J2EE

Java EE

Jakarta EE

I. Partie I

A) Introduction :

J2EE (Java 2 Enterprise Edition) a été lancé en 1999 comme une plateforme dédiée au développement d'applications d'entreprise. Cette édition s'appuie sur J2SE (Java 2 Standard Edition lui meme appelé Java SE), qui contient les API de base du langage Java.

En 2006, avec la sortie de la version 5, J2EE a été renommée Java EE (Enterprise Edition), reflétant une mise à jour majeure et un élargissement des capacités pour les entreprises.

En 2017, Oracle a transféré Java EE à la Fondation Eclipse, et en 2018, le projet a été rebaptisé Jakarta EE. Depuis lors, Jakarta EE continue d'évoluer, fournissant des outils modernes pour les applications d'entreprise dans un environnement open-source.

En effet , Java EE (Java Platform, Enterprise Edition), désormais connue sous le nom de Jakarta EE, est une plateforme de développement pour la création d'applications d'entreprise robustes, sécurisées et évolutives. Elle fournit un ensemble d'API et de spécifications qui facilitent le développement d'applications server-side, comme les services web, les systèmes de gestion d'entreprise, les applications de commerce électronique, et plus encore.

1. Attention ! Java EE n’est pas Java SE :

Java EE (Java Platform, Enterprise Edition), aujourd'hui appelée Jakarta EE, n'est pas simplement "Java", mais une extension de Java standard (JSE ou Java SE, Standard Edition) spécialement conçue pour développer des applications d'entreprise à grande échelle. Alors que Java SE est le socle de base de la plateforme Java, comprenant les fonctionnalités essentielles du langage comme les bibliothèques de classes de base, les entrées/sorties, les collections, et le multithreading, Java EE étend ces fonctionnalités pour répondre aux besoins spécifiques des applications d'entreprise. Voici pourquoi Java EE se distingue de Java SE :

2. Différences entre Java SE et Java EE :

➔ Java SE (Standard Edition) :

Utilisation : Java SE est utilisé pour développer des applications de bureau, des jeux, des utilitaires ou des applications autonomes.
Fonctionnalités de base : Il inclut des classes fondamentales pour des opérations comme la gestion des fichiers, le traitement des données, la gestion de la mémoire, les threads, les structures de données et les interfaces utilisateur de base.
Applications autonomes : Les programmes écrits avec Java SE peuvent s'exécuter indépendamment, sans serveur ou infrastructure complexe.

➔ Java EE (Enterprise Edition) :

Utilisation : Java EE est utilisé pour développer des applications d'entreprise complexes qui nécessitent une gestion avancée des transactions, de la sécurité, de la concurrence et des interactions avec des systèmes distribués ou basés sur le cloud.
Fonctionnalités supplémentaires : Java EE fournit des API supplémentaires et des frameworks conçus pour les applications web, les transactions, la messagerie, la persistance de données, la sécurité, et l'intégration avec d'autres systèmes. Ces extensions sont essentielles pour les grandes entreprises et les organisations qui ont des systèmes complexes et doivent gérer des volumes importants de données.
Architecture client-serveur : Java EE supporte des architectures multi-couches où les composants d'application (logique métier, présentation et données) sont déployés sur différents serveurs pour assurer évolutivité et fiabilité.
Applications orientées serveur : Contrairement à Java SE, les applications Java EE nécessitent souvent un serveur d'applications (comme Apache TomEE, WildFly ou GlassFish) pour héberger les composants. Cela permet de gérer les utilisateurs simultanés, les transactions, les sessions, la sécurité, etc.

3. Est-claire que Java EE n’est pas simplement Java ?

Sinon , voici plus de détaille :

➔ **Portée étendue** : Java EE est conçu pour gérer les besoins d'une application à grande échelle, tandis que Java SE se concentre sur les applications locales ou plus petites. Les spécifications de Java EE répondent à des besoins d’interopérabilité, de performance, et de sécurité qui vont au-delà des applications de bureau ou individuelles.

➔ **Infrastructure serveur** : Java EE est souvent déployé sur des serveurs d'applications qui fournissent un environnement d'exécution pour ces composants avancés. Ce contexte

serveur permet de gérer la montée en charge, la tolérance aux pannes et la gestion de nombreuses connexions simultanées.

➔ **Composants avancés** : Des composants comme les EJB ou les services web sont nécessaires pour construire des systèmes d'entreprise qui nécessitent une gestion complexe des transactions, de la sécurité et de l'intégration avec d'autres systèmes.

4. Exemple concret :

Imagine une application de gestion bancaire. Avec Java SE, tu pourrais développer un programme simple pour effectuer des opérations de base (comme l'ajout ou la soustraction de montants dans des comptes). Mais avec Java EE, tu pourrais créer une architecture complète qui gère les opérations bancaires à grande échelle, où des centaines ou des milliers d'utilisateurs peuvent simultanément faire des transactions sécurisées, avec des règles métier complexes (gestion des prêts, autorisation des crédits, etc.) tout en assurant la robustesse et la fiabilité de chaque opération.

5. Pay atTentioN Java EE n'est pas JavaScript

Java EE (Java Platform, Enterprise Edition) et JavaScript sont souvent confondus en raison de leurs noms similaires, mais ce sont deux technologies totalement distinctes. Voici pourquoi Java EE n'est pas JavaScript :

✔ **Langages différents :**
Java EE est une extension du langage de programmation Java, utilisé pour développer des applications côté serveur, notamment des applications d'entreprise à grande échelle. Java est un langage orienté objet qui doit être compilé et exécuté sur une machine virtuelle (JVM).
JavaScript, en revanche, est un langage de programmation interprété principalement utilisé pour les applications web, côté client (navigateur). Il est utilisé pour rendre les pages web interactives (exemples : animations, validation de formulaires, etc.).

✔ **Utilisations différentes :**
Java EE est principalement utilisé pour le développement d'applications d'entreprise (serveurs d'applications, gestion des bases de données, sécurité, etc.) et nécessite souvent des serveurs comme GlassFish ou Apache TomEE pour fonctionner. Il permet de créer des applications complexes qui gèrent de multiples utilisateurs, des transactions, des sessions et plus encore.
JavaScript est principalement utilisé pour améliorer l'expérience utilisateur dans les pages web. Il est exécuté directement dans le navigateur pour rendre les sites plus dynamiques. Avec l'arrivée de Node.js, JavaScript est également utilisé côté serveur, mais son rôle

d'origine reste le développement d'interfaces interactives côté client.

- ✔ **Exécution :**
  Java EE s'exécute sur une machine virtuelle Java (JVM) après compilation. Le code Java est compilé en bytecode, qui est ensuite exécuté par la JVM sur différents systèmes d'exploitation.
  JavaScript s'exécute directement dans un navigateur web ou sur une machine via un moteur JavaScript (comme V8 pour Chrome ou SpiderMonkey pour Firefox) sans besoin de compilation. Il est interprété à la volée par le navigateur.
  Syntaxe :
  Java EE utilise la syntaxe de Java, qui est fortement typée et nécessite la déclaration explicite des types de variables et d'objets.
  JavaScript a une syntaxe beaucoup plus flexible, avec un typage dynamique, ce qui signifie que les variables n'ont pas besoin d'être déclarées avec un type spécifique.

- ✔ **Ecosystème :**
  Java EE fait partie d'un écosystème serveur orienté entreprise, avec des technologies comme EJB (Enterprise JavaBeans), JPA (Java Persistence API), et JMS (Java Messaging Service). Il est utilisé pour des systèmes back-end lourds.
  JavaScript est un élément clé du développement web front-end, en collaboration avec des technologies comme HTML et CSS pour créer des interfaces utilisateur. Avec des frameworks comme React, Angular, ou Vue.js, il est très populaire dans le développement web moderne.
- ✔ **Exemple concret :**
  Java EE serait utilisé pour gérer la logique métier d'un système de réservation en ligne, en prenant en charge les transactions, la persistance des données, et la gestion des utilisateurs sur le serveur.
  JavaScript serait utilisé sur le front-end de cette même application pour créer une interface utilisateur interactive, où les utilisateurs peuvent cliquer sur des boutons, faire défiler des menus ou remplir des formulaires, qui réagissent en temps réel.

6. Fonctionnement de Java EE

Java EE (désormais Jakarta EE) fonctionne comme une plateforme qui fournit un ensemble de spécifications et de services permettant de construire des applications d'entreprise robustes, distribuées et évolutives. Son architecture repose sur des composants modulaires et une infrastructure serveur pour faciliter le développement d'applications web et d'entreprise. Voici une explication du fonctionnement de Java EE :

- ■ **Architecture de Java EE**
  Java EE suit une architecture multi-couches ou multi-tier, où les différentes parties de l'application sont séparées en couches distinctes. Cela permet une meilleure modularité, réutilisabilité et scalabilité. Les principales couches sont :

Côté client (Client Tier) : La partie visible par l'utilisateur, comme les interfaces utilisateur (UI) développées avec des technologies comme HTML, CSS, JavaScript, ou même des applications Java autonomes.

Côté web (Web Tier) : Reçoit et traite les requêtes HTTP des utilisateurs. Elle contient les composants web comme les Servlets et les JavaServer Pages (JSP), qui génèrent dynamiquement les réponses envoyées au client.

Côté entreprise (Business Tier) : La logique métier réside ici. Des composants comme les Enterprise JavaBeans (EJB) traitent les processus métier, les règles métier et la gestion des transactions. C'est le cœur de l'application.

Côté intégration (Persistence Tier) : Gère la persistence des données dans les bases de données ou autres systèmes de stockage. Les données sont manipulées avec des API comme JPA (Java Persistence API), qui permet de mapper les objets Java aux tables de bases de données relationnelles.

Ces couches communiquent entre elles via des API standardisées et des protocoles comme HTTP, RMI (Remote Method Invocation), ou encore JMS (Java Message Service) pour la messagerie asynchrone.

- **Composants clés de Java EE**
  Java EE repose sur plusieurs composants spécifiques qui facilitent le développement d'applications distribuées et complexes :

  Servlets : Gèrent les requêtes et réponses HTTP. Ils sont généralement utilisés pour créer des applications web dynamiques.

  JSP (JavaServer Pages) : Similaire aux servlets, mais permettent d'incorporer directement du code Java dans des pages HTML, facilitant la création d'interfaces dynamiques.

  EJB (Enterprise JavaBeans) : Ce sont des composants serveur qui encapsulent la logique métier. Ils sont utilisés pour gérer les transactions, la sécurité, le multithreading et la persistance.

  JPA (Java Persistence API) : Fournit un moyen de gérer la persistance des objets Java dans des bases de données relationnelles, en facilitant l'interaction avec les bases de données sans avoir à écrire des requêtes SQL complexes.

  JMS (Java Message Service) : Permet la communication asynchrone entre applications via un système de messagerie. C'est utile dans les systèmes distribués où les applications échangent des messages de manière fiable.

JAX-RS et JAX-WS : Utilisés pour créer des services web RESTful (JAX-RS) ou SOAP (JAX-WS), facilitant ainsi l'interopérabilité entre différentes applications via des protocoles web.

- **Serveur d'applications Java EE**
  Java EE fonctionne généralement dans un serveur d'applications qui prend en charge les différents services et composants de la plateforme. Un serveur d'applications Java EE (comme GlassFish, WildFly, ou Apache TomEE) fournit l'infrastructure nécessaire pour exécuter les composants Java EE, comme les servlets, les EJB, et les services de messagerie. Voici ses rôles :

  Gestion des connexions client-serveur : Le serveur d'applications gère les connexions entrantes des clients (via HTTP ou d'autres protocoles) et distribue les requêtes aux composants appropriés.

  Gestion des transactions : Il fournit un gestionnaire de transactions pour garantir l'intégrité des opérations dans les bases de données et entre différents systèmes.

  Sécurité : Le serveur d'applications gère l'authentification, l'autorisation et la protection des ressources à l'aide de JASPIC (Java Authentication Service Provider Interface for Containers) ou d'autres mécanismes de sécurité.

  Gestion des ressources : Il gère les connexions aux bases de données et autres systèmes externes (comme les services web), optimisant ainsi l'utilisation des ressources grâce à des pools de connexions.

  Tolérance aux pannes et scalabilité : Les serveurs Java EE sont conçus pour être hautement disponibles et évolutifs, ce qui permet de distribuer les composants sur plusieurs serveurs et de gérer des milliers d'utilisateurs simultanés.

- **Cycle de vie d'une requête dans Java EE**
  Client : L'utilisateur fait une requête, par exemple en accédant à une page web via un navigateur.
  Côté Web : La requête est envoyée au serveur web, où elle est traitée par un servlet ou une page JSP.
  Logique métier : Le servlet appelle des composants dans la couche métier (par exemple, un EJB) pour appliquer la logique métier, gérer les transactions ou effectuer des calculs.
  Accès aux données : Si l'application doit accéder à une base de données, elle fait appel à JPA pour récupérer ou sauvegarder des informations dans une base de données.
  Réponse : Les données traitées sont envoyées en réponse au client, souvent sous forme de pages HTML générées dynamiquement par le serveur, ou d'une réponse JSON/XML dans le cas d'un service web.

Session : Le serveur peut gérer des sessions utilisateur pour maintenir l'état entre les requêtes successives d'un même utilisateur.

- **Exemple concret d'une application Java EE**
  Imagine que tu développes une application de commerce électronique. Voici comment Java EE interviendrait :

  Côté client : Un utilisateur passe une commande via une interface web.
  Côté web : Un servlet reçoit la requête d'achat, récupère les informations de l'utilisateur et les articles du panier.
  Côté entreprise : Un EJB gère les règles métiers (vérifie les stocks, applique les promotions, calcule les frais de livraison, etc.).
  Côté données : JPA interagit avec la base de données pour vérifier la disponibilité des articles et stocker les informations de la commande.
  Côté messagerie : Une notification d'achat peut être envoyée au client via un système de messagerie basé sur JMS.

7. Le Modele MVC

L'architecture MVC (Modèle-Vue-Contrôleur) est un modèle de conception couramment utilisé dans les applications Java EE (ou Jakarta EE) pour séparer les responsabilités dans une application, rendant le code plus modulaire, maintenable et évolutif. Elle divise une application en trois parties principales :

Modèle (Model) : Gère les données et la logique métier.
Vue (View) : Représente la partie visuelle, c'est-à-dire l'interface utilisateur.
Contrôleur (Controller) : Gère les interactions de l'utilisateur et met à jour le modèle et la vue en conséquence.
Voici comment l'architecture MVC s'intègre dans une application Java EE.

1. Présentation de l'architecture MVC
Modèle (Model) :
Il représente les données de l'application ainsi que les règles métier. Dans une application Java EE, le modèle peut inclure des EJB (Enterprise JavaBeans), des classes Java ordinaires (POJOs) qui encapsulent la logique métier, ou encore des entités JPA pour la gestion des bases de données.
Vue (View) :
Il s'agit de la représentation visuelle des données que voit l'utilisateur, généralement des pages HTML ou JSP (JavaServer Pages) qui affichent des données envoyées par le contrôleur. Dans Java EE, des technologies comme JSP, Facelets (dans JavaServer Faces - JSF) ou des frameworks front-end modernes (React, Angular) peuvent être utilisées pour créer la vue.
Contrôleur (Controller) :
Le contrôleur est responsable de gérer les requêtes des utilisateurs, de mettre à jour le modèle et de choisir la vue à afficher. Dans Java EE, cela peut être géré par des Servlets, des Managed Beans (avec JSF), ou des contrôleurs dans des frameworks comme Spring MVC ou Struts.
2. Fonctionnement de l'architecture MVC dans Java EE

Voici un flux général du fonctionnement d'une application MVC dans Java EE :

Étape 1 : Requête de l'utilisateur
L'utilisateur fait une requête (soumet un formulaire, accède à une page, etc.) via son navigateur. Cette requête est capturée par le Contrôleur, qui peut être un Servlet ou un Managed Bean dans JSF.
Étape 2 : Traitement dans le Contrôleur
Le Contrôleur analyse la requête, interagit avec le Modèle pour récupérer ou mettre à jour des données.
Par exemple, il peut appeler des EJB ou des objets JPA pour récupérer des informations depuis une base de données ou pour appliquer des règles métier.
Étape 3 : Mise à jour du Modèle
Le Modèle est mis à jour en fonction des actions de l'utilisateur ou de la logique métier. Par exemple, une entité JPA peut être modifiée ou de nouvelles données peuvent être ajoutées dans la base de données.
Étape 4 : Sélection de la Vue
Le Contrôleur choisit la Vue à afficher à l'utilisateur. Il peut transmettre les données du Modèle à la Vue sous forme d'attributs, pour que ces informations soient affichées à l'utilisateur.
Étape 5 : Rendu de la Vue
La Vue (souvent une page JSP, Facelets, ou toute autre technologie front-end) est générée dynamiquement avec les données transmises par le contrôleur et est envoyée à l'utilisateur.
Étape 6 : Réponse à l'utilisateur
L'utilisateur reçoit la Vue mise à jour, sous forme de page web, et peut interagir avec elle pour soumettre de nouvelles requêtes, démarrant un nouveau cycle MVC.
3. Technologies Java EE utilisées dans chaque composant MVC
a. Modèle (Model) :
Le modèle encapsule les données et la logique métier, et dans Java EE, cela peut inclure :

EJB (Enterprise JavaBeans) : Composants serveur qui implémentent la logique métier et gèrent des fonctionnalités comme les transactions, la sécurité et la persistance.
JPA (Java Persistence API) : Utilisée pour la gestion des données et l'accès aux bases de données relationnelles. Elle permet de mapper des objets Java à des tables de bases de données.
POJOs (Plain Old Java Objects) : Des objets Java simples qui représentent les entités métier et contiennent la logique métier, souvent utilisés avec des frameworks comme Spring.
b. Vue (View) :
La vue représente l'interface utilisateur et, en Java EE, elle peut être implémentée avec :

JSP (JavaServer Pages) : Utilisé pour générer des pages HTML dynamiquement avec des données issues du modèle. JSP permet l'incorporation de balises Java et de JavaBeans dans les pages.
JavaServer Faces (JSF) : Un framework Java EE pour la création d'interfaces utilisateur basées sur des composants. Il utilise des Managed Beans pour la logique et des pages Facelets pour le rendu de la vue.
Frameworks modernes front-end : Des frameworks comme Angular, React, ou Vue.js peuvent être utilisés pour des vues plus riches, en interaction avec des services REST ou SOAP fournis par l'application Java EE.
c. Contrôleur (Controller) :
Le contrôleur gère la logique de navigation et la coordination entre le modèle et la vue. Dans Java

EE, cela peut inclure :

Servlets : Ce sont des composants côté serveur qui interceptent les requêtes HTTP, traitent les données et envoient une réponse appropriée. Ils peuvent agir comme contrôleurs dans des applications Java EE basées sur MVC.
Managed Beans (avec JSF) : Des objets managés par le framework JSF, qui agissent comme contrôleurs pour les actions des utilisateurs. Les Managed Beans contrôlent les interactions entre la vue (Facelets) et le modèle (EJB ou JPA).

4. Exemple d'une application MVC dans Java EE
Supposons que nous développons une application Java EE simple où l'utilisateur soumet un formulaire de connexion. Voici comment l'architecture MVC serait implémentée :

Vue (View) : Une page JSP ou JSF est utilisée pour afficher le formulaire de connexion. L'utilisateur entre ses informations de connexion (nom d'utilisateur et mot de passe).

Contrôleur (Controller) : Un Servlet ou un Managed Bean reçoit la requête du formulaire soumis. Il vérifie les informations d'identification en appelant le modèle.

Modèle (Model) : Le contrôleur appelle un EJB qui contient la logique métier pour vérifier les informations de connexion dans une base de données via JPA.

Vue (View) : Après validation, une nouvelle vue est affichée (par exemple, une page de tableau de bord si l'utilisateur est authentifié avec succès, ou un message d'erreur sinon).

5. Avantages de l'architecture MVC dans Java EE
Séparation des responsabilités : Le modèle gère la logique métier, la vue gère l'interface utilisateur, et le contrôleur fait le lien entre les deux. Cela permet de mieux organiser le code et de le rendre plus maintenable.

Réutilisabilité : Le modèle et le contrôleur peuvent être réutilisés avec différentes vues, permettant de changer l'interface utilisateur sans toucher à la logique métier.

Facilité de maintenance : Chaque composant est indépendant des autres, ce qui rend le débogage et la mise à jour plus faciles.

Modularité : Le découplage des composants améliore la modularité de l'application, facilitant son évolution et l'ajout de nouvelles fonctionnalités.

8. Serveur d’Application

Les serveurs d'applications Java EE (ou Jakarta EE) jouent un rôle essentiel dans l'exécution des applications d'entreprise en fournissant une infrastructure complète qui gère les composants Java EE comme les servlets, les EJB (Enterprise JavaBeans), et les services web. Ces serveurs sont conçus pour prendre en charge les applications distribuées, en garantissant des fonctionnalités telles que la gestion des transactions, la sécurité, la scalabilité et la haute disponibilité.

1. Qu'est-ce qu'un serveur d'applications Java EE ?
Un serveur d'applications Java EE est un environnement d'exécution qui fournit un ensemble de services pour héberger des applications Java EE. Il implémente les spécifications Java EE et gère l'exécution des composants serveur comme les servlets, les EJB, et les services web, facilitant ainsi le développement et le déploiement d'applications complexes à grande échelle.

Le serveur d'applications prend en charge des fonctionnalités avancées telles que :

Gestion des transactions distribuées : Pour assurer que toutes les opérations effectuées au sein d'une transaction sont correctement exécutées ou annulées en cas de problème.
Sécurité : Authentification, autorisation et cryptage pour protéger les applications.
Messagerie : Support pour la communication asynchrone via des systèmes de messagerie (par exemple, via JMS - Java Message Service).
Gestion des connexions : Gestion optimisée des connexions avec les bases de données ou autres systèmes externes.
Tolérance aux pannes et scalabilité : Gestion de la haute disponibilité, du clustering, et de la répartition de charge (load balancing) pour maintenir la disponibilité de l'application même en cas de panne.
2. Caractéristiques principales d'un serveur d'applications Java EE
a. Support des composants Java EE
Un serveur d'applications Java EE gère l'exécution des différents types de composants de l'architecture Java EE :

Servlets : Composants côté serveur qui traitent les requêtes HTTP et génèrent des réponses, souvent utilisés pour construire des applications web dynamiques.
EJB (Enterprise JavaBeans) : Composants qui encapsulent la logique métier, gèrent la persistance des données et facilitent la gestion des transactions.
JPA (Java Persistence API) : Fournit un moyen standard de gérer les bases de données en mappant les objets Java sur des tables relationnelles.
JMS (Java Message Service) : Permet la communication asynchrone entre différentes applications via des files d'attente de messages.
JAX-RS et JAX-WS : Gestion des services web, qu'ils soient SOAP (JAX-WS) ou RESTful (JAX-RS), pour permettre l'interopérabilité entre les systèmes.
b. Gestion des transactions
Les serveurs d'applications Java EE gèrent des transactions distribuées, garantissant que toutes les actions dans une transaction (comme la mise à jour de plusieurs bases de données) sont effectuées avec succès ou annulées en cas de problème. L'API JTA (Java Transaction API) permet de gérer cette complexité de manière standard.

c. Sécurité

La sécurité est un aspect central des serveurs d'applications Java EE. Ces serveurs offrent :

Authentification et autorisation : Contrôler qui peut accéder à certaines parties de l'application et à quelles actions ils sont autorisés.
SSL/TLS : Cryptage des communications pour protéger les données échangées entre les clients et le serveur.
JASPIC (Java Authentication Service Provider Interface for Containers) : Gère l'authentification dans les conteneurs web Java EE.
d. Scalabilité et haute disponibilité
Les serveurs Java EE sont conçus pour supporter un grand nombre d'utilisateurs simultanés en fournissant :

Clustering : Permet à plusieurs instances du serveur de travailler ensemble pour gérer la charge et fournir une haute disponibilité.
Load balancing : Répartition de la charge de travail entre différents serveurs pour éviter qu'un serveur ne soit surchargé.
Failover : Si un serveur tombe en panne, une autre instance prend le relais sans interruption du service.
e. Messagerie asynchrone
Grâce à JMS, les serveurs d'applications Java EE permettent de gérer des communications asynchrones entre différentes parties de l'application ou avec d'autres systèmes, ce qui est crucial pour les systèmes distribués.

3. Exemples de serveurs d'applications Java EE
Il existe plusieurs serveurs d'applications qui supportent Java EE/Jakarta EE. Voici quelques-uns des plus populaires :

a. GlassFish
Description : GlassFish est le serveur d'applications de référence pour Java EE, initialement développé par Sun Microsystems (puis Oracle), et aujourd'hui maintenu par la Fondation Eclipse sous le nom de Eclipse GlassFish.
Caractéristiques :
Conformité totale avec les spécifications Java EE/Jakarta EE.
Open-source, largement utilisé pour le développement et la production.
Facile à configurer et à déployer.
b. WildFly (anciennement JBoss AS)
Description : WildFly est un serveur d'applications open-source, initialement développé par Red Hat. Il est souvent utilisé dans les entreprises en raison de ses performances et de sa flexibilité.
Caractéristiques :
Conformité avec Java EE.
Léger et performant, adapté aux environnements cloud.
Support pour le clustering, la gestion des transactions et la sécurité.
c. Apache TomEE
Description : TomEE est une version étendue d'Apache Tomcat qui inclut des fonctionnalités Java EE supplémentaires. C’est une solution légère et flexible, idéale pour les applications web Java EE de taille moyenne.

Caractéristiques :
Basé sur Apache Tomcat avec des fonctionnalités Java EE telles que EJB, JPA, et JMS.
Léger, adapté aux applications web.
d. WebLogic
Description : Développé par Oracle, WebLogic est un serveur d'applications Java EE robuste souvent utilisé dans les environnements d'entreprise. Il est particulièrement apprécié pour sa compatibilité avec les bases de données Oracle.
Caractéristiques :
Très performant et hautement configurable.
Optimisé pour les environnements complexes, notamment avec les bases de données Oracle.
Support complet des fonctionnalités Java EE, avec une gestion avancée des transactions et de la sécurité.
e. IBM WebSphere
Description : IBM WebSphere est un serveur d'applications Java EE orienté entreprise, souvent utilisé dans les grandes organisations. Il est conçu pour gérer des applications d'entreprise critiques nécessitant une haute disponibilité, une scalabilité et une sécurité robustes.
Caractéristiques :
Très fiable et utilisé dans des environnements critiques.
Gestion poussée des transactions, des connexions et de la sécurité.
Intégration forte avec les autres services d'IBM.
4. Rôle du serveur d'applications dans le cycle de vie d'une application Java EE
Lorsqu'une application Java EE est déployée sur un serveur d'applications, celui-ci :

Déploie les composants Java EE (comme les EJB, les servlets, et les pages JSP) et les configure selon les besoins.
Gère les requêtes des utilisateurs : Quand une requête arrive (via HTTP, SOAP, REST, etc.), elle est dirigée vers les composants appropriés pour traitement.
Gère les transactions : Si l'application a besoin de gérer plusieurs opérations (par exemple, mettre à jour une base de données et envoyer un message), le serveur assure que tout est fait de manière atomique (ou annule tout en cas d'erreur).
Assure la sécurité : Le serveur authentifie les utilisateurs, vérifie leurs permissions, et s'assure que seules les personnes autorisées accèdent aux ressources sensibles.
Supervise et gère les ressources : Le serveur d'applications optimise les connexions aux bases de données et autres ressources externes pour garantir des performances optimales.

9. Le Choix de Tomcat

Apache Tomcat est un serveur de servlets open-source qui implémente les spécifications de Java Servlet et JavaServer Pages (JSP). Il est souvent choisi pour des applications web Java qui nécessitent des fonctionnalités de base telles que la gestion des requêtes HTTP, la génération dynamique de pages web, et la gestion de sessions. Cependant, Tomcat ne fournit pas toutes les fonctionnalités d'un serveur d'applications Java EE complet comme EJB (Enterprise JavaBeans), JPA (Java Persistence API) ou la gestion des transactions distribuées.

1. Tomcat vs Serveurs d'Applications Java EE Complets

1.1 Fonctionnalités de Tomcat
Gestion des servlets et JSP : Tomcat est conçu pour exécuter des servlets Java et des JSP. Il gère la requête-réponse HTTP, la gestion des sessions, et la génération dynamique des pages.
Léger et performant : Tomcat est connu pour sa légèreté et sa rapidité. Il est adapté aux applications web de taille moyenne qui n'ont pas besoin des fonctionnalités avancées des serveurs d'applications Java EE complets.
Configuration simple : Comparé aux serveurs d'applications Java EE complets, Tomcat est plus simple à configurer et à utiliser. Il est idéal pour des projets qui ne nécessitent pas une gestion complexe des transactions ou des EJB.
1.2 Limitations de Tomcat
Absence de support pour EJB : Tomcat ne prend pas en charge les EJB, qui sont des composants de serveur pour la logique métier et la gestion des transactions.
Pas de JPA intégré : Tomcat ne fournit pas de support intégré pour JPA pour la gestion de la persistance des données.
Pas de support complet pour les fonctionnalités Java EE avancées : Tomcat n'offre pas de support complet pour certaines spécifications Java EE telles que les transactions distribuées, la messagerie (JMS), ou les fonctionnalités de sécurité avancées.
2. Quand choisir Tomcat ?
Tomcat est un excellent choix pour :

Applications web simples : Si votre application web nécessite uniquement des servlets et des JSP, Tomcat est souvent suffisant et plus léger que les serveurs d'applications Java EE complets.
Projets à faible coût : Tomcat étant open-source et léger, il est une solution économique pour des projets qui n'ont pas besoin des fonctionnalités avancées des serveurs d'applications Java EE.
Développement et prototypage rapide : Sa simplicité en fait un bon choix pour le développement rapide et le prototypage d'applications web.
3. Intégration de Tomcat avec d'autres technologies
Pour les fonctionnalités non fournies par Tomcat, vous pouvez utiliser des bibliothèques et des frameworks externes :

JPA (Java Persistence API) : Pour la gestion de la persistance des données, vous pouvez utiliser des implémentations JPA comme Hibernate ou EclipseLink en conjonction avec Tomcat.
Spring Framework : Spring est un framework léger qui peut être utilisé pour ajouter des fonctionnalités comme la gestion des transactions, la gestion des dépendances, et la persistance des données à une application Tomcat.
JMS (Java Message Service) : Vous pouvez intégrer des bibliothèques de messagerie comme ActiveMQ pour ajouter la fonctionnalité de messagerie asynchrone à votre application Tomcat.
4. Comparaison avec d'autres serveurs d'applications Java EE
Voici comment Tomcat se compare avec d'autres serveurs d'applications Java EE :

Apache Tomcat est un serveur de servlets open-source qui implémente les spécifications de Java Servlet et JavaServer Pages (JSP). Il est souvent choisi pour des applications web Java qui nécessitent des fonctionnalités de base telles que la gestion des requêtes HTTP, la génération dynamique de pages web, et la gestion de sessions. Cependant, Tomcat ne fournit pas toutes les

fonctionnalités d'un serveur d'applications Java EE complet comme EJB (Enterprise JavaBeans), JPA (Java Persistence API) ou la gestion des transactions distribuées.

1. **Tomcat vs Serveurs d'Applications Java EE Complets**

1.1 Fonctionnalités de Tomcat

- **Gestion des servlets et JSP** : Tomcat est conçu pour exécuter des servlets Java et des JSP. Il gère la requête-réponse HTTP, la gestion des sessions, et la génération dynamique des pages.
- **Léger et performant** : Tomcat est connu pour sa légèreté et sa rapidité. Il est adapté aux applications web de taille moyenne qui n'ont pas besoin des fonctionnalités avancées des serveurs d'applications Java EE complets.
- **Configuration simple** : Comparé aux serveurs d'applications Java EE complets, Tomcat est plus simple à configurer et à utiliser. Il est idéal pour des projets qui ne nécessitent pas une gestion complexe des transactions ou des EJB.

1.2 Limitations de Tomcat

- **Absence de support pour EJB** : Tomcat ne prend pas en charge les EJB, qui sont des composants de serveur pour la logique métier et la gestion des transactions.
- **Pas de JPA intégré** : Tomcat ne fournit pas de support intégré pour JPA pour la gestion de la persistance des données.
- **Pas de support complet pour les fonctionnalités Java EE avancées** : Tomcat n'offre pas de support complet pour certaines spécifications Java EE telles que les transactions distribuées, la messagerie (JMS), ou les fonctionnalités de sécurité avancées.

2. **Quand choisir Tomcat ?**

Tomcat est un excellent choix pour :

- **Applications web simples** : Si votre application web nécessite uniquement des servlets et des JSP, Tomcat est souvent suffisant et plus léger que les serveurs d'applications Java EE complets.
- **Projets à faible coût** : Tomcat étant open-source et léger, il est une solution économique pour des projets qui n'ont pas besoin des fonctionnalités avancées des serveurs d'applications Java EE.
- **Développement et prototypage rapide** : Sa simplicité en fait un bon choix pour le développement rapide et le prototypage d'applications web.

3. **Intégration de Tomcat avec d'autres technologies**

Pour les fonctionnalités non fournies par Tomcat, vous pouvez utiliser des bibliothèques et des frameworks externes :

- **JPA (Java Persistence API)** : Pour la gestion de la persistance des données, vous pouvez utiliser des implémentations JPA comme **Hibernate** ou **EclipseLink** en conjonction avec Tomcat.

- **Spring Framework** : Spring est un framework léger qui peut être utilisé pour ajouter des fonctionnalités comme la gestion des transactions, la gestion des dépendances, et la persistance des données à une application Tomcat.
- **JMS (Java Message Service)** : Vous pouvez intégrer des bibliothèques de messagerie comme **ActiveMQ** pour ajouter la fonctionnalité de messagerie asynchrone à votre application Tomcat.

4. **Comparaison avec d'autres serveurs d'applications Java EE**

Voici comment Tomcat se compare avec d'autres serveurs d'applications Java EE :

Fonctionnalité	Tomcat	GlassFish	WildFly (JBoss AS)	WebLogic	WebSphere
Servlets	Oui	Oui	Oui	Oui	Oui
JSP	Oui	Oui	Oui	Oui	Oui
EJB	Non	Oui	Oui	Oui	Oui
JPA	Non intégré	Oui	Oui	Oui	Oui
Transactions	Non	Oui	Oui	Oui	Oui
Messagerie (JMS)	Non	Oui	Oui	Oui	Oui
Clustering	Non	Oui	Oui	Oui	Oui
Sécurité	Basique	Avancée	Avancée	Avancée	Avancée
Configuration	Simple	Complex	Complex	Complex	Complex

10. # Outils et Environnement de Développement pour Java EE

Développer des applications Java EE (ou Jakarta EE) implique l'utilisation de plusieurs outils et environnements pour coder, tester, et déployer les applications. Voici les principaux outils et environnements de développement utilisés dans le contexte de Java EE :

1. Environnements de Développement Intégré (IDE)
Eclipse IDE :

Description : Eclipse est un IDE open-source populaire pour le développement Java. Il fournit des outils robustes pour le développement Java EE, y compris des outils pour le codage, le débogage, le déploiement et le test des applications.
Fonctionnalités Java EE :

Eclipse Enterprise for Java : Inclut des outils pour le développement Java EE, comme la gestion des projets, la configuration des serveurs, et les assistants pour les servlets, JSP, EJB, JPA, etc.
Integration de serveurs : Eclipse permet l'intégration directe avec des serveurs d'applications comme Tomcat, GlassFish, et WildFly.
Outils de débogage : Permet le débogage d'applications Java EE avec des outils de suivi des transactions et des points d'arrêt.
IntelliJ IDEA :

Description : IntelliJ IDEA est un autre IDE très populaire, connu pour ses fonctionnalités avancées et son interface utilisateur conviviale.
Fonctionnalités Java EE :
Support complet pour les spécifications Java EE, y compris des outils pour les servlets, JSP, EJB, JPA, et autres.
Intégration de serveurs : Permet la configuration et le déploiement sur des serveurs d'applications comme Tomcat, Jetty, et d'autres serveurs Java EE.
NetBeans :

Description : NetBeans est un IDE open-source qui supporte le développement Java EE avec des outils intégrés pour coder, tester, et déployer des applications.
Fonctionnalités Java EE :
Support pour les servlets, JSP, EJB, JPA, JSF, et d'autres technologies Java EE.
Intégration de serveurs : Facilite le déploiement sur des serveurs comme GlassFish, Tomcat, et d'autres.

11. Installation de l'eclipse

Voici les étapes pour installer Eclipse sur les systèmes d'exploitation Windows, Linux et macOS :

1. Installation d'Eclipse sur Windows
Téléchargement :

Allez sur le site officiel d'Eclipse : http://eclipse.org/downloads/
Cliquez sur "Download" pour l'édition d'Eclipse que vous souhaitez (par exemple, Eclipse IDE for Java EE Developers).
Extraction :

Une fois le fichier téléchargé (souvent un fichier .zip), décompressez-le dans un répertoire de votre choix, comme C:\Program Files\Eclipse ou un autre répertoire où vous souhaitez installer Eclipse.
Lancement :

Accédez au répertoire où vous avez extrait Eclipse.

Double-cliquez sur le fichier eclipse.exe pour lancer l'application.
Lors du premier lancement, Eclipse vous demandera de spécifier un espace de travail. Choisissez le répertoire où vous souhaitez stocker vos projets et cliquez sur "Launch".
Configuration supplémentaire (optionnel) :

Vous pouvez créer un raccourci sur le bureau pour un accès plus facile. Faites un clic droit sur eclipse.exe, sélectionnez Créer un raccourci, puis déplacez le raccourci vers le bureau.
2. Installation d'Eclipse sur Linux
Téléchargement :

Allez sur le site officiel d'Eclipse : Eclipse Downloads
Cliquez sur "Download" pour l'édition d'Eclipse que vous souhaitez (par exemple, Eclipse IDE for Java Developers).
Extraction :

Ouvrez un terminal.
Accédez au répertoire où vous avez téléchargé le fichier. Le fichier sera généralement un fichier .tar.gz.
Utilisez la commande suivante pour extraire le fichier :

tar -xzf eclipse-java-<version>-linux-gtk-x86_64.tar.gz

Remplacez <version> par le numéro de version du fichier téléchargé.
Déplacement (optionnel) :

Déplacez le répertoire extrait dans /opt pour une installation système globale :

sudo mv eclipse /opt/

Création d'un raccourci (optionnel) :

Créez un fichier de lancement dans /usr/local/bin

sudo nano /usr/local/bin/eclipse

Ajoutez le contenu suivant au fichier, puis sauvegardez et fermez (Ctrl+X, Y, Enter)

#!/bin/bash

/opt/eclipse/eclipse

Rendez le fichier exécutable

sudo chmod +x /usr/local/bin/eclipse

Vous pouvez maintenant lancer Eclipse en tapant eclipse dans le terminal.
Lancement :

Lancez Eclipse en utilisant le raccourci que vous avez créé ou en naviguant vers le répertoire d'installation et en exécutant le fichier eclipse

Installation d'Eclipse sur macOS
Téléchargement :

Allez sur le site officiel d'Eclipse : http://eclipse.org/downloads/
Cliquez sur "Download" pour l'édition d'Eclipse que vous souhaitez (par exemple, Eclipse IDE for Java EE Developers).
Installation :

Une fois le fichier téléchargé (généralement un fichier .dmg), ouvrez-le en double-cliquant sur le fichier.
Faites glisser l'icône d'Eclipse dans le dossier Applications pour l'installer.
Lancement :

Accédez au dossier Applications dans le Finder.
Double-cliquez sur l'icône d'Eclipse pour lancer l'application.
Lors du premier lancement, Eclipse vous demandera de spécifier un espace de travail. Choisissez le répertoire où vous souhaitez stocker vos projets et cliquez sur "Launch".
Création d'un raccourci (optionnel) :

Vous pouvez créer un raccourci dans le Dock en faisant un clic droit sur l'icône d'Eclipse dans le Dock et en sélectionnant Options > Garder dans le Dock.

12. Tomcat dans eclipse

Voici un guide pour installer et intégrer Apache Tomcat dans Eclipse sur les systèmes Windows, Linux et macOS.

1. Installation de Tomcat sur Windows, Linux, et macOS
Étape 1 : Télécharger Apache Tomcat
Accédez au site officiel de Tomcat : Tomcat Downloads
Sous la section Binary Distributions, choisissez la version qui vous convient :
Pour Windows, téléchargez le fichier 32-bit/64-bit Windows Service Installer (fichier .exe).
Pour Linux/macOS, téléchargez l'archive Core (fichier .tar.gz).
Étape 2 : Installation
Windows :

Exécutez le fichier .exe téléchargé et suivez l'assistant d'installation.
Définissez un chemin d'installation, par exemple C:\Program Files\Apache Tomcat.
Configurez un port (par défaut, c'est le port 8080) et définissez un mot de passe pour l'administrateur si demandé.
Complétez l'installation en laissant les paramètres par défaut.
Linux/macOS :

Ouvrez un terminal et accédez au répertoire où vous avez téléchargé l'archive.
Extrayez l'archive avec la commande suivante :

tar -xzf apache-tomcat-9.x.x.tar.gz

Déplacez le répertoire extrait dans /opt pour une installation plus propre (facultatif) :

sudo mv apache-tomcat-9.x.x /opt/tomcat

Pour démarrer Tomcat, exécutez le script startup.sh dans le dossier bin :
/opt/tomcat/bin/startup.sh

2. Intégration de Tomcat dans Eclipse (Windows, Linux et macOS)

13. Une fois Tomcat installé, vous devez l'intégrer dans Eclipse pour pouvoir le gérer directement depuis votre environnement de développement.

    Étape 1 : Installer Eclipse
    Si Eclipse n'est pas encore installé, vous pouvez suivre les instructions pour l'installer en fonction de votre système d'exploitation :

    Windows, Linux, macOS : Téléchargez Eclipse
    Étape 2 : Configuration de Tomcat dans Eclipse
    Ouvrir Eclipse :

    ## Lancez Eclipse sur votre machine.

    Accéder à la vue des serveurs :

    Dans Eclipse, allez dans le menu Window > Show View > Other....
    Dans la fenêtre qui s'ouvre, tapez "Servers" et sélectionnez la vue Servers, puis cliquez sur Open.

Ajouter un serveur Tomcat :

Dans la vue Servers, faites un clic droit et sélectionnez New > Server.
Une fenêtre s’ouvrira, vous demandant de sélectionner le type de serveur. Choisissez Apache Tomcat v9.0 (ou la version que vous avez téléchargée).

Cliquez sur Next.
Spécifier l'emplacement de Tomcat :

Dans la fenêtre suivante, Eclipse vous demandera de fournir le chemin vers le répertoire d'installation de Tomcat.
Sur Windows, il se peut que le chemin soit C:\Program Files\Apache Software Foundation\Tomcat 9.x.
Sur Linux/macOS, le chemin peut être /opt/tomcat ou le répertoire où vous avez extrait Tomcat.
Cliquez sur Finish une fois que vous avez spécifié le chemin.
Ajouter un projet web à Tomcat :

Après avoir ajouté Tomcat à Eclipse, vous pouvez maintenant déployer vos projets web.
Faites un clic droit sur le serveur Tomcat dans la vue Servers et sélectionnez Add and Remove....
Sélectionnez le projet web que vous voulez déployer et cliquez sur Add, puis sur Finish.

14. Démarrer Tomcat depuis Eclipse :

Une fois le projet ajouté, vous pouvez démarrer ou arrêter Tomcat directement depuis Eclipse.
Faites un clic droit sur le serveur Tomcat dans la vue Servers et sélectionnez Start.

15. Configuration de l’éclipse

-Encodage

Allez dans la barre de menus en haut, cliquez sur Window, puis sur Preferences. Une nouvelle fenêtre va s'ouvrir. En haut à gauche de cette fenêtre, vous verrez un champ de recherche. Tapez "encoding" dans ce champ. Dans les sections qui apparaissent dans le panneau de gauche, modifiez l'encodage par défaut en le remplaçant par UTF-8.

16. Désactivation de la vérification de l'orthographe

Allez de nouveau dans le menu Window > Preferences, puis dans le panneau de gauche, accédez à General > Editors > Text Editors. Dans le panneau de droite, cochez la case Show line numbers (Afficher les numéros de ligne). Ensuite, dans le panneau de gauche, sélectionnez le sous-menu Spelling. Dans le panneau de droite qui apparaît, décochez la case Enable spell checking (Activer la vérification orthographique). Pour appliquer les modifications, cliquez sur OK.

17. Création du projet Java EE

1. Installer le plugin Java EE dans Eclipse

Créer un projet Java EE dans Eclipse
Ouvrir Eclipse :
Lancez Eclipse sur votre ordinateur.
Créer un nouveau projet :

Allez dans File > New > Dynamic Web Project.
Donnez un nom à votre projet, par exemple MyFirstWeb
Sélectionnez la version de Java et de Dynamic web module version selon vos besoins (par exemple, Servlet 4.0 ou Servlet 3.1).
Sous Target Runtime, cliquez sur New Runtime pour ajouter Apache Tomcat si vous ne l'avez pas encore configuré.

18. Configurer Apache Tomcat (si non configuré) :

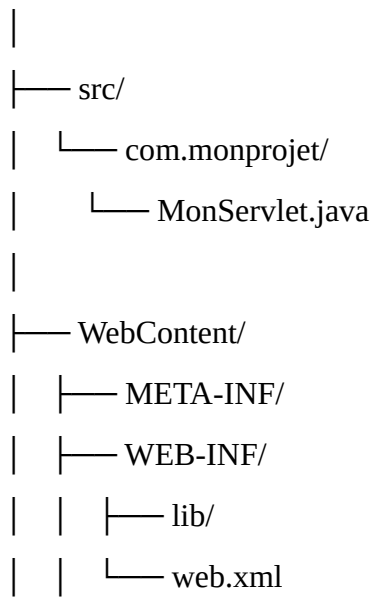
Dans la fenêtre New Server Runtime Environment, sélectionnez Apache > Tomcat v9.0 (ou la version installée) puis cliquez sur Next.
Spécifiez le chemin d'installation de Tomcat sur votre machine (par exemple, C:\Program Files\Apache Software Foundation\Tomcat 9.0 sous Windows ou /opt/tomcat sous Linux/macOS).

Cliquez sur Finish pour valider l'ajout du runtime Tomcat.
Finaliser la création du projet :

Cliquez sur Finish après avoir configuré le runtime et terminé les paramètres du projet.
Eclipse va créer un projet de type Dynamic Web Project avec les répertoires appropriés (comme WebContent (ou webapp, WEB-INF, etc.).

Structure du projet Java EE

MonProjetJavaEE/



Explication du projet

2. Explication des dossiers et fichiers
a) src/
Rôle : Ce dossier contient le code source Java du projet, y compris les Servlets, les Managed Beans (dans les projets JSF), ou les classes métiers.
Chemin : Les classes Java y sont organisées selon leur package. Par exemple, dans le cas de MonServlet.java, il se trouve dans src/com/monprojet/.
b) WebContent/
Rôle : Ce dossier contient tous les fichiers relatifs à la partie web du projet, comme les fichiers JSP, HTML, CSS, JavaScript, et les ressources statiques. C'est le contenu qui sera

exposé directement à l'utilisateur.
i. META-INF/
Rôle : Ce dossier contient des informations de configuration pour les archives WAR ou JAR. Il est principalement utilisé pour les configurations avancées ou pour les projets JAR déployés sous forme de composants.
Fichiers communs : MANIFEST.MF (contient les informations sur les fichiers JAR).
ii. WEB-INF/
Rôle : Ce dossier est crucial pour les applications Java EE. Il contient les configurations du serveur et les ressources que le client ne doit pas directement accéder (comme les bibliothèques ou les fichiers de configuration).
- lib/ :
Ce dossier contient les JAR (bibliothèques Java) nécessaires au projet. Les bibliothèques tierces ou internes qui ne sont pas incluses dans le serveur d'application doivent être placées ici. Par exemple, vous pouvez ajouter ici des fichiers JAR tels que JSTL, Hibernate, etc.
- classes/ :
C'est ici que les fichiers .class (les fichiers compilés des classes Java) sont placés. Eclipse compile

automatiquement les classes Java du dossier src/ et les place ici. Le serveur Tomcat ou tout autre serveur d'application utilise ce dossier pour charger les classes Java nécessaires à l'application.
- web.xml :
Rôle : C'est le descripteur de déploiement de l'application web. Il configure les servlets, les filtres, les écouteurs, les mappages d'URL et bien d'autres paramètres. Par exemple, il définit quelles servlets répondent à quelles URL, configure les paramètres d'initialisation, etc.

19. Création du fichier html et teste avec Tomcat

A compléter ……...

II. Partie 2 : Les coulisses Java EE

La Servlet

Qu'est-ce qu'une Servlet ?

Une Servlet est un composant Java utilisé dans le développement d'applications web, qui fonctionne sur un serveur d'application ou un serveur web. Elle est principalement utilisée pour gérer les requêtes HTTP (comme GET et POST), interagir avec les clients (navigateurs) et générer des réponses dynamiques, souvent en HTML ou JSON.Elle est une classe Java qui étend la classe **HttpServlet** (ou **GenericServlet** dans certaines versions) et répond à une requête client (comme une requête HTTP) en générant une réponse. La Servlet est souvent utilisée pour traiter des données de formulaire, gérer des sessions, et interagir avec une base de données ou d'autres services backend.

2. Cycle de vie d'une Servlet

Le cycle de vie d'une Servlet se compose de trois étapes principales : la création, l'exécution, et la destruction.

Initialisation (init) : La méthode **init()** est appelée une fois lorsqu'une Servlet est créée. C'est ici que l'initialisation de la Servlet se fait, comme la configuration des ressources ou la connexion à une base de données.

Traitement des requêtes (service) : Après initialisation, chaque requête client est traitée par la méthode service(), qui détermine si la requête est une requête GET, POST, etc., et appelle la méthode correspondante (doGet(), doPost(), etc.).

Destruction (destroy) : La méthode destroy() est appelée une fois lorsque la Servlet est supprimée ou lorsqu'elle est arrêtée par le serveur. C'est ici que vous libérez toutes les ressources allouées dans init().

3. Principales méthodes de la Servlet

Les Servlets implémentent plusieurs méthodes définies dans l'**API HttpServlet**. Voici les méthodes les plus courantes et leur rôle :

a) init(ServletConfig config)

Description : La méthode init() est appelée une seule fois lorsque la Servlet est instanciée par le serveur (lors de son chargement). Elle est utilisée pour initialiser la Servlet.

Utilisation : Vous pouvez y placer le code d'initialisation, comme l'établissement d'une connexion à la base de données.

```
@Override
public void init() throws ServletException {
    // Code d'initialisation (connexion DB, configuration, etc.)
}
```

b) **service(HttpServletRequest req, HttpServletResponse resp)**

Description : Cette méthode est automatiquement appelée par le serveur pour traiter une requête. Elle détermine si la requête est un GET, POST, PUT, DELETE, etc., et appelle la méthode appropriée (doGet(), doPost(), etc.).

Utilisation : Vous n'avez généralement pas besoin de la redéfinir, car les méthodes doGet(), doPost(), etc., sont plus spécifiques.

```
@Override
protected void service(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {
    super.service(req, resp);  // Laisse HttpServlet gérer l'appel
}
```

c) doGet(HttpServletRequest req, HttpServletResponse resp)

Description : Cette méthode est appelée lorsque la Servlet reçoit une requête HTTP GET. Elle est utilisée pour récupérer des informations du serveur, comme afficher une page HTML ou envoyer des données JSON.

Utilisation : Vous redéfinissez cette méthode pour répondre aux requêtes GET.

```
@Override
```

protected void doGet(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

```
resp.setContentType("text/html");
PrintWriter out = resp.getWriter();
out.println("<h1>Bonjour depuis doGet!</h1>");
```

}

d) doPost(HttpServletRequest req, HttpServletResponse resp)
Description : Cette méthode est appelée lorsque la Servlet reçoit une requête HTTP POST. Elle est généralement utilisée pour envoyer des données au serveur, comme un formulaire HTML ou une requête API.

Utilisation : Vous redéfinissez cette méthode pour traiter les données POST.

@Override

protected void doPost(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

```
String username = req.getParameter("username");
resp.setContentType("text/html");
PrintWriter out = resp.getWriter();
out.println("<h1>Bonjour " + username + "!</h1>");
```

}

e) doPut(HttpServletRequest req, HttpServletResponse resp)
Description : Cette méthode est appelée pour traiter les requêtes HTTP PUT. Elle est utilisée lorsque des clients envoient des données pour remplacer une ressource existante sur le serveur.

Utilisation : Vous redéfinissez cette méthode pour gérer les requêtes PUT.

@Override

protected void doPut(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

```
// Logique pour gérer la requête PUT
```

}

f) **doDelete(HttpServletRequest req, HttpServletResponse resp)**
Description : Cetcchte méthode est utilisée pour traiter les requêtes HTTP DELETE. Elle est appelée lorsque le client souhaite supprimer une ressource sur le serveur.

@Override

protected void doDelete(HttpServletRequest req, HttpServletResponse resp) throws ServletException, IOException {

    // Logique pour gérer la requête DELETE

}

g) **destroy()**
Description : Cette méthode est appelée une fois lorsque la Servlet est détruite (par exemple, lorsque le serveur est arrêté). Elle est utilisée pour libérer les ressources allouées par la Servlet, comme fermer des connexions de base de données.

Utilisation : Vous redéfinissez cette méthode pour effectuer le nettoyage.

@Override

public void destroy() {

    // Libérer les ressources (fermer la base de données, etc.)

}

Création d'une Servlet dans Eclipse

Servlet qui renvoie une chaîne de caractère
Voici un guide étape par étape pour créer une Servlet dans Eclipse et l'exécuter avec un serveur comme Tomcat.

1. Prérequis
Avoir Eclipse IDE for Java EE Developers installé.
Avoir installé un serveur d'application comme Apache Tomcat et configuré dans Eclipse.
2. Créer un projet Web dynamique
Avant de créer une servlet, il est nécessaire de créer un projet Dynamic Web Project.

Ouvrez Eclipse.
Cliquez sur File > New > Dynamic Web Project.
Donnez un nom à votre projet, par exemple "MonProjetServlet".
Cliquez sur Next et laissez les autres paramètres par défaut.
Dans l'onglet Configuration, choisissez Dynamic Web Module Version 4.0 si vous utilisez Java EE 8 ou 11.
Cliquez sur Finish.
Votre projet "MonProjetServlet" est maintenant créé, et il est prêt à accueillir une Servlet.

3. Ajouter Tomcat comme serveur
Si Tomcat n'est pas déjà configuré, suivez ces étapes pour l'ajouter :

Cliquez sur l'onglet Servers en bas d'Eclipse.
Cliquez droit dans l'espace de l'onglet Servers, puis cliquez sur New > Server.
Choisissez Apache > Tomcat vX.X Server (où X.X est la version de Tomcat que vous avez installé).
Suivez les instructions pour ajouter votre installation de Tomcat à Eclipse.
Cliquez sur Finish.
Tomcat est maintenant configuré comme serveur dans Eclipse.

4. Créer une Servlet
Une fois le projet Web dynamique créé, vous pouvez ajouter une Servlet.

Dans le Project Explorer, faites un clic droit sur le dossier src de votre projet.
Sélectionnez New > Servlet.
Dans la fenêtre qui s'affiche, entrez le package (par exemple, com.monprojet) et le nom de la Servlet (par exemple, MonServlet).
Cliquez sur **Next**, et vous verrez une option pour configurer le mappage URL de la Servlet. Par défaut, Eclipse proposera une URL de mappage, comme **/MonServlet**. Vous pouvez la laisser telle quelle ou la modifier.
Cliquez sur **Finish**.

Voici un exemple de Servlet générée :

```
package com.monprojet;


import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


@WebServlet("/MonServlet")

public class MonServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;


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

        response.getWriter().append("Bienvenue dans ma première Servlet!");
```

```
    }


}
```

NB : Souvent au lieu de voir le package **javax.servlet.*** , vous allez voir j**akarta,servlet,***

lorque vous êtes sous Eclipse

5. **Configurer le fichier web.xml (Optionnel)**
Si vous ne souhaitez pas utiliser les annotations comme **@WebServlet**, vous pouvez configurer la Servlet dans le fichier web.xml.

Ouvrez le fichier **web.xml** situé dans **WebContent/WEB-INF/.**
Ajoutez cette configuration pour mapper la Servlet à une URL :

```
<servlet>
    <servlet-name>MonServlet</servlet-name>
    <servlet-class>com.monprojet.MonServlet</servlet-class>
</servlet>

<servlet-mapping>
    <servlet-name>MonServlet</servlet-name>
    <url-pattern>/MonServlet</url-pattern>
</servlet-mapping>
```

6. Exécuter le projet avec Tomcat
Faites un clic droit sur le projet MonProjetServlet dans l'explorateur de projet.
Sélectionnez **Run As > Run on Server.**
Choisissez **Tomcat** comme serveur et cliquez sur **Finish**.
Cela déploiera votre projet sur Tomcat, et une fenêtre de navigateur s'ouvrira pour afficher le résultat.

Servlet qui renvoie un HTML

```
package com.monprojet;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;
import java.io.PrintWriter;

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

    @Override
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        // Indique au navigateur que la réponse est du HTML
        response.setContentType("text/html");

        // Utilisation d'un PrintWriter pour écrire la réponse HTML
        PrintWriter out = response.getWriter();

        // Génération du contenu HTML



         out.println(" <!Doctype html");
        out.println("<html>");
        out.println("<head>");
        out.println("<title>Bienvenue dans la Servlet</title>");
        out.println("</head>");
        out.println("<body>");
```

```
        out.println("<h1>Bonjour depuis une Servlet Java!</h1>");
        out.println("<p>Ceci est une page HTML générée dynamiquement par une Servlet Java.</p>");
        out.println("</body>");
        out.println("</html>");

        // Ferme l'objet PrintWriter
        out.close();
    }
}
```

2. Explication du Code
Annotation @WebServlet : L'URL de mappage /HelloServlet permet d'accéder à la Servlet à partir de l'URL http://localhost:8080/MonProjetServlet/HelloServlet.
Méthode doGet() : Cette méthode gère les requêtes HTTP GET. Elle définit le type de contenu à text/html, ce qui indique au navigateur que la réponse est du HTML.
PrintWriter : Utilisé pour écrire la réponse HTML ligne par ligne. Le HTML généré est inclus directement dans le flux de réponse.

Servlet qui retourne la vue avec la technologie JSP

JSP (JavaServer Pages) est une technologie Java utilisée dans Java EE (Jakarta EE) pour créer des pages web dynamiques. Contrairement aux Servlets, où tout le contenu HTML est généré à partir du code Java, JSP permet d'intégrer du code Java directement dans des pages HTML à l'aide de balises spéciales. Cela facilite la génération de contenu dynamique, tout en permettant aux développeurs de séparer la logique métier (Java) de la présentation (HTML).

JSP et Servlets ensemble : MVC

Dans une architecture **MVC (Modèle-Vue-Contrôleur)**, les **Servlets** sont souvent utilisées pour gérer la logique métier et les requêtes HTTP, tandis que les **JSP** sont utilisées pour afficher la réponse (Vue).

Créer une page JSP Simple sans Servlet

Une fois le projet créé, vous allez ajouter une page JSP dans ce projet.

Faites un clic droit sur le dossier WebContent dans l'explorateur de projet.
Sélectionnez New > JSP File.
Donnez un nom à la page JSP, par exemple index.jsp, puis cliquez sur Finish.

Ouvrez la page index.jsp que vous venez de créer et ajoutez du contenu HTML et des expressions JSP si nécessaire.

Voici un exemple de code simple pour index.jsp :

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>Exemple JSP</title>
  </head>
  <body>
    <h1>Bienvenue sur ma page JSP</h1>
    <p>Date et heure actuelle : <%= new java.util.Date() %></p>
  </body>
</html>
```

Accéder à la page JSP

Dans le navigateur, accédez à l'URL suivante pour voir la page JSP en action :

http://localhost:8080/MonProjetJSP/index.jsp

Explication du code JSP
Directives JSP : page est une directive utilisée pour définir les paramètres de la page JSP (comme l'encodage ici défini à UTF-8).
Expressions JSP (<%= ... %>) : Cette expression JSP affiche dynamiquement la date et l'heure actuelles en utilisant du code Java.

Une Servlet qui retourne la vue jsp

Créer une Servlet qui renvoie une page JSP dans une application Java EE permet de séparer la logique métier (gérée par la Servlet) de la présentation (gérée par la page JSP).

Faites un clic droit sur le dossier src dans l'explorateur de projet.
Sélectionnez New > Servlet.
Donnez un nom à votre Servlet, par exemple MonServlet, et cliquez sur Finish.
Voici un exemple de code pour la Servlet :

```
package com.monprojet;

import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // Ajoutez des données à passer à la JSP (optionnel)
        String message = "Bienvenue dans la page JSP via Servlet!";
        request.setAttribute("message", message);

        // Redirige vers la page JSP
        RequestDispatcher dispatcher = request.getRequestDispatcher("/index.jsp");
        dispatcher.forward(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

3. Créer une page JSP
Faites un clic droit sur le dossier WebContent (ou webapp) dans l'explorateur de projet.

Sélectionnez New > JSP File.
Donnez un nom à la page, par exemple index.jsp, et cliquez sur Finish.
Voici un exemple de contenu pour la page index.jsp :

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>Page JSP</title>
  </head>
  <body>
    <h1>Page JSP générée via Servlet</h1>
    <p>Message : ${message}</p>
  </body>
</html>
```

. Explication du code
Servlet (MonServlet) : Elle traite les requêtes GET ou POST et redirige la requête vers la page JSP en utilisant le RequestDispatcher. Avant de rediriger, elle ajoute un attribut (message) que la page JSP peut utiliser.

JSP (index.jsp) : Cette page utilise une expression EL (Expression Language) ${message} pour afficher le message envoyé par la Servlet. L'EL permet d'accéder facilement aux données passées depuis la Servlet.

Pourquoi placer votre vue dans le dossier WEB-INF

L'importance de créer des vues JSP dans le dossier WEB-INF dans une application Java EE
Dans une application Java EE, le dossier WEB-INF est une partie protégée du projet web, qui ne peut pas être directement accédée depuis un navigateur. Cela signifie que toute ressource placée dans WEB-INF (comme des pages JSP) est protégée contre l'accès direct depuis l'URL. Cela offre plusieurs avantages importants, notamment en termes de sécurité, de contrôle d'accès et de modèle MVC.

Voici pourquoi il est recommandé de placer des pages JSP dans le dossier WEB-INF :

1. Sécurité : Prévention de l'accès direct
Lorsque vous placez vos fichiers JSP dans WEB-INF, ils ne peuvent pas être directement

accessibles via une URL (comme http://localhost:8080/monProjet/maPage.jsp). Cela empêche les utilisateurs d'accéder directement à des fichiers JSP sans passer par une Servlet ou un contrôleur qui gère la logique de l'application. Cela aide à protéger les vues et à éviter l'exposition involontaire de fichiers sensibles.

Exemple :
Si vous avez une page result.jsp dans le dossier WebContent, les utilisateurs peuvent y accéder directement avec une URL comme :

http://localhost:8080/monProjet/result.jsp

Si cette page est placée sous WEB-INF/jsp/result.jsp, l'accès direct sera bloqué par le serveur. Seule une Servlet pourra rediriger l'utilisateur vers cette page, comme dans le modèle MVC (Modèle-Vue-Contrôleur).

2. Encapsulation des vues dans l'architecture MVC
Dans l'architecture MVC :

Les Servlets ou contrôleurs jouent le rôle de contrôleurs qui gèrent la logique métier, les requêtes HTTP et redirigent les utilisateurs vers les pages JSP.
Les pages JSP sont responsables de l'affichage des données (la vue).
Placer les JSP dans WEB-INF garantit que les utilisateurs ne peuvent pas accéder directement à la vue sans que la logique du contrôleur soit exécutée. Cela permet d'avoir un meilleur contrôle sur les flux de l'application.

Exemple avec MVC :
L'utilisateur accède à une URL gérée par une Servlet (comme /MonServlet).
La Servlet traite les données, puis redirige vers une page JSP située dans WEB-INF/jsp/monJsp.jsp.
L'utilisateur ne peut pas accéder directement à la page JSP sans passer par la Servlet.
3. Contrôle des flux utilisateurs
En plaçant les pages JSP dans WEB-INF, vous forcez les utilisateurs à passer par les contrôleurs appropriés (Servlets). Cela vous permet de :

Vérifier l'authentification et les autorisations avant de rendre une page.
Valider ou traiter les données avant d'afficher la vue (JSP).
Rediriger l'utilisateur en fonction des résultats d'une logique métier (comme un message d'erreur ou une redirection vers une autre page).
4. Gestion des vues complexes
Dans les applications complexes, certaines pages JSP ne doivent être accessibles qu'après que certaines conditions soient remplies (exemple : après soumission d'un formulaire ou après authentification). Si les JSP sont directement accessibles, cela pourrait entraîner des problèmes de navigation et de sécurité. En les plaçant dans WEB-INF, vous pouvez mieux gérer ces situations.

Cas d'utilisation :

Imaginons que vous avez un formulaire de connexion. Si la page dashboard.jsp est placée directement dans WebContent, un utilisateur non authentifié pourrait essayer de l'accéder directement via l'URL. En plaçant cette page dans WEB-INF, vous pouvez forcer le passage par une Servlet qui vérifie si l'utilisateur est authentifié avant de le rediriger vers la page.

5. Meilleure organisation et structure du projet
Placer les JSP dans un sous-dossier comme WEB-INF/jsp/ aide à organiser votre projet de manière plus structurée. Cela rend l'application plus maintenable, en séparant clairement la logique métier (les Servlets) des vues (les JSP). De plus, cela permet d'éviter la prolifération de fichiers JSP dans le dossier racine WebContent.

Exemple de structure :

```
WebContent/
│
├── WEB-INF/
│   ├── jsp/
│   │   ├── index.jsp
│   │
│   └── web.xml
├── css/
├── js/
└── images/
```

Dans cette structure, toutes les pages JSP sont protégées dans WEB-INF/jsp/, et les utilisateurs n'y accèdent qu'à travers une Servlet.

6. **Meilleure intégration avec les frameworks modernes**

Dans de nombreux frameworks Java comme **Spring MVC** ou **Struts**, les pages JSP sont généralement placées sous **WEB-INF** pour suivre le modèle MVC et assurer une séparation claire entre les contrôleurs et les vues. En suivant cette pratique, vous assurez une meilleure compatibilité avec les frameworks modernes, qui utilisent souvent cette convention pour la gestion des vues.

7. **Exemple pratique : Servlet qui redirige vers une JSP dans WEB-INF**

Voici un exemple d'une **Servlet** qui redirige vers une page JSP dans le dossier WEB-INF/jsp/.

Servlet Java :

```
package com.monprojet;

import java.io.IOException;
```

```
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // Ajout de données (optionnel)
        request.setAttribute("message", "Bienvenue dans la page JSP sous WEB-INF");

        // Redirection vers la page JSP sous WEB-INF/jsp/
        RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB-INF/jsp/index.jsp");
        dispatcher.forward(request, response);
    }

    protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        doGet(request, response);
    }
}
```

Page JSP (WEB-INF/jsp/index.jsp) :

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>Page JSP protégée</title>
```

```
  </head>
  <body>
    <h1>Bienvenue dans une JSP sous WEB-INF</h1>
    <p>Message : ${message}</p>
  </body>
</html>
```

Transmettre des variables d'une Servlet à une page JSP

Pour transmettre des variables d'une Servlet à une page JSP, la méthode courante consiste à utiliser l'objet HttpServletRequest pour stocker les données dans des attributs, puis les récupérer dans la JSP.

Voici un guide complet pour réaliser cela :

1. **Création d'une Servlet**
    1. Ouvrez **Eclipse** et créez un projet **Dynamic Web Project** (si vous ne l'avez pas déjà fait).
    2. Faites un clic droit sur le dossier **src**, puis sélectionnez **New > Servlet**.
    3. Donnez un nom à votre Servlet, par exemple MaServlet, et cliquez sur **Finish**.

Voici un exemple de code pour la Servlet MaServlet :

```
package com.monprojet;


import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

```
    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
        // Créer des variables à transmettre à la JSP
        String message = "Bienvenue sur la page JSP!";
        int annee = 2024;

        // Ajouter les variables en tant qu'attributs de la requête
        request.setAttribute("message", message);
        request.setAttribute("annee", annee);

        // Redirection vers la page JSP
        RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB-INF/jsp/maPage.jsp");
        dispatcher.forward(request, response);
    }



    }
}
```

Explication du code de la Servlet :
request.setAttribute("message", message); : On utilise setAttribute pour attacher des variables (ici, message et annee) à l'objet HttpServletRequest. Ces variables seront ensuite accessibles depuis la JSP.
RequestDispatcher dispatcher = request.getRequestDispatcher("/WEB-INF/jsp/maPage.jsp"); : La Servlet redirige la requête vers une page JSP.
dispatcher.forward(request, response); : La méthode forward() envoie la requête et la réponse à la page JSP.
2. Création de la page JSP
Placez la page JSP dans le dossier WEB-INF/jsp/ pour éviter l'accès direct via l'URL.
Faites un clic droit sur WebContent > WEB-INF > jsp et sélectionnez New > JSP File.
Donnez le nom maPage.jsp à la page et cliquez sur Finish.
Voici un exemple de code pour la page JSP maPage.jsp :

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

```
  <head>
    <title>Page JSP</title>
  </head>
  <body>
    <h1>Transmission de variables de la Servlet à JSP avec des scripts</h1>

    <%
      // Récupération des attributs envoyés par la Servlet
      String message = (String) request.getAttribute("message");
      int annee = (Integer) request.getAttribute("annee");
    %>

    <!-- Affichage des variables -->
    <p>Message : <%= message %></p>
    <p>Année : <%= annee %></p>
  </body>
</html>
```

Explication :
<% %> : Ces balises JSP permettent d'inclure du code Java à l'intérieur de la page JSP. Le code entre ces balises est exécuté sur le serveur avant que la page ne soit envoyée au navigateur.

request.getAttribute("message") : Ici, nous utilisons la méthode getAttribute() de l'objet HttpServletRequest pour récupérer les valeurs envoyées depuis la Servlet.

<%= message %> : Cette syntaxe permet d'injecter directement le résultat d'une expression Java dans le code HTML. C'est une manière d'afficher des données dans la page.

Méthodes de transmission des variables :
setAttribute(String name, Object value) : Utilisée pour stocker des données dans l'objet HttpServletRequest. Ces données peuvent être de n'importe quel type Java (String, Integer, List, etc.).

Les Querry Strings : Paramétrés de la requête

Les query strings sont des chaînes de texte ajoutées à l'URL pour transmettre des informations d'une page ou d'une ressource à une autre dans une application web. En Java EE, elles peuvent être utilisées pour passer des informations depuis une Servlet à une JSP ou d'une page à une autre. Les query strings sont particulièrement utiles pour envoyer des données via une requête GET.

Fonctionnement des Query Strings dans Java EE
Une query string est une partie d'une URL qui commence par un point d'interrogation (?) et contient des paires clé-valeur séparées par des esperluettes (&). Par exemple :

http://localhost:8080/MonProjet/MaServlet?nom=Dazro&age=30

Dans cet exemple, la query string est ?nom=Dazro&age=30, où les paramètres nom et age sont transmis à la ressource (ici, une Servlet).

Étapes pour utiliser une Query String entre Servlet et JSP
1. Création d'une Servlet
Dans la Servlet, nous allons récupérer les paramètres passés dans l'URL (query string) et les traiter avant de les transmettre à une page JSP.

```
package com.monprojet;


import java.io.IOException;

import javax.servlet.ServletException;

import javax.servlet.annotation.WebServlet;

import javax.servlet.RequestDispatcher;

import javax.servlet.http.HttpServlet;

import javax.servlet.http.HttpServletRequest;

import javax.servlet.http.HttpServletResponse;


@WebServlet("/MaServlet")

public class MaServlet extends HttpServlet {

    private static final long serialVersionUID = 1L;


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

        // Récupérer les paramètres de la query string

        String nom = request.getParameter("nom");
```

```
        String age = request.getParameter("age");

        // Vérification des valeurs pour éviter les erreurs (paramètres optionnels)
        if (nom == null || nom.isEmpty()) {
            nom = "Inconnu";
        }
        if (age == null || age.isEmpty()) {
            age = "Non spécifié";
        }

        // Ajouter les valeurs récupérées en tant qu'attributs de la requête
        request.setAttribute("nom", nom);
        request.setAttribute("age", age);

        // Rediriger vers la page JSP pour afficher les données
        RequestDispatcher dispatcher =
request.getRequestDispatcher("/WEB-INF/jsp/afficherInfos.jsp");
        dispatcher.forward(request, response);
    }



    }
}
```

Explication :
request.getParameter("nom") et request.getParameter("age") : Ces méthodes récupèrent les valeurs des paramètres nom et age passés dans la query string de l'URL.
Si les paramètres sont vides ou non spécifiés, nous leur donnons une valeur par défaut.

2. Création de la page JSP
Dans la page JSP, nous allons afficher les paramètres récupérés par la Servlet.

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

```
<html>
  <head>
    <title>Afficher les informations</title>
  </head>
  <body>
    <h1>Informations récupérées via la Query String</h1>

    <p>Nom : ${nom}</p>
    <p>Âge : ${age}</p>
  </body>
</html>
```

Explication : A remplacer avec la technologie jsp .....…….

- **${nom}** et **${age}** : Ici, nous utilisons **l'Expression Language (EL)** pour afficher les variables nom et age envoyées par la Servlet.

3. Exécution de l'application
Pour tester cette interaction, accédez à l'URL suivante dans un navigateur :

http://localhost:8080/MonProjet/MaServlet?nom=Dazro&age=30

La page JSP affichera :

Nom : Dazro

Âge : 30

Si les paramètres nom ou age sont manquants, la page affichera les valeurs par défaut que nous avons définies dans la Servlet.

Points importants sur les Query Strings en Java EE

Requête GET : Les query strings sont principalement utilisées avec les requêtes GET. Les données sont visibles dans l'URL et limitées en taille (généralement à environ 2 000 caractères).

Sécurité : Évitez de transmettre des informations sensibles (comme des mots de passe) via une query string, car elles sont visibles dans l'URL.

Avantages :
Simple à utiliser pour transmettre des informations visibles (comme des filtres ou des recherches).
Utile pour les liens directs ou les formulaires simples.
Inconvénients :
Limité en taille (les navigateurs limitent généralement la longueur d'une URL).
Non sécurisé pour des informations confidentielles (puisque les données sont visibles dans l'URL).

JavaBean

Un JavaBean dans le contexte de Java EE est une classe Java qui adhère à certaines conventions spécifiques et est utilisée pour représenter un composant réutilisable qui peut être manipulé visuellement dans des outils de développement. En Java EE, les JavaBeans sont principalement utilisés pour représenter des objets métiers, ou des données, et peuvent être utilisés dans des Servlets, des JSP ou d'autres composants d'une application.

Caractéristiques d'un JavaBean
Un JavaBean doit respecter certaines conventions :
Constructeur sans argument : La classe doit avoir un constructeur par défaut (sans argument), ce qui permet à des frameworks ou à des conteneurs de l'instancier facilement.
Propriétés accessibles par des méthodes getter et setter : Chaque propriété de la classe doit être privée et accessible via des méthodes publiques get et set (pour respecter l'encapsulation).
Sérialisation : La classe JavaBean doit implémenter l'interface Serializable, ce qui permet à son état d'être enregistré ou restauré.

Exemple d'un JavaBean simple

Supposons que nous avons un JavaBean qui représente un utilisateur dans une application web.

```
package com.monprojet.beans;


import java.io.Serializable;


public class Utilisateur implements Serializable {
    private static final long serialVersionUID = 1L;


    private String nom;
    private String email;
    private int age;
```

```
// Constructeur sans argument
public Utilisateur() {
}

// Getter pour le nom
public String getNom() {
    return nom;
}

// Setter pour le nom
public void setNom(String nom) {
    this.nom = nom;
}

// Getter pour l'email
public String getEmail() {
    return email;
}

// Setter pour l'email
public void setEmail(String email) {
    this.email = email;
}

// Getter pour l'âge
public int getAge() {
    return age;
}

// Setter pour l'âge
public void setAge(int age) {
    this.age = age;
```

```
    }
}
```

2. Utilisation du JavaBean dans une Servlet
Dans une Servlet, on crée une instance de Utilisateur et on la transmet à la JSP via request.setAttribute().

```
package com.monprojet;

import com.monprojet.beans.Utilisateur;
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.RequestDispatcher;
import javax.servlet.http.HttpServlet;
import javax.servlet.http.HttpServletRequest;
import javax.servlet.http.HttpServletResponse;

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

    protected void doGet(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException {
        // Créer une instance de Utilisateur
        Utilisateur utilisateur = new Utilisateur();
        utilisateur.setNom("Dazro");
        utilisateur.setEmail("dazro@example.com");
        utilisateur.setAge(30);

        // Ajouter le JavaBean à la requête
        request.setAttribute("utilisateur", utilisateur);

        // Transférer à la page JSP
```

```
        RequestDispatcher dispatcher =
request.getRequestDispatcher("/WEB-INF/jsp/afficherUtilisateur.jsp");
        dispatcher.forward(request, response);
    }



    }
}
```

3. Utiliser le JavaBean dans la JSP

Dans la page JSP, nous allons utiliser du **code Java** classique pour récupérer le bean et afficher ses propriétés.

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
  <head>
    <title>Affichage de l'utilisateur</title>
  </head>
  <body>
    <h1>Détails de l'utilisateur</h1>

    <%
        // Récupérer le JavaBean depuis la requête
        com.monprojet.beans.Utilisateur utilisateur = (com.monprojet.beans.Utilisateur)
request.getAttribute("utilisateur");

        // Vérifier si l'utilisateur n'est pas null
        if (utilisateur != null) {
    %>
        <p>Nom : <%= utilisateur.getNom() %></p>
        <p>Email : <%= utilisateur.getEmail() %></p>
        <p>Âge : <%= utilisateur.getAge() %></p>
    <%
        } else {
```

```
    %>
        <p>Aucun utilisateur trouvé.</p>
    <%
        }
    %>
  </body>
</html>
```

Explication du code JSP :
request.getAttribute("utilisateur") : Nous récupérons l'attribut nommé utilisateur qui a été mis dans la requête par la Servlet.
Cast vers Utilisateur : Le bean est récupéré en tant qu'objet générique, donc nous devons le caster vers le type Utilisateur.
<%= utilisateur.getNom() %> : Cette syntaxe JSP permet d'injecter du code Java dans la page HTML pour afficher les propriétés du JavaBean.
Comparaison avec l'Expression Language (EL)
Sans EL : Vous avez un contrôle direct en utilisant du code Java dans la JSP, mais cela rend le code plus verbeux et mélange la logique métier avec la présentation. Ce n'est pas considéré comme une bonne pratique dans des applications modernes.

Avec EL : L'Expression Language est plus propre et facilite la séparation entre la logique et la présentation, rendant le code plus lisible et maintenable. EL est aussi plus sécurisé en limitant l'accès direct aux objets Java.

Avantages et Inconvénients de l'approche
Avantages :
Contrôle total avec Java : Vous pouvez manipuler directement vos objets en Java sans passer par un autre langage.
Compatibilité : Si vous avez besoin d'une rétrocompatibilité avec des versions plus anciennes de JSP, cette approche peut être utile.
Inconvénients :
Lisibilité : Le code JSP devient plus difficile à lire et à maintenir car il mélange du HTML et du Java.
Séparation des responsabilités : En utilisant directement du code Java dans les JSP, vous mélangez la logique métier et la présentation, ce qui va à l'encontre des principes MVC.

Etude de La Technologie Jsp

En JSP (JavaServer Pages), les balises jouent un rôle essentiel dans la création de pages dynamiques qui intègrent du code Java avec du contenu HTML. Voici une explication détaillée des différentes balises utilisées dans la technologie JSP, notamment les balises de commentaire, balises de déclaration, balises de scriptlet, et balises d'expression.

Les balises JSP

1. Balises de commentaire JSP
Les balises de commentaire JSP sont utilisées pour ajouter des commentaires qui ne sont pas visibles dans le code source HTML généré. Ces commentaires sont ignorés par le serveur et n'apparaissent pas dans la réponse envoyée au navigateur.Les balises de commentaire JSP sont utiles pour insérer des annotations dans le code source JSP sans impacter la sortie HTML visible par le client.

Syntaxe des commentaires JSP : <%-- Ceci est un commentaire JSP --%>

Exemple :

```
<%-- Ce commentaire est invisible dans le code HTML généré --%>
<html>
  <head>
    <title>Page JSP</title>
  </head>
  <body>
    <h1>Bienvenue sur ma page JSP</h1>
  </body>
</html>
```

2. Balises de déclaration JSP
Les balises de déclaration permettent de déclarer des variables ou des méthodes qui peuvent être utilisées ailleurs dans la page JSP. Ces déclarations sont converties en membres de la classe Java générée pour la page JSP. Cette balise pour définir des méthodes ou des variables que vous souhaitez utiliser tout au long de la page. Cependant, les déclarations persistent tout au long du cycle de vie de la JSP.

Syntaxe de déclaration JSP : <%! type nomVariable = valeur; %>

Exemples :

```
<%! int compteur = 0; %>


<html>
  <head>
    <title>Exemple de Déclaration</title>
  </head>
  <body>
    <h1>Compteur : <%= compteur %></h1>
```

```
  </body>
</html>
```

3. Balises de scriptlet JSP

Les balises de scriptlet permettent d'insérer du code Java dans une page JSP. Ce code Java est exécuté à chaque fois que la page JSP est demandée par le client. Les scriptlets peuvent être utilisés pour des boucles, des conditions, ou toute autre logique métier en Java.Les scriptlets sont utilisés pour insérer des blocs de code Java qui s'exécutent sur le serveur et dont les résultats sont intégrés au contenu HTML.

Syntaxe des scriptlets JSP : <% code Java %>

Exemple :

```
<html>
  <head>
    <title>Exemple de Scriptlet</title>
  </head>
  <body>
    <h1>Bienvenue sur la page JSP</h1>

    <%
        for(int i = 1; i <= 5; i++) {
            out.println("<p>Ligne numéro : " + i + "</p>");
        }
    %>
  </body>
</html>
```

4. Balises d'expression JSP

Les balises d'expression sont utilisées pour évaluer des expressions Java et les afficher directement dans la page web. Elles sont particulièrement utiles pour afficher des variables ou des résultats de calculs dans le HTML. La valeur de l'expression est automatiquement convertie en chaîne de caractères et insérée dans la réponse HTML.Les balises d'expression sont idéales pour afficher des résultats simples comme des variables, des méthodes ou des objets Java directement dans le contenu HTML.

Syntaxe des expressions JSP : <%= expression %>

Exemples :

```
<html>
  <head>
    <title>Exemple de Balise d'Expression</title>
  </head>
  <body>
    <h1>La date actuelle est : <%= new java.util.Date() %></h1>
  </body>
</html>
```

Différence entre balises de scriptlet et balises d'expression
Les balises de scriptlet (<% ... %>) exécutent des blocs de code Java, mais n'affichent pas directement de texte dans le HTML (à moins d'utiliser explicitement out.println()).
Les balises d'expression (<%= ... %>) évaluent une expression et affichent immédiatement le résultat dans la page HTML sans avoir besoin d'utiliser out.println().

Les Directives Jsp

Les directives JSP sont des instructions spéciales utilisées dans une page JSP pour donner des informations au conteneur JSP sur la manière dont la page doit être traitée. Elles contrôlent principalement la manière dont la page est compilée et exécutée. Il existe trois principales directives en JSP :

Directive page
Directive include
Directive taglib
Chacune d'entre elles joue un rôle spécifique dans le cycle de vie et le comportement de la page JSP.

1. Directive page
La directive page fournit des informations au conteneur JSP concernant la configuration de la page elle-même. Elle permet de définir des paramètres comme l'encodage, les imports de classes Java, la gestion des exceptions, etc.

Syntaxe : <%@ page attribut="valeur" %>

Principaux attributs de la directive page :

Attribut	Description
`language`	Définit le langage de programmation utilisé dans la page JSP. Par défaut, c'est Java (language="java").
`contentType`	Définit le type de contenu de la réponse. Par exemple, contentType="text/html;charset=UTF-8" pour définir que la page renvoie du HTML avec encodage UTF-8.
`import`	Permet d'importer des classes Java comme dans une classe Java classique (par exemple, import="java.util.Date").
`session`	Indique si la page JSP utilise ou non la session (session="true" ou session="false").
`isErrorPage`	Indique si la page est une page d'erreur (isErrorPage="true") qui peut capturer des exceptions.
`errorPage`	Indique la page à laquelle rediriger en cas d'erreur.
`isELIgnored`	Indique si l'**Expression Language (EL)** doit être désactivée pour cette page (isELIgnored="true").

Exemple :

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ page import="java.util.Date" %>
<html>
  <head>
    <title>Exemple de directive page</title>
  </head>
  <body>
    <h1>Date actuelle : <%= new Date() %></h1>
  </body>
</html>
```

2. Directive include

La directive include permet d'inclure une autre page JSP ou un fichier statique (comme un fichier HTML ou un fichier texte) au moment de la compilation de la page. L'inclusion est statique, ce qui signifie que le contenu du fichier est ajouté à la page principale avant que la JSP ne soit compilée en servlet.

Syntaxe : `<%@ include file="chemin_du_fichier" %>`

Exemples :

```
<html>
<head>
  <title>Page principale</title>
</head>
<body>
  <h1>Contenu principal</h1>
  <%@ include file="footer.jsp" %>
</body>
</html>
```

Dans cet exemple, le fichier footer.jsp est inclus dans la page principale au moment de la compilation.

Inclusion statique : Le contenu de footer.jsp est inséré dans la page avant la compilation de la JSP. Toute modification apportée à footer.jsp après la compilation ne sera pas prise en compte dans la page déjà compilée.

3. Directive taglib
La directive taglib est utilisée pour déclarer une bibliothèque de balises personnalisées (Taglib) dans une page JSP. Les bibliothèques de balises permettent d'étendre la fonctionnalité de JSP avec des balises supplémentaires, souvent utilisées pour les applications web basées sur des frameworks comme JSTL (JSP Standard Tag Library).

Syntaxe : <%@ taglib uri="uri_de_la_taglib" prefix="préfixe" %>

uri : Représente l'URI de la bibliothèque de balises. Il peut s'agir d'un chemin d'accès local ou d'un URI global, comme celui utilisé pour JSTL.
prefix : Définit un préfixe qui sera utilisé dans la page pour accéder aux balises de cette bibliothèque.

Exemples :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head>
  <title>Exemple de JSTL</title>
</head>
<body>
```

```
    <c:forEach var="i" begin="1" end="5">
      <p>Numéro : ${i}</p>
    </c:forEach>
  </body>
</html>
```

Dans cet exemple, la bibliothèque de balises JSTL core est importée avec le préfixe c, permettant l'utilisation de balises comme <c:forEach>.

Comparaison des directives

Directive	Description	Usage
`page`	Configure divers aspects de la page JSP (type de contenu, encodage, session, etc.).	Utilisée pour des paramètres globaux qui affectent la page entière.
`include`	Inclut un fichier au moment de la compilation (inclusion statique).	Utilisée pour inclure du contenu réutilisable comme des en-têtes, pieds de page, etc.
`taglib`	Déclare une bibliothèque de balises personnalisées pour étendre les fonctionnalités JSP.	Utilisée pour ajouter des bibliothèques de balises comme JSTL, facilitant la création d'UI.

JSTL (bibliothèque de balises standard JSP)

La JSTL (JavaServer Pages Standard Tag Library) est une bibliothèque de balises standard pour les pages JSP. Elle fournit un ensemble de balises prêtes à l'emploi pour faciliter le développement des applications web sans avoir à écrire du code Java dans les pages JSP. La JSTL est conforme aux spécifications de la technologie JSP et offre des balises pour la manipulation des données, les boucles, les conditions, les expressions régulières, la gestion des dates, les appels à des services externes, etc.

JSTL aide à séparer la logique de présentation (front-end) de la logique métier, ce qui favorise une approche plus propre et maintenable dans le développement des applications Java EE.

Les principales fonctionnalités de la JSTL sont réparties en plusieurs bibliothèques de balises :

1. **JSTL Core** (c)
2. **JSTL Format** (fmt)
3. **JSTL SQL** (sql)
4. **JSTL XML** (x)
5. **JSTL Functions** (fn)

1. **JSTL Core** (c)

La bibliothèque **JSTL Core** contient les balises les plus couramment utilisées dans les pages JSP pour les opérations de contrôle de flux, les boucles, la gestion des variables et les imports. C'est la bibliothèque centrale que vous utiliserez presque toujours dans une page JSP.

Directive d'import pour JSTL Core :

<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

Balises principales de JSTL Core :
<c:out> : Affiche une valeur (similaire à <%= ... %> en JSP, mais avec des fonctionnalités supplémentaires comme l'échappement des caractères spéciaux HTML).

Exemples :

<c:out value="${variable}" default="Valeur par défaut" />

<c:import> : Permet d'inclure le contenu d'une autre ressource (page JSP, servlet ou page externe) dans la page courante. On utilise cette balise d'importation, le contenu d'un autre serveur FTP et d'un autre site Web est accessible.

Exemples :

<c:import url="footer.jsp" />

<c:set> : Assigne une valeur à une variable dans un scope donné (portée : page, request, session ou application)

Exemples :

<c:set var="nom" value="Dazro" scope="request" />

<c:remove> : Supprime une variable de l'un des scopes

Exemples :

<c:remove var="nom" scope="request" />

<c:if> : Exécute du contenu conditionnellement, si une expression donnée est vraie.

Exemples :

```
<c:if test="${age > 18}">
   <p>Vous êtes majeur.</p>
</c:if>
```

<c:choose>, <c:when>, <c:otherwise> : Fournit un comportement similaire à une instruction switch ou if-else en Java.,

Exemples :

```
<c:choose>
   <c:when test="${age >= 18}">
      <p>Vous êtes majeur.</p>
   </c:when>
   <c:otherwise>
      <p>Vous êtes mineur.</p>
   </c:otherwise>
</c:choose>
```

<c:forEach> : Permet de boucler sur des collections comme les listes, tableaux ou maps.

Exemples

```
<c:forEach var="item" items="${maListe}">
   <p>${item}</p>
</c:forEach>
```

<c:forTokens> : Boucle sur des éléments séparés par des délimiteurs dans une chaîne de caractères.

Exemples

```
<c:forTokens items="java,jsp,jstl" delims="," var="token">
   <p>${token}</p>
</c:forTokens>
```

<c:param>

La balise **<c:param>** est utilisée pour ajouter des paramètres à une URL ou à une requête. Elle peut être utilisée conjointement avec <c:redirect>, <c:url>, ou même dans des formulaires pour transmettre des informations dans la chaîne de requête (query string).

Exemples

```
<c:url var="maUrl" value="page.jsp">
```

```
    <c:param name="param1" value="valeur1" />
    <c:param name="param2" value="valeur2" />
</c:url>
<a href="${maUrl}">Lien vers page.jsp avec paramètres</a>
```

Dans cet exemple, les paramètres param1 et param2 sont ajoutés à l'URL, et le lien généré sera de la forme :
page.jsp?param1=valeur1&param2=valeur2.

Utilisation de <c:param> avec <c:redirect> :

```
<c:redirect url="autrePage.jsp">
    <c:param name="nom" value="Dazro" />
    <c:param name="age" value="30" />
</c:redirect>
```

Dans cet exemple, la page autrePage.jsp sera redirigée avec les paramètres nom et age dans la requête, résultant en une URL :
autrePage.jsp?nom=Dazro&age=30.

<c:url> est utilisée pour créer des URLs dynamiques dans les pages JSP. Elle permet d'ajouter des paramètres à une URL et de gérer le contexte de l'application. Le contexte de l'application est automatiquement ajouté à l'URL si nécessaire.

Exemples :

```
<c:url var="monLien" value="/affichage.jsp">
    <c:param name="id" value="123" />
</c:url>
<a href="${monLien}">Afficher détails</a>
```

Dans cet exemple, l'URL générée sera http://localhost:8080/MonProjet/affichage.jsp?id=123, avec le contexte d'application ajouté automatiquement (/MonProjet).

<c:redirect> est utilisée pour effectuer une redirection HTTP côté serveur. Contrairement à la redirection côté client (qui utilise response.sendRedirect()), cette balise simplifie le processus en ajoutant des paramètres directement dans la balise, si nécessaire.

Exemples :

```
<c:redirect url="page2.jsp">
```

```
    <c:param name="lang" value="fr" />
    <c:param name="theme" value="dark" />
</c:redirect>
```

Dans cet exemple, l'utilisateur sera redirigé vers page2.jsp?lang=fr&theme=dark.

Différence entre <c:redirect> et <c:forward> :

<c:redirect> : Fait une redirection HTTP, ce qui envoie une réponse au client pour lui indiquer de faire une nouvelle requête vers l'URL spécifiée. L'URL dans le navigateur sera mise à jour.

<c:forward> : Transfère la requête au sein du serveur vers une autre ressource sans impliquer le client. L'URL dans le navigateur ne change pas.

Combinaison des balises <c:url>, <c:param> et <c:redirect>

Voici un exemple complet qui combine ces balises pour ajouter des paramètres à une URL et ensuite rediriger l'utilisateur vers cette URL générée :

```
<c:url var="urlRedir" value="result.jsp">
    <c:param name="user" value="Dazro" />
    <c:param name="age" value="30" />
</c:url>

<c:redirect url="${urlRedir}" />
```

Dans cet exemple :

1. L'URL result.jsp est créée avec les paramètres user et age.
2. L'utilisateur est redirigé vers cette URL générée : result.jsp?user=Dazro&age=30.

Balises de fonction JSTL

La bibliothèque JSTL (JavaServer Pages Standard Tag Library) propose des fonctionnalités pratiques pour gérer les boucles, les conditions, la manipulation de données, etc. Parmi les différentes bibliothèques JSTL, la bibliothèque des fonctions (JSTL Functions Library, souvent

importée avec le préfixe fn) permet de manipuler les chaînes de caractères et de réaliser des opérations utiles sur celles-ci.Vous devez importer cette bibliothèque afin de pouvoir l'utiliser comme ceci :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/functions" prefix="fn" %>
```

fn:contains()

Cette fonction vérifie si une chaîne de caractères contient une autre sous-chaîne.

Syntaxe : fn:contains(string, substring)

Exemple :

```
${fn:contains("JavaServer Pages", "Pages")}  <!-- Renvoie true -->
```

fn:containsIgnoreCase()

Elle fonctionne comme `fn:contains()` mais ignore la casse des caractères.

Syntaxe : fn:containsIgnoreCase(string, substring)`

Exemple:

```
${fn:containsIgnoreCase("JavaServer Pages", "pages")}  <!-- Renvoie true -->
```

fn:startsWith()

Cette fonction vérifie si une chaîne commence par une autre sous-chaîne.

Exemple:

```
${fn:startsWith("JavaServer Pages", "Java")}  <!-- Renvoie true -->
```

fn:endsWith()

Elle vérifie si une chaîne se termine par une autre sous-chaîne.

Syntaxe : fn:endsWith(string, suffix)`

Exemple :

```
${fn:endsWith("JavaServer Pages", "Pages")}  <!-- Renvoie true -->
```

fn:indexOf()

Cette fonction retourne l'index de la première occurrence d'une sous-chaîne dans une chaîne donnée.

Syntaxe : fn:indexOf(string, substring)`

Exemple :

```
${fn:indexOf("JavaServer Pages", "Server")}  <!-- Renvoie 4 -->
```

fn:length()

Elle retourne la longueur d'une chaîne, d'un tableau, ou d'une collection.

Syntaxe : fn:length(object)`

Exemple:

${fn:length("Java")} <!-- Renvoie 4 -->

fn:substring()

Cette fonction renvoie une sous-chaîne d'une chaîne donnée à partir de l'index de départ et jusqu'à l'index de fin (non inclus).

Syntaxe: fn:substring(string, begin, end)`

Exemple:

${fn:substring("JavaServer Pages", 0, 10)} <!-- Renvoie "JavaServer" -->

fn:substringAfter()

Elle renvoie la partie de la chaîne située après une sous-chaîne donnée.

Syntaxe: fn:substringAfter(string, substring)`

Exemple :

${fn:substringAfter("JavaServer Pages", "Server")} <!-- Renvoie " Pages" -->

fn:substringBefore()

Elle renvoie la partie de la chaîne située avant une sous-chaîne donnée.

Syntaxe: fn:substringBefore(string, substring)`

Exemple :

${fn:substringBefore("JavaServer Pages", "Server")} <!-- Renvoie "Java" -->

fn:join()

Cette fonction joint les éléments d'un tableau ou d'une liste en une seule chaîne, avec un délimiteur spécifié.

Syntaxe: fn:join(array, delimiter)`

Exemple :

${fn:join(["Java", "JSP", "Servlet"], ", ")} <!-- Renvoie "Java, JSP, Servlet" -->

fn:split()

Elle découpe une chaîne en un tableau en fonction d'un délimiteur donné.

Syntaxe : fn:split(string, delimiter)

- Exemple :

${fn:split("Java, JSP, Servlet", ",")} <!-- Renvoie ["Java", "JSP", "Servlet"] -->

fn:replace()

Cette fonction remplace toutes les occurrences d'une sous-chaîne dans une chaîne par une autre sous-chaîne.

Syntaxe : fn:replace(string, substring, replacement)`

Exemple:

```
  ${fn:replace("JavaServer Pages", "Pages", "Servlet")}  <!-- Renvoie "JavaServer Servlet" -->
  ```
```

fn:toLowerCase()

Elle convertit une chaîne en minuscules.

Syntaxe : fn:toLowerCase(string)

Exemple :

```
 ${fn:toLowerCase("JavaServer Pages")}  <!-- Renvoie "javaserver pages" -->
```

fn:toUpperCase()

Elle convertit une chaîne en majuscules.

Syntaxe: fn:toUpperCase(string)

Exemple :

```
 ${fn:toUpperCase("JavaServer Pages")}  <!-- Renvoie "JAVASERVER PAGES" -->
```

fn:trim()

Cette fonction supprime les espaces en début et en fin d'une chaîne.

Syntaxe : fn:trim(string)

Exemple:

```
  ${fn:trim("   JavaServer Pages   ")}  <!-- Renvoie "JavaServer Pages" -->
```

Les balises de formatage de JSTL

Les balises de formatage de JSTL (JavaServer Pages Standard Tag Library) sont fournies par la bibliothèque **JSTL Formatting**. Elles sont particulièrement utiles pour le formatage des nombres, des dates, des messages et des paramètres, tout en prenant en compte les paramètres régionaux (locales) et les conventions de formatage spécifiques à une langue ou à une région.

Voici les principales balises de formatage de la JSTL, souvent utilisées avec le préfixe **fmt** après avoir importé la bibliothèque :
```

<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>

`<fmt:formatNumber>`

Cette balise permet de formater les nombres en fonction du style choisi (nombre simple, devise, pourcentage).

Syntaxe :

<fmt:formatNumber value="nombre" type="type" pattern="modèle" />

Attributs principaux :
value : Le nombre à formater.
type : Peut être number (nombre simple), currency (devise), ou percent (pourcentage).
pattern : Un modèle personnalisé pour le formatage

Exemples :

<fmt:formatNumber value="123456.789" type="number" /> <!-- 123,456.789 -->

<fmt:formatNumber value="123456.789" type="currency" /> <!-- $123,456.79 (selon la locale) -->

<fmt:formatNumber value="0.85" type="percent" /> <!-- 85% -->

<fmt:parseNumber>

Cette balise analyse (convertit) une chaîne représentant un nombre ou un pourcentage dans un format spécifique.

Syntaxe :

<fmt:parseNumber value="string" type="type" pattern="modèle" />

Exemples

<fmt:parseNumber value="123,456.78" var="numberVar" />

Cet exemple convertit la chaîne "123,456.78" en un objet de type Number.

`<fmt:formatDate>`

Cette balise formate une date ou une heure en fonction d'un style ou d'un modèle spécifié.

Syntaxe : <fmt:formatDate value="date" type="type" pattern="modèle" />

Attributs :

value : La date à formater (peut être un objet java.util.Date ou une chaîne).

type : Peut être date, time ou both (date et heure).
pattern : Un modèle personnalisé (comme dd-MM-yyyy).

Exemples

<fmt:formatDate value="${now}" type="date" dateStyle="short" /> <!-- 10/15/2023 -->

<fmt:formatDate value="${now}" pattern="dd/MM/yyyy HH:mm:ss" /> <!-- 15/10/2023 14:30:15 -->

<fmt:parseDate>
Cette balise analyse une chaîne représentant une date et la convertit en un objet Date.

Syntaxe :<fmt:parseDate value="string" pattern="modèle" var="variable" />

Exemples :

<fmt:parseDate value="15/10/2023" pattern="dd/MM/yyyy" var="dateVar" />

Ce code analyse la chaîne "15/10/2023" et la convertit en une date Java.

<fmt:setLocale>
Cette balise permet de définir les paramètres régionaux (locale) utilisés pour le formatage dans une page JSP.
Syntaxe : <fmt:setLocale value="locale" />

Exemples

<fmt:setLocale value="fr_FR" />

Après cette balise, toutes les balises de formatage utiliseront la locale française (France)

`<fmt:setTimeZone>`

Cette balise définit le fuseau horaire utilisé pour formater les dates et heures.

Syntaxe : <fmt:setTimeZone value="timezone" />

Exemples

<fmt:setTimeZone value="GMT+1" />

Ce code configure le fuseau horaire pour utiliser GMT+1.

`<fmt:message>`

Cette balise permet de récupérer des messages internationalisés à partir d'un fichier de ressources (resource bundle). Elle est utile pour la localisation (i18n).

Syntaxe : <fmt:message key="cléMessage" />

Exemples

<fmt:message key="hello.message" />

Ce code affiche le message associé à la clé hello.message dans le fichier de ressources.

<fmt:bundle>
Cette balise permet de définir un fichier de ressources (resource bundle) qui sera utilisé pour récupérer les messages internationalisés.

Syntaxe : <fmt:bundle basename="nomDuBundle"> <!-- Code ici --></fmt:bundle>
Exemple :
<fmt:bundle basename="messages">
  <fmt:message key="hello.message" />
</fmt:bundle>
Ce code récupère les messages dans le fichier messages.properties.

. <fmt:requestEncoding>
Cette balise permet de définir l'encodage utilisé pour traiter les paramètres de requêtes HTTP.

Syntaxe : <fmt:requestEncoding value="encodage" />
Exemple :
<fmt:requestEncoding value="UTF-8" />

<fmt:timeZone>
Elle permet de définir un fuseau horaire dans une portée limitée.

Syntaxe :<fmt:timeZone value="timeZone"> <!-- Balises de formatage de date/heure ici -->
</fmt:timeZone>
Exemple :
<fmt:timeZone value="PST">
  <fmt:formatDate value="${now}" type="both" />
</fmt:timeZone>
Dans cet exemple, la date et l'heure seront formatées selon le fuseau horaire du Pacifique (PST).

Les Balises xml de JSTL

Les balises XML de la JSTL (JavaServer Pages Standard Tag Library) sont utilisées pour manipuler des documents XML dans des pages JSP. Ces balises permettent de lire, parcourir, et transformer des documents XML directement dans une page JSP, offrant ainsi une gestion simplifiée du XML sans avoir à écrire beaucoup de code Java.

Elles sont généralement utilisées avec le préfixe x après avoir importé la bibliothèque XML de JSTL, comme suit :
<%@ taglib uri="http://java.sun.com/jsp/jstl/xml" prefix="x" %>

Principales balises XML de JSTL
<x:parse>
Cette balise permet de lire et de parser un document XML à partir d'une URL, d'un fichier ou d'une chaîne, pour l'utiliser dans les autres balises XML de JSTL.

Syntaxe : <x:parse var="xmlDoc" xml="${xmlSource}" />
Attributs :
var : Le nom de la variable dans laquelle le document XML est stocké.
xml : L'URL, la chaîne ou l'objet d'entrée XML à parser.
Exemple :
<x:parse var="xmlDoc" xml="${'http://example.com/data.xml'}" />

<x:out>
Cette balise extrait et affiche une partie d'un document XML (ou XPath) spécifié.

Syntaxe :
<x:out select="expressionXPath" />
Attributs :
select : L'expression XPath pour extraire le contenu souhaité.
Exemple :
<x:out select="$xmlDoc//book/title" />
Cet exemple extrait et affiche les titres de tous les éléments <book>.
<x:set>
La balise <x:set> permet de définir une variable ou de stocker une partie de l'arbre XML correspondant à une expression XPath.

Syntaxe :
<x:set var="variable" select="expressionXPath" />
Attributs :
var : Le nom de la variable où l'expression XPath est stockée.
select : L'expression XPath qui sélectionne la partie du document XML.
Exemple :
<x:set var="bookTitles" select="$xmlDoc//book/title" />
Ici, les titres des livres sont stockés dans la variable bookTitles.
<x:forEach>
La balise <x:forEach> est utilisée pour itérer sur les nœuds sélectionnés dans un document XML

avec une expression XPath, similaire à la boucle c:forEach.
Syntaxe :
<x:forEach select="expressionXPath" var="variable"> <!-- Contenu -->
</x:forEach>
Attributs :
select : L'expression XPath définissant les nœuds à itérer.
var : Le nom de la variable pour chaque élément itéré.
Exemple :
<x:forEach select="$xmlDoc//book" var="book">
  <x:out select="$book/title" />
</x:forEach>
Cet exemple affiche les titres de tous les éléments <book> trouvés dans le document XML.
 <x:if>
Cette balise évalue une condition basée sur une expression XPath et affiche le contenu à l'intérieur du bloc si la condition est vraie.
Syntaxe :
<x:if select="expressionXPath">
  <!-- Contenu à afficher si vrai -->
</x:if>
Attributs :
select : L'expression XPath qui détermine si le contenu doit être affiché.
Exemple :
<x:if select="$xmlDoc//book[@category='fiction']">
  <x:out select="$xmlDoc//book/title" />
</x:if>
Ce code vérifie si un livre appartient à la catégorie "fiction" et affiche son titre.
 <x:choose>, <x:when>, <x:otherwise>
Ces balises permettent de réaliser des conditions complexes, similaires à une structure switch ou if-else-if.

Syntaxe :
<x:choose>
  <x:when select="expressionXPath">
    <!-- Contenu -->
  </x:when>
  <x:otherwise>
    <!-- Contenu par défaut -->
  </x:otherwise>
</x:choose>
Exemple :
<x:choose>
  <x:when select="$xmlDoc//book[@category='fiction']">
    Fiction Book: <x:out select="$xmlDoc//book/title" />
  </x:when>

```
  <x:otherwise>
    Non-Fiction Book: <x:out select="$xmlDoc//book/title" />
  </x:otherwise>
</x:choose>
```
<x:transform>
Cette balise est utilisée pour transformer un document XML à l'aide d'une feuille de style XSLT.
Syntaxe :
<x:transform xml="sourceXML" xslt="sourceXSLT" />
Attributs :
xml : Le document XML à transformer.
xslt : Le document XSLT qui définit la transformation.
Exemple :
<x:transform xml="${xmlDoc}" xslt="${'styles.xsl'}" />
Ce code applique une transformation XSLT sur le document XML.
<x:remove>
Cette balise est utilisée pour supprimer des nœuds dans un document XML en fonction d'une expression XPath.
Syntaxe :
<x:remove select="expressionXPath" />
Exemple :
<x:remove select="$xmlDoc//book[@category='old']" />
Ce code supprime tous les livres ayant une catégorie "old" du document XML.

Les balises sql de jstl

Les balises SQL de JSTL (JavaServer Pages Standard Tag Library) permettent d'exécuter des requêtes SQL directement dans une page JSP, offrant ainsi un moyen simple et rapide d'interagir avec une base de données dans une application Java EE sans avoir besoin d'écrire du code Java complexe.

Ces balises SQL sont généralement utilisées avec le préfixe sql après avoir importé la bibliothèque SQL de JSTL :
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>

<sql:setDataSource>
Cette balise est utilisée pour définir la source de données (base de données) que les autres balises SQL vont utiliser pour exécuter des requêtes.
Syntaxe :
<sql:setDataSource var="db" driver="driverClass" url="jdbcUrl" user="nomUtilisateur" password="motDePasse" />
Attributs :
var : Nom de la variable qui stocke la source de données.

driver : Le nom de la classe du driver JDBC (ex : com.mysql.cj.jdbc.Driver pour MySQL).
url : L'URL JDBC de la base de données (ex : jdbc:mysql://localhost:3306/nomBase).
user : Le nom d'utilisateur pour se connecter à la base de données.
password : Le mot de passe pour l'utilisateur.
Exemple :

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

<sql:query>
Cette balise exécute une requête SQL de type SELECT et stocke les résultats dans une variable pour une utilisation ultérieure dans la page JSP.
Syntaxe :

```
<sql:query dataSource="dataSource" var="result">
  SELECT * FROM table
</sql:query>
```

Attributs :
dataSource : La source de données définie avec <sql:setDataSource>.
var : Le nom de la variable qui stocke le résultat de la requête (généralement un objet ResultSet).

Exemple :

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

Cet exemple exécute une requête qui sélectionne tous les enregistrements de la table employees et stocke le résultat dans la variable result.

<sql:update>
Cette balise exécute une requête SQL de type INSERT, UPDATE, ou DELETE, ou toute autre requête qui ne retourne pas de données (par exemple, la création de tables).
Syntaxe :

```
<sql:update dataSource="dataSource" var="rowsAffected">
  INSERT INTO table (col1, col2) VALUES (?, ?)
</sql:update>
```

Attributs :
dataSource : La source de données définie avec <sql:setDataSource>.
var : Nom de la variable qui stocke le nombre de lignes affectées par la requête.
Exemple :

```
<sql:update dataSource="${db}" var="rowsAffected">
  INSERT INTO employees (name, role) VALUES ('John Doe', 'Manager')
</sql:update>
```

Ce code insère un nouvel employé dans la table employees et stocke le nombre de lignes affectées dans rowsAffected.

<sql:param>
Cette balise est utilisée pour passer des paramètres à une requête SQL. Elle est particulièrement

utile pour prévenir les injections SQL et pour rendre les requêtes plus dynamiques.
Syntaxe :
<sql:param value="valeur" />
Attributs :
value : La valeur du paramètre à passer dans la requête SQL.
Exemple :

```
<sql:query dataSource="${db}" var="result">
  SELECT * FROM employees WHERE id = ?
  <sql:param value="${employeeId}" />
</sql:query>
```

Dans cet exemple, la valeur de employeeId est passée comme paramètre dans la requête SELECT.

<sql:transaction>
Cette balise permet de grouper plusieurs opérations SQL dans une transaction. Si une opération échoue, toutes les autres seront annulées (rollback).
Syntaxe :

```
<sql:transaction dataSource="dataSource">
  <!-- Opérations SQL ici -->
</sql:transaction>
```

Attributs :
dataSource : La source de données définie avec <sql:setDataSource>.
Exemple :

```
<sql:transaction dataSource="${db}">
  <sql:update>
    DELETE FROM employees WHERE id = ?
    <sql:param value="${employeeId}" />
  </sql:update>
  <sql:update>
    INSERT INTO audit_log (action, employee_id) VALUES ('delete', ?)
    <sql:param value="${employeeId}" />
  </sql:update>
</sql:transaction>
```

Cet exemple regroupe la suppression d'un employé et l'insertion d'une ligne dans une table d'audit dans une transaction unique.
<sql:dateParam>
Cette balise est utilisée pour passer un paramètre de type Date dans une requête SQL.
Syntaxe :

```
<sql:dateParam value="dateValue" type="date|time|timestamp" />
```

Attributs :
value : La valeur du paramètre de date à passer.
type : Le type de date (date, time ou timestamp).
Exemple :

```
<sql:query dataSource="${db}" var="result">
  SELECT * FROM orders WHERE order_date = ?
  <sql:dateParam value="${orderDate}" type="date" />
</sql:query>
```

Exemple Complet d'Utilisation des Balises SQL

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%@ taglib uri="http://java.sun.com/jsp/jstl/sql" prefix="sql" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<html>
<head>
  <title>Exemple JSTL SQL</title>
</head>
<body>

<!-- Définir la source de données -->
<sql:setDataSource var="db" driver="com.mysql.cj.jdbc.Driver"
                url="jdbc:mysql://localhost:3306/mydb"
                user="root" password="password" />

<!-- Exécuter une requête SELECT -->
<sql:query dataSource="${db}" var="result">
  SELECT * FROM employees
</sql:query>

<!-- Afficher les résultats -->
<table border="1">
  <tr>
    <th>ID</th>
    <th>Nom</th>
    <th>Rôle</th>
  </tr>
  <c:forEach var="row" items="${result.rows}">
    <tr>
      <td><c:out value="${row.id}" /></td>
      <td><c:out value="${row.name}" /></td>
      <td><c:out value="${row.role}" /></td>
    </tr>
  </c:forEach>
</table>

</body>
</html>
```

Pratique .....…..…..…...……...……..