3. Injection de Dépendances : Injecte automatiquement les ressources nécessaires dans les composants (comme les EJB ou les connexions aux bases de données).
 4. Gestion de la Concurrence : Gère l'accès concurrent aux ressources partagées.
 5. Communication Asynchrone : Facilite la gestion des messages asynchrones (via JMS).

1. Interopérabilité et Standardisation :

1. En suivant les spécifications définies par les JSR, les conteneurs assurent que les composants respectent les standards et peuvent fonctionner de manière interopérable avec d'autres implémentations.
2. Les conteneurs implémentent les fonctionnalités définies par les JSR, mais ils peuvent également offrir des optimisations et des fonctionnalités supplémentaires.

Exemple de Fonctionnement

Conteneur Web

- **JSR 340 (Java Servlet 3.1) :**

- Spécification : Décrit les règles et les comportements que les Servlets doivent suivre.
- Conteneur : Implémente ces règles en fournissant un environnement où les Servlets peuvent être déployés et exécutés.
- Services Fournis : Gestion des requêtes et des réponses HTTP, gestion des sessions, sécurité web, etc.

Les conteneurs en Java EE transforment les abstractions définies par les JSR en des implémentations concrètes et fonctionnelles, souvent représentées par des Implementations de Référence (RI). Ils fournissent un environnement d'exécution complet pour les composants d'application, offrant une multitude de services essentiels et garantissant que les implémentations sont conformes aux standards.

Analogie : Traduction d'un Manuel de Conception en Produit Réel

1. Manuel de Conception (JSR) :

1. Imaginez un manuel détaillé qui décrit comment construire un produit. Ce manuel inclut toutes les spécifications, les standards, et les directives nécessaires pour garantir que le produit fonctionne correctement et est conforme aux normes.
2. En Java EE, ce manuel est représenté par les JSR (Java Specification Requests).

2. Prototype de Référence (RI) :

1. Un ingénieur utilise ce manuel pour construire un prototype fonctionnel du produit. Ce prototype sert de modèle pour montrer que le manuel peut être suivi pour créer un produit fonctionnel.
2. En Java EE, ce prototype est l'Implementation de Référence (RI).

3. Usine de Production (Conteneur) :

1. L'usine utilise le manuel et le prototype comme guide pour produire le produit à grande échelle. Elle s'assure que chaque produit fabriqué est conforme aux spécifications du manuel et fonctionne comme le prototype.
2. En Java EE, cette "usine" est le conteneur.



Jakarta EE : L'Évolution de Java EE

Jakarta EE est l'évolution de Java EE (Java Platform, Enterprise Edition), sous l'égide de la **Fondation Eclipse**. Elle continue de fournir un cadre robuste et standardisé pour le développement d'applications d'entreprise.

Historique

1. Java EE :

1. Initialement développé par Sun Microsystems, puis maintenu par Oracle après l'acquisition de Sun.
2. Offrait une plateforme standardisée pour développer des applications d'entreprise avec des composants modulaires (EJB, JPA, JMS, etc.).

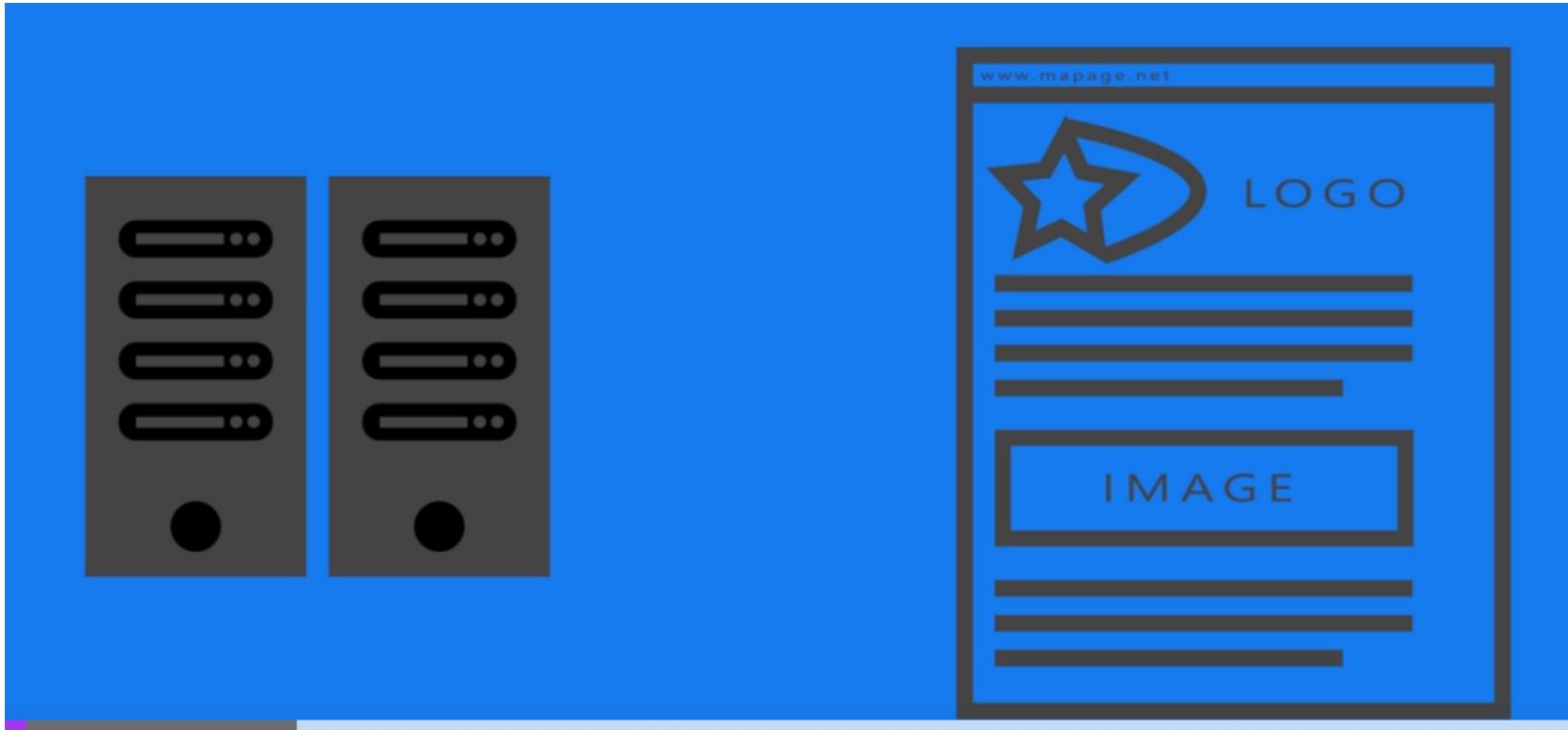
2. Transition vers Jakarta EE :

1. En 2017, Oracle a transféré la gestion de Java EE à la Fondation Eclipse.
2. Le nom "Jakarta EE" a été adopté pour continuer le développement de la plateforme sous une nouvelle gouvernance.

Jakarta EE continue l'héritage de Java EE en offrant une plateforme robuste et standardisée pour le développement d'applications d'entreprise. Sous la gouvernance de la Fondation Eclipse, elle vise à évoluer de manière ouverte et collaborative, assurant que les applications restent modernes, portables et interopérables. Les conteneurs jouent un rôle clé en fournissant l'infrastructure nécessaire pour gérer les composants et les services essentiels, simplifiant ainsi le développement et la gestion des applications d'entreprise.

Serveur d'application Web Java

Utilise un Language de Programmation pour generer du Contenu Web



Un serveur d'application est un logiciel situé entre l'utilisateur et l'architecture transactionnelle.

Il permet de communiquer avec des bases de données, de consulter des web-services, effectuer des recherches, piloter un robot, ...

L'objectif d'un serveur d'application est le même qu'un mainframe : permettre à partir d'un client aussi léger que possible d'effectuer des traitements distants sur une machine puissante, en mode transactionnel.

Les utilisateurs y accèdent par le biais d'un navigateur.

De l'autre côté, le serveur séparent les niveaux : accès aux données, traitement métier et présentation.

Ils travaillent à partir de composants objets réutilisables tels que les **EJB**.

Ces composants peuvent assurer de manière plus ou moins cachée, les fonctions de moniteur transactionnel, la persistance des données, la gestion de la montée en charge, ..

Serveur apache PHP

Serveur apache

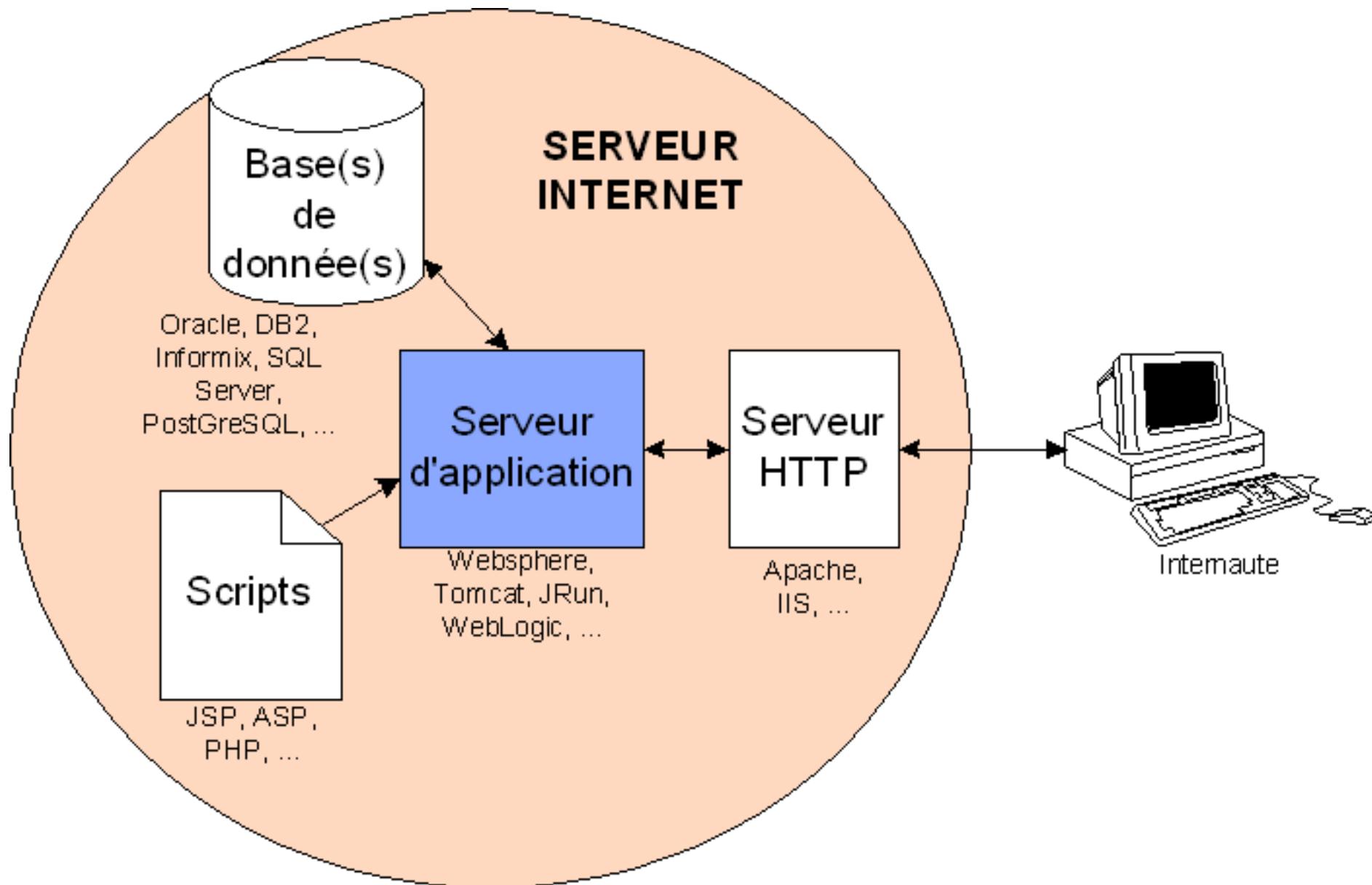
Serveur d'application

JAVA

Conteneurs de servlet



Servlet = Petit programme serveur



Installer un serveur Tomcat

Le conteneur de Servlets - Tomcat comment ça marche ?

Structure d'une application Web Java

Simple Repertoire Avec du Contenu Web

Fichiers | Accueil | Partage | Affichage | webapps

Épingler sur l'accès rapide Copier Coller Copier le chemin d'accès Coller le raccourci Presse-papiers

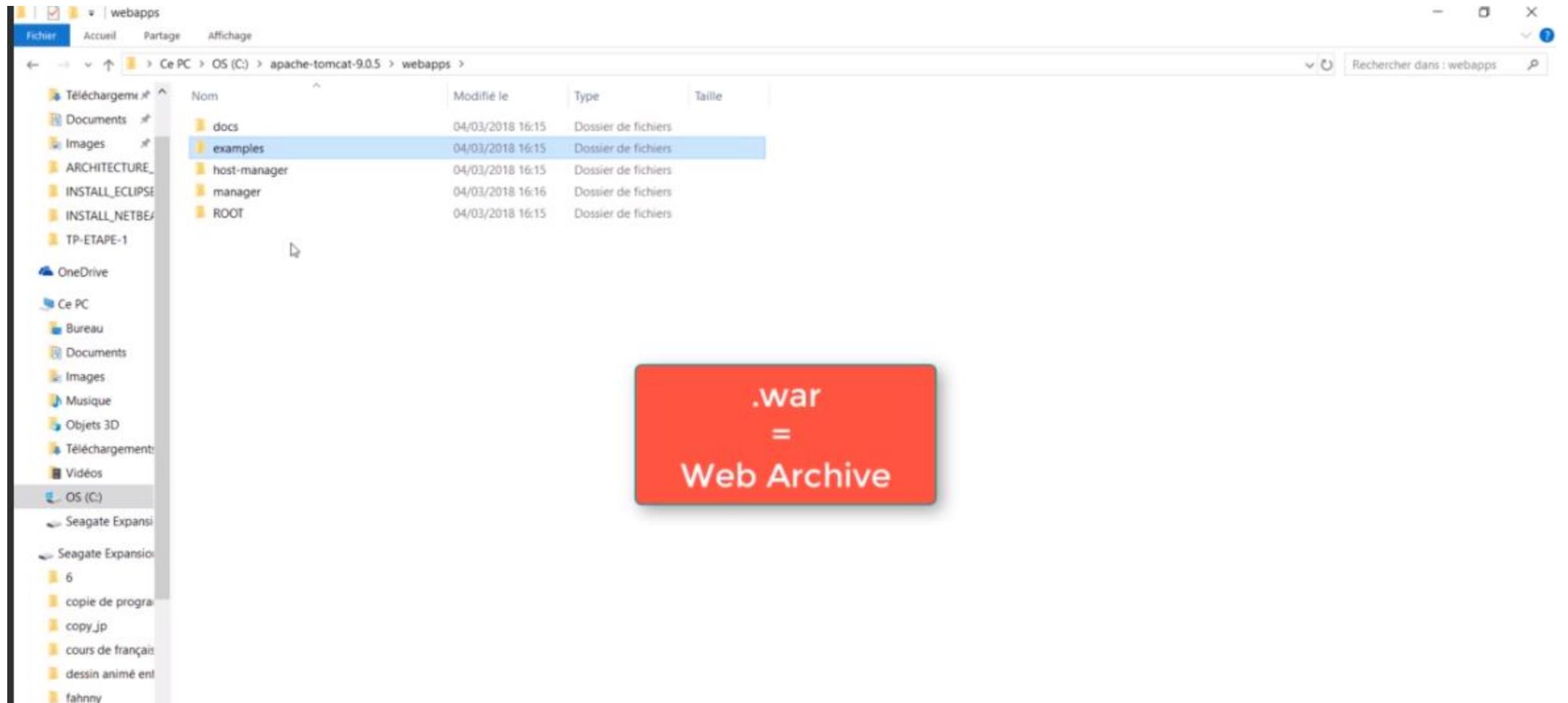
Déplacer vers Copier vers Supprimer Renommer Nouveau dossier Nouvel élément Accès rapide Propriétés Ouvrir Sélectionner tout Aucun Inverser la sélection

Nouveau Ouvrir Sélectionner

← → ↑ ↓ Ce PC > Téléchargements > apache-tomcat-10.1.24 > webapps

Rechercher dans : webapps

Images	Nom	Modifié le	Type	Taille
Projet_1 gestion_ecole	docs	2024-05-09 17:41	Dossier de fichiers	
chap_livre	examples	2024-05-09 17:41	Dossier de fichiers	
projet1	host-manager	2024-05-09 17:41	Dossier de fichiers	
version_word des articles	manager	2024-05-09 17:41	Dossier de fichiers	
[OxTorrent.com] Killjoys.S0	ROOT	2024-05-09 17:41	Dossier de fichiers	
Android_Kotlin				
dv2				
jee				
OneDrive				



.war
=

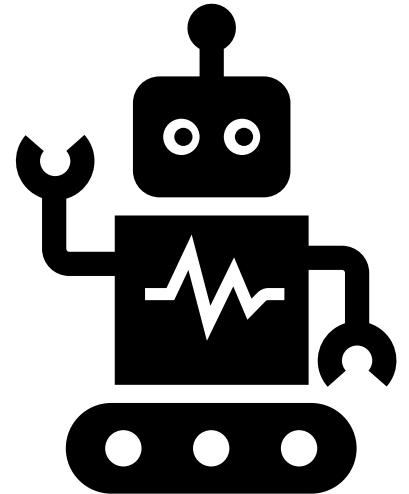
Web Archive

Première application Web Java

Créer un Simple Répertoire et Un Index.html « hello world »

Le faire Depuis Neatbeans

Fournir du contenu dynamique - Introduction aux Servlets Java EE



La Servlet Http

Une Servlet est une classe Java utilisée pour créer des applications web dynamiques. Elle s'exécute sur un serveur web ou un serveur d'applications et génère des réponses aux requêtes HTTP, généralement en produisant du HTML ou en manipulant des données.

Fonctionnement d'une Servlet

1. Réception des Requêtes :

1. Une servlet est appelée par le serveur pour traiter une requête HTTP reçue d'un client (comme un navigateur web).
2. La requête HTTP peut être une requête GET, POST, PUT, DELETE, etc.

2. Traitement des Requêtes :

1. La servlet traite la requête en exécutant la logique d'application nécessaire.
2. Cela peut inclure des opérations comme lire des paramètres de requête, interagir avec une base de données, ou appeler d'autres services.

3. Génération de Réponses :

1. Après le traitement, la servlet génère une réponse HTTP.
2. Cette réponse peut inclure du contenu HTML, JSON, XML, ou d'autres types de données.

Cycle de Vie d'une Servlet

Le cycle de vie d'une servlet est géré par le conteneur de servlets et inclut les étapes suivantes :

Chargement et Initialisation :

La servlet est chargée en mémoire et une instance est créée par le conteneur de servlets.

La méthode init() est appelée pour initialiser la servlet.

Traitement des Requêtes :

Pour chaque requête HTTP, la méthode service() de la servlet est appelée.

Cette méthode appelle ensuite doGet(), doPost(), ou d'autres méthodes spécifiques en fonction du type de requête.

Destruction :

Lorsque la servlet n'est plus nécessaire, la méthode destroy() est appelée par le conteneur pour nettoyer les ressources.

- Retourner une page Web
- Retourner un fichier PDF
- Envoyer un email
- Afficher les twitts
- Contrôler un robot
- Piloter une ch. de montage
- etc...

SERVLET

1) Prendre en charge la requête

2) Effectuer des traitements

3) Retourner une réponse

New JAVA CLASS



```
PrintWriter out = response.getWriter();

out.println(
"<html>
<body>
<h1>Hello : Personne</h1>
</body>
</html>“
);
```



<default config>



433,0/634,0MB



Projects X Services Files

> firstApp-1.0-SNAPSHOT [master]
> FirstAppcore
OdcProjet1-1.0-SNAPSHOT
> Web Pages
> RESTful Web Services
> Source Packages
> Other Sources
> Dependencies
> Java Dependencies
> Project Files
pom.xml
nb-configuration.xml
> Projet1 [main]

BookServlet.java [-/M] X Chien.java [-/M] X Livre.java X HelloServeur.java X pom.xml [OdcProjet1] X

Source Graph Effective History

```
10 <properties>
11     <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>
12     <jakartaee>10.0.0</jakartaee>
13 </properties>
14
15 <dependencies>
16     <dependency>
17         <groupId>jakarta.platform</groupId>
18         <artifactId>jakarta.jakartaee-api</artifactId>
19         <version>${jakartaee}</version>
20         <scope>provided</scope>
21     </dependency>
22 </dependencies>
23
24 <build>
25     <plugins>
26         <plugin>
27             <groupId>org.apache.maven.plugins</groupId>
28             <artifactId>maven-compiler-plugin</artifactId>
29             <version>3.8.1</version>
30             <configuration>
```

Terminal - ...ygdrive/c/Program Files/NetBeans-21

Output

Transferring Maven repository index: Central Repository

1%

16:21

INS Windows (CRLF)



Taper ici pour rechercher

02:57
2024-06-05

32°C

FRA

MVN REPOSITORY

Indexed Artifacts (36.0M)

Popular Categories

- Testing Frameworks & Tools
- Android Packages
- Logging Frameworks
- Java Specifications
- JVM Languages
- JSON Libraries
- Language Runtime
- Core Utilities
- Mocking
- Web Assets
- Annotation Libraries

Search for groups, artifacts, categories

Categories | Popular | Contact Us

Home » jakarta.platform » jakarta.jakartaee-api » 8.0.0

Jakarta EE Platform API » 8.0.0

Jakarta EE Platform API

License	EPL 2.0 GPL
Categories	Java Specifications
Tags	jakarta standard api platform specs
Organization	Eclipse Foundation
Date	Sep 10, 2019
Files	pom (6 KB) jar (1.9 MB) View All
Repositories	Central
Ranking	#1818 in MvnRepository (See Top Artifacts) #57 in Java Specifications
Used By	268 artifacts
Vulnerabilities	Vulnerabilities from dependencies: CVE-2019-17091

Note: There is a new version for this artifact

Indexed Repositories (2024)

- Central
- Atlassian
- Hortonworks
- JCenter
- Sonatype
- JBossEA
- KtorEAP
- Atlassian Public
- WSO2 Releases
- WSO2 Public

Popular Tags

- aar
- android
- apache
- api
- application
- arm
- assets
- build
- build-system
- bundle
- client
- clojure
- cloud
- config
- cran
- data
- database
- eclipse
- example
- extension
- framework

MVN REPOSITORY

Indexed Artifacts (36.0M)

Popular Categories

- Testing Frameworks & Tools
- Android Packages
- Logging Frameworks
- Java Specifications
- JVM Languages
- JSON Libraries
- Language Runtime
- Core Utilities
- Mocking
- Web Assets
- Annotation Libraries

Search for groups, artifacts, categories

Home » jakarta.platform » jakarta.jakartaee-web-api

Jakarta EE Web Profile API

Jakarta EE Web Profile API

License	EPL 2.0 GPL
Categories	Java Specifications
Tags	jakarta standard web api platform specs
Ranking	#6769 in MvnRepository (See Top Artifacts) #102 in Java Specifications
Used By	63 artifacts

Central (11)

	Version	Vulnerabilities	Repository	Usages	Date
11.0.x	11.0.0-M2		Central	1	Apr 16, 2024
	11.0.0-M1		Central	1	Dec 20, 2023
10.0.x	10.0.0		Central	23	Sep 13, 2022
	10.0.0-RC1		Central	1	Sep 13, 2022
9.1.0		Central	16	May 25, 2021	

Exécuter une Servlet Http, le chemin d'accès

```
@WebServlet(name="HelloServelet",urlPatterns={"/Hello"} )  
public class HelloServeur extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws  
ServletException, IOException {  
    resp.setContentType("text/html");  
  
    PrintWriter out = resp.getWriter();  
    out.println("<html><body><h1>Hello from HelloServeur</h1></body></html>");  
}  
}
```

Générer une Servlet avec Netbeans

Générer une Servlet avec Eclipse

Utiliser d'autres classes dans la Servlet

Projects Files Services

firstApp-1.0-SNAPSHOT

- > Web Pages
- > RESTful Web Services
- > Source Packages
 - > com.mycompany.Controleur
 - > com.mycompany.dto
 - Personne.java
 - > com.mycompany.firstapp
 - > com.mycompany.firstapp.resources
- > Test Packages
- > Other Sources
- > Dependencies
- > Java Dependencies
- > Project Files
 - pom.xml
 - nb-configuration.xml
- > mavenproject1

HelloServeur.java x Personne.java x

Source History

```
23
24
25     @Override
26     protected void doGet(HttpServletRequest request, HttpServletResponse response)
27             throws ServletException, IOException {
28         PrintWriter out = response.getWriter();
29
30         Personne sory = new Personne("sory","millimono");
31         out.println("<html><body><h1>Hello : "+sory.toString() + "</h1></body></html>");
32     }
33 }
34 }
```

com.mycompany.Controleur.HelloServeur > doGet >

Output x

Run (firstApp-1.0-SNAPSHOT) x Apache Tomcat or TomEE Log x Apache Tomcat or TomEE x

```
unaeploying ...
undeploy?path=/firstApp
OK - Application non déployée pour le chemin de contexte [/firstApp]
In-place deployment at C:\Users\Lenovo\Documents\NetBeansProjects\firstApp\target\firstApp-1.0-SNAPSHOT
Deployment is in progress...
deploy?config=file%3A%2FC%3A%2FUsers%2FLenovo%2FAppData%2FLocal%2FTemp%2Fcontext13152939815560662136.xml&path=/fi
OK - Application déployée pour le chemin de contexte [/firstApp]
Start is in progress...
start?path=/firstApp
OK - Application démarrée pour le chemin de contexte [/firstApp]
```

doGet - Navigator x

Members <empty>

HelloServeur :: HttpServlet

- HelloServeur()
- doGet(HttpServletRequest request, HttpServletResponse response)

Taper ici pour rechercher

31:46 30°C 21:49 INS Windows (CRLF)

Windows Taskbar icons: File Explorer, FileZilla, Microsoft Edge, Opera, FileZilla, WhatsApp, Microsoft Word, Microsoft Excel, Microsoft Powerpoint, Microsoft Teams, Microsoft Edge DevTools, Microsoft Store, Microsoft Edge DevTools.

```
public class Personne
{
    private String lastName;
    private String firstName;

    public Personne() {
    }

    public Personne(String lastName, String firstName) {
        this.lastName = lastName;
        this.firstName = firstName;
    }
}
```



File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help

Apache NetBeans IDE 21



Search (Ctrl+I)

— □ X



Projects X Files Services

> firstApp-1.0-SNAPSHOT
> FirstAppcore
 Source Packages
 com.mycompany.firstappcore
 FirstAppcore.java
 Personne.java
 Dependencies
> Java Dependencies
> Project Files

HelloServeur.java X Personne.java X FirstAppcore.java X

Source History

```
15  
16  
17  /**  
18  *  
19  * @author Lenovo  
20  */  
21  @.WebServlet(name = "HelloServeur", urlPatterns = {"/Hello"})  
22  public class HelloServeur extends HttpServlet {  
23  
24  
25  @Override
```

com.mycompany.Controleur.HelloServeur >

Output X

```
Apache Tomcat or TomEE Log X Apache Tomcat or TomEE X Run (firstApp-1.0-SNAPSHOT) X  
profile mode: false  
debug mode: false  
force redeploy: true  
Undeploying ...  
undeploy?path=/firstApp  
OK - Application non déployée pour le chemin de contexte [/firstApp]  
In-place deployment at C:\Users\Lenovo\Documents\NetBeansProjects\firstApp\target\firstApp-1.0-SNAPSHOT  
Deployment is in progress...  
deploy?config=file%3A%2FC%3A%2FUsers%2FLenovo%2FAppData%2FLocal%2FTemp%2Fcontext3390656747983659544.xml&path=/fir  
OK - Application déployée pour le chemin de contexte [/firstApp]  
Start is in progress...  
start?path=/firstApp  
OK - Application démarrée pour le chemin de contexte [/firstApp]
```

Navigator X

deploy deploy-file
install install-file
resources copy-resources
site attach-descriptor



Fichiers .war

File Edit View Navigate Source Refactor Run Debug Profile Team Tools Window Help firstApp-1.0-SNAPSHOT - Apache NetBeans I... Search (Ctrl+I) — □ X

Projects Files Services

- > generated-test-sources
- > maven-archiver
- > maven-status
- > test-classes
- firstApp-1.0-SNAPSHOT.war
 - > <default package>
 - > META-INF
 - > META-INF.maven.com.mycompany.firstApp
 - > WEB-INF
 - > WEB-INF.classes.Controleur
 - > WEB-INF.classes.META-INF
 - > WEB-INF.classes.com.mycompany.Controleur
 - > WEB-INF.classes.com.mycompany.dto
 - > WEB-INF.classes.com.mycompany.firstapp
 - > WEB-INF.classes.com.mycompany.firstapp.resources
 - > WEB-INF.lib
 - > FirstAppcore-1.0.jar
- nb-configuration.xml
- pom.xml
- FirstAppcore
 - > src
 - > target
- pom.xml

Navigator X

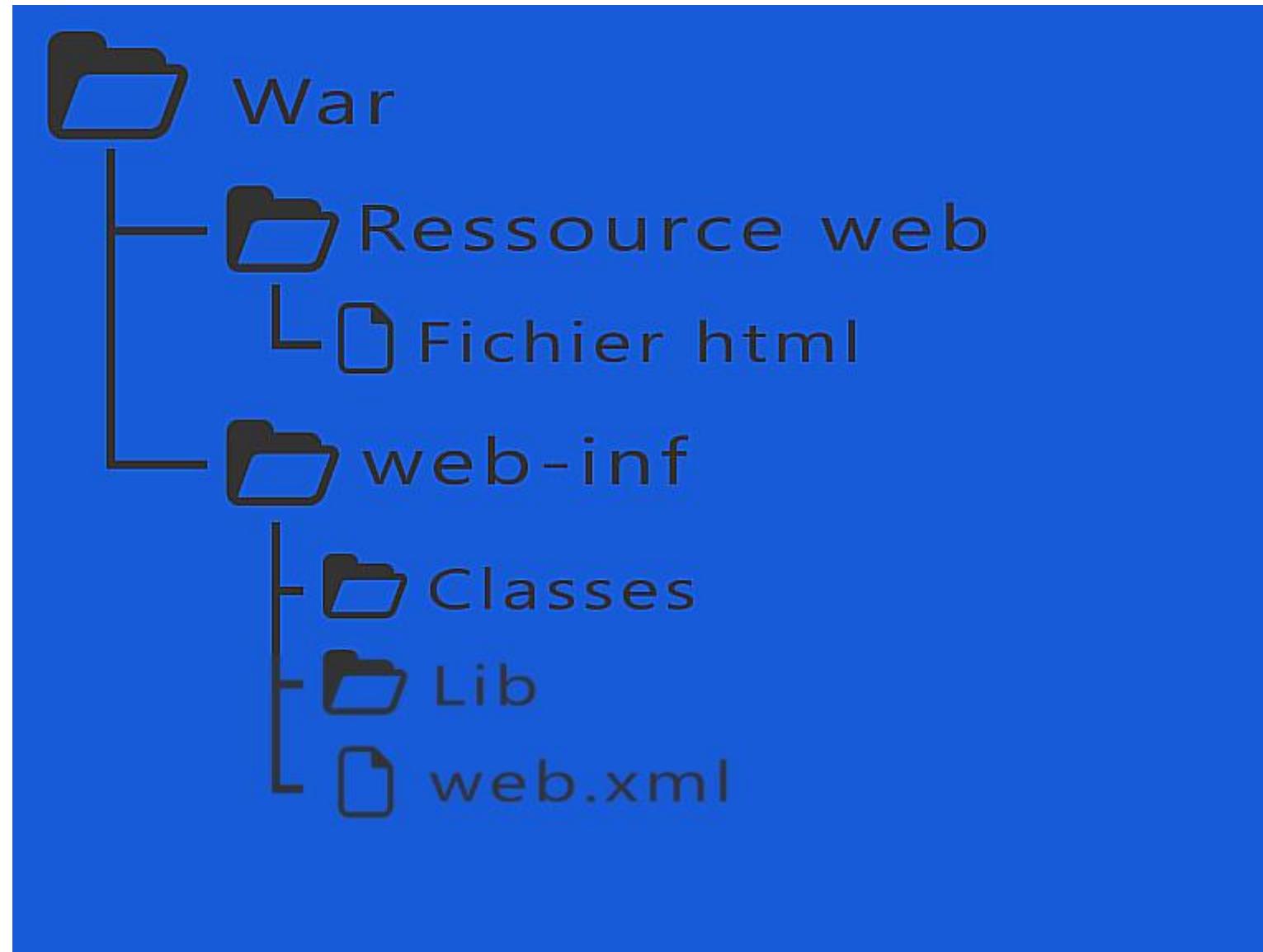
POM model

- Model Version : 4.0.0
- GroupId : com.mycompany
- ArtifactId : firstApp

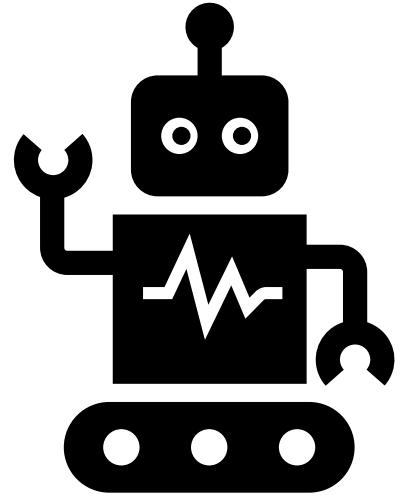
Output X

Apache Tomcat or TomEE Log X Apache Tomcat or TomEE X Run (firstApp-1.0-SNAPSHOT) X

```
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance" xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.org/xsd/maven-4.0.0.xsd">  
    <modelVersion>4.0.0</modelVersion>  
    <groupId>com.mycompany</groupId>  
    <artifactId>firstApp</artifactId>  
    <version>1.0-SNAPSHOT</version>  
    <packaging>war</packaging>  
    <name>firstApp-1.0-SNAPSHOT</name>  
  
    <properties>  
        <project.build.sourceEncoding>UTF-8</project.build.sourceEncoding>  
        <jakartae>10.0.0</jakartae>  
    </properties>  
  
    <dependencies>  
        <dependency>  
            <groupId>jakarta.platform</groupId>  
            <artifactId>jakarta.jakartae-api</artifactId>  
            <version>${jakartae}</version>  
            <scope>provided</scope>  
        </dependency>  
        <dependency>  
            <groupId>com.mycompany</groupId>  
            <artifactId>FirstAppcore</artifactId>  
        </dependency>  
    </dependencies>  
</project>
```



Les Servlets - concepts fondamentaux



Content type

```
@WebServlet(name = "HelloServeur", urlPatterns = {"/Hello"})
public class HelloServeur extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        Personne sory = new Personne("sory","millimono");
        out.println("<html><body><h1>Hello : "+sory.toString() + "</h1></body></html>");
    }
}
```

Liste des types MIME communs

[Liste des types MIME communs - HTTP | MDN \(mozilla.org\)](https://developer.mozilla.org/en-US/docs/Web/HTTP/Basics_of_HTTP/MIME_types/Common_types)

Jeu de caractères / Charsets

ISO-8859-1

=

LATIN 1

UTF-8

Générer un autre type de contenu
comme du PDF

The latest iText Suite 8.0.4 release features GraalVM Native Image support for iText Core, plus improved PDF/UA creation and digital signature validation. >

HARNESS THE POWER OF PDF

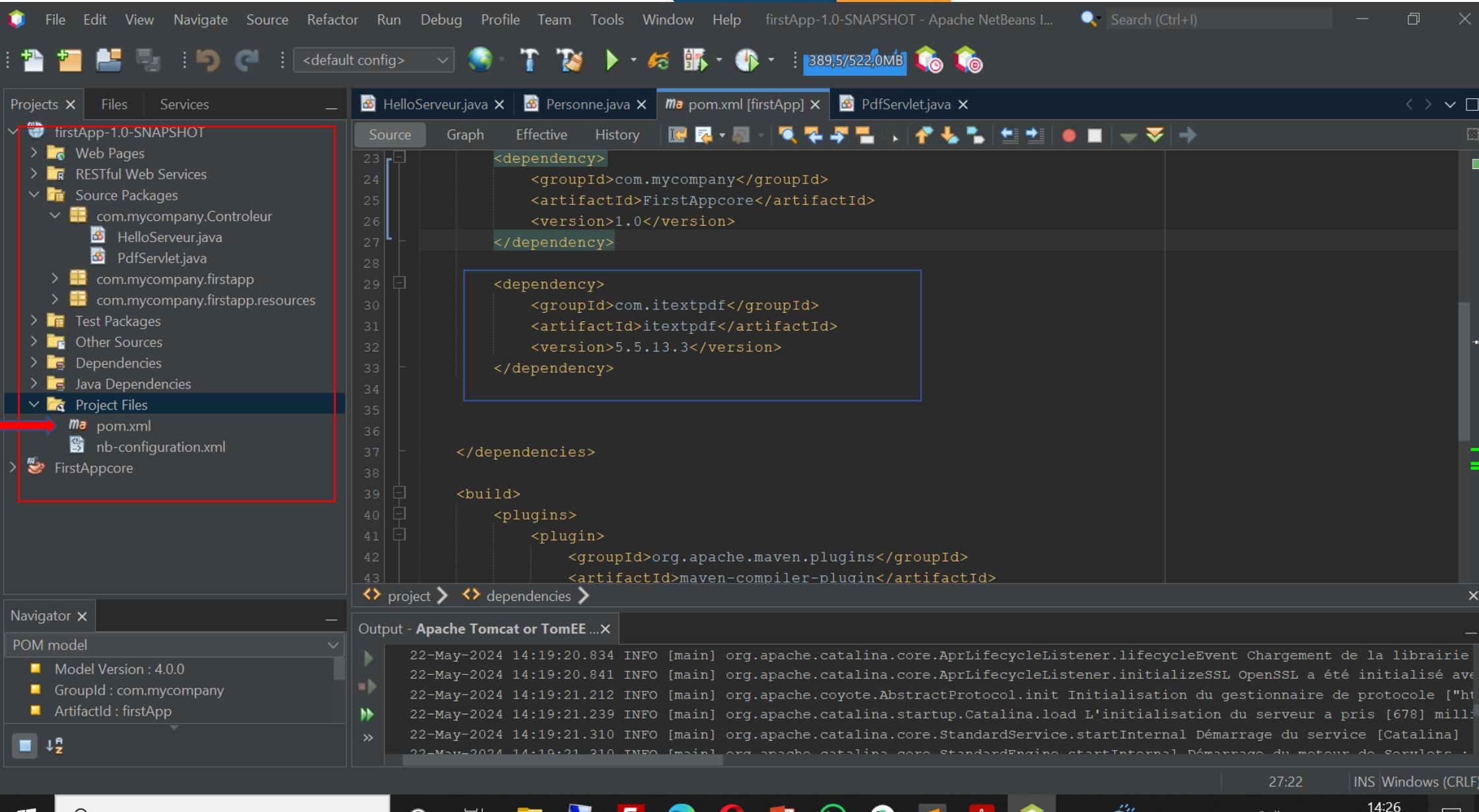
Generate & Manipulate PDFs Effortlessly

The preferred PDF technology, by developers for developers.

For small Enterprises, large Corporations, and Government Institutions.

Get started and try for yourself now >





The screenshot shows the Apache NetBeans IDE interface with a Maven project named "firstApp-1.0-SNAPSHOT".

Projects View: Shows the project structure under "firstApp-1.0-SNAPSHOT". A red box highlights the "Project Files" section, which contains the "pom.xml" file. A red arrow points from the "Get started and try it" button on the left sidebar to this "pom.xml" file.

pom.xml Content:

```
<dependency>
    <groupId>com.mycompany</groupId>
    <artifactId>FirstAppcore</artifactId>
    <version>1.0</version>
</dependency>

<dependency>
    <groupId>com.itextpdf</groupId>
    <artifactId>itextpdf</artifactId>
    <version>5.5.13.3</version>
</dependency>

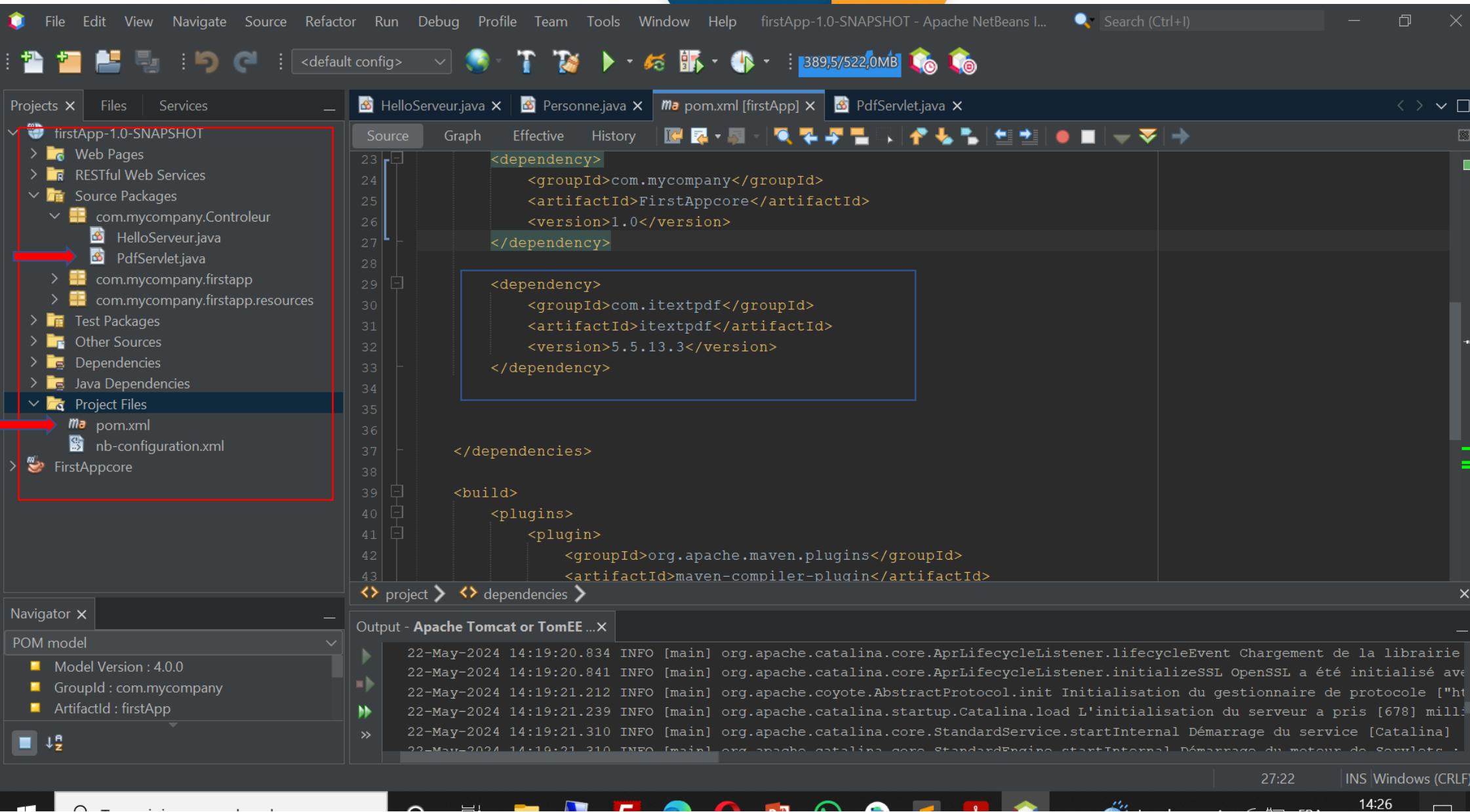
</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
        
```

Output View: Displays logs for "Apache Tomcat or TomEE ...".

```
22-May-2024 14:19:20.834 INFO [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent Chargement de la librairie
22-May-2024 14:19:20.841 INFO [main] org.apache.catalina.core.AprLifecycleListener.initializeSSL OpenSSL a été initialisé ave
22-May-2024 14:19:21.212 INFO [main] org.apache.coyote.AbstractProtocol.init Initialisation du gestionnaire de protocole ["ht
22-May-2024 14:19:21.239 INFO [main] org.apache.catalina.startup.Catalina.load L'initialisation du serveur a pris [678] milli
22-May-2024 14:19:21.310 INFO [main] org.apache.catalina.core.StandardService.startInternal Démarrage du service [Catalina]
22-May-2024 14:19:21.310 INFO [main] org.apache.catalina.core.StandardEngine.startInternal Démarrage du moteur de Servlets ...

```



The screenshot shows the Apache NetBeans IDE interface with a Maven project named "firstApp-1.0-SNAPSHOT".

Projects View: Shows the project structure under "firstApp-1.0-SNAPSHOT". A red box highlights the "Source Packages" node, which contains the "com.mycompany.Controleur" package. Inside this package are the files "HelloServeur.java" and "PdfServlet.java". Another red arrow points from the "Project Files" node to the "pom.xml" file.

pom.xml Content:

```
<dependency>
    <groupId>com.mycompany</groupId>
    <artifactId>FirstAppcore</artifactId>
    <version>1.0</version>
</dependency>

<dependency>
    <groupId>com.itextpdf</groupId>
    <artifactId>itextpdf</artifactId>
    <version>5.5.13.3</version>
</dependency>

</dependencies>

<build>
    <plugins>
        <plugin>
            <groupId>org.apache.maven.plugins</groupId>
            <artifactId>maven-compiler-plugin</artifactId>
        
```

Output View: Displays logs for the Apache Tomcat or TomEE server. The logs show the startup of the Catalina service and the loading of the StandardEngine.

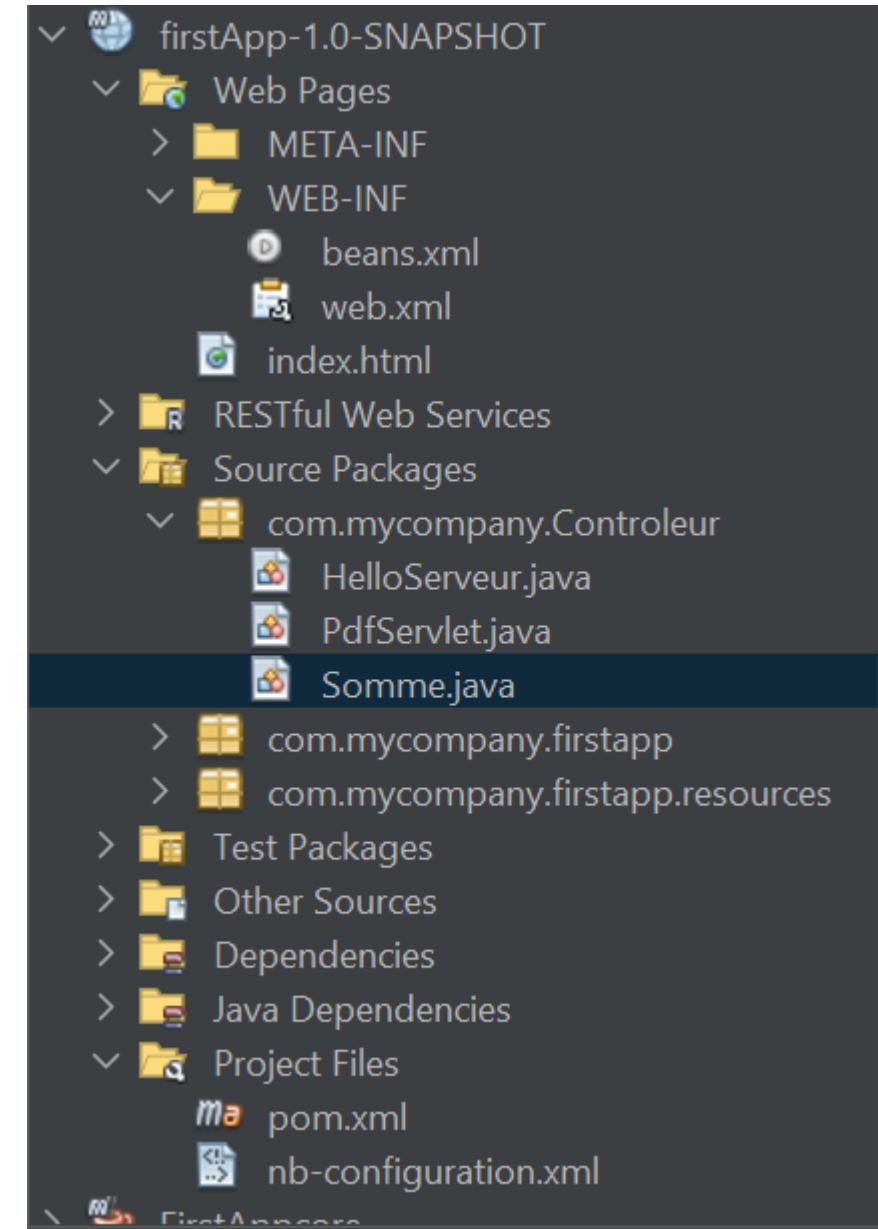
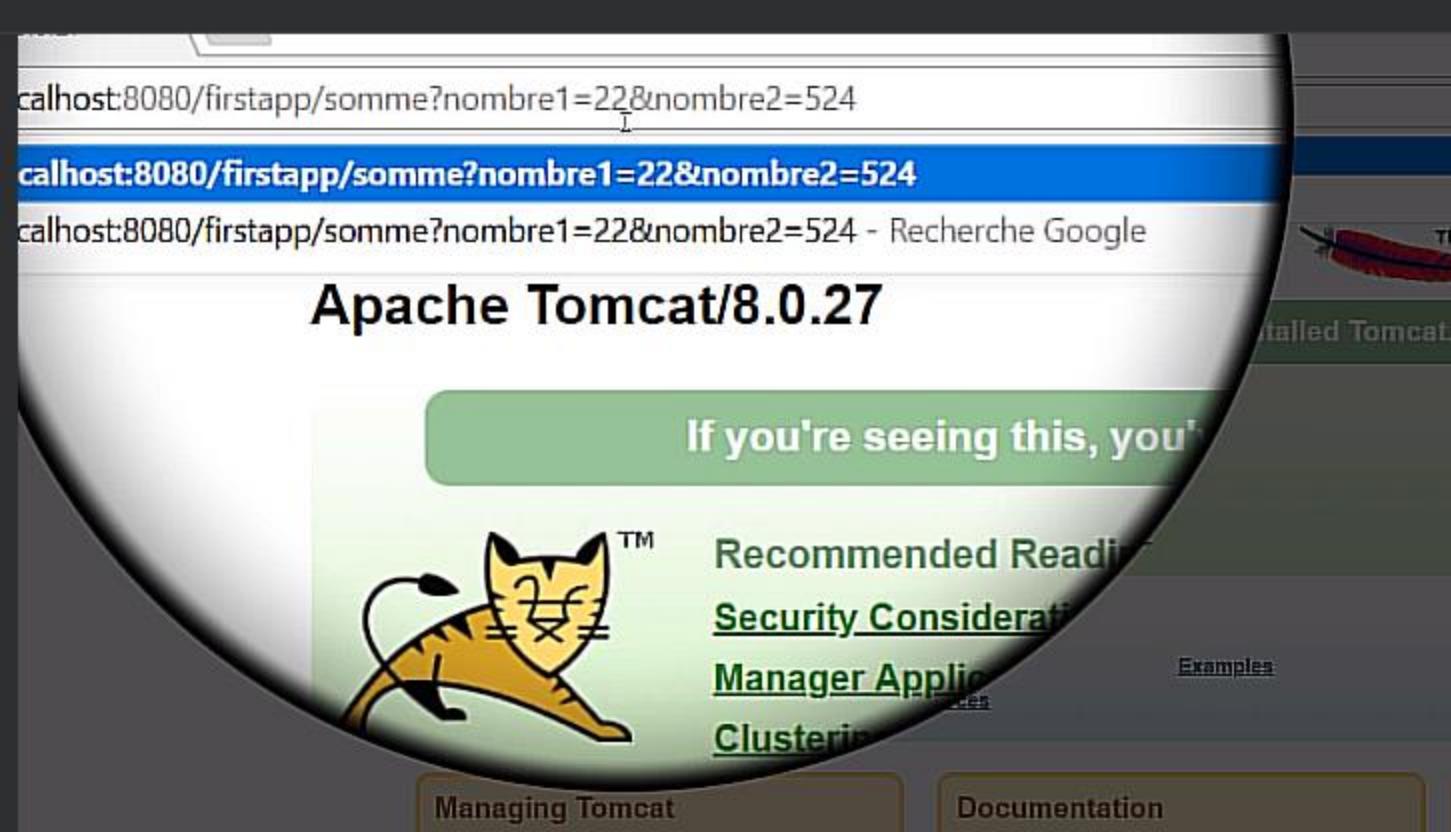
```
22-May-2024 14:19:20.834 INFO [main] org.apache.catalina.core.AprLifecycleListener.lifecycleEvent Chargement de la librairie
22-May-2024 14:19:20.841 INFO [main] org.apache.catalina.core.AprLifecycleListener.initializeSSL OpenSSL a été initialisé ave
22-May-2024 14:19:21.212 INFO [main] org.apache.coyote.AbstractProtocol.init Initialisation du gestionnaire de protocole ["ht
22-May-2024 14:19:21.239 INFO [main] org.apache.catalina.startup.Catalina.load L'initialisation du serveur a pris [678] milli
22-May-2024 14:19:21.310 INFO [main] org.apache.catalina.core.StandardService.startInternal Démarrage du service [Catalina]
22-May-2024 14:19:21.310 INFO [main] org.apache.catalina.core.StandardEngine.startInternal Démarrage du moteur de Servlets :
```

```
<dependency>  
  
<groupId>com.itextpdf</g  
roupId>  
  
<artifactId>itextpdf</artif  
actId>  
  
<version>5.5.13.3</versio  
n>  
</dependency>
```

```
@WebServlet(name = "PdfServlet", urlPatterns = {"/Pdf"})  
public class PdfServlet extends HttpServlet {  
  
    @Override  
    protected void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException  
    {  
        response.setContentType("application/pdf");  
        try{  
            Document doc =new Document();  
            PdfWriter.getInstance(doc, response.getOutputStream());  
            doc.open();  
            doc.add(  
                new Paragraph("hello toi")  
            );  
            doc.close();  
  
        } catch (Exception e)  
        {  
            e.printStackTrace();  
        } } }
```

Le descripteur de déploiement web.xml

Transmission de paramètres

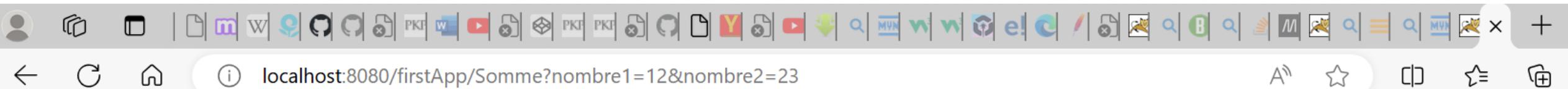


```
@WebServlet(name = "Somme", urlPatterns = {"/Somme"})
public class Somme extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        String nombre1 = request.getParameter("nombre1");
        String nombre2 = request.getParameter("nombre2");

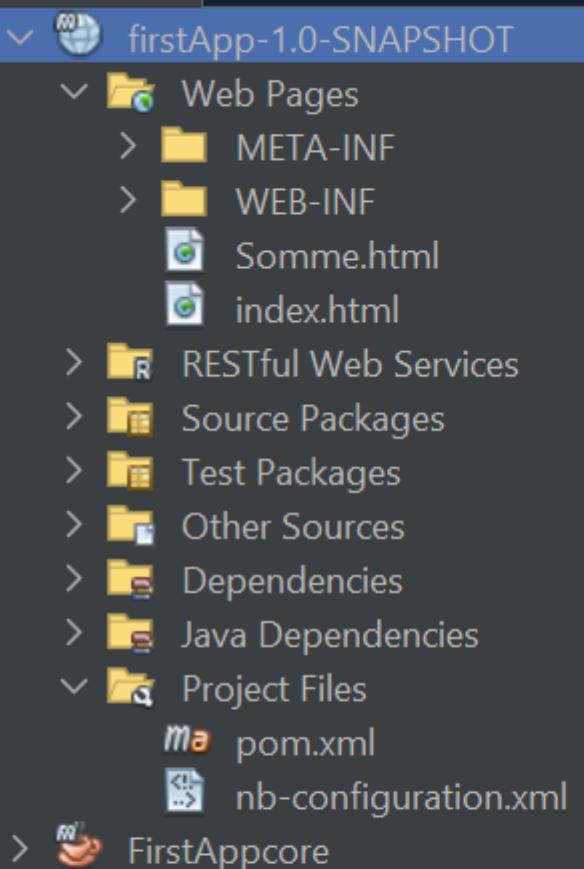
        int somme = Integer.parseInt(nombre1) + Integer.parseInt(nombre2);

        out.print("<HTML> <BODY> la somme est "+somme+" </BODY></HTML>");
    }
}
```



la somme est 24

Traitement des formulaires



```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Calculate Sum</title>
</head>
<body>
    <h1>Calculate Sum of Two Numbers</h1>
    <form action="Somme" method="GET">
        <div>
            <label for="nombre1">Number 1:</label>
            <input type="text" id="nombre1" name="nombre1" required>
        </div>
        <div>
            <label for="nombre2">Number 2:</label>
            <input type="text" id="nombre2" name="nombre2" required>
        </div>
        <button type="submit">Calculate Sum</button>
    </form>
</body>
</html>
```

La méthode de soumission POST

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Calculate Sum</title>
</head>
<body>
  <h1> Authentification </h1>
  <form action="Login" method="GET">
    <div>
      <label for="email">Email</label>
      <input type="text" id="email" name="email" required>
    </div>
    <div>
      <label for="password">Password:</label>
      <input type="password" id="password" name="password" required>
    </div>
    <button type="submit">Calculate Sum</button>
  </form>
</body>
</html>
```

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8">
  <title>Calculate Sum</title>
</head>
<body>
  <h1> Authentification </h1>
  <form action="Login" method="POST">
    <div>
      <label for="email">Email</label>
      <input type="text" id="email" name="email" required>
    </div>
    <div>
      <label for="password">Password:</label>
      <input type="password" id="password" name="password" required>
    </div>
    <button type="submit">Calculate Sum</button>
  </form>
</body>
</html>
```

Exécutions simultanées d'une Servlet - Thread Safety

```
@WebServlet(name = "ThreadSafeServlet", urlPatterns = {"/TestThread"})
public class ThreadSafeServlet extends HttpServlet {
    int solde;

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        solde = Integer.parseInt(request.getParameter("retait"));

        try {
            Thread.sleep(10000);

        } catch (Exception e) {
        }
        PrintWriter out = response.getWriter();
        out.print("<HTML> <BODY> vous avez retiré "+solde+" </BODY> </HTML>");    }
```



```
@WebServlet(name = "ThreadSafeServlet", urlPatterns = {"/TestThread"})
public class ThreadSafeServlet extends HttpServlet {

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException
    {
        response.setContentType("text/html");
        int solde = Integer.parseInt(request.getParameter("retait"));

        try {
            Thread.sleep(10000);

        } catch (Exception e) {
        }

        PrintWriter out = response.getWriter();
        out.print("<HTML> <BODY> vous avez retiré "+solde+" </BODY> </HTML>");

    }

}
```

Session HTTP - suivi de l'utilisateur

UserServlet.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Login</title>
</head>
<body>
    <form action="UserServlet" method="post">
        <label for="username">Username:</label>
        <input type="text" id="username" name="username" required>
        <button type="submit">Login</button>
    </form>
</body>
</html>
```

selectBook.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Select Book</title>
</head>
<body>
    <h1>Select a Book</h1>
    <form action="BookServlet" method="post">
        <input type="hidden" name="action" value="selectBook">
        <label for="book">Choose a book:</label>
        <select id="book" name="book">
            <option value="Book A">Book A</option>
            <option value="Book B">Book B</option>
            <option value="Book C">Book C</option>
        </select>
        <button type="submit">Select</button>
    </form>
</body>
</html>
```

payBook.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Pay for Book</title>
</head>
<body>
    <h1>Pay for Book</h1>
    <form action="BookServlet" method="post">
        <input type="hidden" name="action" value="payBook">
        <button type="submit">Pay</button>
    </form>
</body>
</html>
```

SelecteBook.java

Durée de vie de la session utilisateur

La durée de vie d'une session utilisateur dans une application Java EE peut être configurée de différentes manières

Configurer le temps d'inactivité de la session dans le descripteur de déploiement (web.xml) :

Nous pouvons spécifier la durée de la session en minutes dans le fichier web.xml de notre application.

```
<web-app 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/web-app_4_0.xsd"
    version="4.0">

    <session-config>
        <!-- Durée de vie de la session en minutes -->
        <session-timeout>30</session-timeout>
    </session-config>

</web-app>
```

Durée de vie de la session utilisateur

La durée de vie d'une session utilisateur dans une application Java EE peut être configurée de différentes manières

Configurer le temps d'inactivité de la session programmatiquement dans le code :
Nous pouvons définir la durée de la session au niveau du code dans nos Servlets.

```
@WebServlet("/UserServlet")
public class UserServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        String username = request.getParameter("username");
        HttpSession session = request.getSession(true);

        synchronized (session) {
            session.setAttribute("username", username);
            // Durée de vie de la session en secondes (1800 secondes = 30 minutes)
            session.setMaxInactiveInterval(1800);
        }

        response.sendRedirect("selectBook.html");
    }
}
```

Durée de vie de la session utilisateur

La durée de vie d'une session utilisateur dans une application Java EE peut être configurée de différentes manières

Configurer la durée de vie de la session au niveau du conteneur :

Certains serveurs d'applications permettent de configurer la durée de vie de la session au niveau du conteneur.

Cette méthode dépend du serveur d'applications que vous utilisez (par exemple, Apache Tomcat, JBoss, WebSphere). Vous pouvez généralement configurer la durée de vie de la session via les fichiers de configuration spécifiques au serveur.

Pour Apache Tomcat, par exemple, vous pouvez configurer la durée de vie de la session dans le fichier context.xml.

<Context>

 <Manager sessionTimeout="30"/>

</Context>

Session après usage

```
session.removeAttribute("selectedBook");
session.removeAttribute("username");
```

Or Logout avec :

```
session.invalidate();
```

Coopération ou transfert de contrôle

En Java EE (Jakarta EE), le transfert de contrôle d'un servlet à un autre ou à une page HTML est un concept fondamental pour la gestion du flux de traitement des requêtes HTTP.

1. Forwarding (Redirection Interne) Le forwarding permet de transférer la requête d'un servlet à un autre servlet ou à une page JSP (ou HTML) sur le même serveur sans que le client (le navigateur) en soit informé. Cela se fait en utilisant **RequestDispatcher**.

```
// Transfert de contrôle à une autre page JSP
```

```
RequestDispatcher dispatcher = request.getRequestDispatcher("/result.jsp");
dispatcher.forward(request, response);
```

2. Redirection (Redirection Externe)

La redirection est utilisée pour informer le client (navigateur) qu'il doit effectuer une nouvelle requête vers une autre URL. Contrairement au forwarding, cela implique une nouvelle requête HTTP de la part du client. Cela se fait en utilisant **HttpServletResponse.sendRedirect**.

```
// Si la connexion est réussie, redirection vers la page de bienvenue
if (isValidUser)
    { response.sendRedirect("welcome.html"); }
else
    { response.sendRedirect("login.html"); }
```

Comparaison entre Forwarding et Redirection

- **Forwarding :**

- Effectué sur le serveur sans que le client soit informé.
- Utilise RequestDispatcher.forward.
- La barre d'adresse du navigateur ne change pas.
- Plus efficace pour les transferts internes car il n'implique pas une nouvelle requête HTTP.

- **Redirection :**

- Le client est informé et doit faire une nouvelle requête.
- Utilise HttpServletResponse.sendRedirect.
- La barre d'adresse du navigateur change pour refléter la nouvelle URL.
- Utile pour rediriger vers une autre application ou un autre domaine.

error.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Error</title>
</head>
<body>
    <h1>Invalid parameters</h1>
    <p>One or more parameters are not valid integers. Please check your input and try again.</p>
</body>
</html>
```

Somme.java

```
@WebServlet(  
    name = "Somme",  
    urlPatterns = {"Somme"})  
public class Somme extends HttpServlet {  
    private static final long serialVersionUID = 1L;  
  
    public Somme() {  
    }  
  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {  
        response.setContentType("text/html");  
        PrintWriter out = response.getWriter();  
        String nombre1 = request.getParameter("nombre1");  
        String nombre2 = request.getParameter("nombre2");  
        if (this.isInteger(nombre1) && this.isInteger(nombre2)) {  
            int somme = Integer.parseInt(nombre1) + Integer.parseInt(nombre2);  
            out.print("<html><body>La somme est " + somme + "</body></html>");  
        } else {  
            RequestDispatcher dispatcher = request.getRequestDispatcher("error.html");  
            dispatcher.forward(request, response);  
        }  
    }  
  
    private boolean isInteger(String str) {  
        if (str == null) {  
            return false;  
        } else {  
            try {  
                Integer.parseInt(str);  
            } catch (NumberFormatException e) {  
                return false;  
            }  
            return true;  
        }  
    }  
}
```

Utiliser un Servlet

```
RequestDispatcher dispatcher = request.getRequestDispatcher("/ErrorServlet");
dispatcher.forward(request, response);
```

Portée des variables en mémoire (Scope)

En Java EE (Jakarta EE), la portée des variables en mémoire est essentielle pour comprendre où et comment les données sont accessibles au sein d'une application web.

Portée de la Requête (Request Scope)

Description : Les objets sont disponibles uniquement pendant la durée de vie d'une requête HTTP. Une fois la requête terminée, les objets sont supprimés.

Utilisation : Pour stocker des données temporaires qui ne sont nécessaires que pour une seule requête, comme les paramètres de formulaire ou les résultats de traitement.

```
request.setAttribute("user", user);
```

Portée de la Session (Session Scope)

- **Description** : Les objets sont disponibles pendant la durée de vie de la session utilisateur. La session commence lorsqu'un utilisateur accède à l'application et se termine lorsque l'utilisateur se déconnecte ou que la session expire.
- **Utilisation** : Pour stocker des informations spécifiques à l'utilisateur qui doivent persister entre les différentes requêtes, comme les informations de connexion ou les préférences utilisateur.

```
HttpSession session = request.getSession();
session.setAttribute("user", user);
```

Portée de l'Application (Application Scope)

- **Description** : Les objets sont disponibles pendant la durée de vie de l'application web. Ils sont partagés entre tous les utilisateurs et toutes les sessions.
- **Utilisation** : Pour stocker des données globales accessibles à tous les utilisateurs, comme les configurations de l'application ou les caches partagés.

```
ServletContext context = getServletContext();
context.setAttribute("config", config);
```

Portée de la Page (Page Scope)

- **Description** : Les objets sont disponibles uniquement dans la page JSP où ils sont définis. Cette portée est limitée à l'exécution de la page JSP.
- **Utilisation** : Pour stocker des données temporaires spécifiques à une page JSP particulière.

```
<jsp:useBean id="user" scope="page" class="com.example.User"/>
```

La compréhension des différentes portées de variables en mémoire en Java EE est cruciale pour le développement efficace d'applications web. Chaque portée a des utilisations spécifiques qui permettent de gérer correctement les données au sein de l'application, assurant ainsi une bonne organisation et une performance optimale.

request

>

session

>

application

très courte durée
une seule
requête donc un
seul utilisateur

longue durée
1 seul
utilisateur

très longue
durée
tous les
utilisateurs

Somme.html

```
<!DOCTYPE html>
<html lang="en">
<head>
    <meta charset="UTF-8">
    <title>Calculate Sum</title>
</head>
<body>
    <h1>Calculate Sum of Two Numbers</h1>
    <form action="Somme" method="GET">
        <div>
            <label for="nombre1">Number 1:</label>
            <input type="text" id="nombre1" name="nombre1" required>
        </div>
        <div>
            <label for="nombre2">Number 2:</label>
            <input type="text" id="nombre2" name="nombre2" required>
        </div>
        <div>
            <input type="checkbox" id="format" name="format" value="pdf">
            <label for="format">Generate PDF</label>
        </div>
        <button type="submit">Calculate Sum</button>
    </form>
</body>
</html>
```

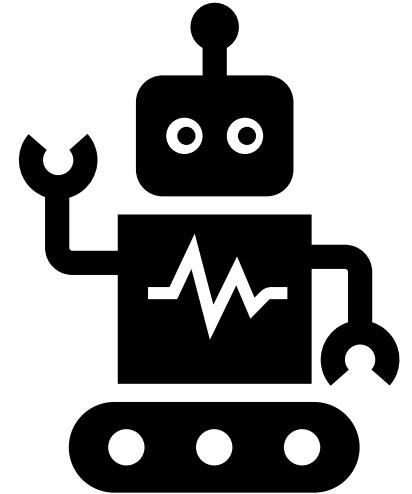
Somme.java

```
@WebServlet(  
    name = "Somme",  
    urlPatterns = {"Somme"})  
public class Somme extends HttpServlet {  
    private static final long serialVersionUID = 1L;  
  
    public Somme() {}  
  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException,  
    IOException {  
        response.setContentType("text/html");  
        String nombre1 = request.getParameter("nombre1");  
        String nombre2 = request.getParameter("nombre2");  
        String format = request.getParameter("format");  
        if (this.isInteger(nombre1) && this.isInteger(nombre2)) {  
            int somme = Integer.parseInt(nombre1) + Integer.parseInt(nombre2);  
            PrintWriter out;  
            if (format != null && !format.isEmpty()) {  
                if ("pdf".equalsIgnoreCase(format)) {  
                    request.setAttribute("somme", somme);  
                    RequestDispatcher dispatcher = request.getRequestDispatcher("/PdfServlet");  
                    dispatcher.forward(request, response);  
                } else {  
                    response.getWriter().println("Somme : " + somme);  
                }  
            }  
        }  
    }  
}
```

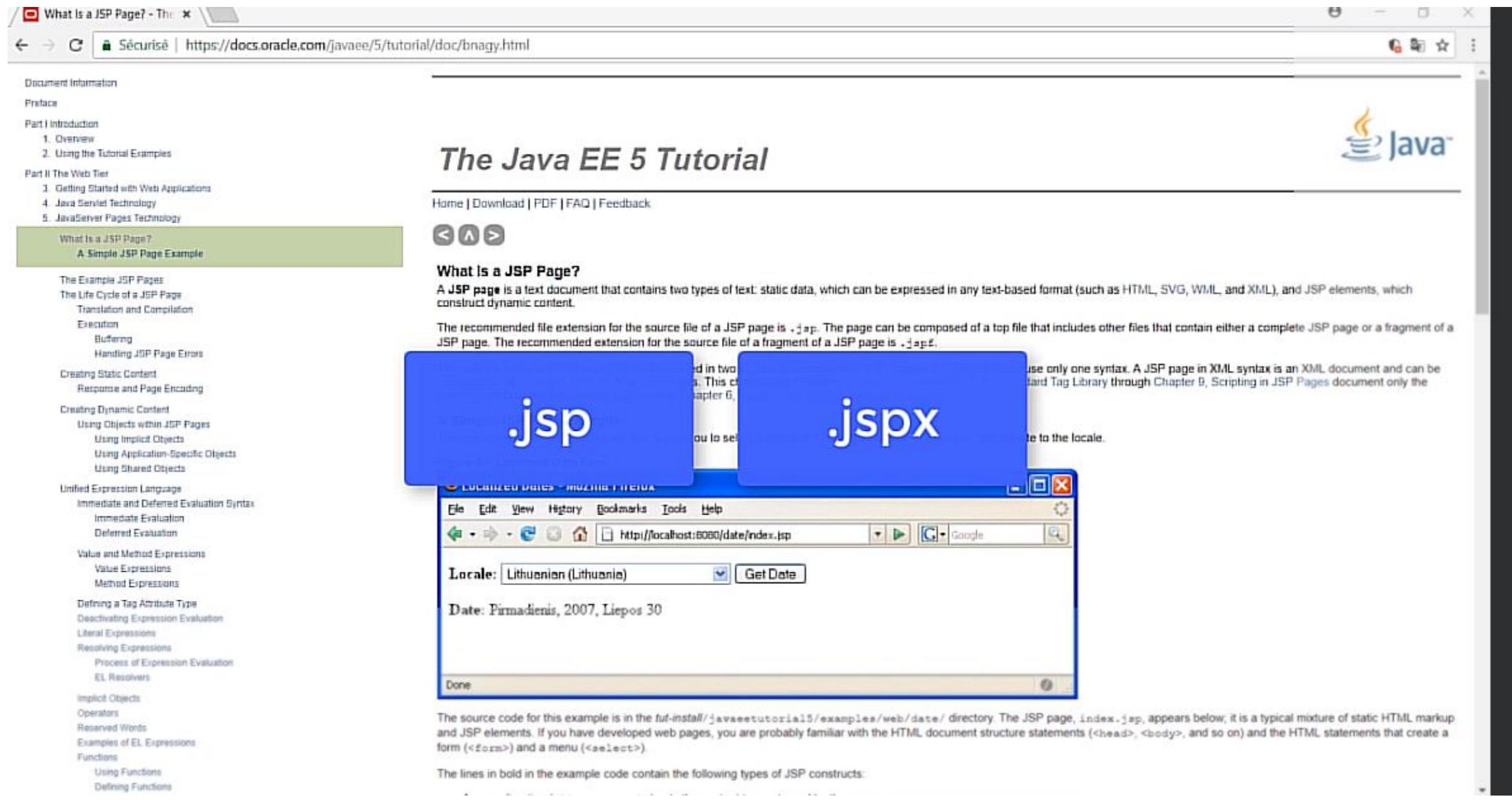
PdfServlet.html

```
@WebServlet(  
    name = "PdfServlet",  
    urlPatterns = {"/PdfServlet"}  
)  
public class PdfServlet extends HttpServlet {  
    public PdfServlet() {  
    }  
  
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws  
ServletException, IOException {  
        response.setContentType("application/pdf");  
        response.setHeader("Content-Disposition", "inline; filename=somme.pdf");  
        Integer somme = (Integer)request.getAttribute("somme");  
        String message;  
        if (somme == null) {  
            message = "hello world";  
        } else {  
            message = "La somme est : " + String.valueOf(somme);  
        }  
  
        try {  
            Document doc = new Document();  
        }  
    }  
}
```

Les pages JSP - Introduction



JSP - Qu'est ce que c'est ?



Les JSP (JavaServer Pages) sont une technologie Java utilisée pour développer des pages web dynamiques basées sur HTML, XML ou d'autres types de documents. Elles permettent de mélanger le code Java avec le balisage HTML pour générer du contenu dynamique.

1. Qu'est-ce qu'une JSP ?

- Une JSP est une page web qui contient du code Java entre des balises HTML. Elle est compilée en un servlet par le serveur d'applications lors de la première requête et le code Java est exécuté côté serveur pour générer du HTML dynamique.

2. Structure de base d'une JSP

- Une JSP se compose de trois parties principales :
 - **Directives** : Instructions pour le conteneur JSP (ex. : page, include).
 - **Scriptlets** : Code Java inclus dans la page.
 - **Expressions** : Code Java évalué et inséré dans le flux de réponse.
 - **Déclarations** : Code Java déclaré au niveau de la classe servlet générée.

Exemples de base

Directives JSP

Les directives fournissent des informations au conteneur JSP. Elles commencent par `<%@` et se terminent par `%>`.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<%@ page import="java.util.*" %>
```

Scriptlets

Les scriptlets contiennent du code Java qui est exécuté lorsque la page JSP est demandée. Ils sont inclus entre `<%` et `%>`.

```
<%
  Date date = new Date();
%>
<p>Date et heure actuelles : <%= date %></p>
```

Exemples de base

Expressions

Les expressions permettent d'insérer des valeurs dynamiques directement dans le contenu HTML. Elles commencent par `<%=` et se terminent par `%>`.

```
<p>Date et heure actuelles : <%= new Date() %></p>
```

Déclarations

Les déclarations permettent de déclarer des variables et des méthodes. Elles commencent par `<%!` et se terminent par `%>`.

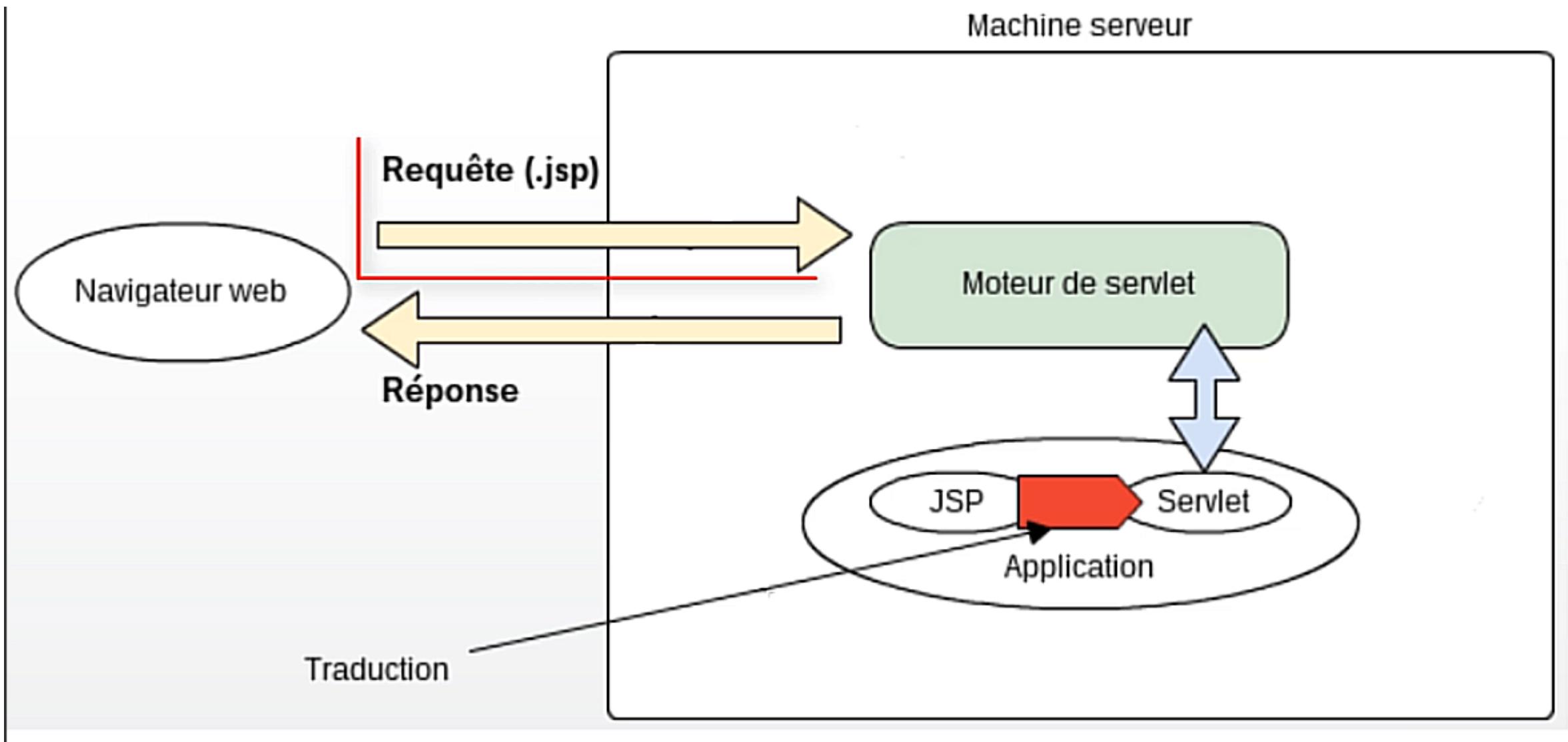
```
<%
private String getGreeting() {
    return "Bonjour!";
}
%>
<p><%= getGreeting() %></p>
```

hello.jsp

```
<%--  
 Document      : hello  
 Created on   : 28 mai 2024, 17 h 37 min 51 s  
 Author       : Lenovo  
--%>
```

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>  
<%@ page import="java.util.Date" %>  
<!DOCTYPE html>  
<html>  
<head>  
    <meta charset="UTF-8">  
    <title>JSP Example</title>  
</head>  
<body>  
    <h1>Welcome to JSP</h1>  
    <p>Current Date and Time: <%= new Date() %></p>  
</body>  
</html>
```

JSP - Comment ça marche ?



Cycle de vie d'une JSP

Traduction : La page JSP est traduite en un servlet Java par le conteneur JSP.

Compilation : Le servlet généré est compilé en bytecode.

Chargement : Le bytecode du servlet est chargé en mémoire.

Initialisation : La méthode `jsplInit()` est appelée pour initialiser le servlet.

Exécution : La méthode `_jspService()` est appelée pour traiter les requêtes.

Destruction : La méthode `jspDestroy()` est appelée avant que le servlet ne soit détruit.

Avantages des JSP

Séparation de la logique et de la présentation : Les développeurs peuvent séparer le code Java de la logique métier et le code HTML de la présentation.

Facilité d'utilisation : Les JSP sont faciles à écrire et à maintenir.

Compatibilité avec les servlets : Les JSP sont compilées en servlets et peuvent utiliser toutes les fonctionnalités des servlets.

Variables implicites

Les variables implicites dans une page JSP sont des objets pré-déclarés et pré-initialisés par le conteneur JSP, ce qui permet aux développeurs d'accéder facilement à des informations couramment utilisées sans avoir besoin de les déclarer explicitement.

Variables Implicites dans JSP

request

Type : HttpServletRequest

Description : Représente la requête HTTP envoyée par le client au serveur. Contient des informations telles que les paramètres de la requête, les en-têtes, et les attributs.

```
String paramValue = request.getParameter("paramName");
```

response

Type : HttpServletResponse

Description : Représente la réponse HTTP que le serveur envoie au client. Utilisée pour configurer la réponse, envoyer des en-têtes HTTP, rediriger le client, etc.

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

Les variables implicites dans une page JSP sont des objets pré-déclarés et pré-initialisés par le conteneur JSP, ce qui permet aux développeurs d'accéder facilement à des informations couramment utilisées sans avoir besoin de les déclarer explicitement.

Variables Implicites dans JSP

session

Type : HttpSession

Description : Représente la session utilisateur associée à la requête. Permet de stocker des informations spécifiques à l'utilisateur sur plusieurs requêtes.

```
String userName = (String) session.getAttribute("userName");
```

out

Type : JspWriter

Description : Utilisée pour envoyer du contenu à la réponse HTTP. Similaire à PrintWriter, mais avec des fonctionnalités supplémentaires pour la gestion des tampons de sortie.

```
out.println("Hello, World!");
```

Les variables implicites dans une page JSP sont des objets pré-déclarés et pré-initialisés par le conteneur JSP, ce qui permet aux développeurs d'accéder facilement à des informations couramment utilisées sans avoir besoin de les déclarer explicitement.

Variables Implicites dans JSP

pageContext

Type : PageContext

Description : Fournit un accès à tous les autres objets implicites ainsi qu'à des informations supplémentaires sur la page JSP.

```
pageContext.include("anotherPage.jsp");
```

application

Type : ServletContext

Description : Représente le contexte de l'application web. Utilisée pour accéder aux ressources partagées entre toutes les servlets et JSP de l'application.

```
String appName = application.getInitParameter("appName");
```

Les variables implicites dans une page JSP sont des objets pré-déclarés et pré-initialisés par le conteneur JSP, ce qui permet aux développeurs d'accéder facilement à des informations couramment utilisées sans avoir besoin de les déclarer explicitement.

Variables Implicites dans JSP

config

Type : ServletConfig

Description : Contient les paramètres de configuration pour la JSP. Utilisée pour obtenir des informations de configuration spécifiques à la servlet/JSP.

```
String jsplnInitParam = config.getInitParameter("initParamName");
```

page

Type : Object

Description : Représente la page JSP elle-même. Pratiquement identique à this dans un servlet.

// Utilisé rarement, principalement pour des méthodes utilitaires

example.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ page import="java.util.Date" %>
<!DOCTYPE html>
<html>
<head>
    <title>Example JSP Page</title>
</head>
<body>
    <h1>Example JSP Page</h1>

    <h2>Using implicit variables</h2>
    <p>Request Parameter 'name': <%= request.getParameter("name") %></p>

    <%
        // Set a session attribute
        session.setAttribute("lastAccessTime", new Date());
    %>
    <p>Last Access Time (from session): <%= session.getAttribute("lastAccessTime") %></p>

    <%
        // Using the application context to get an initialization parameter
        String appName = application.getInitParameter("appName");
    %>
    <p>Application Name: <%= appName %></p>

    <p>Current Date and Time: <%= new Date() %></p>

    <%
        // Including another JSP
        pageContext.include("footer.jsp");
    %>
</body>
</html>
```

footer.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<hr>
<p>This is the footer included using pageContext.include().</p>
```

Importer des classes dans une JSP

```
<%@ page import="java.util.Date, com.mycompany.firstappcore.Personne" %>
```

```
<%@ page import="com.example.model.User, java.util.Date" %>
```

JSP EL (Expression Language)

JSP EL (Expression Language) est un langage simple utilisé dans les pages JSP pour accéder facilement aux objets et manipuler les données sans avoir à écrire du code Java complexe. Il permet d'accéder aux propriétés des objets, d'appeler des méthodes, et d'évaluer des expressions de manière concise et intuitive.

Principes de Base

1. Syntaxe :

- Les expressions EL sont délimitées par \${}.
- Par exemple : \${user.name} accède à la propriété name de l'objet user.

2. Objets Implicites :

- EL fournit plusieurs objets隐式的 pour accéder aux données stockées dans différents portées :
 - pageScope : Accède aux attributs dans la portée de la page.
 - requestScope : Accède aux attributs dans la portée de la requête.
 - sessionScope : Accède aux attributs dans la portée de la session.
 - applicationScope : Accède aux attributs dans la portée de l'application.

JSP EL (Expression Language)

BookServlet.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Payment Confirmation</title>
</head>
<body>
    <h1>Payment Confirmation</h1>
    <p>User ${sessionScope.username} has successfully paid for the book:
    ${sessionScope.selectedBook}</p>
</body>
</html>
```

JSP EL (Expression Language)

payBook.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Pay for Book</title>
</head>
<body>
    <h1>Pay for Book</h1>
    <form action="BookServlet.jsp" method="post">
        <input type="hidden" name="action" value="payBook">
        <button type="submit">Pay</button>
    </form>
</body>
</html>
```

Exemple avec un Objet Java

```
com.mycompany.firstappcore;  
public class Livre {  
    String numLivre;  
  
    public String getNumLivre() {  
        return numLivre;  
    }  
  
    public void setNumLivre(String numLivre) {  
        this.numLivre = numLivre;  
    }  
}
```

Exemple avec un Objet Java

```
@WebServlet({"/BookServlet"})
public class BookServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    public BookServlet() {
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        HttpSession session = request.getSession(false);
        if (session != null && session.getAttribute("username") != null) {
            String action = request.getParameter("action");
            synchronized(session) {
                String selectedBook;
                if ("selectBook".equals(action)) {
                    selectedBook = request.getParameter("book");
                    Livre livre = new Livre();
                    livre.setNumLivre(selectedBook);
                    session.setAttribute("Livre", livre);
                }
            }
        }
    }
}
```

Exemple avec un Objet Java

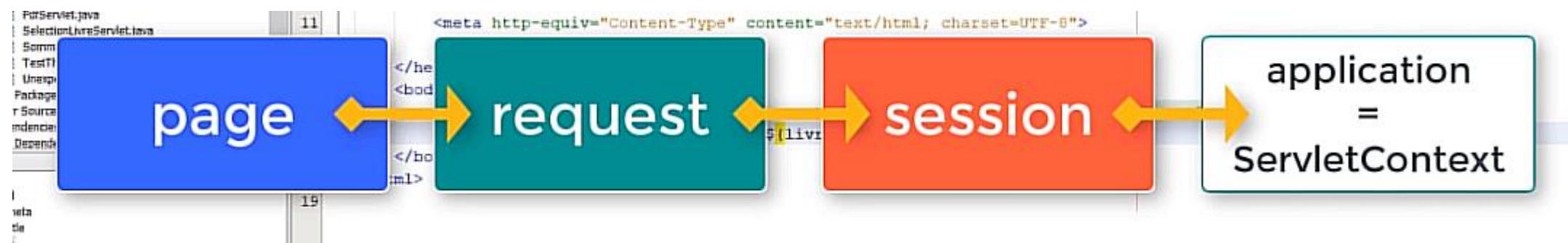
BookServlet.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Payment Confirmation</title>
</head>
<body>
    <h1>Payment Confirmation</h1>
    <p>User ${sessionScope.username} has successfully paid for the book:
    ${sessionScope.Livre.numLivre}</p>
</body>
</html>
```

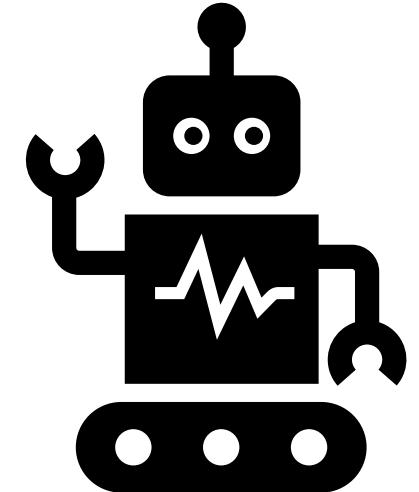
Exemple avec un Objet Java

BookServlet.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Payment Confirmation</title>
</head>
<body>
    <h1>Payment Confirmation</h1>
    <p>User ${sessionScope.username} has successfully paid for the book: ${Livre.numLivre}</p>
</body>
</html>
```



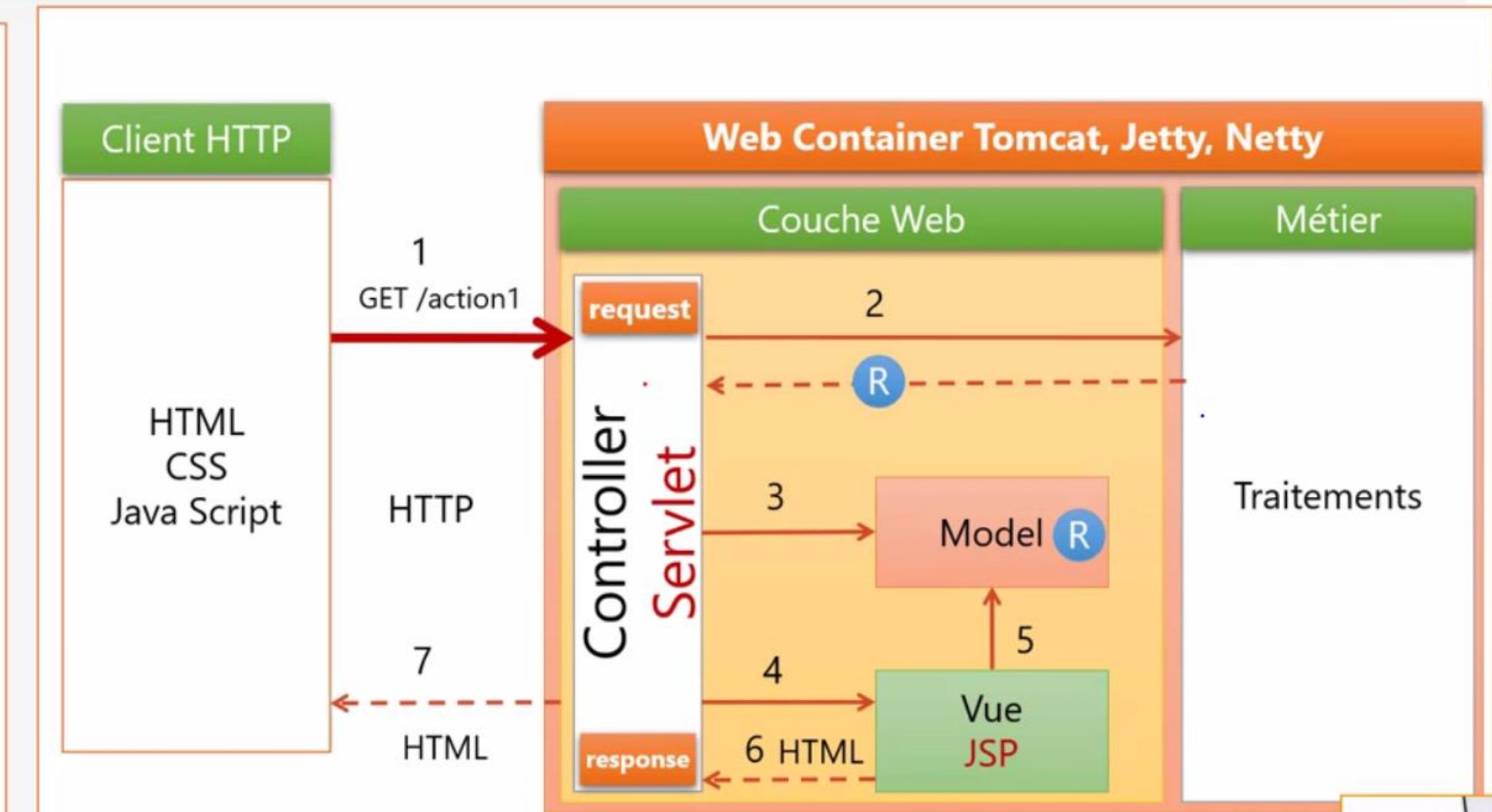
Architectures standards et technologies associées - Introduction à Java EE



Le Design Pattern MVC pour les applications Web Java

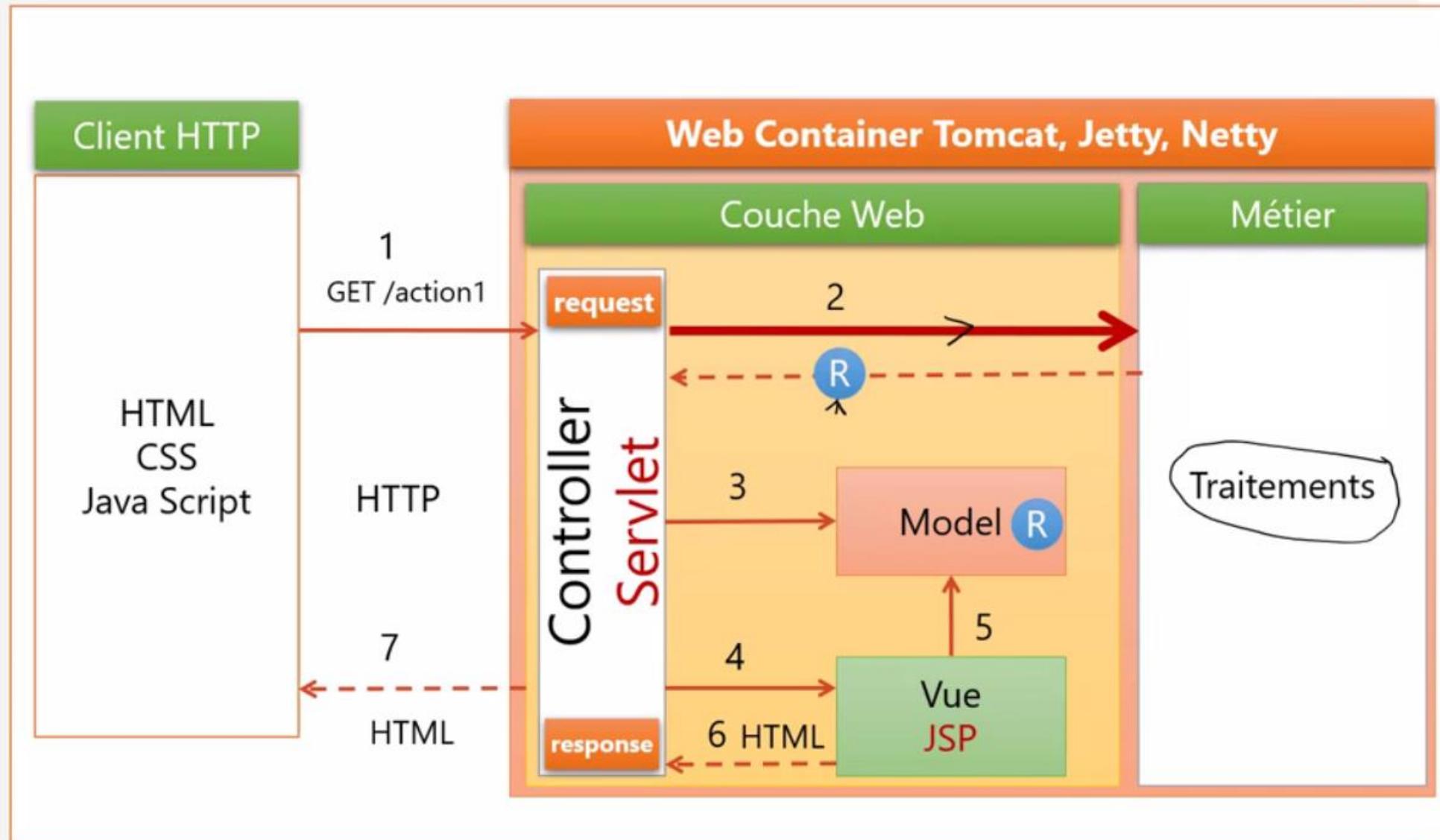
Architecture Web JEE : HTTP, Servlet, JSP, MVC

1 – Le client envoie une requête HTTP de type GET ou POST vers le contrôleur représenté par un composant Web JEE : **SERVLET**. Pour lire les données de la requête HTTP le contrôleur utilise l'objet **request** de type **HttpServletRequest**



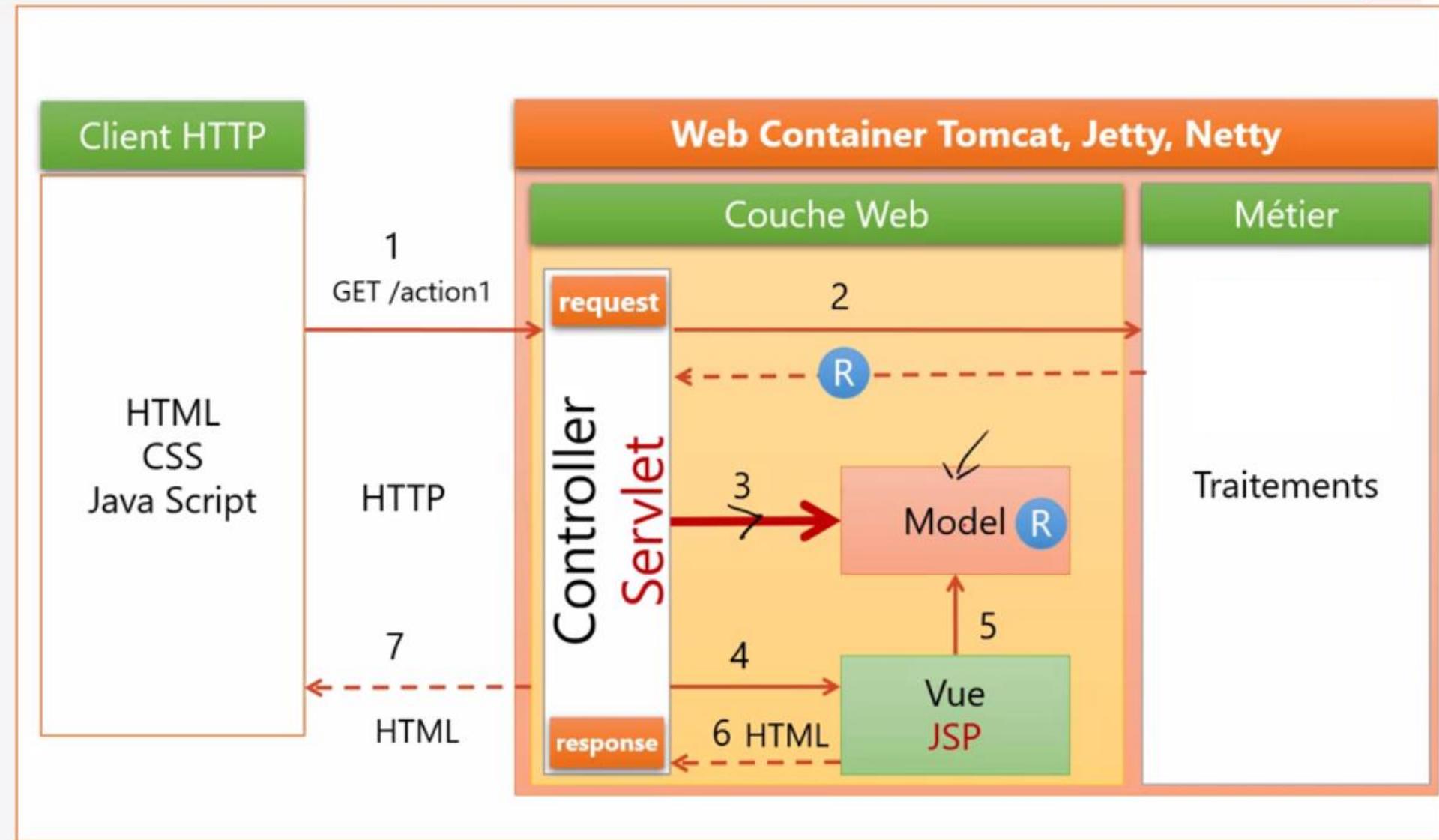
Architecture Web JEE

2 – Le contrôleur fait appel à la couche métier pour effectuer les traitements et récupère les résultats R



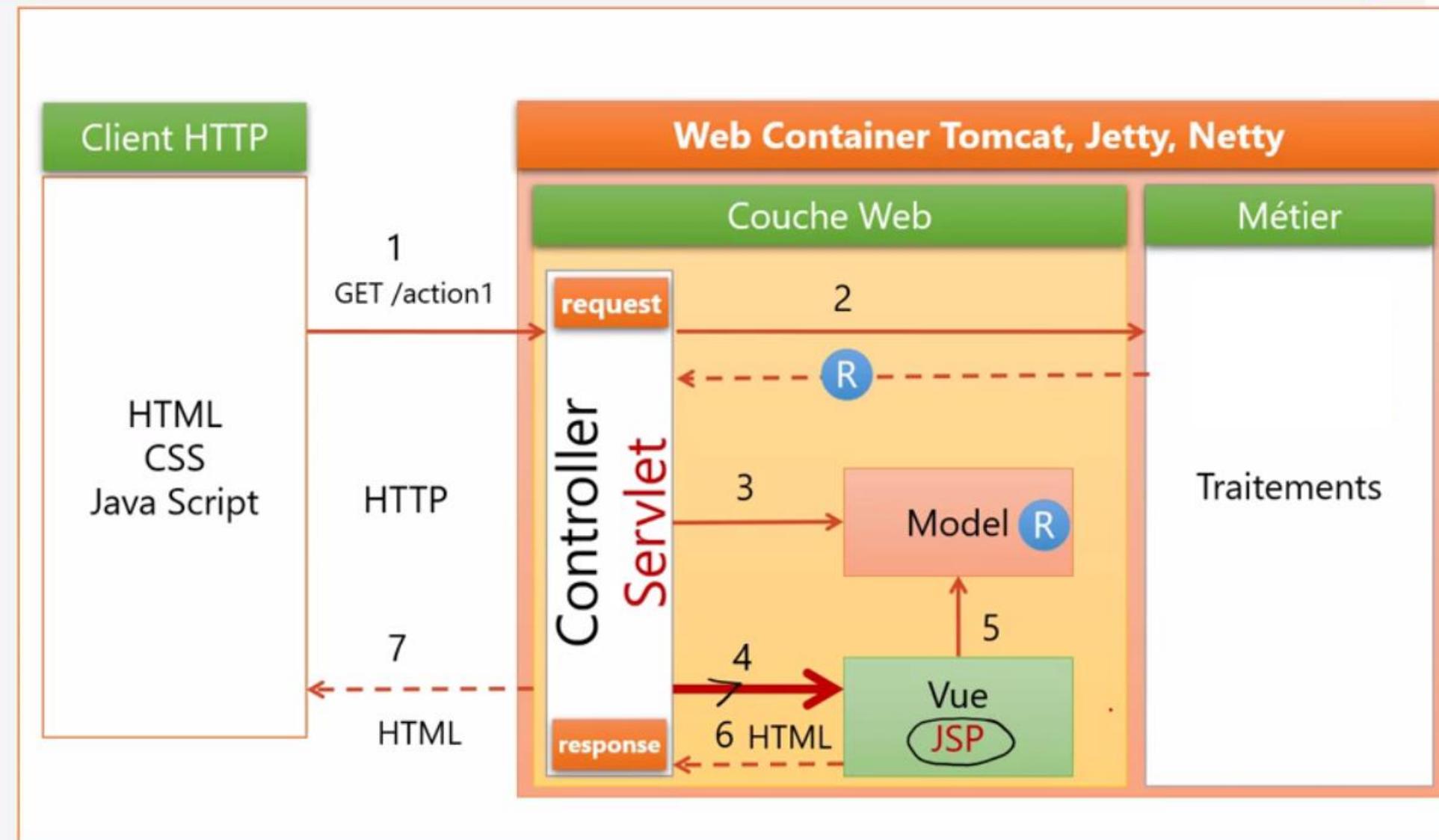
Architecture Web JEE

3 – Le contrôleur Stocke le résultat R dans le modèle M. Le modèle est généralement une objet qui permet de stocker toutes les données qui seront affichées dans la vue. Généralement, le contrôleur stocke le modèle dans l'objet request ou session.



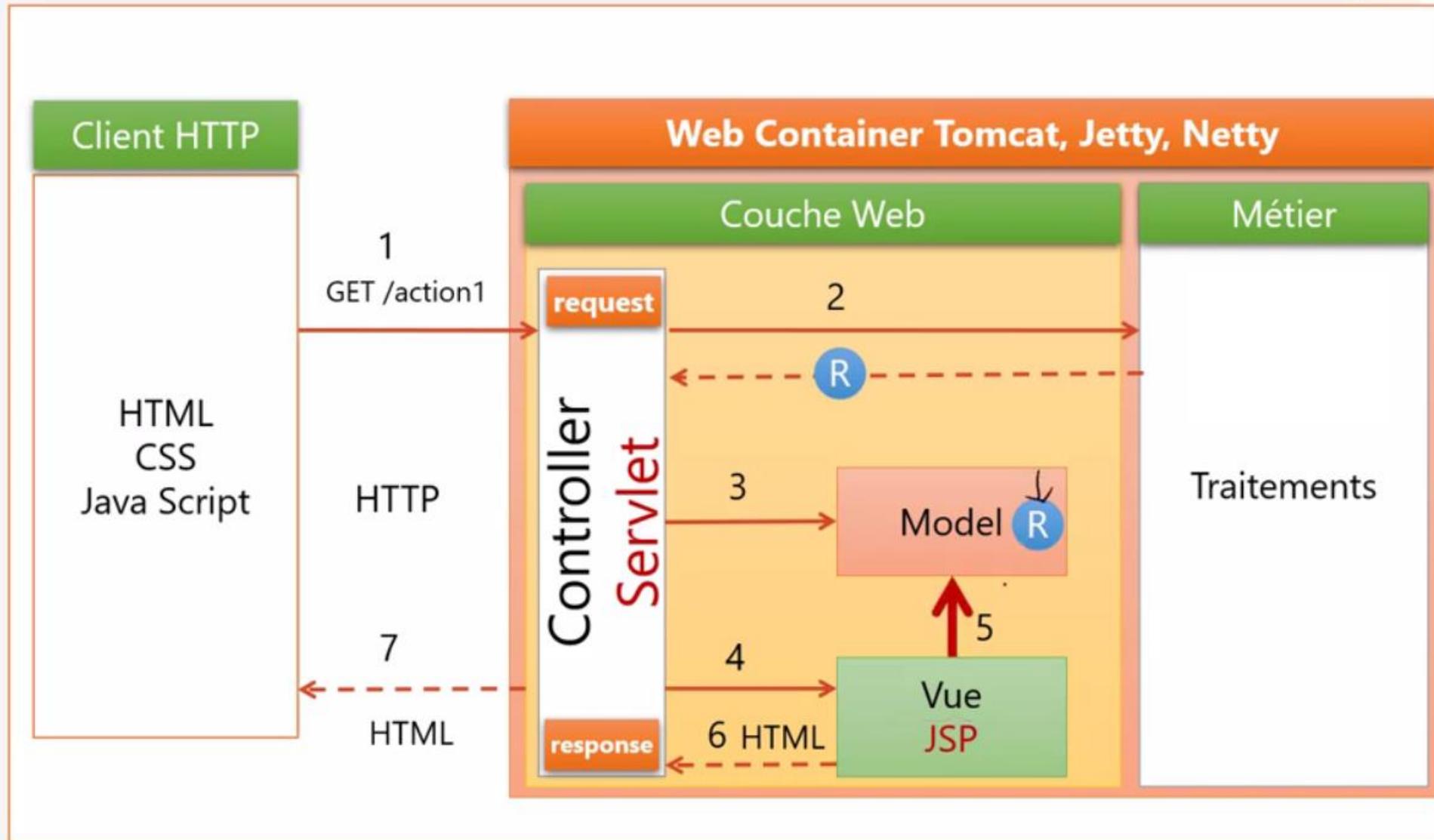
Architecture Web JEE

4 – Le contrôleur fait appel à la vue JSP (Java Server Pages) en lui transmettant les mêmes objets request et response. Cette opération s'appelle Forwarding.



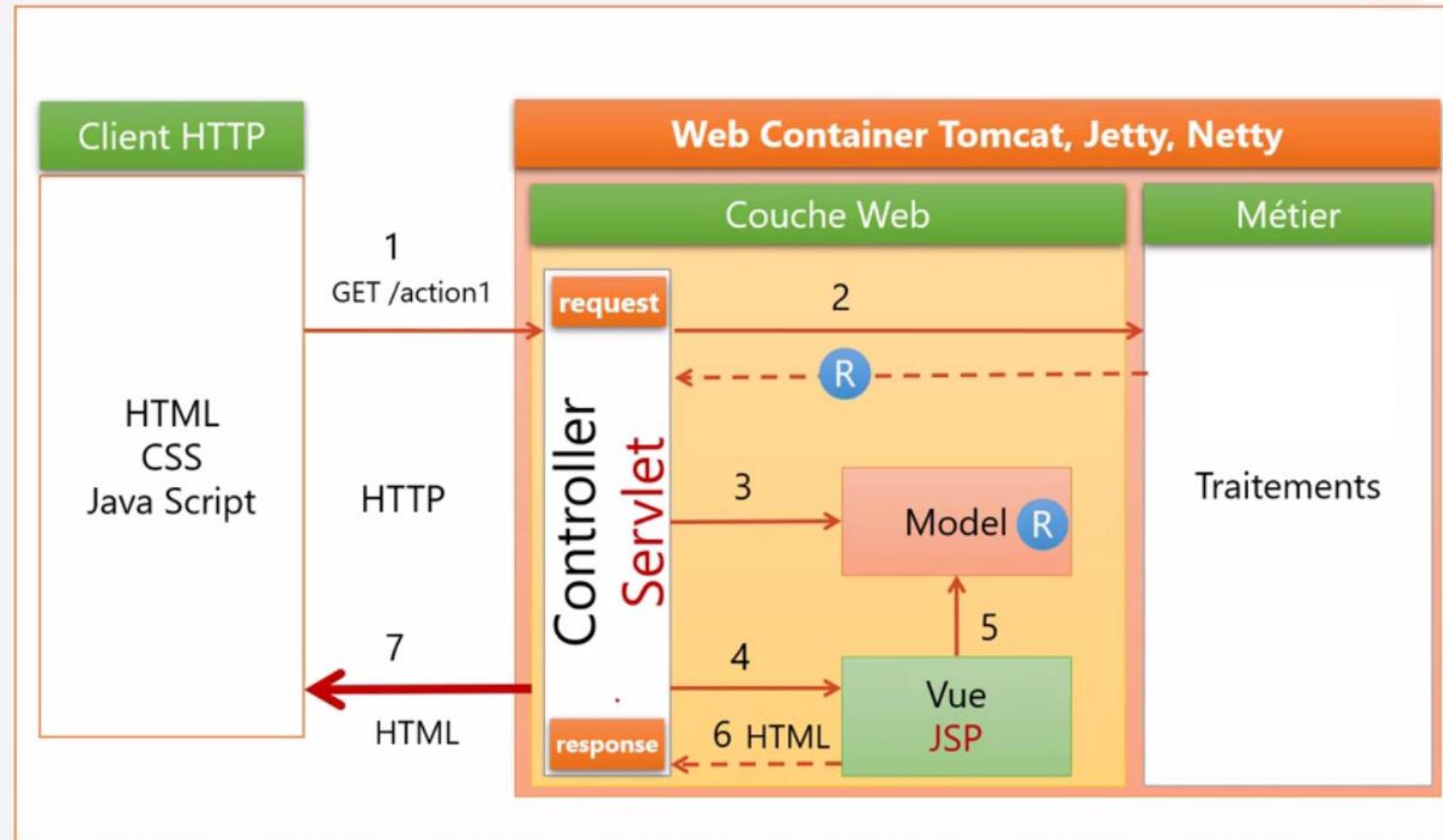
Architecture Web JEE

5 – La vue JSP récupère le résultat à partir du modèle. La vue retrouve le modèle dans l'objet request ou session.



Architecture Web JEE

7 – La page HTML générée est envoyée dans le corps de la réponse HTTP.




```
if (format == null || format.isEmpty()) {
    // Afficher un message de salutation si le format n'est pas envoyé
    //PrintWriter out = response.getWriter();
    //out.print("<html><body>La somme est " + somme + "</body></html>");

    RequestDispatcher dispatcher = request.getRequestDispatcher("/AfficherSomme.jsp");
    dispatcher.forward(request, response);
}

} else if ("pdf".equalsIgnoreCase(format)) {
    // Définir l'attribut "somme" dans la requête
    request.setAttribute("somme", somme);
    RequestDispatcher dispatcher = request.getRequestDispatcher("/PdfServlet");
    dispatcher.forward(request, response);
} else {
    // Afficher la somme si le format est envoyé mais n'est pas "pdf"
    PrintWriter out = response.getWriter();
    out.print("<html><body>La somme est " + somme + "</body></html>");

}

}

private boolean isInteger(String str) {
    if (str == null) {
        return false;
    }
    try {
```

AfficherSomme.jsp

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
    <title>Sum Result</title>
</head>
<body>
    <h1>Sum Result</h1>
    <p>La somme est : ${requestScope.somme}</p>
</body>
</html>
```

La JSTL (Java Standard Tag Library)

La Java Standard Tag Library (JSTL) est une collection de balises personnalisées qui enrichissent les capacités des JavaServer Pages (JSP).

JSTL offre un ensemble de fonctionnalités standardisées pour gérer les opérations courantes dans les applications web, réduisant ainsi la nécessité d'écrire du code Java directement dans les pages JSP. Cela améliore la lisibilité et la maintenabilité du code JSP

Pourquoi Utiliser JSTL ?

Séparation des Préoccupations :

Sépare la logique de présentation de la logique métier en utilisant des balises plutôt que du code Java dans les JSP.

Lisibilité et Maintenabilité :

Facilite la lecture et la maintenance des pages JSP en encapsulant la logique dans des balises compréhensibles.

Standardisation :

Utilise un ensemble standardisé de balises, ce qui permet une meilleure portabilité et réutilisabilité du code.

Les Bibliothèques de Balises JSTL

JSTL est divisée en plusieurs bibliothèques de balises spécialisées :

- 1.Core Tags (c:): Balises pour les structures de contrôle et les opérations de base.
- 2.Formatting Tags (fmt:): Balises pour l'internationalisation et la formatage de données.
- 3.SQL Tags (sql:): Balises pour les interactions de base avec les bases de données.
- 4.XML Tags (x:): Balises pour la manipulation et la transformation XML.
- 5.Functions Tags (fn:): Fonctions utilitaires pour les opérations sur les chaînes.

Core Tags (c:)

Les balises **Core** sont utilisées pour les opérations courantes telles que les boucles, les conditions, l'importation de ressources, et la gestion des variables.

Affiche la valeur d'une expression, avec échappement des caractères spéciaux XML.
`<c:out value="${user.name}" />`

Définir une variable ou un attribut dans une portée spécifiée (page, request, session, application).
`<c:set var="username" value="John Doe" scope="session" />`

Évaluer une condition et inclure du contenu si la condition est vraie.
`<c:if test="${not empty user}">
 <p>Welcome, ${user.name}!</p>
</c:if>`

Core Tags (c:)

Équivalent d'une structure switch/case

```
<c:choose>
  <c:when test="${user.role == 'admin'}">
    <p>Welcome, Admin!</p>
  </c:when>
  <c:otherwise>
    <p>Welcome, User!</p>
  </c:otherwise>
</c:choose>
```

Parcourir une collection ou un tableau

```
<c:forEach var="book" items="${bookList}">
  <p>${book.title}</p>
</c:forEach>
```

Formatting Tags (fmt:)

Les balises de formatage sont utilisées pour l'internationalisation (i18n) et la localisation des messages, des nombres, des dates, etc.

Récupère un message localisé à partir d'un fichier de ressources

```
<fmt:setBundle basename="messages" />  
<fmt:message key="welcome.message" />
```

Formate un nombre selon le format spécifié

```
<fmt:formatNumber value="${price}" type="currency" />
```

Formate une date selon le format spécifié

```
<fmt:formatDate value="${now}" pattern="yyyy-MM-dd" />
```

Analyse une chaîne en un nombre selon le format spécifié

```
<fmt:parseNumber var="num" value="${param.number}" />
```

Analyse une chaîne en une date selon le format spécifié

```
<fmt:parseDate var="date" value="${param.date}" pattern="yyyy-MM-dd" />
```

SQL Tags (sql:)

Les balises SQL permettent de réaliser des opérations de base sur une base de données directement depuis les JSP. Cependant, leur utilisation est déconseillée pour des raisons de sécurité et de séparation des préoccupations. Elles peuvent être utiles pour des prototypes rapides.

Définir une source de données pour les opérations SQL

```
<sql:setDataSource var="ds" driver="com.mysql.jdbc.Driver"  
url="jdbc:mysql://localhost:3306/mydb" user="root" password="password" />
```

Exécuter une requête SQL et stocker le résultat dans une variable

```
<sql:query dataSource="${ds}" var="result">  
    SELECT * FROM users  
</sql:query>
```

SQL Tags (sql:)

Exécuter une mise à jour SQL (INSERT, UPDATE, DELETE)

```
<sql:update dataSource="${ds}">
    INSERT INTO users (name, email) VALUES ('John Doe', 'john.doe@example.com')
</sql:update>
```

XML Tags (x:)

Les balises XML permettent de manipuler et de transformer des documents XML.

Analyse un document XML à partir d'une chaîne ou d'une URL et le stocke dans une variable

```
<x:parse var="doc" xml="${xmlString}" />
```

Évalue une expression XPath et affiche le résultat

```
<x:out select="$doc/root/element" />
```

Parcourt les nœuds d'un document XML

```
<x:forEach select="$doc/root/elements/element" var="element">
  <p><x:out select="$element/@attribute" /></p>
</x:forEach>
```

Functions Tags (fn:)

Les balises de fonctions fournissent des fonctions utilitaires pour les opérations sur les chaînes et autres

Vérifie si une chaîne contient une sous-chaîne spécifiée

`${fn:contains(user.email, 'example.com')}`

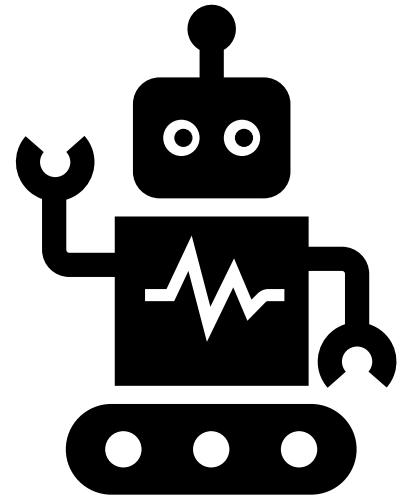
Convertit une chaîne en majuscules

`${fn:toUpperCase(user.name)}`

Retourne la longueur d'une chaîne, d'un tableau ou d'une collection

`${fn:length(bookList)}`

EJB, CDI et JPA



Jakarta Enterprise Beans (EJB) et Jakarta Persistence API (JPA) sont deux technologies clés de Jakarta EE (anciennement Java EE) utilisées pour développer des applications d'entreprise robustes, scalables et maintenables.

Elles permettent de séparer la logique métier des opérations de persistance et de transaction, facilitant ainsi le développement et la gestion des applications

Fondamentaux des EJB (Enterprise JavaBeans)

Web MVC 2

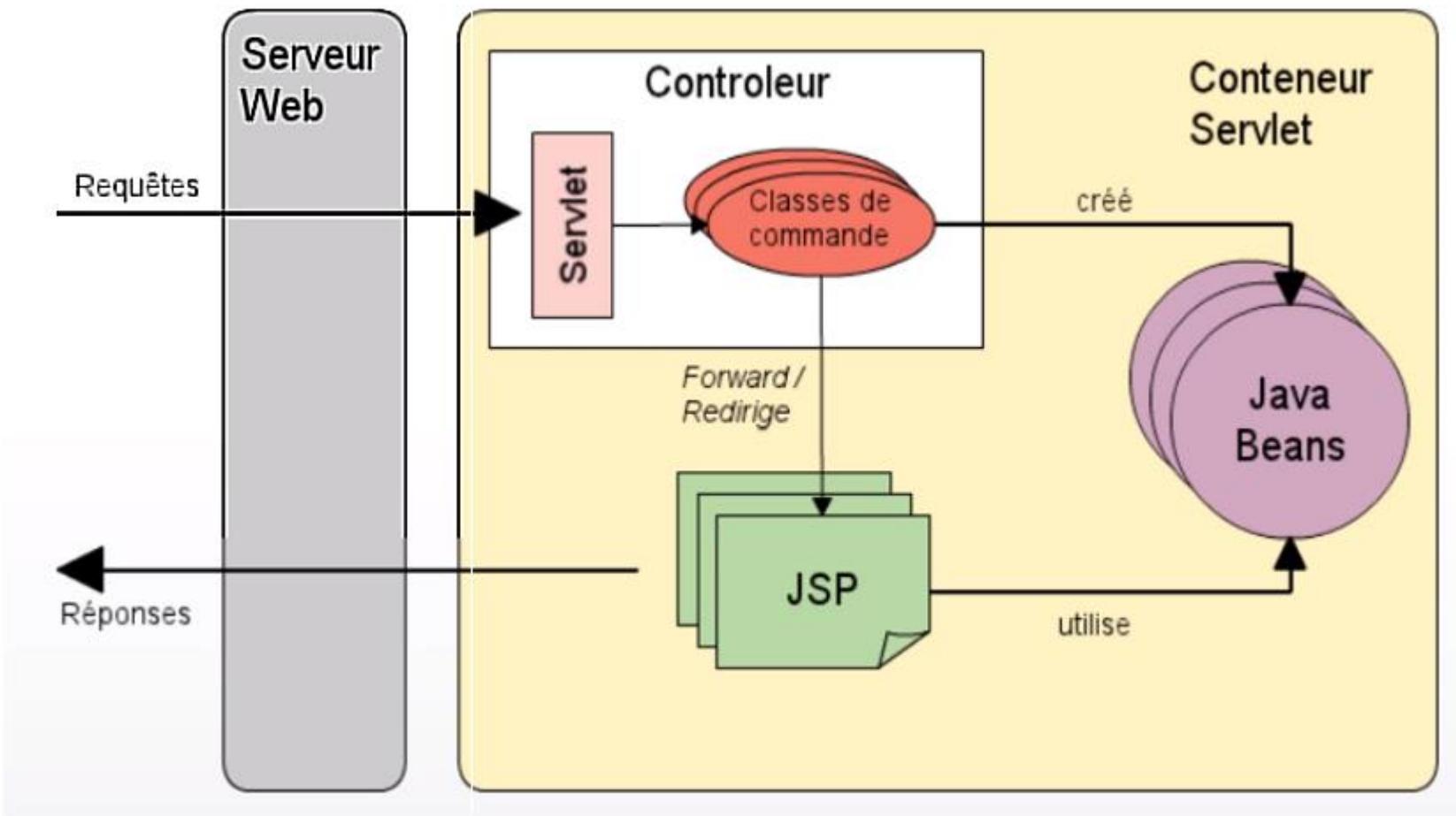
Le Web MVC (Model-View-Controller) en Java EE est un modèle d'architecture utilisé pour structurer les applications web de manière à séparer les différentes préoccupations :

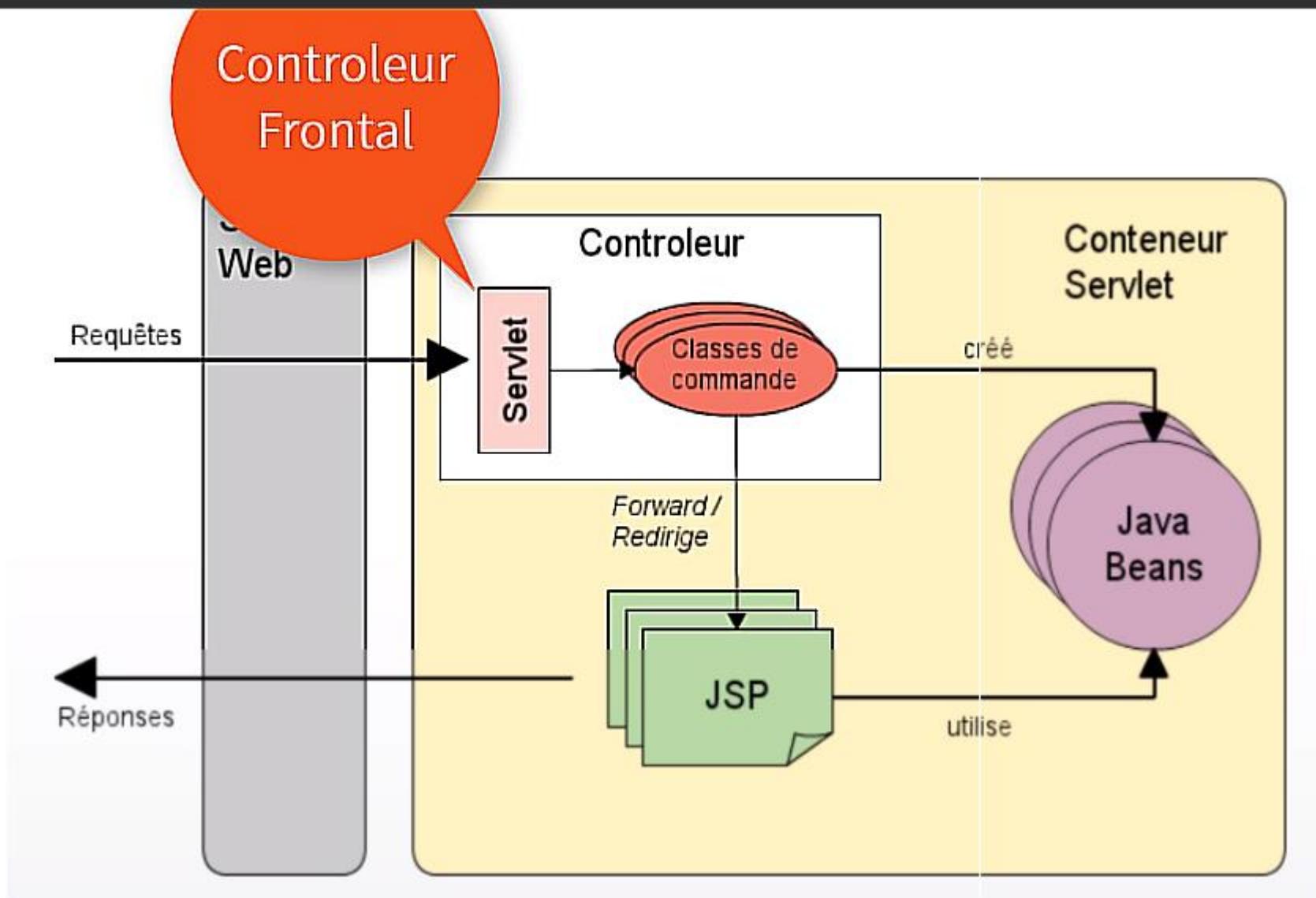
1. Modèle (Model) : Représente les données de l'application et la logique métier. En Java EE, cela peut inclure des entités JPA, des EJB, ou d'autres composants Java qui manipulent les données.

2. Vue (View) : Représente la couche de présentation, c'est-à-dire ce que l'utilisateur voit et avec quoi il interagit. Les technologies courantes pour cette couche incluent JSP (JavaServer Pages), JSF (JavaServer Faces), ou des frameworks de templating comme Thymeleaf.

3. Contrôleur (Controller) : Gère les interactions de l'utilisateur, traite les requêtes HTTP, et coordonne les réponses en utilisant les modèles et les vues. Dans Java EE, les servlets sont souvent utilisés pour ce rôle, mais des frameworks comme Spring MVC ou JSF peuvent aussi être employés.

Web MVC 2





EJB est une spécification pour les composants côté serveur qui encapsulent la logique métier d'une application. Ils sont gérés par un conteneur EJB qui fournit des services comme la gestion des transactions, la sécurité, la mise en cache et la gestion des ressources.

Les EJB sont utilisés pour encapsuler la logique métier d'une application. Cela permet de séparer les préoccupations de la présentation (vue) et de la logique métier (contrôleur).

Types d'EJB :

- 1.Session Beans** : Représentent une logique métier à courte durée de vie.
 - 1.Stateless (sans état)** : Ne maintiennent pas d'état entre les appels de méthodes.
 - 2.Stateful (avec état)** : Maintiennent l'état entre les appels de méthodes.
 - 3.Singleton** : Un seul instance par application.
- 2.Message-Driven Beans (MDB)** : Répondent à des messages asynchrones (souvent JMS).

Services fournis par le conteneur EJB :

Gestion des Transactions : Le conteneur peut gérer automatiquement les transactions, facilitant la gestion de la consistance des données.

Les EJB fournissent un support transactionnel intégré et une gestion de la concurrence. Cela garantit que les opérations critiques sont exécutées correctement et en toute sécurité.

Sécurité : Gestion des autorisations et des authentifications

Les EJB peuvent utiliser les mécanismes de sécurité de Jakarta EE pour restreindre l'accès aux méthodes en fonction des rôles utilisateur.

Injection de Dépendances : Permet l'injection de ressources et de dépendances nécessaires à l'exécution des beans.

Les EJB facilitent l'injection de dépendances, ce qui simplifie la gestion des objets et des services nécessaires à l'application.

Scalabilité :

Les EJB sont conçus pour être déployés dans des environnements distribués et peuvent être facilement mis à l'échelle pour gérer des charges de travail importantes.

```
package com.example.ejb;

import jakarta.ejb.EJB;
import jakarta.ejb.Stateless;
import jakarta.ejb.LocalBean;

/**
 *
 * @author Lenovo
 */
@Stateless
@LocalBean
public class HelloBean {

    public String from()
    {
        return "EJB";
    }

}
```

```
@WebServlet(name = "HelloServeur", urlPatterns = {"/Hello"})
public class HelloServeur extends HttpServlet {

    //@@EJB
    //private MyStatelessBean myStatelessBean;

    @EJB
    private HelloBean Bean;

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response)
        throws ServletException, IOException {
        response.setContentType("text/html;charset=UTF-8");
        PrintWriter out = response.getWriter();
        String name = "Sory";
        // Code existant
        out.println("<html><body><h1>Hello : Personne "+Bean.from()+""
</h1></body></html>");
    }
}
```

Notre exemple:

```
@Stateless  
@LocalBean  
public class BookServiceBean {  
  
    private static final Map<String, Livre> books = new HashMap<>();  
  
    static {  
        Livre book1 = new Livre();  
        book1.setNumLivre("1");  
        book1.setTitle("Book A");  
        books.put("1", book1);  
  
        Livre book2 = new Livre();  
        book2.setNumLivre("2");  
        book2.setTitle("Book B");  
        books.put("2", book2);  
  
        Livre book3 = new Livre();  
        book3.setNumLivre("3");  
        book3.setTitle("Book C");  
        books.put("3", book3);  
    }  
  
    public Livre findBookById(String id) {  
        return books.get(id);  
    }  
}
```

SelectBook.html

```
<!DOCTYPE html>
<html>
<head>
    <title>Select Book</title>
</head>
<body>
    <h1>Select a Book</h1>
    <form action="BookServlet" method="post">
        <input type="hidden" name="action" value="selectBook">
        <label for="book">Choose a book:</label>
        <select id="book" name="book">
            <option value="1">Book A</option>
            <option value="2">Book B</option>
            <option value="3">Book C</option>
        </select>
        <button type="submit">Select</button>
    </form>
</body>
</html>
```

```
public class Livre {  
    String numLivre;  
    String Title;  
  
    public String getTitle() {  
        return Title;  
    }  
  
    public void setTitle(String Title) {  
        this.Title = Title;  
    }  
  
    public String getNumLivre() {  
        return numLivre;  
    }  
  
    public void setNumLivre(String numLivre) {  
        this.numLivre = numLivre;  
    }  
}
```

```
package com.mycompany.Controleur;

@WebServlet("/BookServlet")
public class BookServlet extends HttpServlet {
    private static final long serialVersionUID = 1L;

    @EJB
    private BookServiceBean bookServiceBean;

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        HttpSession session = request.getSession(false);
        if (session == null || session.getAttribute("username") == null) {
            response.sendRedirect("UserServlet.html");
            return;
        }

        String action = request.getParameter("action");

        synchronized (session) {
            if ("selectBook".equals(action)) {
                String selectedBook = request.getParameter("book");

```


Fondamentaux des JPA(Java Persistence API)

Java Persistence API (JPA) est une spécification qui fournit un moyen de gérer les données relationnelles dans les applications Java. Dans le contexte de Jakarta EE et du modèle MVC (Model-View-Controller), JPA joue un rôle crucial dans la gestion de la persistance des données.

Abstraction des Détails de la Base de Données :

JPA fournit une abstraction des opérations CRUD (Create, Read, Update, Delete), permettant aux développeurs de se concentrer sur la logique métier plutôt que sur les détails de la base de données.

Modèle de Programmation Orienté Objet :

JPA permet de manipuler les données de la base de données sous forme d'objets Java, ce qui simplifie le développement et améliore la lisibilité du code.

Transactions :

JPA gère les transactions automatiquement, garantissant l'intégrité des données et simplifiant la gestion des transactions dans les applications.

Support pour les Requêtes JPQL :

JPA utilise JPQL (Java Persistence Query Language), un langage de requête orienté objet similaire à SQL, mais conçu pour manipuler les entités JPA.

Exemple d'Utilisation des JPA dans un Projet Jakarta EE MVC

Étape 1 : Créer l'Entité JPA

Livre.java (modifié pour inclure les annotations JPA) :

Object Relational Mapping

ID Generation

Auto

Table

Identity

Property Access

Field

Property

Mapping to Table

@Basic

@Column

@Transient

Simple Java Types

Large Objects

@Lob - CLOB/BLOB

Enumerated Types

String

Ordinal

Temporal Types

Embeddables

Relationship Concepts

Roles

Directionality

Cardinality

Ordinality

Many-to-one

One-to-one

One-to-many

Many-to-many

Collection Mapping

- JPA est une spécification Java EE pour la gestion de la persistance et le mapping objet-relationnel (ORM).
- Elle permet aux développeurs de travailler avec des bases de données relationnelles en utilisant des objets Java.

Principaux avantages de JPA :

- Abstraction de la gestion des bases de données.
- Simplification des opérations CRUD.
- Standardisation et compatibilité avec différents fournisseurs d'implémentations (Hibernate, EclipseLink, etc.).

Configuration du fichier persistence.xml

Le fichier persistence.xml est un fichier de configuration essentiel pour toute application utilisant JPA. Il se trouve dans le répertoire META-INF de votre projet. Ce fichier contient les informations nécessaires pour se connecter à la base de données et configurer les aspects de la persistance.

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence version="3.0" xmlns="https://jakarta.ee/xml/ns/persistence"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="https://jakarta.ee/xml/ns/persistence
  https://jakarta.ee/xml/ns/persistence/persistence_3_0.xsd">
  <persistence-unit name="myPU">
    <class>com.example.jpa.Livre</class>
    <properties>
      <property name="jakarta.persistence.jdbc.url"
        value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/>
      <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
      <property name="jakarta.persistence.jdbc.user" value="sa"/>
      <property name="jakarta.persistence.jdbc.password" value="" />
      <property name="jakarta.persistence.schema-generation.database.action"
        value="create"/>
    </properties>
  </persistence-unit>
</persistence>
```

3. Création d'entités JPA

Une entité JPA est une classe POJO (Plain Old Java Object) qui est mappée à une table de base de données.

Qu'est-ce qu'une Entité JPA ?

Une entité JPA est une classe Java simple (POJO - Plain Old Java Object) qui représente une table dans une base de données relationnelle. Chaque instance de cette classe correspond à une ligne de la table. Les entités sont utilisées pour mapper les objets Java aux tables de la base de données et vice versa.

Configuration de base d'une Entité JPA

Pour configurer une entité JPA, nous devons annoter votre classe Java avec des annotations JPA spécifiques.

Etapes de base pour créer une entité JPA :

- 1.Définir la classe comme une entité :** Utiliser l'annotation `@Entity` pour indiquer que la classe est une entité JPA.
- 2.Définir la clé primaire :** Utiliser l'annotation `@Id` pour définir le champ de la clé primaire. Nous pouvons également utiliser `@GeneratedValue` pour indiquer que la clé primaire sera générée automatiquement.
- 3.Définir les colonnes :** Utiliser l'annotation `@Column` pour personnaliser le mapping des champs aux colonnes de la table de la base de données.

```
import jakarta.persistence.Column;
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;

@Entity
public class Livre {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    //or @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(nullable = false)
    private String numLivre;

    private String title;

    /// Generer les Getters et les Setters
}
```

- **@Entity** : Indique que la classe est une entité.
- **@Id** : Spécifie le champ de clé primaire.
- **@GeneratedValue** : Spécifie la stratégie de génération de la clé primaire (AUTO, IDENTITY, SEQUENCE, TABLE).
- **@Column** : Personnalise le mapping du champ à une colonne de la table de la base de données. Peut être utilisé pour définir des contraintes telles que nullable = false, unique = true, etc.
- **@Table** : Spécifie la table de la base de données à laquelle l'entité est mappée. Si omise, le nom de la classe sera utilisé par défaut.

JPA Entity - Using Super Classes

Lors de la modélisation de notre domaine en utilisant JPA, nous pouvons avoir des entités abstraites qui servent de base à d'autres entités concrètes. Cela permet de définir des propriétés et des comportements communs dans une superclasse abstraite, puis de spécialiser ces comportements dans des sous-classes concrètes.

```
@MappedSuperclass
public abstract class Personne implements Serializable {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String nom;
    private String prenom;
    private String email;

    // Getters et setters

}
```

JPA Entity - Using Super Classes

Lors de la modélisation de notre domaine en utilisant JPA, nous pouvons avoir des entités abstraites qui servent de base à d'autres entités concrètes. Cela permet de définir des propriétés et des comportements communs dans une superclasse abstraite, puis de spécialiser ces comportements dans des sous-classes concrètes.

```
@Entity  
public class Client extends Personne {  
  
    private String adresse;  
  
    // Getters et setters  
  
    public String getAdresse() {  
        return adresse;  
    }  
  
    public void setAdresse(String adresse) {  
        this.adresse = adresse;  
    }  
}
```

```
@Entity  
public class Employe extends Personne {  
  
    private String departement;  
  
    // Getters et setters  
  
    public String getDepartement() {  
        return departement;  
    }  
  
    public void setDepartement(String departement) {  
        this.departement = departement;  
    }  
}
```

JPA Entity - Super Class Field Overriding

L'annotation **@AttributeOverride** permet de redéfinir les mappings des attributs hérités d'une superclasse dans une entité JPA

```
@Entity
@GeneratedValue(strategy = GenerationType.IDENTITY)
@NoArgsConstructor
@AllArgsConstructor
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    private String departement;

    // Getters et setters

    public String getDepartement() {
        return departement;
    }

    public void setDepartement(String departement) {
        this.departement = departement;
    }
}
```

JPA Entity - Mapping Simple Java Types

Dans JPA, le mapping des types Java simples aux types de base de données est une tâche fondamentale. JPA prend en charge le mapping des types de données Java courants tels que les chaînes de caractères, les nombres, les dates, et les booléens aux types correspondants dans la base de données.

L'annotation **@Basic** est utilisée pour spécifier que l'attribut d'une entité JPA est un attribut de base. Par défaut, tous les attributs de type simple (comme String, int, boolean, etc.) sont considérés comme des attributs de base et sont implicitement annotés avec **@Basic**. Cependant, cette annotation peut être explicitement utilisée pour définir certains comportements supplémentaires.

JPA Entity - Customizing Database Columns

Lors de la modélisation des entités JPA, nous pouvons personnaliser les colonnes de la base de données pour répondre à des besoins spécifiques. Cela peut inclure la modification des noms de colonnes, la définition de contraintes, la personnalisation des types de données, et bien plus encore. L'annotation principale utilisée pour cette personnalisation est **@Column**

Annotation @Column

L'annotation **@Column** permet de spécifier les détails de mapping entre un champ de l'entité et une colonne de la table de la base de données.

propriétés couramment utilisées :

- **name** : Le nom de la colonne dans la base de données.
- **nullable** : Indique si la colonne peut contenir des valeurs nulles (true par défaut).
- **unique** : Indique si la colonne doit avoir des valeurs uniques (false par défaut).
- **length** : La longueur maximale de la colonne (utilisée pour les types de données String).
- **precision** : La précision de la colonne (utilisée pour les types de données décimales).
- **scale** : L'échelle de la colonne (utilisée pour les types de données décimales).
- **insertable** : Indique si la colonne doit être incluse dans les instructions SQL d'insertion (true par défaut).
- **updatable** : Indique si la colonne doit être incluse dans les instructions SQL de mise à jour (true par défaut).
- **columnDefinition** : Permet de spécifier une définition de colonne SQL directement.

JPA Entity - Customizing Database Columns

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    // Getters et setters

    public String getDepartement() {
        return departement;
    }

    public void setDepartement(String departement) {
        this.departement = departement;
    }
}
```

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")
    private String nom;

    // Getters et setters

    public String getDepartement() {
        return departement;
    }

    public void setDepartement(String departement) {
        this.departement = departement;
    }
}
```

JPA Entity - Transient Fields

Dans JPA, les champs transitoires sont des champs d'entité qui ne sont pas persistés dans la base de données. Ils sont utilisés pour stocker des données temporaires qui ne doivent pas être sauvegardées dans la base de données.

Utilisation de l'Annotation @Transient

L'annotation `@Transient` est utilisée pour marquer un champ comme transitoire. Les champs annotés avec `@Transient` ne sont pas inclus dans les opérations de persistance JPA.

```
@Entity
public class Client extends Personne {

    private String adresse;

    @Transient
    private String ageCategory;

    // Getters et setters

    public String getAdresse() {
        return adresse;
    }

    public void setAdresse(String adresse) {
        this.adresse = adresse;
    }
}
```

JPA Entity - Field Access Type

En JPA, le type d'accès aux champs détermine comment l'implémentation JPA accède aux propriétés de l'entité : par les champs ou par les méthodes getter/setter. Les deux types d'accès principaux sont :

- 1. Field Access (Accès par champ)**
- 2. Property Access (Accès par propriété)**

Field Access (Accès par champ)

Avec l'accès par champ, JPA accède directement aux champs de l'entité. Les annotations sont placées sur les champs eux-mêmes.

```
@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "employee_name", nullable = false, length = 50)
    private String name;

    @Column(name = "employee_age", nullable = true)
    private Integer age;

    // Getters et setters

    public Long getId() {
        return id;
    }

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

Property Access (Accès par propriété)

Avec l'accès par propriété, JPA utilise les méthodes getter et setter pour accéder aux champs. Les annotations sont placées sur les méthodes getter.

```
@Entity
public class Employee {

    private Long id;
    private String name;
    private Integer age;

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

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

    @Column(name = "employee_name", nullable = false, length = 50)
    public String getName() {
        return name;
    }
}
```

Spécification du Type d'Accès

Le type d'accès est déterminé par l'emplacement des annotations de JPA. La règle est simple :

- Si l'annotation `@Id` est placée sur un champ, l'accès par champ est utilisé.
- Si l'annotation `@Id` est placée sur une méthode getter, l'accès par propriété est utilisé.

Mixtes

Bien que cela ne soit pas recommandé, nous pouvons mélanger les types d'accès dans une même entité en utilisant l'annotation `@Access`. Cela peut être utile dans des cas particuliers où nous devons personnaliser le comportement de l'accès.

```
@Entity
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Access(AccessType.PROPERTY)
    private String name;

    @Access(AccessType.FIELD)
    @Column(name = "employee_age", nullable = true)
    private Integer age;

    // Getters et setters

    public Long getId() {
        return id;
    }

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

    @Column(name = "employee_name", nullable = false, length = 50)
```

Avantages et Inconvénients

Field Access

Avantages :

- Plus simple à écrire et à lire.
- Accède directement aux champs, ce qui peut être légèrement plus performant.

Inconvénients :

- Ne prend pas en compte la logique métier présente dans les getters/setters.
- Peut entraîner des problèmes si la logique métier doit être exécutée lors de l'accès ou de la modification des champs.

Property Access

Avantages :

- Permet l'encapsulation et l'ajout de logique métier dans les getters/setters.
- Plus flexible si des validations ou transformations sont nécessaires lors de l'accès ou de la modification des champs.

Inconvénients :

- Peut rendre le code un peu plus verbeux et complexe.
- Légèrement plus lent en raison des appels de méthode supplémentaires

JPA Mapping Types

En JPA, nous pouvons mapper divers types de données Java à des colonnes de base de données, y compris les types spéciaux comme les énumérations (enum) et les grands objets (LOB - Large Objects) tels que les images ou les fichiers de texte volumineux.

JPA Entity - Mapping Enum Types

Mapping des Enumérations Les énumérations (enum) peuvent être mappées à des colonnes de base de données de deux manières :

- **ORDINAL** : Le numéro ordinal de l'énumération (sa position dans la déclaration) est stocké.
- **STRING** : Le nom de l'énumération est stocké.

```
public enum EmployeeType
{
    FULL_TIME,
    PART_TIME,
    CONTRACT
}
```

```
/**  
 *  
 * @author Lenovo  
 */  
@Entity  
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,  
unique = true))  
public class Employe extends Personne {  
  
@Column(name = "departement_name", nullable = false, length = 50)  
private String departement;  
  
@Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")  
private String nom;  
  
@Enumerated(EnumType.STRING)  
@Column(name = "employee_type", nullable = false)  
private EmployeeType employeeType;  
  
// Getters et setters  
  
}
```

JPA - Mapping Large Objects (Eg images)

Les grands objets tels que les images, les fichiers audio ou les textes volumineux peuvent être mappés en utilisant l'annotation @Lob.

```
@Entity
public class Document {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "document_name", nullable = false, length = 100)
    private String name;

    @Lob
    @Column(name = "document_content", nullable = false)
    private byte[] content;

    // Getters et setters

}
```

JPA - Lazy and Eager Fetching of Entity State

Dans JPA, les stratégies de chargement des entités permettent de contrôler quand les données associées sont récupérées de la base de données.

Les deux principales stratégies de chargement sont :

- Chargement hâtif (Eager Fetching)
- Chargement paresseux (Lazy Fetching)

Ces stratégies sont principalement utilisées pour les associations entre entités, telles que:

@OneToOne, @OneToMany, @ManyToOne, et @ManyToMany.

Chargement Paresseux (Lazy Fetching)

Avec le chargement paresseux, les données associées ne sont récupérées que lorsqu'elles sont réellement accédées. Cela permet de retarder le chargement des données associées jusqu'à ce qu'elles soient nécessaires, ce qui peut améliorer les performances en réduisant la quantité de données récupérées initialement.

```
@Entity
public class Document {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "document_name", nullable = false, length = 100)
    private String name;

    @Lob
    @Basic(fetch = FetchType.LAZY)
    @Column(name = "document_content", nullable = false)
    private byte[] content;

    // Getters et setters
}
```

Chargement Hâtif (Eager Fetching)

Avec le chargement hâtif, les données associées sont récupérées immédiatement lors de la récupération de l'entité principale. Cela signifie que toutes les entités liées sont chargées en même temps que l'entité principale, même si elles ne sont pas utilisées immédiatement.

```
@Entity
public class Document {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "document_name", nullable = false, length = 100)
    private String name;

    @Lob
    @Basic(fetch = FetchType.EAGER)
    @Column(name = "document_content", nullable = false)
    private byte[] content;

    // Getters et setters
}
```

Comparaison des Deux Stratégies

- **Chargement Hâtif (EAGER) :**

- **Avantage** : Toutes les données nécessaires sont disponibles immédiatement après la récupération de l'entité.
- **Inconvénient** : Peut entraîner une surcharge de performance si les données de grande taille ne sont pas toujours nécessaires.

- **Chargement Paresseux (LAZY) :**

- **Avantage** : Améliore les performances initiales en ne chargeant pas les données volumineuses jusqu'à ce qu'elles soient nécessaires.
- **Inconvénient** : Nécessite une gestion correcte du contexte de persistance pour éviter les exceptions LazyInitializationException.

Par Défaut pour les Relations

1. @OneToOne : FetchType.EAGER par défaut.
2. @ManyToOne : FetchType.EAGER par défaut.
3. @OneToMany : FetchType.LAZY par défaut.
4. @ManyToMany : FetchType.LAZY par défaut.

Par Défaut pour les Attributs Simples

Pour les attributs simples (comme String, int, etc.), si on utilise pas d'annotation comme @Basic, ils seront traités avec un chargement hâtif implicite.

JPA Mapping Java Date/Time Types

Mapping des Types Date/Time en Jakarta Persistence

Avec Jakarta Persistence, nous pouvons mapper les types de date et d'heure de Java aux types correspondants dans la base de données en utilisant les annotations appropriées.

Nous pouvons utiliser `@Temporal` pour les types `java.util.Date` et `java.util.Calendar`, et les types de l'API Java Time (`java.time.*`) sont pris en charge nativement.

Exemple avec `java.util.Date` et `@Temporal`

```
@Temporal(TemporalType.DATE) @Column(name = "event_date", nullable = false)  
private Date eventDate;
```

```
@Temporal(TemporalType.TIME) @Column(name = "event_time", nullable = false)  
private Date eventTime;
```

```
@Temporal(TemporalType.TIMESTAMP) @Column(name = "event_timestamp", nullable = false)  
private Date eventTimestamp;
```

Exemple avec l'API Java Time (java.time.*)

```
@Column(name = "meeting_date", nullable = false)
```

```
private LocalDate meetingDate;
```

```
@Column(name = "meeting_time", nullable = false)
```

```
private LocalTime meetingTime;
```

```
@Column(name = "meeting_timestamp", nullable = false)
```

```
private LocalDateTime meetingTimestamp;
```

JPA - Mapping Embeddable Classes

Dans JPA, une classe embeddable est une classe dont les propriétés peuvent être intégrées dans une autre entité.

Cela permet de réutiliser des groupes de champs communs entre différentes entités sans avoir à les dupliquer.

Les classes embeddables sont définies avec l'annotation **@Embeddable**, et les entités qui les utilisent les intègrent avec l'annotation **@Embedded**.

```
@Embeddable
public class Address {

    private String street;
    private String city;
    private String state;
    private String zipCode;

    // Getters et setters

}
```

JPA - Mapping Embeddable Classes

```
/**  
 *  
 * @author Lenovo  
 */  
@Entity  
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,  
unique = true))  
public class Employe extends Personne {  
  
    @Column(name = "departement_name", nullable = false, length = 50)  
    private String departement;  
  
    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")  
    private String nom;  
  
    @Embedded  
    private Address address;  
  
    // Getters et setters  
}
```

JPA - Mapping Embeddable Classes

Annotation **@AttributeOverride**

Si nous souhaitons personnaliser le mapping des colonnes d'un champ embeddable, nous pouvons utiliser l'annotation **@AttributeOverride**. Cela nous permet de spécifier des noms de colonnes différents pour les propriétés de la classe embeddable.

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")
    private String nom;

    @Enumerated(EnumType.STRING)
    @Column(name = "employee_type", nullable = false)
    private EmployeeType employeeType;

    @Embedded
    @AttributeOverrides({
        @AttributeOverride(name = "street", column = @Column(name = "home_street")),
        @AttributeOverride(name = "city", column = @Column(name = "home_city")),
        @AttributeOverride(name = "state", column = @Column(name = "home_state")),
        @AttributeOverride(name = "zipCode", column = @Column(name = "home_zip_code"))
    })
    private Address homeAddress;
}
```

Utilisation de **@EmbeddedId** pour les Clés Composées

Les classes embeddables peuvent également être utilisées pour les clés composées en les annotant avec **@EmbeddedId** dans l'entité.

```
@Embeddable  
public class EmployeeId implements Serializable {  
  
    private String departmentId;  
    private Long employeeNumber;  
  
    // Getters et setters  
  
}
```

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @EmbeddedId
    private EmployeeId id;

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")
    private String nom;

    @Enumerated(EnumType.STRING)
    @Column(name = "employee_type", nullable = false)
    private EmployeeType employeeType;

    // Getters et setters
}
```

JPA - An Intro to Mapping Primary Keys

Dans JPA (Jakarta Persistence API), le mapping des clés primaires est essentiel pour identifier de manière unique chaque enregistrement dans une table de base de données. Une clé primaire peut être une seule colonne (clé primaire simple) ou une combinaison de colonnes (clé primaire composite).

La manière la plus courante de définir une clé primaire est d'utiliser une seule colonne avec l'annotation `@Id`. Il est également fréquent d'utiliser des générateurs de clés pour créer des valeurs uniques automatiquement.

```
@Id @GeneratedValue(strategy = GenerationType.AUTO) private Long id;
```

Génération des Clés Primaires

JPA fournit plusieurs stratégies pour la génération des clés primaires :

- **AUTO** : JPA choisit automatiquement la stratégie la plus appropriée pour la base de données utilisée.
- **IDENTITY** : Utilise une colonne de type auto-increment.
- **SEQUENCE** : Utilise une séquence de base de données.
- **TABLE** : Utilise une table spécifique pour générer les valeurs des clés primaires

```
@Id @GeneratedValue(strategy = GenerationType.SEQUENCE, generator = "product_seq")
```

```
@SequenceGenerator(name = "product_seq", sequenceName = "product_sequence",  
allocationSize = 1)
```

```
private Long id;
```

Clé Primaire Composite

Une clé primaire composite utilise plusieurs colonnes pour identifier de manière unique un enregistrement. JPA permet de définir des clés composites en utilisant une classe embeddable annotée avec `@Embeddable`.

JPA - An Intro to Entity Relationship Mapping

En JPA (Jakarta Persistence API), les entités peuvent être liées entre elles par des relations. Les relations entre les entités sont mappées en utilisant des annotations spécifiques. JPA prend en charge quatre types principaux de relations entre les entités :

1. One-to-One (@OneToOne)
2. One-to-Many (@OneToMany)
3. Many-to-One (@ManyToOne)
4. Many-to-Many (@ManyToMany)

Ces relations peuvent être unidirectionnelles ou bidirectionnelles.

One-to-One (@OneToOne)

Une relation One-to-One signifie qu'une entité est liée à une autre entité de manière unique. Chaque instance de l'entité source a une correspondance unique dans l'entité cible.

```
public class Employe extends Personne {  
  
    @EmbeddedId  
    private EmployeeId id;  
  
    @Column(name = "departement_name", nullable = false, length = 50)  
    private String departement;  
  
    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")  
    private String nom;  
  
    @Enumerated(EnumType.STRING)  
    @Column(name = "employee_type", nullable = false)  
    private EmployeeType employeeType;  
  
    @OneToOne  
    @JoinColumn(name = "parking_spot_id")  
    private ParkingSpot parkingSpot;
```

```
/**  
 *  
 * @author Lenovo  
 */  
@Entity  
public class ParkingSpot {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
  
    private String location;  
  
    // Getters et setters  
}
```

One-to-Many (@OneToMany)

Une relation One-to-Many signifie qu'une entité source est liée à plusieurs instances de l'entité cible.

```
@Entity
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    @OneToMany(mappedBy = "department")
    private List<Employe> employees;

    // Getters et setters
}
```

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @EmbeddedId
    private EmployeeId id;

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")
    private String nom;

    @Enumerated(EnumType.STRING)
    @Column(name = "employee_type", nullable = false)
    private EmployeeType employeeType;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
```

Many-to-One (@ManyToOne)

Une relation Many-to-One est l'inverse de One-to-Many. Chaque instance de l'entité source est liée à une seule instance de l'entité cible, mais l'entité cible peut avoir plusieurs entités sources qui lui sont associées.

Une relation Many-to-One unidirectionnelle signifie qu'une entité source (par exemple, Employee) est liée à une seule instance d'une entité cible (par exemple, Department), mais l'entité cible n'a pas de référence explicite à l'entité source. Seule l'entité source connaît la relation.

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @EmbeddedId
    private EmployeeId id;

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")
    private String nom;

    @Enumerated(EnumType.STRING)
    @Column(name = "employee_type", nullable = false)
    private EmployeeType employeeType;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;
```

```
@Entity
public class Department {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;
    // Getters et setters
}
```

Many-to-Many (@ManyToMany)

Une relation Many-to-Many signifie qu'une instance de l'entité source peut être liée à plusieurs instances de l'entité cible, et vice versa

```
@Entity
public class Student {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    private String name;

    @ManyToMany
    @JoinTable(
        name = "student_course",
        joinColumns = @JoinColumn(name =
"student_id"),
        inverseJoinColumns = @JoinColumn(name =
"course_id")
    )
    private List<Course> courses;

    // Getters et setters
}
```

```
@Entity
public class Course {

    @Id
    @GeneratedValue(strategy =
GenerationType.AUTO)
    private Long id;

    private String title;

    @ManyToMany(mappedBy = "courses")
    private List<Student> students;

    // Getters et setters
}
```

- **JPA Collection Mapping of Embeddable Objects and Collection Table**
- **JPA Collection Mapping of Strings (Or other Simple Java types)**

JPA permet de mapper des collections de types simples (comme String) ou des objets embeddables dans des entités. Il existe plusieurs manières de gérer ces mappings, notamment en utilisant des collections d'objets embeddables ou en mappant des collections dans une table de collection.

1. Mapping de Collections d'Objets Embeddables

Exemple d'Objets Embeddables Supposons que nous avons une entité Employee qui a une collection de Address.

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @EmbeddedId
    private EmployeeId id;

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")
    private String nom;

    @Enumerated(EnumType.STRING)
    @Column(name = "employee_type", nullable = false)
    private EmployeeType employeeType;

    @ManyToOne
    @JoinColumn(name = "department_id")
    private Department department;

    @OneToOne
    @JoinColumn(name = "parking_spot_id")
```

```
@Embeddable  
public class Address {  
  
    private String street;  
    private String city;  
    private String state;  
    private String zipCode;  
  
    // Getters et setters
```

2. Mapping de Collections de Types Simples (Strings)

JPA permet également de mapper des collections de types simples comme String, Integer, etc. Exemple de Collection de Strings Supposons que nous avons une entité Book qui a une collection de genres (genres).

```
@Entity
public class Book
{
    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "book_title", nullable = false)
    private String title;

    @ElementCollection
    @CollectionTable(name = "book_genre", joinColumns = @JoinColumn(name = "book_id"))
    @Column(name = "genre")
    private List<String> genres;
}
```

Le mapping des collections en JPA permet de gérer efficacement les relations entre entités et leurs attributs complexes ou multiples. En utilisant `@ElementCollection` et `@CollectionTable`, vous pouvez facilement gérer des collections de types simples comme des `String` ou des objets embeddables dans vos entités JPA, rendant le modèle de données plus flexible et plus représentatif des relations réelles.

Persistable Collection Types

JPA prend en charge plusieurs types de collections pour les entités et les objets embeddables. Les types de collections persistables couramment utilisés incluent :

1.java.util.List

2.java.util.Set

3.java.util.Map

1. java.util.List

List est une collection ordonnée qui permet des éléments en double. Les éléments sont ordonnés par leur position dans la liste.

Exemple : Entité avec List

Supposons que nous avons une entité Employee avec une liste de numéros de téléphone

```
@Entity
@AttributeOverride(name = "email", column = @Column(name = "employee_email", nullable = false,
unique = true))
public class Employe extends Personne {

    @EmbeddedId
    private EmployeeId id;

    @Column(name = "departement_name", nullable = false, length = 50)
    private String departement;

    @Column(name = "employee_name", columnDefinition = "VARCHAR(50) NOT NULL")
    private String nom;

    @Enumerated(EnumType.STRING)
    @Column(name = "employee_type", nullable = false)
    private EmployeeType employeeType;

    @ElementCollection
    @CollectionTable(name = "employee_phone", joinColumns = @JoinColumn(name = "employee_id"))
    @Column(name = "phone_number")
    private List<String> phones;
```

2. java.util.Set Set est une collection qui ne permet pas d'éléments en double.

Les éléments ne sont pas ordonnés.

Exemple : Entité avec Set

Supposons que nous avons une entité Book avec un ensemble de genres (genres).

```
@Entity
public class Book {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;

    @Column(name = "book_title", nullable = false)
    private String title;

    @ElementCollection
    @CollectionTable(name = "book_genre", joinColumns = @JoinColumn(name = "book_id"))
    @Column(name = "genre")
    private Set<String> genres;
```

3. java.util.MapMap

est une collection qui associe des clés à des valeurs. Les clés ne peuvent pas être dupliquées, mais les valeurs peuvent l'être.

Exemple : Entité avec Map

Supposons que nous avons une entité Library avec une carte des livres (books), où chaque entrée de la carte représente un ISBN et un titre de livre.

```
import jakarta.persistence.Embeddable;

@Embeddable
public class BookInfo {
    private String title;
    private String author;

    // Getters et setters

    public String getTitle() {
        return title;
    }

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

    public String getAuthor() {
        return author;
    }

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

```
@Entity
public class Library {

    @Id
    @GeneratedValue(strategy =
GenerationType.AUTO)
    private Long id;

    @Column(name = "library_name", nullable =
false)
    private String name;

    @ElementCollection
    @CollectionTable(name = "library_books",
joinColumns = @JoinColumn(name =
"library_id"))
    @MapKeyColumn(name = "isbn")
    @Column(name = "book_info")
    private Map<String, BookInfo> books;

    // Getters et setters
}
```

Enterprise JavaBeans (EJBs) - What Are They?

Enterprise JavaBeans (EJBs) - Qu'est-ce que c'est?

Enterprise JavaBeans (EJBs) est une architecture Java EE (Jakarta EE) pour le développement et le déploiement de composants d'application distribués, basés sur des transactions, dans l'environnement serveur.

Les EJBs sont conçus pour simplifier le développement des applications à grande échelle, distribuées et transactionnelles.

Types de EJBs

Il existe trois principaux types de EJBs :

1.Session Beans

2.Message-Driven Beans (MDBs)

3.Entity Beans (dépréciés à partir de EJB 3.0, remplacés par JPA)

1. Session Beans

Les Session Beans représentent des unités de logique métier dans une application.

Il existe trois types de Session Beans :

- **Stateless Session Beans** : Ils ne maintiennent pas l'état entre les appels du client. Chaque appel est indépendant.
- **Stateful Session Beans** : Ils maintiennent l'état entre les appels du client. L'état est conservé tant que la session du client est active.
- **Singleton Session Beans** : Ils sont instanciés une seule fois par le conteneur EJB et partagés entre tous les clients.

Exemple de Stateless Session Bean

- Ils ne maintiennent pas l'état entre les appels du client. Chaque appel est indépendant.

```
import jakarta.ejb.Stateless;  
  
@Stateless  
public class CalculatorBean {  
  
    public int add(int a, int b) {  
        return a + b;  
    }  
  
    public int subtract(int a, int b) {  
        return a - b;  
    }  
}
```

Exemple de Stateful Session Bean

- Ils maintiennent l'état entre les appels du client. L'état est conservé tant que la session du client est active.

```
import jakarta.ejb.Stateful;

@Stateful
public class ShoppingCartBean {

    private List<String> items = new ArrayList<>();

    public void addItem(String item) {
        items.add(item);
    }

    public List<String> getItems() {
        return items;
    }
}
```

Exemple de Singleton Session Bean

Ils sont instanciés une seule fois par le conteneur EJB et partagés entre tous les clients.

```
import jakarta.ejb.Singleton;

@Singleton
public class ConfigurationBean {

    private String config;

    public String getConfig() {
        return config;
    }

    public void setConfig(String config) {
        this.config = config;
    }
}
```

2. Message-Driven Beans (MDBs)

Les MDBs permettent de traiter des messages asynchrones. Ils sont souvent utilisés pour interagir avec des systèmes de messagerie comme JMS (Java Message Service).

```
@MessageDriven(  
    activationConfig = {  
        @ActivationConfigProperty(propertyName = "destinationType", propertyValue =  
"jakarta.jms.Queue")  
    }  
)  
public class OrderListenerBean implements MessageListener {  
  
    @Override  
    public void onMessage(Message message) {  
        // Traitement du message  
    }  
}
```

3. Entity Beans

Les Entity Beans étaient utilisés pour représenter des objets persistants dans une base de données relationnelle. Ils ont été dépréciés à partir de EJB 3.0 et ont été remplacés par JPA (Java Persistence API).

Caractéristiques des EJBs

- **Transactions** : Les EJBs peuvent participer à des transactions distribuées, gérées par le conteneur EJB.
- **Sécurité** : Le conteneur EJB fournit des mécanismes de sécurité pour authentifier et autoriser les appels aux EJBs.
- **Concurrence** : Le conteneur EJB gère les problèmes de concurrence pour les EJBs.
- **Interception** : Les EJBs peuvent utiliser des intercepteurs pour appliquer des logiques transversales comme la journalisation et la gestion des exceptions.

Avantages des EJBs

- **Simplification du développement** : Les EJBs simplifient le développement des applications distribuées et transactionnelles en externalisant la gestion des transactions, la sécurité et la concurrence au conteneur EJB.
- **Réutilisation des composants** : Les EJBs encouragent la réutilisation des composants de logique métier.
- **Scalabilité** : Les EJBs sont conçus pour être distribués et peuvent être déployés sur plusieurs serveurs pour améliorer la scalabilité.

Inconvénients des EJBs

- **Complexité** : Les EJBs peuvent introduire une certaine complexité dans le développement et le déploiement des applications.
- **Performance** : Le surcoût du conteneur EJB peut affecter les performances dans certains scénarios.

Utilisation des EJB avec JPA pour Stocker les Valeurs

Les Enterprise JavaBeans (EJBs) et Jakarta Persistence API (JPA) sont souvent utilisés ensemble dans les applications Java EE (ou Jakarta EE) pour gérer la logique métier et la persistance des données.

Concepts Clés

- 1. EJB (Enterprise JavaBeans)** : Fournissent un modèle standard pour encapsuler la logique métier.
- 2. JPA (Jakarta Persistence API)** : Fournit une API standard pour la gestion de la persistance et du mappage objet-relationnel (ORM).
- 3. EntityManager (em)** : Interface principale de JPA utilisée pour interagir avec le contexte de persistance.

Structure d'un Projet EJB et JPA

- 1. Entités JPA** : Représentent les objets persistants mappés à des tables de base de données.
- 2. Session Beans EJB** : Encapsulent la logique métier et utilisent l'**EntityManager** pour la persistance des entités.
- 3. Unités de Persistance (persistence.xml)** : Définissent la configuration de persistance JPA.

Étapes pour Utiliser les EJB avec JPA

1.Définir les Entités JPA

2.Configurer persistence.xml

3.Créer un Session Bean pour la Logique Métier

4.Utiliser l'EntityManager dans le Session Bean

1. Définir les Entités JPA

```
import jakarta.persistence.*;  
  
@Entity  
public class Employee {  
  
    @Id  
    @GeneratedValue(strategy = GenerationType.AUTO)  
    private Long id;  
  
    @Column(name = "employee_name", nullable = false, length = 50)  
    private String name;  
  
    private double salary;  
  
    // Getters et setters  
}
```

2. Configurer persistence.xml

Qu'est-ce qu'une unité de persistance (Persistence Unit) ?

Une unité de persistance est un concept clé en JPA (Java Persistence API) qui définit les détails nécessaires pour configurer et gérer l'accès à une base de données. Elle regroupe toutes les classes entitées, les informations de connexion, et les paramètres de gestion des transactions pour une application.

Structure d'une Unité de Persistance

Une unité de persistance est définie dans le fichier persistence.xml, situé dans le répertoire META-INF de notre application. éléments principaux :

- 1. Nom de l'unité de persistance :** Chaque unité de persistance doit avoir un nom unique.
- 2. Classes d'entité :** Les classes qui représentent les tables de la base de données.
- 3. Propriétés de connexion :** Informations de connexion à la base de données telles que l'URL, le pilote, le nom d'utilisateur et le mot de passe.
- 4. Paramètres supplémentaires :** Autres configurations comme le mode de génération du schéma, le cache, et les extensions spécifiques au fournisseur JPA

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence xmlns="https://jakarta.ee/xml/ns/persistence" version="3.0">
    <persistence-unit name="myPU">
        <class>com.example.jpa.Employee</class>
        <class>com.example.jpa.Department</class>
        <properties>
            <property name="jakarta.persistence.jdbc.url"
value="jdbc:h2:mem:test;DB_CLOSE_DELAY=-1"/>
                <property name="jakarta.persistence.jdbc.driver" value="org.h2.Driver"/>
                <property name="jakarta.persistence.jdbc.user" value="sa"/>
                <property name="jakarta.persistence.jdbc.password" value="" />
                <property name="jakarta.persistence.schema-generation.database.action"
value="create"/>
            </properties>
        </persistence-unit>
    </persistence>
```

Éléments de persistence.xml

1.<persistence-unit name="myPU"> :

- Nom unique de l'unité de persistance. Dans cet exemple, l'unité de persistance s'appelle "myPU".

2.<class> :

- Liste des classes d'entités. Ici, Employee et Department sont des classes d'entités mappées aux tables de la base de données.

3.<properties> :

- Propriétés de configuration de la connexion à la base de données et d'autres paramètres

Propriétés de Connexion

- jakarta.persistence.jdbc.url : URL de connexion à la base de données.
- jakarta.persistence.jdbc.driver : Pilote JDBC utilisé pour se connecter à la base de données.
- jakarta.persistence.jdbc.user : Nom d'utilisateur pour la connexion.
- jakarta.persistence.jdbc.password : Mot de passe pour la connexion.

Autres Propriétés

- jakarta.persistence.schema-generation.database.action : Détermine l'action à effectuer sur le schéma de la base de données (par exemple, create, update, drop-and-create)

Utilisation d'une Unité de Persistance

L'unité de persistance est utilisée par l'EntityManager pour gérer les entités et les transactions. L'EntityManager est initialisé en utilisant le nom de l'unité de persistance définie dans persistence.xml

```
public class EmployeeService {  
  
    private EntityManagerFactory emf = Persistence.createEntityManagerFactory("myPU");  
    private EntityManager em = emf.createEntityManager();  
  
    public void addEmployee(String name, String department) {  
        em.getTransaction().begin();  
        Employee employee = new Employee();  
        employee.setName(name);  
        employee.setDepartment(department);  
        em.persist(employee);  
        em.getTransaction().commit();  
    }  
}
```

3. Créer un Session Bean pour la Logique Métier

```
@Stateless
public class EmployeeService {

    @PersistenceContext(unitName = "myPU")
    private EntityManager em;

    public void createEmployee(String name, double salary) {
        Employee employee = new Employee();
        employee.setName(name);
        employee.setSalary(salary);
        em.persist(employee);
    }

    public Employee findEmployee(Long id) {
        return em.find(Employee.class, id);
    }

    public void updateEmployee(Long id, String name, double salary) {
        Employee employee = em.find(Employee.class, id);
        if (employee != null) {
            employee.setName(name);
            employee.setSalary(salary);
        }
    }
}
```

@PersistenceContext

Injecte l'Entity Manager configuré dans persistence.xml.

L'EntityManager est l'interface principale de JPA pour interagir avec le contexte de persistance.

EntityManager (em)

- **persist** : Utilisé pour sauvegarder une nouvelle entité dans la base de données.
- **find** : Utilisé pour récupérer une entité par sa clé primaire.
- **merge** : Utilisé pour mettre à jour une entité existante.
- **remove** : Utilisé pour supprimer une entité de la base de données.

4. Utiliser l'EntityManager dans le Session Bean

Le Session Bean utilise l'EntityManager pour interagir avec la base de données.

```
public class EmployeeServlet extends HttpServlet {  
  
    @EJB  
    private EmployeeService employeeService;  
  
    protected void doPost(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        String name = request.getParameter("name");  
        double salary = Double.parseDouble(request.getParameter("salary"));  
  
        employeeService.createEmployee(name, salary);  
  
        response.getWriter().write("Employee created successfully!");  
    }  
  
    protected void doGet(HttpServletRequest request, HttpServletResponse response)  
        throws ServletException, IOException {  
        Long id = Long.parseLong(request.getParameter("id"));  
        Employee employee = employeeService.findEmployee(id);  
  
        if (employee != null) {
```

Exemple avec Employe

JPQL (Java Persistence Query Language)

JPQL est une version orientée objet de SQL. Elle est utilisée pour effectuer des requêtes sur des entités JPA. Contrairement à SQL qui opère directement sur les tables de la base de données, JPQL opère sur les entités et les attributs des entités.

Syntaxe de JPQL

JPQL utilise une syntaxe similaire à SQL mais opère sur les entités JPA :

- **SELECT** : Récupère les entités ou les attributs des entités.
- **FROM** : Spécifie les entités sur lesquelles la requête opère.
- **WHERE** : Filtre les résultats de la requête.
- **JOIN** : Permet de faire des jointures entre les entités.

Syntaxe de JPQL

1. Requête de Sélection Simple **SELECT e FROM Employee e**

2. Requête avec Filtrage

SELECT e FROM Employee e WHERE e.department = :department

3. Requête avec Jointure

SELECT e FROM Employee e JOIN e.department d WHERE d.name = :deptName

4. Requête avec Agrégation

SELECT COUNT(e) FROM Employee e

Exemple de Requêtes JPQL

1. Requête de Sélection Simple

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.TypedQuery;
import java.util.List;

public class EmployeeService {

    @PersistenceContext
    private EntityManager em;

    public List<Employee> getAllEmployees() {
        TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e",
Employee.class);
        return query.getResultList();
    }
}
```

Détails sur les Méthodes et Fonctionnalités

1. createQuery

La méthode createQuery est utilisée pour créer une requête JPQL.

Elle prend deux arguments :

- La chaîne de la requête JPQL.
- La classe du type de résultat attendu.

```
TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e", Employee.class);
```

2. setParameter

La méthode setParameter est utilisée pour définir les paramètres de la requête. Les paramètres sont marqués avec : dans la requête JPQL

```
query.setParameter("department", department);
```

3. getResultList

La méthode getResultList est utilisée pour exécuter la requête et obtenir les résultats sous forme de liste.

```
List<Employee> results = query.getResultList();
```

4. getSingleResult

La méthode getSingleResult est utilisée pour exécuter une requête qui retourne un seul résultat

```
Long count = query.getSingleResult();
```

2. Requête avec Filtrage

```
public List<Employee> getEmployeesByDepartment(String department) {  
    TypedQuery<Employee> query =  
  
        em.createQuery("SELECT e FROM Employee e WHERE e.department = :department", Employee.class);  
        query.setParameter("department", department);  
        return query.getResultList();  
}
```

3. Requête avec Jointure

```
public List<Employee> getEmployeesWithDepartmentName() {  
    TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e JOIN e.department d  
        WHERE d.name = :deptName", Employee.class);  
    query.setParameter("deptName", "Sales");  
    return query.getResultList();  
}
```

Fonctionnalités Avancées de JPQL Fonctions d'agrégation :

COUNT, SUM, AVG, MAX, MIN

```
public Long getEmployeeCount() {  
    TypedQuery<Long> query = em.createQuery("SELECT COUNT(e) FROM Employee e", Long.class);  
    return query.getSingleResult();  
}
```

Groupement et Tri : Utilisation de GROUP BY et ORDER BY

```
public List<Object[]> getEmployeeCountByDepartment() {  
    TypedQuery<Object[]> query = em.createQuery("SELECT e.department, COUNT(e) FROM Employee e  
GROUP BY e.department", Object[].class);  
    return query.getResultList();  
}
```

Sous-requêtes : Utilisation de sous-requêtes pour des requêtes plus complexes

```
public List<Employee> getEmployeesInLargeDepartments()
{
    TypedQuery<Employee> query =
        em.createQuery("SELECT e FROM Employee e WHERE e.department IN
        (SELECT d FROM Department d WHERE d.size > 50)", Employee.class);
    return query.getResultList();
}
```

Utilisation de JPQL en Java

Pour exécuter des requêtes JPQL en Java, nous utilisons l'interface EntityManager pour créer et exécuter les requêtes.

```
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.persistence.TypedQuery;
import java.util.List;

public class EmployeeService {

    @PersistenceContext
    private EntityManager em;

    public List<Employee> getAllEmployees() {
        TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e",
Employee.class);
        return query.getResultList();
    }

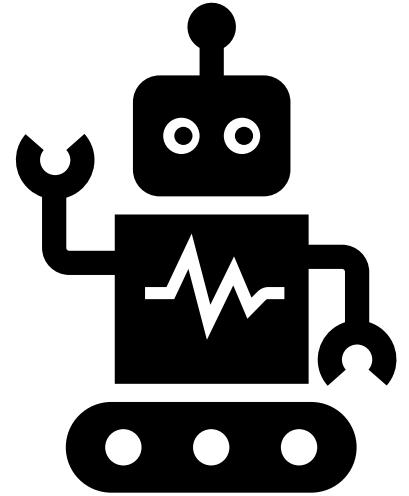
    public List<Employee> getEmployeesByDepartment(String department) {
        TypedQuery<Employee> query = em.createQuery("SELECT e FROM Employee e WHERE
e.department = :department", Employee.class);
        query.setParameter("department", department);
    }
}
```

Requêtes Dynamiques

JPQL permet également de créer des requêtes dynamiques en combinant des parties de requêtes

```
String baseQuery = "SELECT e FROM Employee e";
if (department != null) {
    baseQuery += " WHERE e.department = :department";
}
TypedQuery<Employee> query = em.createQuery(baseQuery, Employee.class);
if (department != null) {
    query.setParameter("department", department);
}
List<Employee> results = query.getResultList();
```

Sécurité, Transactions et API REST



What are Transactions?

Une transaction est une unité de travail qui est exécutée de manière entièrement ou non. Autrement dit, une transaction doit être complétée avec succès dans son intégralité, sinon aucune de ses parties ne doit être appliquée. Les transactions sont essentielles pour maintenir l'intégrité et la cohérence des données dans les applications, notamment celles qui interagissent avec des bases de données.

Principe ACID

Les transactions respectent généralement les propriétés ACID :

- 1. Atomicité** : La transaction est tout ou rien. Si une partie de la transaction échoue, la transaction entière échoue et l'état du système revient à son état initial.
- 2. Cohérence** : Une transaction apporte le système d'un état cohérent à un autre état cohérent.
- 3. Isolation** : Les transactions concurrentes s'exécutent de manière isolée les unes des autres.
- 4. Durabilité** : Une fois que la transaction est validée, ses modifications sont permanentes et persistantes, même en cas de panne système.

Transactions en Jakarta EE

En Jakarta EE, les transactions sont principalement gérées par le conteneur de manière transparente pour le développeur. Jakarta EE utilise JTA (Java Transaction API) pour la gestion des transactions. Il y a deux types principaux de gestion des transactions :

1. Transactions gérées par le conteneur (CMT - Container Managed Transactions)

:

1. Le conteneur Jakarta EE gère automatiquement les frontières transactionnelles.
2. On utilise des annotations pour déclarer les exigences transactionnelles.

2. Transactions gérées par le bean (BMT - Bean Managed Transactions) :

1. On a un contrôle total sur le début, la validation et l'annulation des transactions.
2. Utilisation explicite de l'API de transaction pour gérer les transactions.

Transactions Gérées par le Conteneur (CMT)

Les transactions gérées par le conteneur sont les plus couramment utilisées car elles simplifient le développement.

Le conteneur gère automatiquement le début, la validation et l'annulation des transactions.

```
import jakarta.ejb.Stateless;
import jakarta.transaction.Transactional;

@Stateless
public class TransactionalBean {

    @Transactional
    public void performTransactionalOperation() {
        // Code transactionnel
    }
}
```

Dans cet exemple, le conteneur démarre automatiquement une transaction lorsqu'il entre dans la méthode `performTransactionalOperation` et la valide lorsqu'il en sort, sauf si une exception est lancée.

Transactions Gérées par le Bean (BMT)

Les transactions gérées par le bean donnent plus de contrôle au développeur. Le développeur doit explicitement démarrer, valider et annuler les transactions.

```
import jakarta.ejb.Stateless;
import jakarta.transaction.UserTransaction;
import jakarta.annotation.Resource;

@Stateless
public class ManualTransactionBean {

    @Resource
    private UserTransaction utx;

    public void performOperation() {
        try {
            utx.begin();
            // Code transactionnel
            utx.commit();
        } catch (Exception e) {
            try {
                utx.rollback();
            } catch (Exception rollbackEx) {
                // Gestion de l'erreur de rollback
            }
        }
    }
}
```

Dans cet exemple, la méthode `performOperation` démarre explicitement une transaction avec `utx.begin()`, exécute du code transactionnel, et valide la transaction avec `utx.commit()`. Si une exception est levée, la transaction est annulée avec `utx.rollback()`

Gestion des Transactions dans les Applications Web

Dans les applications web Jakarta EE, les transactions sont souvent gérées automatiquement par le conteneur lorsqu'elles sont associées à des composants EJB (Enterprise JavaBeans). Cependant, elles peuvent également être gérées dans des servlets ou des beans de gestion en utilisant des annotations de transactions ou des API de transactions.

Exemple d'application web utilisant des transactions

Supposons que nous ayons une application web où nous souhaitons gérer les transactions pour l'ajout d'un nouvel employé :

```
import java.io.IOException;

@WebServlet("/addEmployee")
public class AddEmployeeServlet extends
HttpServlet {

    @EJB
    private EmployeeService employeeService;

    protected void doPost(HttpServletRequest
request, HttpServletResponse response)
            throws ServletException,
IOException {
        String name =
request.getParameter("name");
        String department =
request.getParameter("department");

        try {
            employeeService.addEmployee(name,
department);
            response.getWriter().write("Employee added successfully!");
        } catch (Exception e) {
            response.sendError(HttpServletRequest
```

```
import jakarta.ejb.Stateless;
import jakarta.persistence.EntityManager;
import jakarta.persistence.PersistenceContext;
import jakarta.transaction.Transactional;

@Stateless
public class EmployeeService {

    @PersistenceContext
    private EntityManager em;

    @Transactional
    public void addEmployee(String name, String department) {
        Employee employee = new Employee();
        employee.setName(name);
        employee.setDepartment(department);
        em.persist(employee);
    }
}
```

Dans cet exemple, l'ajout d'un nouvel employé est géré transactionnellement par le conteneur grâce à l'annotation `@Transactional` sur la méthode `addEmployee`.

Les transactions en Jakarta EE sont essentielles pour assurer l'intégrité et la cohérence des données. Elles peuvent être gérées automatiquement par le conteneur, offrant ainsi une simplicité et une sécurité accrues, ou manuellement par le développeur pour un contrôle plus précis. Comprendre comment utiliser et configurer les transactions est crucial pour le développement d'applications d'entreprise robustes et fiables.

CMT Transactions Management Attributes

En Jakarta EE, les transactions gérées par le conteneur (CMT) permettent de déléguer la gestion des transactions au conteneur, simplifiant ainsi le développement d'applications.

Le conteneur gère automatiquement les débuts, les validations et les annulations des transactions. Plusieurs attributs peuvent être utilisés pour configurer le comportement transactionnel d'un EJB.

Attributs de Gestion des Transactions CMT

- 1.REQUIRED
- 2.REQUIRES_NEW
- 3.MANDATORY
- 4.NOT_SUPPORTED
- 5.SUPPORTS
- 6.NEVER

1. REQUIRED

Cet attribut est le plus couramment utilisé. S'il existe déjà une transaction, la méthode s'exécutera dans cette transaction. Sinon, une nouvelle transaction sera démarrée

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;

@Stateless
public class ExampleBean {

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void doWork() {
        // Code qui s'exécute dans une transaction
    }
}
```

REQUIRES_NEW

Cet attribut crée toujours une nouvelle transaction, suspendant toute transaction existante. La nouvelle transaction est indépendante de la transaction précédente.

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;

@Stateless
public class ExampleBean {

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void doWork() {
        // Code qui s'exécute dans une nouvelle transaction
    }
}
```

3. MANDATORY

Cet attribut exige qu'une transaction soit déjà en cours. Si aucune transaction n'est en cours, une exception sera levée.

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;

@Stateless
public class ExampleBean {

    @TransactionAttribute(TransactionAttributeType.MANDATORY)
    public void dowork() {
        // Code qui exige une transaction existante
    }
}
```

4. NOT_SUPPORTED

Cet attribut indique que la méthode ne doit pas s'exécuter dans une transaction. Si une transaction existe, elle sera suspendue pendant l'exécution de la méthode.

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;

@Stateless
public class ExampleBean {

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public void doWork() {
        // Code qui s'exécute sans transaction
    }
}
```

5. SUPPORTS

Cet attribut indique que la méthode peut s'exécuter dans une transaction si une transaction existe, mais elle n'en créera pas de nouvelle si aucune transaction n'est en cours.

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;

@Stateless
public class ExampleBean {

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public void doWork() {
        // Code qui s'exécute sans transaction
    }
}
```

6. NEVER

Cet attribut indique que la méthode ne doit jamais s'exécuter dans une transaction. Si une transaction est en cours, une exception sera levée

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;

@Stateless
public class ExampleBean {

    @TransactionAttribute(TransactionAttributeType.NEVER)
    public void doWork() {
        // Code qui s'exécute sans transaction, exception si une transaction existe
    }
}
```

Utilisation des Attributs dans un Scénario Réel

Contexte

Supposons que nous ayons une application de gestion de commandes où :

- Nous validons la commande (`validateOrder`) dans une transaction existante.
- Nous créons une nouvelle transaction pour le paiement (`processPayment`).
- Nous enregistrons les logs sans transaction (`logOrder`).

```
import jakarta.ejb.Stateless;
import jakarta.ejb.TransactionAttribute;
import jakarta.ejb.TransactionAttributeType;

@Stateless
public class OrderService {

    @TransactionAttribute(TransactionAttributeType.REQUIRED)
    public void validateOrder(Order order) {
        // Validation de la commande dans une transaction existante
    }

    @TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
    public void processPayment(Payment payment) {
        // Traitement du paiement dans une nouvelle transaction
    }

    @TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
    public void logOrder(Order order) {
        // Enregistrement des logs sans transaction
    }
}
```

Scénario Global

- 1. Validation de la Commande (validateOrder) :** Cette méthode est responsable de vérifier que la commande est valide avant de procéder.
- 2. Traitement du Paiement (processPayment) :** Cette méthode traite le paiement de la commande.
- 3. Enregistrement des Logs (logOrder) :** Cette méthode enregistre des informations sur la commande pour le suivi et l'audit.

```
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public void validateOrder(Order order) {
    // Validation de la commande dans une transaction existante
}
```

- **TransactionAttributeType.REQUIRED** : Cette méthode s'exécute dans une transaction existante si elle est présente. Sinon, une nouvelle transaction est démarrée.
- **Scénario** : Si validateOrder est appelée dans le cadre d'une transaction plus large, elle utilisera cette transaction. Si elle est appelée indépendamment, une nouvelle transaction sera créée.
- **Implication** : Cela garantit que toutes les validations de commande sont atomiques et qu'elles font partie d'une transaction cohérente.

```
@TransactionAttribute(TransactionAttributeType.REQUIRES_NEW)
public void processPayment(Payment payment) {
    // Traitement du paiement dans une nouvelle transaction
}
```

- **TransactionAttributeType.REQUIRES_NEW** : Cette méthode crée toujours une nouvelle transaction, suspendant toute transaction existante.
- **Scénario** : Indépendamment de toute transaction en cours, processPayment s'exécute dans une nouvelle transaction.
- **Implication** : Cela permet d'isoler le traitement du paiement des autres opérations transactionnelles. Si le paiement échoue, seule cette transaction sera annulée, sans affecter d'autres transactions en cours

```
@TransactionAttribute(TransactionAttributeType.NOT_SUPPORTED)
public void logOrder(Order order) {
    // Enregistrement des logs sans transaction
}
```

- **TransactionAttributeType.NOT_SUPPORTED** : Cette méthode s'exécute en dehors de toute transaction. Si une transaction est en cours, elle sera suspendue pendant l'exécution de cette méthode.
- **Scénario** : logOrder est utilisée pour enregistrer des informations sur la commande sans être impliquée dans une transaction.
- **Implication** : Cela garantit que l'enregistrement des logs n'affecte pas les transactions en cours et n'est pas annulé même si la transaction principale échoue

Implications et Bénéfices

1. Isolation des Transactions :

- En utilisant **REQUIRES_NEW** pour le paiement, le traitement du paiement est isolé des autres opérations, ce qui permet de gérer les échecs de paiement de manière indépendante.

2. Flexibilité Transactionnelle :

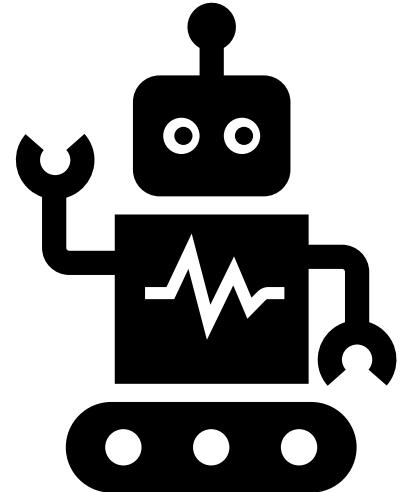
- Avec REQUIRED, validateOrder s'adapte au contexte transactionnel existant, garantissant une intégrité transactionnelle lorsqu'elle est appelée dans le cadre d'une transaction plus large.

3. Logs Fiables :

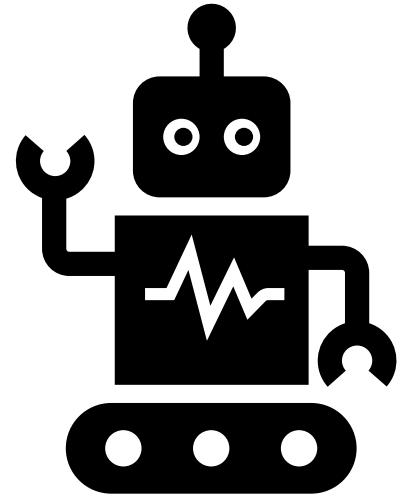
- NOT_SUPPORTED pour logOrder garantit que les opérations de logging ne sont pas influencées par les transactions et sont toujours exécutées, même en cas d'échec de la transaction principale.

Les attributs de gestion des transactions CMT en Jakarta EE permettent une flexibilité considérable dans la gestion des transactions. Ils permettent aux développeurs de spécifier facilement comment et quand les transactions doivent être utilisées, tout en déléguant la complexité de la gestion des transactions au conteneur. En utilisant ces attributs correctement, nous pouvons garantir l'intégrité et la cohérence de vos opérations transactionnelles dans vos applications d'entreprise.

Java API for Rest Web Services (JAX-RS 2.1)



Java Message Service (JMS) et JavaMail



Jakarta Messaging (JMS)

Jakarta Messaging (anciennement Java Message Service - JMS) est une API Java qui permet aux applications de créer, envoyer, recevoir et lire des messages. Elle permet la communication entre différentes composantes d'une application distribuée et est utilisée pour obtenir une communication asynchrone.

Concepts clés :

- 1. Message** : Les données transférées entre applications. Les messages peuvent contenir différents types de charges utiles, comme du texte, des objets ou des flux de bytes.
- 2. Queue (File d'attente)** : Un modèle de messagerie point-à-point où chaque message est envoyé à une file d'attente spécifique. Chaque message est reçu et traité par un seul consommateur.
- 3. Topic (Sujet)** : Un modèle de publication-abonnement où les messages sont diffusés à plusieurs consommateurs. Chaque message publié à un sujet est reçu par tous les abonnés actifs.
- 4. ConnectionFactory** : Un objet administré utilisé par un client pour créer des connexions avec un fournisseur JMS.
- 5. Destination** : La cible des messages envoyés ou reçus, représentée par une file d'attente ou un sujet.
- 6. Connection** : Une connexion active avec le broker de messages.
- 7. Session** : Un contexte à thread unique pour produire et consommer des messages.
- 8. Producer (Producteur)** : Un objet qui envoie des messages à une destination.
- 9. Consumer (Consommateur)** : Un objet qui reçoit des messages d'une destination

Exemple d'utilisation :

```
// Obtenir une connexion JNDI
InitialContext ctx = new InitialContext();
QueueConnectionFactory qcf = (QueueConnectionFactory) ctx.lookup("jms/QueueConnectionFactory");
Queue queue = (Queue) ctx.lookup("jms/Queue");

// Créer une connexion
QueueConnection qc = qcf.createQueueConnection();
QueueSession qs = qc.createQueueSession(false, Session.AUTO_ACKNOWLEDGE);

// Créer un producteur
QueueSender sender = qs.createSender(queue);

// Créer et envoyer un message
TextMessage message = qs.createTextMessage();
message.setText("Ceci est un message de notification.");
sender.send(message);

// Fermer la connexion
sender.close();
qs.close();
qc.close();
```

Jakarta Mail

Vue d'ensemble :

Jakarta Mail (anciennement JavaMail) est une API utilisée pour envoyer et recevoir des courriels à partir d'applications Java. Elle supporte plusieurs protocoles de messagerie comme SMTP, POP3 et IMAP.

Concepts clés :

1.Session : Une session de messagerie qui contient des propriétés et des informations de configuration sur le serveur de messagerie.

2.Message : L'objet qui représente un email.

3.Transport : Un objet utilisé pour envoyer des messages.

4.Store : Un objet utilisé pour recevoir des messages.

5.Folder : Un objet qui représente un dossier de messages dans une boîte de réception

Exemple d'utilisation :

```
// Définir les propriétés de la session
Properties props = new Properties();
props.put("mail.smtp.host", "smtp.example.com");
props.put("mail.smtp.port", "587");
props.put("mail.smtp.auth", "true");
props.put("mail.smtp.starttls.enable", "true");

// Obtenir la session de messagerie
Session session = Session.getInstance(props, new javax.mail.Authenticator() {
    protected PasswordAuthentication getPasswordAuthentication() {
        return new PasswordAuthentication("username", "password");
    }
});

try {
    // Créer un message
    Message message = new MimeMessage(session);
    message.setFrom(new InternetAddress("from@example.com"));
    message.setRecipients(Message.RecipientType.TO, InternetAddress.parse("to@example.com"));
    message.setSubject("Sujet de l'email");
    message.setText("Ceci est le contenu de l'email.");

    // Envoyer le message
}
```

