INF1521 Développement d'Application avancée : Java EE & Spring Boot

Profs: AMIDOU Abdou-Raoufou
E-mail: amidouabdouraoufou@gmail.com Institut de Formation aux Normes et Technologies de l'Informatique
Sokodé - Togo

2025-09-23

Sommaire

Ce cours présente les bases de Java EE, un ensemble de spécifications pour développer des applications d'entreprise robustes, modulaires et évolutives. Nous abordons les composants essentiels tels que les servlets, JSP et EJB, afin de comprendre le fonctionnement des applications web et distribuées en Java. L'objectif est d'acquérir une vision claire de l'architecture et des outils qu'offre Java EE pour la construction de solutions professionnelles. Dans le cours suivant, nous prolongerons cette étude en introduisant Spring Boot, un framework moderne qui simplifie et accélère le développement d'applications en s'appuyant sur les concepts vus avec Java EE.

Contents

1 Partie 1 : Les généralités de Java EE

1.1 Introducrion au langage Java EE

Java EE (**Java Platform, Enterprise Edition**) est une plateforme logicielle développée initialement par **Sun Microsystems**, puis gérée par **Oracle**, et aujourd'hui par la fondation **Eclipse** sous le nom **Jakarta EE**. Elle est conçue pour faciliter le développement d'applications d'entreprise robustes, sécurisées et distribuées, comme des applications web, des services REST/SOAP, ou encore des systèmes bancaires.

1.1.1 Définition

Java EE est une **extension de Java SE** (Java Standard Edition) qui ajoute des **API** et des **spécifications** destinées aux applications d'entreprise. Elle fournit :

- Des **composants serveurs** (Servlets, JSP, JSF) pour créer des applications web.
- Des **services métiers** (EJB – Enterprise JavaBeans, JPA – Java Persistence API).
- Des **outils de communication** (JAX-RS pour REST, JAX-WS pour SOAP).
- Des **mécanismes de sécurité et de transactions** (JAAS, JTA).

1.1.2 Prérequis

Avant d'apprendre Java EE, il est conseillé de maîtriser :

- **Java SE** (bases du langage, POO, collections, exceptions, threads).
- **Concepts du web** (HTTP, HTML, CSS, éventuellement un peu de JavaScript côté client).
- **Notions de bases de données** (SQL, JDBC).
- **Notions sur les serveurs applicatifs** (Tomcat, WildFly, GlassFish, Payara).

1.1.3 Comparaison avec Java SE

- **Java SE** :
  - Base du langage Java.
  - Sert à développer des applications *standalone*, outils desktop, ou petites applis.
  - Fournit les bibliothèques de base (Collections, I/O, Concurrency, etc.).
- **Java EE** :
  - Extension de Java SE pour des applications multi-couches et distribuées.
  - Fournit des API avancées (JPA, JMS, JAX-RS, EJB).
  - S'exécute sur des serveurs d'applications spécialisés.

En résumé : Java SE = fondations, Java EE = solutions avancées pour applications professionnelles.

1.1.4 Comparaison avec JavaScript

- **Java EE** :
  - Langage Java, compilé en bytecode et exécuté sur la JVM.

  - Utilisé côté serveur pour gérer la logique métier, la persistance et la sécurité.
  - Typé statiquement, fortement orienté objet.
- **JavaScript** :
  - Langage interprété, exécuté dans le navigateur (ou côté serveur avec Node.js).
  - Utilisé principalement côté client pour rendre les pages web dynamiques et interactives.
  - Typé dynamiquement et orienté objet basé sur prototypes.

NB : malgré la ressemblance des noms, Java EE et JavaScript n'ont quasiment rien en commun : le premier est une plateforme serveur lourde, le second un langage de script léger.

1.1.5 Cas d'utilisation concrets de Java EE

Java EE est utilisé principalement dans les grandes applications critiques nécessitant robustesse, sécurité, scalabilité et interopérabilité. Quelques exemples concrets :

- **Applications bancaires et financières** : gestion de comptes, virements, services en ligne.
  - Utilisation de JTA pour les transactions sécurisées.
  - Haute sécurité et scalabilité pour des millions d'utilisateurs.
- **Applications de e-commerce** : sites de vente en ligne.
  - Gestion des sessions et paniers avec Servlets et JSP.
  - Persistance avec JPA.
  - Déploiement sur des serveurs robustes (WildFly, GlassFish, Payara).
- **Systèmes de gestion d'entreprise (ERP/CRM)** : ressources humaines, stock, facturation.
  - Architecture multi-couches (présentation, logique métier, persistance).
  - Services REST avec JAX-RS et SOAP avec JAX-WS.
- **Applications gouvernementales et administratives** : portails de services publics en ligne.
  - Grande fiabilité et sécurité renforcée.
  - Interopérabilité via normes ouvertes.
- **Télécommunications et systèmes distribués** : plateformes de gestion de SMS, facturation opérateurs.
  - Communication asynchrone avec JMS.
  - Haute disponibilité et performance.
- **Applications de santé** : gestion des dossiers médicaux électroniques, rendez-vous patients.
  - Respect des normes de confidentialité et sécurité des données.
  - Intégration avec d'autres systèmes via web services.

1.2 Under the hood(sous le capot) de Java EE / Jakarta EE

1.2.1 Comment ça fonctionne ?

Java EE repose sur une **architecture en couches** qui facilite le développement et la maintenance.

Client Le client peut être un navigateur, une application mobile ou un autre service. Il envoie des requêtes HTTP/HTTPS ou utilise des API REST/SOAP.

Web Layer (couche Web) Gérée par des **Servlets**, **JSP** ou des frameworks comme **JSF**. Elle reçoit la requête et la transmet à la couche métier.

Business Layer (couche métier) Implémentée avec des **EJB (Enterprise Java Beans)** ou **CDI (Contexts and Dependency Injection)**. Elle contient la logique métier et est transactionnelle et distribuée.

Data Layer (couche de persistance) Basée sur **JPA (Java Persistence API)** pour interagir avec la base de données. Elle permet de manipuler des objets Java reliés aux tables de la BD sans écrire de SQL natif (ORM).

Serveur d'application Le serveur d'application est le cœur de Java EE. Il gère :

- le cycle de vie des composants (Servlets, EJB, etc.),
- la sécurité (authentification, autorisation),
- les transactions,
- le pooling de connexions BD,
- la gestion des threads et la scalabilité.

1.2.2 Les principales spécifications de Java EE

- **Servlets / JSP / JSF** : pour la partie Web.
- **EJB (Enterprise Java Beans)** : logique métier et transactions.
- **JPA (Java Persistence API)** : gestion des données.
- **JAX-RS / JAX-WS** : services REST et SOAP.
- **JMS (Java Message Service)** : communication asynchrone via files de messages.
- **CDI (Contexts and Dependency Injection)** : injection de dépendances.
- **JAAS / Security API** : sécurité.

1.2.3 Exemple d'un flux typique

1. Le client fait une requête `HTTP GET /produits`.
2. Le serveur d'application (via un Servlet ou un endpoint REST JAX-RS) reçoit la requête.
3. La logique métier (EJB ou CDI) traite la demande.
4. La couche de persistance (JPA) récupère les données dans la base.
5. La réponse (JSON, HTML, XML) est renvoyée au client.

1.2.4 A Retenir

Java EE fournit une **infrastructure prête à l'emploi** qui gère automatiquement les parties complexes (transactions, sécurité, communication, persistance), permettant aux développeurs de se concentrer sur la logique métier.

1.3 Le Modèle MVC et son lien avec Java EE

1.3.1 Origine et principe

Le modèle MVC (Model–View–Controller) est un **patron d'architecture logicielle** apparu dans les années 1970 avec Smalltalk. Il permet de séparer les responsabilités d'une application en trois parties indépendantes :

- **Model (Modèle)** : la logique métier et les données,
- **View (Vue)** : la présentation et l'interface utilisateur,
- **Controller (Contrôleur)** : la gestion des interactions et du flux des requêtes.

Cette séparation rend le code plus modulaire, testable et maintenable.

1.3.2 Les composants du modèle MVC

Modèle (Model) Représente les **données** de l'application et leur traitement. Il contient la logique métier et accède aux ressources de persistance (base de données, fichiers, API). Exemple : une classe `Produit` avec ses attributs (`id`, `nom`, `prix`) et ses méthodes.

Vue (View) Gère la **représentation visuelle** des données (interface utilisateur). Elle affiche les informations reçues du modèle sans contenir de logique métier. Exemple : une page HTML/JSF affichant la liste des produits.

Contrôleur (Controller) Fait le lien entre la vue et le modèle. Il reçoit les actions de l'utilisateur (requêtes HTTP), appelle la logique métier du modèle et choisit la vue adaptée. Exemple : une Servlet recevant une requête, invoquant un service et redirigeant vers une JSP.

1.3.3 Cycle d'exécution MVC

1. L'utilisateur interagit avec la **Vue**.
2. Le **Contrôleur** intercepte l'action et appelle le **Modèle**.
3. Le **Modèle** exécute la logique métier et renvoie le résultat.
4. Le **Contrôleur** choisit la **Vue** appropriée.
5. La **Vue** affiche les résultats à l'utilisateur.

1.3.4 MVC dans Java EE

Dans une application Java EE, le modèle MVC est utilisé pour organiser le code de la manière suivante :

- **Model (Modèle)** : Entités JPA (`@Entity`), logique métier avec EJB ou CDI, persistance avec JPA/Hibernate.
- **View (Vue)** : JSP, JSF ou frameworks comme Thymeleaf.
- **Controller (Contrôleur)** : Servlets ou Beans managés JSF.

1.3.5 Résumé

Le modèle MVC sépare clairement la logique métier, la présentation et le contrôle du flux. Dans Java EE, cette architecture se traduit par :

- Modèle : entités JPA, EJB, CDI,
- Vue : JSP, JSF,
- Contrôleur : Servlets, Beans managés.

Cela rend les applications plus **claires, évolutives et maintenables**.

1.4 Outils de développement : Eclipse for Java Developer et Tomcat

1.4.1 Pré-requis communs

- Installer un JDK (Java Development Kit), version LTS recommandée (17 ou 21).
- Vérifier l'installation avec : `java -version` et `javac -version`.
- Télécharger Eclipse IDE for Java Developers depuis le site officiel.
- Télécharger Apache Tomcat (version 9 ou 10 selon le besoin).

1.5 Installation et configuration sous Windows

Installation du JDK

1. Télécharger l'installateur Temurin JDK (17 ou 21).
2. Lancer le fichier MSI et installer avec les options par défaut.
3. Vérifier l'installation avec : `java -version`.
4. (Optionnel) Configurer la variable `JAVA_HOME` :
   - `JAVA_HOME = C:\Program Files\Eclipse Adoptium\jdk-21`
   - Ajouter `%JAVA_HOME%\bin` au `PATH`.

Installation d'Eclipse

1. Télécharger l'installateur Eclipse.
2. Choisir *Eclipse IDE for Java Developers*.
3. Lancer Eclipse et choisir un workspace.

Installation de Tomcat Deux méthodes :

- **Avec installateur Windows** : exécuter `apache-tomcat-*.exe`, installer comme service Windows, pointer vers le JDK.
- **Avec archive ZIP** : décompresser dans `C:\Tools\tomcat`, définir `CATALINA_HOME`, démarrer avec `startup.bat`.

Configuration dans Eclipse

1. Installer les plugins **Web Tools Platform (WTP)** ou **JST Server Adapters**.

2. Ajouter Tomcat : *Window → Preferences → Server → Runtime Environments → Add.*
3. Créer un serveur dans la vue *Servers*, et déployer un projet *Dynamic Web Project.*

Réglages utiles

- Ports : par défaut 8080/8005/8009, modifiables dans Eclipse.
- Utilisateur *Manager* : ajouter un utilisateur dans `conf/tomcat-users.xml`.
- Mémoire JVM : créer `bin\setenv.bat` et ajouter :
  `set "JAVA_OPTS=-Xms512m -Xmx1024m -XX:+UseG1GC"`

1.6 Installation et configuration sous Linux

Installation du JDK

- Sous Ubuntu/Debian :
  `sudo apt install temurin-21-jdk`
  `java -version`
- Sous Fedora :
  `sudo dnf install temurin-21-jdk`

Installation d'Eclipse

1. Télécharger et exécuter l'installateur Eclipse (https://www.eclipse.org/downloads) ou zip (https://www.eclipse.org/downloads/packages/)
2. Ou installer via Snap/Flatpak si disponible.

Installation de Tomcat

1. Décompresser l'archive dans `/opt/tomcat`.
2. Créer un utilisateur *tomcat* et attribuer les droits.
3. Démarrer avec `bin/startup.sh`.

Configuration en service systemd (optionnel)

```
[Unit]
Description=Apache Tomcat
After=network.target

[Service]
Type=forking
Environment="JAVA_HOME=/usr/lib/jvm/temurin-21-jdk"
Environment="CATALINA_HOME=/opt/tomcat"
ExecStart=/opt/tomcat/bin/startup.sh
ExecStop=/opt/tomcat/bin/shutdown.sh
User=tomcat
```

```
Group=tomcat

[Install]
WantedBy=multi-user.target
```

Réglages utiles

- Définir dans `~/.bashrc` :
  ```
  export JAVA_HOME=/opt/jdk-21
  export PATH=$PATH:$JAVA_HOME/bin
  ```
- Créer `bin/setenv.sh` :
  ```
  export JAVA_OPTS="-Xms512m -Xmx1024m -XX:+UseG1GC"
  ```

1.7 Installation et configuration sous macOS

Installation du JDK

- Télécharger le fichier PKG Temurin et installer.
- Vérifier avec `java -version`.
- (Alternative Homebrew) :
  ```
  brew install --cask temurin
  ```

Installation d'Eclipse

1. Télécharger l'installateur Eclipse et choisir *Eclipse IDE for Java Developers*.

Installation de Tomcat

- Avec Homebrew :
  ```
  brew install tomcat
  brew services start tomcat
  ```
- Ou bien décompresser l'archive dans `~/Tools/tomcat` et lancer `bin/startup.sh`.

Réglages utiles

- Ajouter dans `~/.zshrc` :
  ```
  export JAVA_HOME="$(/usr/libexec/java_home -v 21)"
  export PATH="$PATH:$JAVA_HOME/bin"
  ```
- Créer `bin/setenv.sh` pour les paramètres mémoire.

1.8 Vérification du fonctionnement

1. Vérifier le JDK : `java -version`.
2. Vérifier Tomcat : accéder à `http://localhost:8080`.
3. Vérifier Eclipse : démarrer Tomcat depuis la vue *Servers* et déployer un projet.
4. Vérifier le débogage : lancer Tomcat en mode *Debug* dans Eclipse.

1.9 Réglages Eclipse recommandés

- Compiler compliance level : 17 ou 21.
- Encodage projet : UTF-8.
- Ajouter Tomcat comme runtime dans les préférences.
- Vérifier les ports et la configuration du serveur.

1.10 Problèmes fréquents

- Tomcat absent dans Eclipse : installer JST Server Adapters via Marketplace.
- Port 8080 occupé : changer le port dans la configuration serveur.
- Erreur 403 sur /manager : ajouter un utilisateur dans `tomcat-users.xml`.
- Timeout au démarrage : vérifier les ports et la configuration `server.xml`.

1.11 Créer un projet Java EE sous Eclipse et le lier avec Tomcat

Rappel :

1. Java JDK installé (au moins Java 8).
2. Eclipse IDE for Enterprise Java Developers.
3. Apache Tomcat installé sur votre ordinateur.

1.11.1 Étape 1 : Installer Tomcat dans Eclipse

1. Ouvrez Eclipse.
2. Allez dans **Window → Preferences → Server → Runtime Environments**.
3. Cliquez sur **Add...**.
4. Sélectionnez **Apache → Tomcat v9.0 Server ou 10.1 selon votre version d'apache → Next**.
5. Cliquez sur **Browse...** pour indiquer le chemin où Tomcat est installé.
6. Cliquez sur **Finish → OK**.

1.11.2 Étape 2 : Créer un projet Dynamic Web Project

1. Dans Eclipse : **File → New → Dynamic Web Project**.
2. Nom du projet : `CoursJavaee`.
3. Target runtime : choisissez **Apache Tomcat v9.0 ou 10.1 selon votre version d'apache**.
4. Configuration : **Dynamic Web Module Version 3.1 ou 6 selon la derniere version de la servlet**.
5. Cliquez sur **Next → Next → Finish**.

La structure simplifiée du projet :

```
CoursJavaee/
 Java Resources/src/
```

```
Webapp ou WebContent selon votre version d'éclipse/
    META-INF/
    WEB-INF/
         web.xml
```

1.11.3 Étape 3 : Créer le fichier HTML simple

1. Clique droit sur Webapp → New → HTML File.
2. Nom du fichier : index.html
3. Clique sur Finish.
4. Eclipse va créer le fichier index.html dans Webapp.
5. effacer le contenu de index.html et copier-coller ce code suivant :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Page d'accueil</title>
</head>
<body>
<h1>Bienvenue sur le cours de Java EE avec Mr Arrow Djiwa !</h1>
<p>Ceci est un exemple simple.</p>
</body>
</html>
```

1.11.4 Étape 4 : Déployer le projet sur Tomcat

1. Cliquez droit sur votre projet → **Run As → Run on Server**.
2. Choisissez **Tomcat v9.0 Server ou 10.1 selon votre version d'apache → Finish**.
3. Eclipse démarre Tomcat et déploie le projet.
4. Dans le navigateur, accédez à :
   `http://localhost:8080/CoursJavaee`
5. **Resultat attendu dans le navigateur :**

```
Bienvenue sur le cours de Java EE avec Mr Arrow Djiwa !
Ceci est un exemple simple.
```

1.11.5 Structure standard d'une application Java EE

- **Racine de l'application :** La racine correspond au dossier portant le nom du projet. Elle contient l'intégralité des dossiers et fichiers de l'application.
- **Dossier `WEB-INF` :** Ce dossier est spécial et obligatoire. Il doit être placé directement sous la racine de l'application. Il contient :
  - Le fichier de configuration de l'application `web.xml`.
  - Un dossier `classes`, destiné à contenir les classes compilées (`.class`).

  - Un dossier `lib`, contenant les bibliothèques nécessaires au projet (`.jar`).
- **Règles de visibilité :**
  - Les fichiers et dossiers placés directement sous la racine sont **publics** et accessibles via leur URL.
  - Les fichiers et dossiers placés sous `WEB-INF` sont **privés** et ne peuvent pas être accédés directement par le client.

NB : Il est essentiel de respecter cette organisation. Une application ne suivant pas cette structure risque de ne pas pouvoir être déployée ni exécutée correctement par le serveur.

1.11.6 Organisation d'un projet Java EE dans Eclipse

Racine du projet :

Dans Eclipse, chaque projet Java EE a un dossier racine portant le nom du projet.

Ce dossier contient tous les fichiers et dossiers nécessaires à l'application.

Dossier `src` :

Contient le code source Java (.java) de l'application.

Eclipse compile automatiquement ces fichiers et les place dans le dossier `build/classes` ou `WEB-INF/classes` selon le type de projet.

Dossier `Webapp` ou `WebContent` (ou encore `WebRoot`) :

Correspond à la racine web de l'application.

Contient les fichiers HTML, JSP, CSS, JavaScript, images, etc.

C'est dans ce dossier que se trouve le dossier `WEB-INF`.

Dossier `WEB-INF` :

Contient :

- Le fichier de configuration `web.xml`.
- Le dossier `classes` pour les fichiers compilés.
- Le dossier `lib` pour les bibliothèques (.jar).

Dossier `lib` :

Contient toutes les bibliothèques tierces nécessaires au projet (situé dans `WEB-INF/lib`).

Configuration du projet :

Eclipse crée automatiquement un fichier `.project` et un fichier `.classpath` pour gérer les paramètres du projet et les dépendances Java.

Pour un projet dynamique Web, Eclipse ajoute des paramètres spécifiques pour Tomcat ou tout autre serveur Java EE.

2 Partie 2 :Les notions essentiels de Java EE

2.1 Étude du contrôleur de MVC en Java EE : Servlet

2.1.1 Rappel sur le rincipe de fonctionnement de HTTP et ses méthodes (GET, POST, etc.)

1. Qu'est-ce que HTTP ? HTTP (HyperText Transfer Protocol) est le langage de communication entre un navigateur (client) et un serveur web. Il fonctionne sur le principe **requête (client) → réponse(serveur)** :

- Le client (navigateur, application mobile, Postman, etc.) envoie une **requête HTTP** au serveur.
- Le serveur analyse et renvoie une **réponse HTTP**.

C'est comparable à un client qui passe une commande (requête) dans un restaurant, et le serveur du restaurant qui apporte le plat (réponse).

2. Structure d'une requête HTTP Une requête HTTP contient :

1. Une **méthode** (GET, POST, PUT, DELETE, ...).
2. Une **URL** (exemple : `/produits/1`).
3. Des **en-têtes (headers)** : informations comme le type de contenu, cookies, etc.
4. Un **corps (body)** : utilisé uniquement pour certaines méthodes (POST, PUT).

3. Structure d'une réponse HTTP Une réponse HTTP contient :

1. Un **code de statut** (200 = OK, 404 = introuvable, 500 = erreur serveur).
2. Des **en-têtes (headers)** : type de la réponse (HTML, JSON, ...).
3. Un **corps (body)** : le contenu renvoyé (page web, données JSON, ...).

4. Les principales méthodes HTTP

GET

- Sert à **récupérer** des données depuis le serveur.
- Exemple : afficher une page web ou récupérer une liste d'articles.
- Les paramètres sont envoyés dans l'**URL**, par exemple : `http://monSite.com/produits?id=5`.
- Non sécurisé pour les mots de passe car visible dans l'URL.
- Idéal pour : lecture, recherche, navigation.

POST

- Sert à **envoyer des données** au serveur (formulaire, login, création de compte).
- Les données sont envoyées dans le **corps (body)** de la requête.
- Exemple (formulaire login) : `POST /login`
  `Body : { "username": "Zey", "password": "1234" }`
- Plus sécurisé que GET.
- Idéal pour : formulaires, enregistrement, authentification.

PUT

- Sert à **mettre à jour** une ressource existante.
- Exemple : modifier un profil utilisateur.
- Les informations mises à jour sont envoyées dans le **body**.

DELETE

- Sert à **supprimer** une ressource.
- Exemple : `DELETE /utilisateur/12`.
- Cela demande au serveur d'effacer l'utilisateur ayant l'ID 12.

5. Exemple concret (Restaurant)

- **GET** : lire le menu sans rien changer (juste récupérer des infos).
- **POST** : passer une commande (créer quelque chose de nouveau).
- **PUT** : modifier la commande (changer de plat).
- **DELETE** : annuler la commande.

6. Recap. rapide

- HTTP c̄ommunication client serveur.
- GET : récupérer des données.
- POST : envoyer ou créer des données.
- PUT : mettre à jour des données.
- DELETE : supprimer des données.

2.1.2 Comportement du serveur d'application Java EE lorsqu'une requête arrive

a-Le chemin d'une requête dans un serveur d'application Java EE

1. **Le client envoie une requête HTTP**
   Exemple : le navigateur tape `http://localhost:8080/monApp/hello`. La requête arrive au **serveur d'application** (Tomcat, WildFly, GlassFish. . .).
2. **Le serveur web intégré reçoit la requête**
   Le serveur d'application a un module qui comprend HTTP (comme Tomcat). Il analyse :

- le **port** (8080 par ex.),
- le **contexte de l'application** (`/monApp`),
- le **chemin exact** (`/hello`).

Cela permet de savoir quelle application et quelle ressource doit répondre.

3. **Le serveur trouve la ressource correspondante**
   - Si l'URL correspond à une **Servlet**, le serveur regarde dans la configuration (annotation `@WebServlet` ou `web.xml`) pour savoir quelle classe Java doit être exécutée.
   - Si c'est une **JSP**, il la traduit d'abord en Servlet puis l'exécute.
   - Si c'est un **fichier statique** (HTML, CSS, image...), il le renvoie directement sans passer par une Servlet.
4. **Cycle de vie de la Servlet** Une fois la Servlet identifiée :
   - Si ce n'est pas déjà fait, le serveur la **charge en mémoire** et appelle une seule fois `init()`.
   - Pour chaque requête :
     - Le serveur crée un objet **HttpServletRequest** (qui contient toutes les infos de la requête : paramètres, en-têtes, cookies, etc.).
     - Il crée aussi un objet **HttpServletResponse** (qui servira à construire la réponse).
     - Il appelle la méthode `service()`, qui redirige vers `doGet()`, `doPost()`, `doPut()` ou `doDelete()` selon la méthode HTTP.
5. **Génération de la réponse**
   La Servlet exécute son code : logique métier, accès à la base de données, etc. Elle écrit la réponse (HTML, JSON, XML, ...) dans l'objet `HttpServletResponse`. Le serveur envoie cette réponse HTTP au client.
6. **Envoi au navigateur**
   Le navigateur reçoit la réponse (par exemple une page HTML) et l'affiche à l'écran.

b- Où Placé ma Servlet sans IDE comme Eclipse? Dans un projet **Java EE** (ou Jakarta EE) sans IDE, il faut respecter la structure standard d'une application web Java. Une application web est généralement empaquetée sous forme de fichier **WAR** (Web ARchive), qui a une arborescence bien définie.

```
MonProjet/
 src/                    → Contient le code source Java (servlets, classes, etc.)
     com/monsite/servlets/
          MaServlet.java

 web/                    → Contient les fichiers accessibles par le web
     index.jsp
     autres.jsp

 WEB-INF/                → Spécial, non accessible directement par URL
 web.xml            → Fichier de configuration (déclaration servlets, mapping...)
```

```
classes/          → Contient les .class compilés des servlets et autres classes
com/monsite/servlets/
MaServlet.class
lib/              → Contient les fichiers JAR nécessaires (JDBC, JSTL, etc.)
```

Placement des servlets :

1. **Pendant le développement (avant compilation)** Les servlets doivent être placées dans `src/com/monsite/servlets/`. Exemple : `src/com/monsite/servlets/MaServlet.java`
2. **Après compilation** Les fichiers `.class` doivent être placés dans : `WEB-INF/classes/com/monsite/servl`
3. **Configuration (si les annotations @WebServlet ne sont pas utilisées)** Les servlets et leurs mappings doivent être déclarés dans le fichier `WEB-INF/web.xml`.

Exemple de configuration web.xml :

```
<servlet> <servlet-name>MaServlet</servlet-name> <servlet-class>com.monsite.servlets.MaServ
```

```
<servlet-mapping>
<servlet-name>MaServlet</servlet-name>
<url-pattern>/maServlet</url-pattern>
</servlet-mapping>
```

Résumé :

- Code source (`.java`) → `src/`
- Classes compilées (`.class`) → `WEB-INF/classes/`
- Déclaration → `WEB-INF/web.xml` (ou via `@WebServlet` à partir de Java EE 6+)

c-Recap. simplifié Quand une requête arrive sur un serveur Java EE :

1. Le serveur la reçoit et regarde **quelle application** est concernée.
2. Il identifie la ressource demandée (Servlet, JSP, fichier statique).
3. Il crée des objets `HttpServletRequest` et `HttpServletResponse`.
4. Il appelle la méthode correspondante de la Servlet (`doGet`, `doPost`, ...).
5. La Servlet génère la réponse.
6. Le serveur renvoie cette réponse au client.
7. Client → Serveur → Servlet → Réponse

2.1.3 Création manuelle d'une Servlet dans Eclipse : Servlet = classe classique en Java

1. Création d'une classe Java

1. Déplier le dossier `Java Ressources` puis
2. Cliquer droit sur `src/main/java` puis `New>Class` et remplissez :

3. `Package` : com.ifnti.servlets
4. `Name` : Home
5. Pour la `Superclasse` : laisser vide puis
6. cliquer sur `Finish`

2. Transformer une classe Java classique en Servlet

```
package com.coursjavaee.servlets;

public class Home {

}
```

3. Hériter de `HttpServlet` et l'importer Pour transformer cette classe en servlet, il faut qu'elle hérite de `HttpServlet` :

```
package com.coursjavaee.servlets;
import jakarta.servlet.http.HttpServlet;

public class Home extends HttpServlet {

}
```

Pourquoi hériter de `HttpServlet` ?

- `HttpServlet` est une classe abstraite qui gère la logique des requêtes HTTP.
- Elle fournit des méthodes comme `doGet`, `doPost`, `doPut`, `doDelete`.
- Il suffit de redéfinir celles que l'on souhaite gérer.

3. Redéfinir `doGet` et `doPost` avec `@Override`

```
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;



public class Home extends HttpServlet {

@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response) {

}
```

```
}
```

Explications :

- `@Override` : indique que la méthode remplace une méthode de la classe parente (`HttpServlet`).

L'objet `HttpServletRequest` représente la requête HTTP envoyée par le client (navigateur ou application) vers le serveur. Il contient toutes les informations liées à cette requête.

Rôles principaux de HttpServletRequest :

- Récupérer les données envoyées par le client (paramètres, formulaires, URL, cookies, etc.).
- Identifier le type de requête (GET, POST, etc.).
- Fournir des informations sur le client (adresse IP, en-têtes HTTP, etc.).

Méthodes utiles de HttpServletRequest :

- `getParameter(String name)` : permet de récupérer un paramètre envoyé par le client (par exemple un champ d'un formulaire HTML).
- `getMethod()` : retourne la méthode HTTP utilisée (GET, POST, etc.).
- `getRequestURI()` : donne l'URI demandée par le client.
- `getHeader(String name)` : permet d'accéder à un en-tête HTTP spécifique.
- `getCookies()` : retourne les cookies envoyés par le client.

Exemple d'utilisation :

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

L'objet `HttpServletResponse` représente la réponse HTTP que le serveur renvoie au client. Il permet de construire et d'envoyer le résultat au navigateur.

Rôles principaux de HttpServletResponse :

- Définir le type de contenu renvoyé (HTML, JSON, image, etc.).
- Écrire du contenu dans la réponse.
- Gérer les en-têtes et le code de statut HTTP (200, 404, 500, etc.).
- Effectuer une redirection ou envoyer des cookies.

Méthodes utiles de HttpServletResponse :

- `setContentType(String type)` : définit le type MIME du contenu (`"text/html"`, `"application/json"`, etc.).
- `getWriter()` : retourne un flux d'écriture pour envoyer du texte/HTML.
- `setStatus(int sc)` : définit un code de statut HTTP.
- `sendRedirect(String location)` : redirige le client vers une autre ressource.

- `addCookie(Cookie cookie)` : ajoute un cookie dans la réponse.

Exemple d'utilisation :

```
response.setContentType("text/html");
response.getWriter().println("<h2>Bonjour utilisateur !</h2>");
```

Interaction entre les deux L'objet `HttpServletRequest` est utilisé pour lire et analyser la requête envoyée par le client, tandis que l'objet `HttpServletResponse` permet de construire et d'envoyer la réponse.

Exemple complet :

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
response.setCharacterEncoding( "UTF-8" );
PrintWriter out = response.getWriter();
out.println("<!DOCTYPE html>");
out.println("<html>");
out.println("<head>");
out.println("<meta charset=\"utf-8\" />");
out.println("<title>Test</title>");
out.println("</head>");
out.println("<body>");
out.println("<p>Ceci est une page générée depuis une
servlet.</p>");
out.println("</body>");
out.println("</html>");
}
```

4. Gérer les erreurs et Importer les packages nécessaires

```
import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;



public class Home extends HttpServlet {

@Override
```

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
response.setContentType("text/html");
response.getWriter().println("<h2>Servlet transformée depuis MaClasse !</h2>");
}

}
```

Explications :

Lors de la création d'une servlet, certaines méthodes comme `doGet()` et `doPost()` peuvent lever des exceptions spécifiques. Les plus courantes sont `ServletException` et `IOException`.

1. **ServletException**
   - Cette exception appartient au package `jakarta.servlet`.
   - Elle est levée lorsqu'une erreur liée à l'exécution de la servlet se produit (problème de logique métier, problème d'initialisation, etc.).
   - Elle permet d'indiquer au conteneur de servlets (comme Tomcat) qu'une anomalie spécifique au traitement de la requête est survenue.
   - Import nécessaire :
     `import jakarta.servlet.ServletException;`
2. **IOException**
   - Cette exception appartient au package `java.io`.
   - Elle est levée lorsqu'une erreur d'entrée/sortie se produit, par exemple lors de la lecture de la requête ou de l'écriture de la réponse.
   - Dans une servlet, elle survient souvent quand on écrit dans le flux de sortie `response.getWriter()`.
   - Import nécessaire :
     `import java.io.IOException;`

Exemple de signature d'une méthode `doGet` utilisant ces exceptions :

```
@Override
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
// Code de traitement
}
```

5. Ajouter un mapping pour la servlet

- Avec annotation (**Servlet 3.0+**) :
  `@WebServlet("/home")`
- Ou dans `web.xml` :
  `<servlet>`
  `<servlet-name>Home</servlet-name>`

```
<servlet-class>com.ifnti.servlets.Home</servlet-class>
</servlet>

<servlet-mapping>
<servlet-name>Home</servlet-name>
<url-pattern>/home</url-pattern>
</servlet-mapping>
```

6. Déployer et tester

1. Déployer le projet sur Tomcat via Eclipse : `Run As > Run on Server`.
2. Accéder à la servlet via le navigateur :
   `http://localhost:8080/coursjavaee/home`
3. Vérifier que le contenu s'affiche correctement.

Recap. : créer une classe Java classique, hériter de `HttpServlet`, importer les classes nécessaires, redéfinir `doGet`/`doPost` avec `@Override` et gérer les exceptions, ajouter un mapping via annotation ou `web.xml`, puis déployer et tester.

2.1.4 Créer directement une Servlet dans Eclipse

1. Créer une Servlet

1. Clic droit sur le dossier `src` > `New > Servlet`.
2. Donner un nom à la servlet, par exemple `HelloServlet`.
3. Cliquer sur **Finish**.

Eclipse génère une classe Java semblable à :

```
package com.demo; //nom du package de ton choix

import java.io.IOException;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServlet;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;

public class HelloServlet extends HttpServlet {
private static final long serialVersionUID = 1L;

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

```
response.getWriter().println("<h1>Hello, this is my first Servlet!</h1>");
}
}
```

2. Déclarer la Servlet (si nécessaire)

- Avec les annotations (**Servlet 3.0+**) :
  ```
  @WebServlet("/hello")
  public class HelloServlet extends HttpServlet {
  ...
  }
  ```
- Sinon, la déclaration se fait dans `web.xml` :
  ```
  <servlet>
  <servlet-name>HelloServlet</servlet-name>
  <servlet-class>com.demo.HelloServlet</servlet-class>
  </servlet>

  <servlet-mapping>
  <servlet-name>HelloServlet</servlet-name>
  <url-pattern>/hello</url-pattern>
  </servlet-mapping>
  ```

NB: Ceci n'est qu'une étape simplifiée sinon vous pouvez faire **Next** , **Next** pour filtrer plus d'avantage!

3. Déployer et tester

1. Clic droit sur le projet > `Run As > Run on Server`.
2. Choisir **Tomcat** et lancer.
3. Dans un navigateur, entrer : `http://localhost:8080/ServletDemo/hello`.
4. Résultat attendu : `Hello, this is my first Servlet!`.

La documentation complète des apis se trouve ici : https://jakarta.ee/specifications/platform/11/apidocs/.

2.1.5 Le fichier `web.xml` et le mapping des Servlets

Le fichier `web.xml`, appelé **descripteur de déploiement**, est utilisé pour configurer une application web Java EE/Jakarta EE. Il se trouve généralement dans le dossier :

`WebContent/WEB-INF/web.xml`

Avant l'arrivée des annotations (`@WebServlet`), c'était le principal moyen de déclarer et de mapper les servlets, filtres, écouteurs, pages d'erreurs, etc.

Déclaration d'une Servlet On décrit la servlet avec un nom logique et sa classe Java :

```
<servlet>
<servlet-name>MaPremiereServlet</servlet-name>
<servlet-class>com.example.MaPremiereServlet</servlet-class>
</servlet>
```

Mapping d'une Servlet On associe l'alias défini ci-dessus à une ou plusieurs URL :

```
<servlet-mapping>
<servlet-name>MaPremiereServlet</servlet-name>
<url-pattern>/maServlet</url-pattern>
</servlet-mapping>
```

Exemple complet

```
<web-app xmlns="http://xmlns.jcp.org/xml/ns/javaee"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://xmlns.jcp.org/xml/ns/javaee
http://xmlns.jcp.org/xml/ns/javaee/web-app_3_1.xsd"
version="3.1">

<!-- Déclaration de la servlet -->
<servlet>
<servlet-name>MaPremiereServlet</servlet-name>
<servlet-class>com.example.MaPremiereServlet</servlet-class>
</servlet>

<!-- Mapping de la servlet -->
<servlet-mapping>
<servlet-name>MaPremiereServlet</servlet-name>
<url-pattern>/maServlet</url-pattern>
</servlet-mapping>

</web-app>
```

Plusieurs patterns possibles Une servlet peut être appelée par plusieurs URL :

```
<servlet-mapping>
<servlet-name>MaPremiereServlet</servlet-name>
<url-pattern>/servlet1</url-pattern>
<url-pattern>/servlet2</url-pattern>
</servlet-mapping>
```

Types de `url-pattern`

- **Chemin exact** :
  `<url-pattern>/login</url-pattern>`
- **Pattern avec joker** :
  `<url-pattern>/users/*</url-pattern>`
- **Extension** :
  `<url-pattern>*.do</url-pattern>`

Balises supplémentaires Il existe d'autres balises utiles dans `web.xml`, comme :

- `<load-on-startup>` : pour charger une servlet au démarrage du serveur.
- `<init-param>` : pour définir des paramètres d'initialisation.

```
<servlet>
<servlet-name>ConfigurableServlet</servlet-name>
<servlet-class>com.example.ConfigurableServlet</servlet-class>
<init-param>
<param-name>theme</param-name>
<param-value>dark</param-value>
</init-param>
<load-on-startup>1</load-on-startup>
</servlet>
```

En résumé, la balise `<servlet>` permet de déclarer une servlet (nom et classe), tandis que la balise `<servlet-mapping>` définit l'association entre le nom et l'URL correspondante.

2.2 Etude de la Vue du modèle MVC de Java EE : JSP (JavaServer Pages)

2.2.1 Qu'est-ce que JSP ?

JSP est une technologie Java pour créer des pages web dynamiques.

- **Dynamique** signifie que le contenu de la page peut changer selon l'utilisateur, la base de données, ou d'autres conditions.
- C'est comme HTML, mais avec la possibilité d'inclure du code Java directement dans la page.
- JSP est souvent utilisé avec **Servlets**, car une JSP est en réalité transformée en Servlet par le serveur avant d'être exécutée.

2.2.2 Pourquoi utiliser JSP ?

- Pour séparer **le contenu dynamique (Java)** et **le contenu statique (HTML)**.
- Pour rendre le développement web plus simple que de coder toute la logique dans des Servlets pures.
- Pour intégrer facilement des données de la base de données, des formulaires, ou des sessions utilisateur.

2.2.3 Comment fonctionne JSP ?

Le fonctionnement est le suivant :

1. L'utilisateur fait une requête HTTP pour accéder à une page `.jsp`.
2. Le serveur d'application (Tomcat, GlassFish, ...) reçoit la requête.
3. Si c'est la première fois que la JSP est demandée :
   - La JSP est transformée en **Servlet Java** par le serveur.
   - Le Servlet est compilé en **bytecode Java**.
4. Le Servlet généré est exécuté :
   - Il traite les données (ex: récupère des infos de la base, lit les paramètres du formulaire).
   - Il génère du HTML qui est renvoyé au navigateur de l'utilisateur.
5. Les requêtes suivantes utilisent directement le Servlet compilé (plus rapide).

Récap. simple : JSP = HTML + Java → Servlet → HTML envoyé au navigateur.

2.2.4 Cycle de vie d'une JSP, ses méthodes et exemples d'utilisation

Le cycle de vie d'une JSP est géré par le conteneur de servlets (exemple : Tomcat). Une JSP est traduite en servlet, compilée et exécutée. Les étapes sont les suivantes :

1. **Traduction en servlet** : la page `.jsp` est traduite en une classe servlet Java.
2. **Compilation** : la servlet générée est compilée en bytecode.
3. **Chargement et instanciation** : le conteneur charge la classe et crée une instance de la JSP.
4. **Initialisation (`jspInit()`)** : exécutée une seule fois lors du premier chargement, utilisée pour initialiser des ressources.
5. **Traitement des requêtes (`_jspService()`)** : appelée pour chaque requête, cette méthode est générée automatiquement et contient le code HTML, JSP et les expressions.
6. **Destruction (`jspDestroy()`)** : appelée une seule fois avant que la JSP soit retirée de la mémoire, utilisée pour libérer des ressources.

Méthodes principales

- **jspInit()** : appelée une seule fois lors de l'initialisation.

```
<%!
public void jspInit() {
System.out.println("JSP initialisée !");
// Exemple : ouverture de connexion
}
%>
```

- **_jspService(HttpServletRequest, HttpServletResponse)** : appelée pour chaque requête. Cette méthode est générée automatiquement par le conteneur et exécute le code HTML et JSP.

```
<html>
<body>
<h2>Bonjour, le temps actuel est : <%= new java.util.Date() %></h2>
</body>
</html>
```

- **jspDestroy()** : appelée une seule fois avant la destruction.

```
<%!
public void jspDestroy() {
System.out.println("JSP détruite !");
// Exemple : fermer une connexion BD
}
%>
```

Exemple concret : JSP avec connexion à une base de données

```
<%@ page import="java.sql.*" %>
<%!
Connection con;

public void jspInit() {
try {
Class.forName("org.postgresql.Driver");
con = DriverManager.getConnection(
"jdbc:postgresql://localhost:5432/maDB",
"user", "password");
System.out.println("Connexion BD établie !");
} catch (Exception e) {
e.printStackTrace();
}
}

public void jspDestroy() {
try {
if (con != null && !con.isClosed()) {
con.close();
System.out.println("Connexion BD fermée !");
}
} catch (Exception e) {
e.printStackTrace();
}
}
%>
```

```
<html>
<body>
<h2>Liste des utilisateurs</h2>
<ul>
<%
Statement stmt = con.createStatement();
ResultSet rs = stmt.executeQuery("SELECT nom FROM utilisateurs");
while(rs.next()) {
out.println("<li>" + rs.getString("nom") + "</li>");
}
rs.close();
stmt.close();
%>
</ul>
</body>
</html>
```

Quand utiliser les méthodes du cycle de vie d'une JSP ? Les différentes méthodes du cycle de vie d'une JSP n'ont pas le même rôle ni la même fréquence d'exécution. Voici leurs usages typiques :

- **jspInit()**
  Exécutée une seule fois au démarrage de la JSP. On l'utilise pour initialiser des ressources ou charger des données réutilisables.
  Exemples :
  - établir une connexion à une base de données,
  - lire des paramètres de configuration,
  - initialiser un compteur global (nombre de visiteurs).
- **_jspService(HttpServletRequest, HttpServletResponse)**
  Méthode générée automatiquement par le conteneur et exécutée à chaque requête HTTP (GET ou POST). C'est ici que s'exécutent le code HTML, JSP, JSTL et EL.
  Exemples :
  - afficher une page web dynamique (liste de produits, résultats de recherche),
  - traiter les paramètres envoyés par un formulaire,
  - générer une réponse en fonction des données de l'utilisateur.
- **jspDestroy()**
  Exécutée une seule fois avant que la JSP soit détruite par le conteneur. On l'utilise pour libérer les ressources utilisées.
  Exemples :
  - fermer une connexion à la base de données,
  - sauvegarder des données temporaires,
  - libérer des objets lourds (sessions, flux, sockets).

Résumé sous forme de tableau

Méthode	Moment d'exécution	Utilisation typique
jspInit()	Une seule fois au début	Initialisation (connexion BD, configuration)
_jspService()	À chaque requête	Génération de la réponse dynamique (HTML, JSP, JSTL)
jspDestroy()	Une seule fois à la fin	Libération des ressources (fermeture BD, nettoyage)

En résumé : **jspInit()** prépare le terrain, **_jspService()** traite chaque requête, et **jspDestroy()** nettoie avant la fin.

2.2.5 Syntaxe JSP

- **Scriptlets** `<% ... %>` : Code Java directement dans la page.
  ```
  <%
  String nom = "Zey";
  out.println("Bonjour " + nom);
  %>
  ```
- **Expressions** `<%= ... %>` : Pour afficher une valeur directement.
  ```
  <p>La date actuelle est : <%= new java.util.Date() %></p>
  ```
- **Déclarations** `<%! ... %>` : Pour déclarer des variables ou méthodes.
  ```
  <%! int compteur = 0; %>
  ```

2.2.6 Avantages et limites de JSP

Avantages :

- Intégration facile avec Java EE et bases de données.
- Séparation partielle du contenu statique et dynamique.
- Supporte les tags et JSTL pour réduire le code Java dans la JSP.

Limites :

- Mélanger trop de Java dans le HTML peut rendre la page difficile à maintenir.
- Moins utilisé dans les projets modernes (frameworks comme JSF, Spring MVC, ou Thymeleaf sont plus populaires).

2.2.7 Exemple simple complet

```
<%@ page language="java" contentType="text/html; charset=UTF-8" %>
<html>
<head><title>Page JSP Exemple</title></head>
<body>
<h1>Bonjour depuis JSP !</h1>
<%
String utilisateur = request.getParameter("user");
if(utilisateur == null) utilisateur = "Invité";
```

```
%>
<p>Bienvenue, <%= utilisateur %> !</p>
</body>
</html>
```

2.2.8 Créer une JSP dans Eclipse

1.Créer une JSP

1. Faire un clic droit sur le dossier `Webqpp` → **New** → **JSP File**.
2. Donner un nom au fichier, par exemple `index.jsp`.
3. Cliquer sur **Finish**.

Remarque : Eclipse crée un squelette HTML pour la JSP automatiquement.

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Insert title here</title>
</head>
<body>


</body>
</html>
```

2.Lancer la JSP

1. Clic droit sur le projet → **Run As** → **Run on Server**.
2. Sélectionner le serveur (ex : Tomcat).
3. Eclipse déploie le projet et ouvre le navigateur.

Exemple de test : `http://localhost:8080/coursjavaee/index.jsp` Affiche : **Rien !** car rien n'est écrit entre les balises <body>

3.Notes pratiques

- Tout fichier JSP doit être dans `Webapp` ou ses sous-dossiers.
- Les fichiers `.jsp` sont automatiquement compilés en Servlet par Tomcat.
- Pour inclure du code Java, utiliser les **scriptlets** <% ... %> ou mieux les **JSTL** pour séparer Java et HTML.(à étudier plus tard!)

2.2.9 Placement des JSP et bonnes pratiques

Pourquoi ne pas placer les JSP directement dans WebContent ou webapp ?

- Les fichiers JSP placés directement dans `WebContent` (ou `webapp`) sont **accessibles directement par l'utilisateur via l'URL**.
- Exemple : si vous avez `WebContent/index.jsp`, quelqu'un peut taper `http://localhost:8080/MonProje` et accéder à cette page.
- Cela peut poser **des problèmes de sécurité**, surtout si la JSP contient du code sensible ou sert uniquement à générer du contenu via des Servlets.
- Cela mélange la **présentation (JSP)** et l'accès direct à l'utilisateur, ce qui **brise la logique MVC (Model-View-Controller)**.

La place des JSP dans WEB-INF

- Les fichiers JSP peuvent être déplacés dans `WEB-INF` (ex : `WEB-INF/views/`) pour **les protéger contre un accès direct via l'URL**.
- Les pages JSP dans `WEB-INF` **ne peuvent pas être atteintes directement par le navigateur**, elles ne sont accessibles que **via un Servlet**.
- Cela force l'utilisation de la **logique MVC** :
  1. L'utilisateur envoie une requête → Servlet.
  2. Le Servlet traite les données → Forward vers la JSP dans `WEB-INF`.
  3. La JSP génère le HTML → renvoie au navigateur.

2.2.10 Forwarding d'une JSP depuis un Servlet et utilisation de RequestDispatcher

Étapes pour faire un forwarding d'une JSP depuis un Servlet :

1. **Créer une JSP** (ex : `WEB-INF/views/index.jsp`) - Mettre la JSP dans `WEB-INF` pour qu'elle ne soit pas accessible directement par l'URL.
2. **Créer un Servlet** (ex : `ExempleServlet.java`) - Ce Servlet va recevoir la requête HTTP de l'utilisateur.
3. **Traiter les données dans le Servlet** - Par exemple, récupérer des paramètres ou interroger une base de données. - Stocker des données à transmettre à la JSP dans l'objet `request` :
   `request.setAttribute("user", "Zey");`
4. **Obtenir un RequestDispatcher** - Le `RequestDispatcher` permet de dispatcher (envoyer) la requête vers une autre ressource (JSP ou Servlet) :
   `RequestDispatcher rd = request.getRequestDispatcher("/WEB-INF/views/index.jsp");`
5. **Faire le forward vers la JSP** - La méthode `forward()` transfère la requête et la réponse vers la JSP :
   `rd.forward(request, response);`
6. **La JSP génère le HTML** - La JSP peut récupérer les attributs stockés dans `request` :
   `<p>Bienvenue, <%= request.getAttribute("user") %> !</p>`

RequestDispatcher et ses méthodes :

- **forward(ServletRequest request, ServletResponse response)** - Transfère la requête et la réponse vers une autre ressource. - L'URL du navigateur reste inchangée. - Après le `forward()`, le contrôle ne revient jamais au Servlet initial.
- **include(ServletRequest request, ServletResponse response)** - Inclut le contenu d'une autre ressource dans la réponse en cours. - Utile pour inclure un header, footer ou un menu dans plusieurs pages. - Exemple :
  `RequestDispatcher rd = request.getRequestDispatcher("/header.jsp");`
  `rd.include(request, response);`

Différence Forward / Include :

- **forward** :
  - Effet sur l'URL : URL inchangée
  - Contrôle après exécution : ne revient jamais au Servlet initial
- **include** :
  - Effet sur l'URL : URL inchangée
  - Contrôle après exécution : revient au Servlet après inclusion

Exemple complet :

```
// ExempleServlet.java
protected void doGet(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {
request.setAttribute("user", "Zey");
RequestDispatcher rd = request.getRequestDispatcher("/WEB-INF/views/index.jsp");
rd.forward(request, response);
}
```

```
<!-- index.jsp -->
<p>Bienvenue, <%= request.getAttribute("user") %> !</p>
```

2.2.11 Transmission et récupération des données entre client, serveur et JSP

1. Données issues du client : les paramètres

- Lorsqu'un utilisateur remplit un formulaire ou passe des informations dans l'URL, le client envoie des **paramètres** au serveur.
- Ces paramètres sont transmis via **GET** ou **POST** :
  - **GET** : paramètres dans l'URL (`?user=Zey&age=25`)
  - **POST** : paramètres envoyés dans le corps de la requête

Exemple HTML (formulaire) :

```
<form action="<%= request.getContextPath() %>/" method="post">
```

```
Nom : <input type="text" name="nom">
<input type="submit" value="Envoyer">
</form>
```

Ici, `nom` est un paramètre que le serveur recevra.

Explication de `request.getContextPath()`

Quand on écrit dans un formulaire :

```
<form action="<%= request.getContextPath() %>/" method="post">
```

il y a deux choses importantes à comprendre :

`request.getContextPath()`

- C'est une méthode de l'objet implicite `request` (de type `HttpServletRequest`).
- Elle retourne le **contexte de l'application web**, c'est-à-dire le chemin racine où l'application est déployée.

Exemple :

- Si l'application est déployée sous `http://localhost:8080/MonApp`, alors `request.getContextPath()` = `/MonApp`.
- Si elle est à la racine (`/`), alors `request.getContextPath()` = `""` (chaîne vide).

Le caractère `/` après Lorsque l'on écrit :

```
<%= request.getContextPath() %>/
```

- On concatène le chemin de contexte avec `/`.
- Résultat : cela donne l'URL vers la **racine de l'application**, donc la servlet mappée sur `"/"`.

Exemple complet Supposons :

- Application déployée sous `http://localhost:8080/MonApp`
- Servlet mappée avec `@WebServlet("/")`

Alors, dans un JSP :

```
<form action="<%= request.getContextPath() %>/" method="post">
```

le code généré côté navigateur est :

```
<form action="/MonApp/" method="post">
```

Ce qui enverra bien la requête POST vers la **servlet `"/"`**, et non directement au JSP.

Alternative moderne Au lieu d'utiliser le vieux scriptlet `<%= ... %>`, on préfère utiliser l'EL (Expression Language) :

```
<form action="${pageContext.request.contextPath}/" method="post">
```

Cette approche est plus lisible, plus propre, et évite le mélange de Java et de HTML.

Comparaison entre `action="/"` et `action="<%= request.getContextPath() %>/"`

Cas 1 : `action="/"`

- Ici, le `/` est interprété par le navigateur comme la racine du serveur.
- Exemple : si l'application est déployée sous `http://localhost:8080/MonApp`, alors
  `<form action="/" method="post">`
  pointera vers `http://localhost:8080/`, en ignorant complètement le contexte `/MonApp`.
- Résultat : la servlet n'est pas appelée, sauf si l'application est déployée à la racine.

Cas 2 : `action="<%= request.getContextPath() %>/"`

- Ici, on utilise dynamiquement le contexte de l'application.
- Exemple : si l'application est déployée sous `http://localhost:8080/MonApp`, alors
  `<form action="<%= request.getContextPath() %>/" method="post">`
  est rendu côté navigateur comme :
  `<form action="/MonApp/" method="post">`
- Résultat : la requête POST est envoyée correctement vers la servlet mappée sur `"/"` à l'intérieur de l'application.

Conclusion

- `action="/"` fonctionne uniquement si l'application est déployée à la racine.
- `action="<%= request.getContextPath() %>/"` (ou en EL `${pageContext.request.contextPath}/`) est la solution portable et robuste, car elle s'adapte automatiquement à l'emplacement de déploiement de l'application.

2. Récupération des paramètres par le serveur

- Le Servlet récupère les paramètres envoyés par le client via la requête HTTP.
- Méthodes principales :

```
String nom = request.getParameter("nom"); // un seul paramètre
String[] valeurs = request.getParameterValues("choix"); // plusieurs valeurs
Enumeration<String> nomsParams = request.getParameterNames(); // tous les noms
```

- Les paramètres sont toujours de type `String`, même pour des nombres ou des dates.

3. Transmission des données du serveur à la JSP : les attributs

- Une fois les données traitées, le Servlet peut les transmettre à la JSP via des **attributs**.
- Ces attributs sont stockés dans l'objet `request`.

Exemple côté Servlet :

```
// Dans le Servlet
String nom = request.getParameter("nom");
request.setAttribute("nomUtilisateur", nom); // stocke l'attribut
RequestDispatcher rd = request.getRequestDispatcher("/WEB-INF/views/index.jsp");
rd.forward(request, response);
```

Exemple côté JSP :

```
<p>Bonjour, <%= request.getAttribute("nomUtilisateur") %> !</p>
```

Les attributs permettent donc de transmettre des variables du serveur (Servlet) à la JSP de manière sécurisée, sans exposer directement l'URL.

4. Résumé du flux

1. **Client → Serveur** : envoi de paramètres via GET ou POST.
2. **Serveur (Servlet)** : récupération des paramètres avec `request.getParameter()`.
3. **Serveur → JSP** : transmission des données traitées via `request.setAttribute()`.
4. **JSP** : récupération des attributs avec `request.getAttribute()` et affichage.

Ce mécanisme permet de séparer la logique (Servlet) de la présentation (JSP), en respectant le modèle MVC.

2.2.12 Étude en détail des Balises JSP

En JSP (Java Server Pages), il existe plusieurs types de balises spéciales qui permettent d'insérer du code Java dans une page HTML.

Syntaxe en JSP Dans ce chapitre, nous allons étudier la **syntaxe en JSP**. Nous allons découvrir l'utilisation des éléments de base qui constituent la structure d'une page JSP.

Les éléments de JSP Les éléments principaux de JSP permettent d'intégrer du code Java et de combiner celui-ci avec du HTML. On distingue notamment :

- le **scriptlet**,
- la **déclaration**,
- et l'**expression**.

Le Scriptlet Un **scriptlet** est une section de code Java insérée directement dans une page JSP. Il peut contenir :

- des instructions Java,
- des déclarations de variables ou de méthodes,
- ou encore des expressions valides dans le langage de script utilisé par la page.

La syntaxe d'un scriptlet est la suivante :

```
<%
// code Java
%>
```

Son équivalent en syntaxe XML est :

```
<jsp:scriptlet>
// Code Java
</jsp:scriptlet>
```

Remarque : tout texte, balises HTML ou autres éléments JSP doivent être placés **en dehors** du scriptlet.

Exemple de Scriptlet

```
<html>
<head>
<title>Exemple Scriptlet JSP</title>
</head>
<body>
<h2>Exemple de Scriptlet</h2>

<%
// Code Java dans un scriptlet
java.util.Date date = new java.util.Date();
%>

<p>La date et l'heure actuelles sont : <%= date %></p>
</body>
</html>
```

Dans cet exemple :

- Le code Java `java.util.Date date = new java.util.Date();` est écrit à l'intérieur du **scriptlet**.
- L'expression `<%= date %>` est utilisée pour afficher directement la valeur de l'objet `date` dans la page HTML générée.

La Déclaration Une **déclaration** permet de définir des variables ou des méthodes accessibles dans toute la page JSP. La syntaxe est la suivante :

```
<%!
type variableName;
type methodName() { ... }
%>
```

Exemple de Déclaration

```
<html>
<head>
<title>Exemple Déclaration JSP</title>
</head>
<body>
<h2>Exemple de Déclaration</h2>

<%!
int compteur = 0;
String message() {
return "Bienvenue sur JSP !";
}
%>

<p>Compteur : <%= ++compteur %></p>
<p>Message : <%= message() %></p>
</body>
</html>
```

Ici :

- La variable `compteur` est déclarée une seule fois et sa valeur est conservée entre les requêtes.
- La méthode `message()` peut être appelée dans la page JSP.

L'Expression JSP Une **expression JSP** sert à évaluer et afficher directement une valeur. Elle est utile pour intégrer le résultat d'une variable ou d'une méthode au sein du HTML.

La syntaxe est la suivante :

```
<%= expression %>
```

Exemple d'Expression

```
<html>
<head>
<title>Exemple Expression JSP</title>
</head>
<body>
<h2>Exemple d'Expression</h2>

<p>La somme de 5 et 3 est : <%= 5 + 3 %></p>
<p>L'heure actuelle est : <%= new java.util.Date() %></p>
</body>
</html>
```

Ici :

- L'expression `<%= 5 + 3 %>` affiche directement le résultat `8`.
- L'expression `<%= new java.util.Date() %>` affiche la date et l'heure actuelles.

Les Commentaires JSP Les **commentaires** en JSP permettent d'insérer des annotations ou des explications dans le code source, qui ne seront pas envoyées au client et donc ne seront pas visibles dans le navigateur. Ils sont utiles pour documenter le code ou masquer temporairement une portion.

Il existe deux types de commentaires :

- Les **commentaires JSP** : ils sont spécifiques à JSP et sont complètement ignorés par le serveur.
- Les **commentaires HTML** : ils sont envoyés au navigateur et peuvent être visibles via le code source de la page.

Syntaxe des Commentaires JSP

```
<%-- Ceci est un commentaire JSP --%>
```

Syntaxe des Commentaires HTML

```
<!-- Ceci est un commentaire HTML -->
```

Exemple de Commentaires

```
<html>
<head>
<title>Exemple Commentaires JSP</title>
</head>
<body>
<h2>Exemple de Commentaires</h2>

<%-- Ceci est un commentaire JSP : il ne sera pas visible
dans le code source envoyé au navigateur --%>

<!-- Ceci est un commentaire HTML :
il sera visible dans le code source de la page -->

<p>Bonjour, ceci est une page JSP avec commentaires.</p>
</body>
</html>
```

Dans cet exemple :

- Le commentaire JSP (`<%-- ... --%>`) est totalement supprimé lors de la compilation de la page.
- Le commentaire HTML (`<!-- ... -->`) est envoyé au navigateur et reste visible dans le code source.

Structures de contrôle en JSP Les **structures de contrôle** permettent de gérer le flux d'exécution du code Java dans une page JSP. Comme JSP permet d'insérer du code Java via les scriptlets (`<% %>`), on peut utiliser toutes les structures classiques de Java :

- Conditionnelles : `if`, `if...else`, `switch`
- Boucles : `for`, `while`, `do...while`
- Contrôle de flux : `break`, `continue`, `return`

Exemple : Conditionnelle

```
<%
int heure = 10;
if (heure < 12) {
%>
<p>Bonjour !</p>
<%
} else {
%>
<p>Bonsoir !</p>
<%
}
%>
```

Boucles en JSP Les boucles permettent de répéter un bloc de code plusieurs fois. On utilise les boucles Java classiques dans les scriptlets.

Boucle `for`

```
<%
for (int i = 1; i <= 5; i++) {
%>
<p>Compteur : <%= i %></p>
<%
}
%>
```

Boucle `while`

```
<%
int i = 1;
```

```
while (i <= 5) {
%>
<p>Valeur i : <%= i %></p>
<%
i++;
}
%>
```

Boucle `do...while`

```
<%
int i = 1;
do {
%>
<p>i vaut : <%= i %></p>
<%
i++;
} while (i <= 5);
%>
```

Les balises Directives

Directive `page` La directive **page** configure la page JSP et permet de définir des propriétés globales pour la page entière. Elle est placée au début de la page JSP et ne génère pas de sortie HTML.

Syntaxe générale

```
<%@ page attribut1="valeur1" attribut2="valeur2" ... %>
```

Attributs principaux de `page` Voici les principaux attributs de la directive `page` avec leur description et un exemple :

- **language** : Langage de script utilisé (toujours `java`). *Exemple* : %@ page language="java"
- **contentType** : Type MIME et encodage de la page. *Exemple* : %@ page contentType="text/html; charset=UTF-8"
- **import** : Classes Java à importer. *Exemple* : %@ page import="java.util.Date, java.text.SimpleDateFormat
- **session** : Indique si la page utilise une session (`true` ou `false`). *Exemple* : %@ page session="true"
- **isThreadSafe** : Indique si la page est thread-safe (`true` ou `false`). *Exemple* : %@ page isThreadSafe="false"
- **errorPage** : Page JSP à afficher en cas d'erreur. *Exemple* : %@ page errorPage="erreur.jsp"
- **buffer** : Taille du tampon de sortie. *Exemple* : %@ page buffer="16kb"

- **autoFlush** : Indique si le tampon est automatiquement vidé (`true` ou `false`). *Exemple* : %@ page autoFlush="true"
- **isErrorPage** : Indique si la page peut traiter les exceptions (`true` ou `false`). *Exemple* : %@ page isErrorPage="true"

Exemple concret de `page`

```
<%@ page language="java" contentType="text/html; charset=UTF-8"
import="java.util.Date" session="true" buffer="8kb" %>
<html>
<head>
<title>Exemple Directive Page</title>
</head>
<body>
<h2>Date actuelle : <%= new Date() %></h2>
</body>
</html>
```

Directive `include` La directive **include** permet d'inclure le contenu d'un autre fichier JSP ou HTML au moment de la compilation, utile pour réutiliser des parties communes (ex : header, footer).

Syntaxe

```
<%@ include file="nom_fichier.jsp" %>
```

Exemple de `include`

```
<!-- header.jsp -->
<h1>Bienvenue sur mon site</h1>
<hr>

<!-- page principale -->
<%@ include file="header.jsp" %>
<p>Contenu principal de la page...</p>
```

Directive `taglib` La directive **taglib** permet d'utiliser des bibliothèques de tags JSP (custom tags ou JSTL). Elle définit un préfixe pour les balises et l'URI de la bibliothèque.

Syntaxe

```
<%@ taglib prefix="prefix" uri="URI_de_la_bibliothèque" %>
```

Exemple avec JSTL Core

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

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

Remarques :

- 'prefix="c"' définit le préfixe à utiliser pour les balises JSTL Core.
- 'uri' identifie la bibliothèque de tags.
- La directive `taglib` ne produit pas de sortie HTML directement, elle permet seulement d'utiliser des balises personnalisées.

Les balises d'action JSP et leurs attributs

1. **`<jsp:forward>`** Redirige la requête vers une autre ressource côté serveur.
   - **Attributs** :
     - `page` : chemin de la ressource cible
   - **Exemple** :
     ```
     <jsp:forward page="autrePage.jsp" />
     ```

2. **`<jsp:include>`** Inclut le contenu d'une autre ressource à l'exécution.
   - **Attributs** :
     - `page` : chemin de la ressource à inclure
     - `flush` : vide le buffer avant inclusion (true/false)
   - **Exemple** :
     ```
     <jsp:include page="footer.jsp" flush="true" />
     ```

3. **`<jsp:param>`** Passe un paramètre à une ressource incluse ou redirigée.
   - **Attributs** :
     - `name` : nom du paramètre
     - `value` : valeur du paramètre
   - **Exemple** :
     ```
     <jsp:include page="produit.jsp">
     <jsp:param name="id" value="123" />
     </jsp:include>
     ```

4. **`<jsp:useBean>`** Crée ou récupère un bean Java pour la page JSP.

   - **Attributs** :
     - `id` : nom de la variable JSP
     - `class` : nom complet de la classe JavaBean
     - `scope` : portée du bean (`page`, `request`, `session`, `application`)
   - **Exemple** :

     ```
     <jsp:useBean id="utilisateur" class="com.example.Utilisateur" scope="session" />
     ```

5. **`<jsp:setProperty>`** Assigne une valeur à une propriété d'un bean.

   - **Attributs** :
     - `name` : identifiant du bean
     - `property` : nom de la propriété (‘*‘ pour toutes les propriétés via les paramètres de requête)
     - `value` : valeur à assigner
   - **Exemple** :

     ```
     <jsp:setProperty name="utilisateur" property="nom" value="Alice" />
     <jsp:setProperty name="utilisateur" property="*" />
     ```

6. **`<jsp:getProperty>`** Lit la valeur d'une propriété d'un bean et l'insère dans la page.

   - **Attributs** :
     - `name` : identifiant du bean
     - `property` : nom de la propriété à récupérer
   - **Exemple** :

     ```
     <p>Nom de l'utilisateur : <jsp:getProperty name="utilisateur" property="nom" /></p>
     ```

7. **`<jsp:plugin>`** Intègre des applets Java dans la page JSP.

   - **Attributs** :
     - `type` : type de plugin (‘applet‘ ou ‘bean‘)
     - `code` : nom de la classe de l'applet
     - `width` / `height` : dimensions
   - **Exemple** :

     ```
     <jsp:plugin type="applet" code="MonApplet.class" width="300" height="200" />
     ```

8. **`<jsp:element>`** Génère dynamiquement une balise XML ou HTML.

   - **Attributs** :
     - `name` : nom de la balise à générer
   - **Exemple** :

```
<jsp:element name="p">
<jsp:body>Texte dynamique dans un paragraphe</jsp:body>
</jsp:element>
```

9. `<jsp:attribute>` Définit un attribut pour une balise créée avec `<jsp:element>` ou un tag personnalisé.

- **Attributs** :
  - `name` : nom de l'attribut
- **Exemple** :

```
<jsp:element name="div">
<jsp:attribute name="class">important</jsp:attribute>
Contenu du div
</jsp:element>
```

10. `<jsp:body>` Contenu du corps d'un tag ou d'un élément généré dynamiquement.

- **Exemple** :

```
<jsp:element name="section">
<jsp:body>
<p>Contenu complexe dans la section</p>
</jsp:body>
</jsp:element>
```

Attributs communs à plusieurs balises d'action

- `page` : chemin de la ressource cible (‘<jsp:include>‘, ‘<jsp:forward>‘)
- `flush` : vide le buffer avant l'action (‘<jsp:include>‘)
- `name` : nom de la variable ou du paramètre (‘<jsp:useBean>‘, ‘<jsp:setProperty>‘, ‘<jsp:getProperty>‘, ‘<jsp:param>‘)
- `scope` : portée du bean (‘page‘, ‘request‘, ‘session‘, ‘application‘) (‘<jsp:useBean>‘)
- `property` : nom de la propriété du bean (‘<jsp:setProperty>‘, ‘<jsp:getProperty>‘)
- `value` : valeur assignée à une propriété ou un paramètre (‘<jsp:setProperty>‘, ‘<jsp:param>‘)

Objets implicites en JSP En JSP, les objets implicites sont fournis automatiquement par le conteneur pour chaque page. Ils permettent d'accéder facilement à des fonctionnalités clés sans les créer explicitement. Il existe 9 objets implicites principaux.

1. `request` Représente la requête HTTP envoyée par le client.

- **Classe** : `jakarta.servlet.http.HttpServletRequest`
- **Exemple** :

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

```
%>
<p>Bonjour, <%= nom %> !</p>
```

2. `response` Représente la réponse HTTP envoyée au client.

- **Classe** : `jakarta.servlet.http.HttpServletResponse`
- **Exemple** :

```
<%
response.setContentType("text/html;charset=UTF-8");
response.addHeader("Cache-Control", "no-cache");
%>
```

3. `out` Flux de sortie utilisé pour envoyer du contenu au client.

- **Classe** : `jakarta.servlet.jsp.JspWriter`
- **Exemple** :

```
<%
out.println("<p>Bonjour depuis l'objet out !</p>");
%>
```

4. `session` Représente la session utilisateur. Permet de stocker des informations persistantes pour un utilisateur spécifique.

- **Classe** : `jakarta.servlet.http.HttpSession`
- **Exemple** :

```
<%
session.setAttribute("utilisateur", "Alice");
String utilisateur = (String) session.getAttribute("utilisateur");
%>
<p>Utilisateur connecté : <%= utilisateur %></p>
```

5. `application` Représente le contexte de l'application web. Partagé entre toutes les sessions.

- **Classe** : `jakarta.servlet.ServletContext`
- **Exemple** :

```
<%
application.setAttribute("versionApp", "1.0");
String version = (String) application.getAttribute("versionApp");
%>
<p>Version de l'application : <%= version %></p>
```

6. `config` Fournit des informations sur la configuration de la servlet JSP.

- **Classe** : `jakarta.servlet.ServletConfig`
- **Exemple** :

```
<%
String nomServlet = config.getServletName();
%>
<p>Nom de la servlet : <%= nomServlet %></p>
```

7. `pageContext` Accès centralisé à tous les autres objets implicites et méthodes utilitaires (forward, include...).

- **Classe** : `jakarta.servlet.jsp.PageContext`
- **Exemple** :
  ```
  <%
  pageContext.setAttribute("message", "Bienvenue !");
  %>
  <p><%= pageContext.getAttribute("message") %></p>
  ```

8. `page` Représente l'instance actuelle de la page JSP (équivalent à `this` en Java).

- **Classe** : classe générée automatiquement pour la JSP
- **Exemple** :
  ```
  <%
  out.println("Référence de la page : " + page.toString());
  %>
  ```

9. `exception` Représente l'exception survenue sur la page, uniquement dans les pages d'erreur.

- **Classe** : `java.lang.Throwable`
- **Exemple** :
  ```
  <%@ page isErrorPage="true" %>
  <p>Une erreur est survenue : <%= exception.getMessage() %></p>
  ```

Résumé des objets implicites

- `request` : informations sur la requête client
- `response` : construction de la réponse HTTP
- `out` : flux de sortie vers le client
- `session` : stockage de données par utilisateur
- `application` : stockage de données global à l'application
- `config` : informations sur la configuration de la servlet
- `pageContext` : accès centralisé à tous les objets et méthodes utilitaires
- `page` : référence de l'instance de la page JSP
- `exception` : exception survenue sur la page (page d'erreur uniquement)

L'objet implicite `request` en détail L'objet `request` est un objet implicite fourni par JSP, instance de la classe `HttpServletRequest`. Il représente la **requête HTTP** envoyée par le client

(navigateur) vers le serveur. Cet objet permet d'accéder à toutes les informations transmises par l'utilisateur ou le navigateur : paramètres de formulaire, en-têtes HTTP, cookies, etc.

Utilisation courante Les principales fonctionnalités de l'objet `request` sont les suivantes :

- **Récupérer les paramètres d'un formulaire** :
  ```
  <!-- Exemple HTML -->
  <form action="traitement.jsp" method="post">
  Nom : <input type="text" name="nom"><br>
  <input type="submit" value="Envoyer">
  </form>
  ```
  Dans `traitement.jsp` :
  ```
  <%
  String nom = request.getParameter("nom");
  out.println("Bonjour " + nom);
  %>
  ```
- **Récupérer tous les paramètres** :
  ```
  <%
  java.util.Enumeration<String> nomsParams = request.getParameterNames();
  while(nomsParams.hasMoreElements()){
  String param = nomsParams.nextElement();
  out.println(param + " = " + request.getParameter(param) + "<br>");
  }
  %>
  ```
- **Accéder aux en-têtes HTTP** (exemple : navigateur utilisé) :
  ```
  <%
  String navigateur = request.getHeader("User-Agent");
  out.println("Votre navigateur est : " + navigateur);
  %>
  ```
- **Obtenir l'adresse IP du client** :
  ```
  <%
  String ip = request.getRemoteAddr();
  out.println("Votre adresse IP est : " + ip);
  %>
  ```
- **Stocker et transmettre des données via un `forward`** :
  ```
  <%
  request.setAttribute("message", "Bienvenue !");
  RequestDispatcher rd = request.getRequestDispatcher("affiche.jsp");
  rd.forward(request, response);
  %>
  ```
  Dans `affiche.jsp` :
  ```
  <%= request.getAttribute("message") %>
  ```

Résumé

- `request` représente la requête HTTP côté serveur.
- Il permet d'accéder aux **paramètres**, aux **en-têtes**, aux **cookies**, aux **informations sur le client** et de manipuler des **attributs**.
- Sa portée est **limitée à une seule requête**, contrairement à `session` qui persiste plus longtemps.

Les en-têtes de l'objet `request` Un **en-tête HTTP** est une information envoyée par le client (navigateur) au serveur lors de chaque requête. L'objet implicite `request` permet d'y accéder via la méthode :

```
request.getHeader("Nom-de-l-entete");
```

Voici quelques en-têtes les plus courants avec exemples :

- **User-Agent** : indique le navigateur et le système utilisé par le client.
  ```
  <%
  String navigateur = request.getHeader("User-Agent");
  out.println("Navigateur : " + navigateur);
  %>
  ```
  Exemple de sortie :
  ```
  Mozilla/5.0 (Windows NT 10.0; Win64; x64) Chrome/120.0.0.0 Safari/537.36
  ```
- **Host** : donne le nom de domaine et le port du serveur demandé.
  ```
  <%
  String host = request.getHeader("Host");
  out.println("Hôte : " + host);
  %>
  ```
  Exemple de sortie :
  ```
  www.monsite.com:8080
  ```
- **Accept** : liste les types de contenus que le client peut accepter.
  ```
  <%
  String accept = request.getHeader("Accept");
  out.println("Types acceptés : " + accept);
  %>
  ```
  Exemple de sortie :
  ```
  text/html,application/xhtml+xml,application/xml;q=0.9,*/*;q=0.8
  ```
- **Accept-Language** : indique les langues préférées du client.
  ```
  <%
  String lang = request.getHeader("Accept-Language");
  out.println("Langue préférée : " + lang);
  %>
  ```
  Exemple de sortie :
  ```
  fr-FR,fr;q=0.9,en;q=0.8
  ```

- **Accept-Encoding** : précise les types de compression supportés.
  ```
  <%
  String encoding = request.getHeader("Accept-Encoding");
  out.println("Encodage supporté : " + encoding);
  %>
  ```
  Exemple de sortie :
  ```
  gzip, deflate, br
  ```
- **Referer** : contient l'URL de la page précédente (source de la requête).
  ```
  <%
  String referer = request.getHeader("Referer");
  out.println("Page précédente : " + referer);
  %>
  ```
  Exemple de sortie :
  ```
  https://www.google.com/
  ```
- **Connection** : indique le type de connexion.
  ```
  <%
  String connection = request.getHeader("Connection");
  out.println("Connexion : " + connection);
  %>
  ```
  Exemple de sortie :
  ```
  keep-alive
  ```
- **Cookie** : contient les cookies envoyés par le client.
  ```
  <%
  String cookies = request.getHeader("Cookie");
  out.println("Cookies : " + cookies);
  %>
  ```
  Exemple de sortie :
  ```
  JSESSIONID=ABC123XYZ456; theme=dark
  ```

Lister tous les en-têtes Pour afficher tous les en-têtes reçus dans une requête HTTP :

```
<%
java.util.Enumeration<String> headers = request.getHeaderNames();
while(headers.hasMoreElements()){
String headerName = headers.nextElement();
out.println(headerName + " : " + request.getHeader(headerName) + "<br>");
}
%>
```

Exemple : proposer un téléchargement selon le système d'exploitation Le code JSP suivant détecte le système d'exploitation de l'utilisateur à partir de l'en-tête `User-Agent`, puis propose un fichier adapté en téléchargement :

```
<%@ page import="java.io.*" %>
```

```
<%
// Récupération de l'User-Agent
String userAgent = request.getHeader("User-Agent");
String fichier = "";

// Détection basique du système d'exploitation
if(userAgent != null){
if(userAgent.toLowerCase().contains("windows")){
fichier = "logiciel_windows.exe";
} else if(userAgent.toLowerCase().contains("mac")){
fichier = "logiciel_mac.dmg";
} else if(userAgent.toLowerCase().contains("linux")){
fichier = "logiciel_linux.tar.gz";
} else {
fichier = "logiciel_universel.zip"; // fallback
}
}

// Définition des en-têtes HTTP pour forcer le téléchargement
response.setContentType("application/octet-stream");
response.setHeader("Content-Disposition",
"attachment; filename=\"" + fichier + "\"");

// Lecture du fichier depuis le répertoire /downloads de l'application
String chemin = application.getRealPath("/downloads/" + fichier);
FileInputStream inStream = new FileInputStream(chemin);
OutputStream outStream = response.getOutputStream();

byte[] buffer = new byte[4096];
int bytesRead = -1;
while ((bytesRead = inStream.read(buffer)) != -1) {
outStream.write(buffer, 0, bytesRead);
}

inStream.close();
outStream.close();
%>
```

Explication du code

Le fonctionnement de ce code peut se résumer ainsi :

- **Lecture du User-Agent** : L'en-tête `User-Agent` est récupéré avec `String userAgent = request.getHeader("User-Agent");`

Il permet d'identifier le système d'exploitation de l'utilisateur.

- **Détection du système d'exploitation** : On vérifie si le User-Agent contient `windows`, `mac` ou `linux`. Selon le cas, on choisit un fichier spécifique : `logiciel_windows.exe`, `logiciel_mac.dmg`, `logiciel_linux.tar.gz`, ou bien un fichier universel par défaut.
- **Préparation de la réponse HTTP** :
  ```
  response.setContentType("application/octet-stream");
  response.setHeader("Content-Disposition",
  "attachment; filename=\"" + fichier + "\"");
  ```
  Cela indique au navigateur qu'il doit télécharger le fichier (et non l'afficher).
- **Lecture et écriture du fichier** : On ouvre le fichier dans le répertoire `/downloads` de l'application avec un `FileInputStream`, puis on écrit son contenu dans la réponse via `OutputStream`.
- **Envoi du fichier au client** : Une boucle lit le fichier par blocs de 4096 octets et les transmet au navigateur, jusqu'à la fin. Enfin, on ferme les flux.

Méthodes principales L'objet implicite `request` en JSP, instance de `HttpServletRequest`, permet d'accéder à toutes les informations envoyées par le client au serveur, telles que les paramètres de formulaire, les en-têtes HTTP, les cookies, la session, et bien plus. Les méthodes principales, regroupées par catégorie, avec des exemples pratiques en JSP et Servlet sont :

Paramètres de Requête

getParameter

Description Récupère la première valeur d'un paramètre envoyé par un formulaire ou dans l'URL.

Exemple JSP <String nom = request.getParameter("nom"); out.println("Bonjour " + nom);

Exemple Servlet protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException String nom = request.getParameter("nom"); response.getWriter().println("Bonjour " + nom);

getParameterValues

Description Récupère toutes les valeurs d'un paramètre (cases à cocher, liste multiple).

Exemple JSP <String[] langages = request.getParameterValues("langage"); if(langages != null) for(String l : langages) out.println(l + " ");

Exemple Servlet protected void doPost(HttpServletRequest request, HttpServletResponse response) throws ServletException, IOException String[] langages = request.getParameterValues("langage"); if(langages != null) for(String l : langages) response.getWriter().print(l + " ");

getParameterNames et getParameterMap

Description Permettent de lister tous les paramètres et leurs valeurs.

Exemple JSP <java.util.Enumeration<String> noms = request.getParameterNames(); while(noms.hasMoreE
String n = noms.nextElement(); out.println(n + " = " + request.getParameter(n) + "
");

java.util.Map<String,String[]> map = request.getParameterMap(); map.forEach((k,v) -> out.println(k
+ " -> " + java.util.Arrays.toString(v) + "
"));

En-têtes HTTP

getHeader, getHeaders, getHeaderNames

Exemple JSP <String userAgent = request.getHeader("User-Agent"); out.println("Navigateur
: " + userAgent);

java.util.Enumeration<String> headers = request.getHeaderNames(); while(headers.hasMoreElements())
String h = headers.nextElement(); out.println(h + " : " + request.getHeader(h) + "
");

Cookies

getCookies <javax.servlet.http.Cookie[] cookies = request.getCookies(); if(cookies != null)
for(javax.servlet.http.Cookie c : cookies) out.println(c.getName() + " = " + c.getValue() +
"
");

Attributs de requête

setAttribute, getAttribute, removeAttribute <request.setAttribute("msg", "Bonjour!");
request.getRequestDispatcher("/affiche.jsp").forward(request,response);

Chemins et URI <out.println("URI : " + request.getRequestURI()); out.println("URL : " +
request.getRequestURL()); out.println("Query : " + request.getQueryString()); out.println("ContextPath
: " + request.getContextPath()); out.println("ServletPath : " + request.getServletPath());
out.println("PathInfo : " + request.getPathInfo());

Méthodes HTTP et sécurité <out.println("Méthode : " + request.getMethod()); out.println("Protocole
: " + request.getProtocol()); out.println("isSecure ? " + request.isSecure());

Upload de fichiers

Exemple Servlet MultipartConfig @WebServlet("/upload") @MultipartConfig public class UploadServlet extends HttpServlet protected void doPost(HttpServletRequest request, HttpServletResponse resp) throws ServletException, IOException Part p = request.getPart("file"); String fileName = p.getSubmittedFileName(); try (InputStream in = p.getInputStream(); java.nio.file.OutputStream out = java.nio.file.Files.newOutputStream(java.nio.file.Paths.get("/tmp/" + fileName))) in.transferTo(out);

Téléchargement de fichiers selon OS «String ua = request.getHeader("User-Agent"); String file = ua.toLowerCase().contains("windows") ? "soft$_w$*in.exe*" : *ua.toLowerCase*().*contains*("*mac*")?"*soft*$_m$*ac.dm* "*soft*$_l$*inux.tar.gz*";

response.setContentType("application/octet-stream"); response.setHeader("Content-Disposition","attachment; filename="¨ + file + "¨");

FileInputStream in = new FileInputStream(application.getRealPath("/downloads/" + file)); OutputStream out = response.getOutputStream(); byte[] buffer = new byte[4096]; int bytesRead; while((bytesRead = in.read(buffer)) != -1) out.write(buffer,0,bytesRead); in.close(); out.close();

Gestion de la session et authentification <HttpSession session = request.getSession(); session.setAttribute("user","Zey");

out.println("User=" + request.getRemoteUser()); if(request.isUserInRole("admin")) out.println(" -> admin");

Asynchrone (Servlet 3.x) protected void doGet(HttpServletRequest req, HttpServletResponse resp) AsyncContext ac = req.startAsync(); ac.start(() -> // traitement long en arrière-plan ac.complete(););

Lecture du corps de la requête <BufferedReader r = request.getReader(); String line; while((line = r.readLine()) != null) out.println(line + "
");

Recap. L'objet `request` est central pour JSP/Servlet, permettant d'accéder à toutes les données transmises par le client. Il est conseillé de manipuler la logique côté Servlet et d'utiliser EL/JSTL pour l'affichage JSP.

Objet implicite `response` Dans une JSP, `response` est un objet implicite de type `HttpServletResponse`, fourni par le serveur (Tomcat, GlassFish...). Il représente **la réponse HTTP que le serveur envoie au client** (navigateur). Il n'est donc pas nécessaire de le créer, il est automatiquement disponible dans la JSP.

Principales fonctionnalités de `response` L'objet `response` permet de :

1. **Envoyer du contenu au client** Grâce au `Writer` ou `OutputStream` :
   ```
   <%
   response.getWriter().println("Bonjour depuis JSP !");
   ```

```
%>
```

Ici, `getWriter()` renvoie un `PrintWriter` pour écrire du texte (HTML, JSON...).

2. **Définir le type de contenu (MIME type)** Avec `setContentType(String type)` :

```
<%
response.setContentType("text/html"); // HTML
response.setContentType("application/json"); // JSON
%>
```

3. **Ajouter des en-têtes HTTP** Pour contrôler le cache, redirections, cookies, etc. :

```
<%
response.setHeader("Cache-Control", "no-cache");
response.setHeader("Custom-Header", "Valeur");
%>
```

4. **Rediriger vers une autre page ou URL** Avec `sendRedirect(String location)` :

```
<%
response.sendRedirect("https://www.example.com");
%>
```

Cela envoie une réponse HTTP 302 au navigateur pour qu'il aille à l'URL spécifiée.

5. **Envoyer des codes de statut HTTP** Avec `setStatus(int sc)` ou des méthodes dédiées :

```
<%
response.setStatus(HttpServletResponse.SC_NOT_FOUND); // 404
response.sendError(HttpServletResponse.SC_FORBIDDEN, "Accès interdit");
%>
```

6. **Gérer les cookies** Avec `addCookie(Cookie c)` :

```
<%
Cookie cookie = new Cookie("username", "Zey");
cookie.setMaxAge(3600); // 1 heure
response.addCookie(cookie);
%>
```

7. **Gérer les en-têtes de type redirection, cache et téléchargement** Exemple pour forcer le téléchargement d'un fichier :

```
<%
response.setContentType("application/octet-stream");
response.setHeader("Content-Disposition", "attachment; filename=\"exemple.txt\"");
%>
```

Exemple JSP complet

```
<%@ page import="javax.servlet.http.Cookie" %>
<%
// Définir le type de contenu
response.setContentType("text/html");
```

```
// Ajouter un cookie
Cookie userCookie = new Cookie("user", "Zey");
userCookie.setMaxAge(3600);
response.addCookie(userCookie);

// Ajouter un en-tête HTTP
response.setHeader("Custom-Header", "ValeurTest");

// Écrire du contenu HTML
response.getWriter().println("<h1>Bienvenue sur ma page JSP</h1>");
response.getWriter().println("<p>Contenu généré par l'objet response</p>");

// Redirection conditionnelle
// response.sendRedirect("autrePage.jsp");
%>
```

Méthodes importantes de `response`

- `getWriter()` : Écrire du texte dans la réponse.
- `getOutputStream()` : Écrire des données binaires (images, fichiers...).
- `setContentType(String type)` : Définir le type MIME.
- `setHeader(String name, String value)` : Ajouter ou modifier un en-tête HTTP.
- `addCookie(Cookie c)` : Ajouter un cookie.
- `sendRedirect(String location)` : Rediriger le client vers une autre URL.
- `setStatus(int sc)` : Définir le code HTTP.
- `sendError(int sc, String msg)` : Envoyer un code d'erreur avec message.

Les en-têtes HTTP dans `response` Les **en-têtes HTTP** sont des informations que le serveur envoie au client **avant le corps de la réponse**. Ils contrôlent le comportement du navigateur, la mise en cache, la sécurité, les redirections et d'autres paramètres.

En JSP, on les manipule avec la méthode :

```
response.setHeader(String nom, String valeur);
```

ou pour certains en-têtes particuliers, des méthodes dédiées comme `setContentType()` ou `addCookie()`.

1. Content-Type **But** : Indique le type MIME du contenu (HTML, JSON, image...). **Exemple** :

```
<%
response.setContentType("text/html");
%>
```

Le navigateur sait qu'il doit interpréter la réponse comme du HTML.

2. Content-Disposition **But** : Contrôle comment le contenu est présenté (inline ou téléchargement). **Exemple** :

```
<%
response.setHeader("Content-Disposition", "attachment; filename=\"fichier.txt\"");
%>
```

Le navigateur va proposer un téléchargement du fichier `fichier.txt`.

3. Cache-Control / Pragma / Expires **But** : Contrôler le cache côté client. **Exemple** :

```
<%
response.setHeader("Cache-Control", "no-cache, no-store, must-revalidate");
response.setHeader("Pragma", "no-cache");
response.setDateHeader("Expires", 0);
%>
```

Cela empêche le navigateur de mettre la page en cache.

4. Location **But** : Utilisé pour les redirections (souvent avec code 3xx). **Exemple** :

```
<%
response.setStatus(HttpServletResponse.SC_MOVED_TEMPORARILY);
response.setHeader("Location", "nouvellePage.jsp");
%>
```

Le navigateur est redirigé vers `nouvellePage.jsp`.

Remarque : On préfère généralement `response.sendRedirect("nouvellePage.jsp");` qui fait la même chose automatiquement.

5. Set-Cookie **But** : Ajouter un cookie dans la réponse. **Exemple** :

```
<%
Cookie c = new Cookie("user", "Zey");
c.setMaxAge(3600);
response.addCookie(c);
%>
```

Le navigateur reçoit le cookie `user=Zey` valable 1 heure.

6. Exemples supplémentaires d'en-têtes utiles

- **X-Frame-Options** : Sécurité pour empêcher l'affichage en iframe.
  ```
  <%
  response.setHeader("X-Frame-Options", "DENY");
  %>
  ```
- **Strict-Transport-Security (HSTS)** : Forcer HTTPS.

```
<%
response.setHeader("Strict-Transport-Security", "max-age=31536000; includeSubDomains");
%>
```

- **Custom Header** : Ajouter un en-tête personnalisé.

```
<%
response.setHeader("X-Mon-Header", "ValeurTest");
%>
```

L'objet implicite `response` Dans une JSP, `response` est un objet implicite de type `HttpServletResponse`, fourni par le serveur. Il représente la réponse HTTP que le serveur envoie au client. Il permet de :

- Écrire du contenu dans la réponse.
- Définir le type MIME.
- Ajouter des en-têtes HTTP.
- Gérer les cookies.
- Envoyer des redirections ou des erreurs.
- Contrôler le cache et la sécurité.

1. getWriter() **But** : Obtenir un `PrintWriter` pour envoyer du texte (HTML, JSON, texte brut) dans la réponse. **Exemple** :

```
<%
response.setContentType("text/html");
PrintWriter out = response.getWriter();
out.println("<h1>Bonjour depuis JSP</h1>");
%>
```

2. getOutputStream() **But** : Obtenir un `ServletOutputStream` pour envoyer des données binaires (images, fichiers...). **Exemple** :

```
<%
response.setContentType("image/png");
ServletOutputStream out = response.getOutputStream();
FileInputStream in = new FileInputStream("image.png");
byte[] buffer = new byte[1024];
int len;
while ((len = in.read(buffer)) != -1) {
out.write(buffer, 0, len);
}
in.close();
out.close();
%>
```

3. setContentType(String type) **But** : Définir le type MIME de la réponse pour que le navigateur sache comment l'interpréter. **Exemples** :

```
<%
response.setContentType("text/html");       // HTML
response.setContentType("application/json"); // JSON
response.setContentType("application/pdf");  // PDF
%>
```

4. setHeader(String name, String value) **But** : Ajouter ou modifier un en-tête HTTP personnalisé. **Exemple** :

```
<%
response.setHeader("X-Custom-Header", "ValeurTest");
%>
```

5. addHeader(String name, String value) **But** : Ajouter un en-tête sans remplacer ceux existants du même nom. **Exemple** :

```
<%
response.addHeader("X-Custom-Header", "Valeur1");
response.addHeader("X-Custom-Header", "Valeur2");
%>
```

6. setDateHeader(String name, long date) **But** : Définir un en-tête HTTP avec une valeur de date. **Exemple** :

```
<%
response.setDateHeader("Expires", System.currentTimeMillis() + 3600*1000); // expire dans 1h
%>
```

7. addCookie(Cookie c) **But** : Ajouter un cookie à la réponse. **Exemple** :

```
<%
Cookie cookie = new Cookie("username", "Zey");
cookie.setMaxAge(3600); // 1 heure
response.addCookie(cookie);
%>
```

8. sendRedirect(String location) **But** : Rediriger le client vers une autre URL. Envoie un code HTTP 302. **Exemple** :

```
<%
response.sendRedirect("autrePage.jsp");
%>
```

9. setStatus(int sc) **But** : Définir le code de statut HTTP. **Exemple** :

```
<%
response.setStatus(HttpServletResponse.SC_NOT_FOUND); // 404
%>
```

10. sendError(int sc) **But** : Envoyer un code d'erreur HTTP au client avec le contenu standard du serveur. **Exemple** :

```
<%
response.sendError(HttpServletResponse.SC_FORBIDDEN, "Accès interdit");
%>
```

11. setBufferSize(int size) **But** : Définir la taille du tampon de sortie avant que les données soient envoyées au client. **Exemple** :

```
<%
response.setBufferSize(8192); // 8 Ko
%>
```

12. flushBuffer() **But** : Forcer l'envoi du contenu du tampon au client. **Exemple** :

```
<%
response.getWriter().println("Données temporaires");
response.flushBuffer();
%>
```

13. reset() / resetBuffer() **But** : Réinitialiser le tampon de sortie, utile avant de modifier le type de contenu ou d'envoyer un code d'erreur. **Exemple** :

```
<%
response.reset();
response.setContentType("text/plain");
response.getWriter().println("Nouveau contenu après reset");
%>
```

14. setContentLength(int length) **But** : Indiquer la taille du contenu en octets. **Exemple** :

```
<%
String text = "Bonjour JSP";
response.setContentLength(text.length());
response.getWriter().write(text);
%>
```

15. setCharacterEncoding(String charset) **But** : Définir l'encodage des caractères de la réponse. **Exemple** :

```
<%
response.setCharacterEncoding("UTF-8");
response.setContentType("text/html;charset=UTF-8");
%>
```

HTTP Status Codes en JSP Les **codes de statut HTTP** sont des nombres à 3 chiffres renvoyés par le serveur dans la réponse HTTP pour indiquer le résultat d'une requête. Ils permettent au client (navigateur ou application) de savoir si la requête a réussi, échoué, ou doit être redirigée.

Catégories principales de codes

- **1xx – Informations** : Codes provisoires indiquant que la requête a été reçue et est en cours de traitement. Rarement utilisés dans JSP.
- **2xx – Succès** :
  - 200 OK : La requête a réussi et le serveur renvoie le contenu demandé.
  - 201 Created : Ressource créée avec succès (ex. POST).
- **3xx – Redirections** :
  - 301 Moved Permanently : La ressource a été déplacée définitivement.
  - 302 Found / Moved Temporarily : Redirection temporaire.
  - 304 Not Modified : La ressource n'a pas changé (cache).
- **4xx – Erreurs côté client** :
  - 400 Bad Request : Requête invalide.
  - 401 Unauthorized : Authentification requise.
  - 403 Forbidden : Accès interdit.
  - 404 Not Found : Page ou ressource introuvable.
- **5xx – Erreurs côté serveur** :
  - 500 Internal Server Error : Erreur générale du serveur.
  - 503 Service Unavailable : Serveur indisponible temporairement.

Gérer les codes HTTP en JSP Dans JSP, l'objet implicite `response` permet de définir le code de statut HTTP avec plusieurs méthodes :

1. setStatus(int sc) Définit le code de statut sans message détaillé. **Exemple** :

```
<%
response.setStatus(HttpServletResponse.SC_NOT_FOUND);
%>
```

2. sendError(int sc, String msg) Envoie un code d'erreur HTTP avec un message optionnel et affiche la page d'erreur par défaut du serveur. **Exemple** :

```
<%
response.sendError(HttpServletResponse.SC_FORBIDDEN, "Accès interdit");
%>
```

3. Redirections (codes 3xx) Méthode 1 : sendRedirect(String location) Envoie un code 302 (Found) pour rediriger vers une autre URL. **Exemple** :

```
<%
response.sendRedirect("login.jsp");
%>
```

Méthode 2 : Redirection manuelle Pour définir une redirection permanente :

```
<%
response.setStatus(HttpServletResponse.SC_MOVED_PERMANENTLY);
response.setHeader("Location", "nouvellePage.jsp");
%>
```

4. Exemples pratiques Exemple 1 – Page non trouvée

```
<%
response.setStatus(HttpServletResponse.SC_NOT_FOUND);
out.println("<h1>404 - Page non trouvée</h1>");
%>
```

Exemple 2 – Accès interdit

```
<%
response.sendError(HttpServletResponse.SC_FORBIDDEN, "Vous n'avez pas la permission d'accéde
%>
```

Exemple 3 – Redirection vers une page de login

```
<%
response.sendRedirect("login.jsp");
%>
```

Le traitement du formulaire en JSP

1. Récupération des chaînes de caractères Dans un formulaire, les champs texte `<input type="text">` ou `<textarea>` renvoient des chaînes de caractères.

Exemple HTML :

```
<form action="traitement.jsp" method="post">
```

```
Nom : <input type="text" name="nom"><br>
Prénom : <input type="text" name="prenom"><br>
<input type="submit" value="Envoyer">
</form>
```

Exemple JSP :

```
<%
String nom = request.getParameter("nom");
String prenom = request.getParameter("prenom");

out.println("Nom : " + nom + "<br>");
out.println("Prénom : " + prenom);
%>
```

2. Récupération des nombres Pour traiter des nombres, on récupère la chaîne et on la convertit.

Exemple JSP :

```
<%
String ageStr = request.getParameter("age");
int age = 0;
if(ageStr != null && !ageStr.isEmpty()){
age = Integer.parseInt(ageStr);
}
out.println("Âge : " + age);
%>
```

Gestion des exceptions :

```
try {
age = Integer.parseInt(ageStr);
} catch(NumberFormatException e) {
out.println("Veuillez saisir un nombre valide !");
}
```

3. Boutons radio (choix unique) Les boutons radio permettent de choisir une seule option dans un groupe.

Exemple HTML :

```
<form action="traitement.jsp" method="post">
Sexe :
<input type="radio" name="sexe" value="Homme">Homme
<input type="radio" name="sexe" value="Femme">Femme
<input type="submit" value="Envoyer">
```

```
</form>
```

Exemple JSP :

```
<%
String sexe = request.getParameter("sexe");
out.println("Sexe : " + sexe);
%>
```

4. Cases à cocher (choix multiples) Les cases à cocher permettent de choisir plusieurs options.

Exemple HTML :

```
<form action="traitement.jsp" method="post">
Langages connus :
<input type="checkbox" name="langages" value="Java">Java
<input type="checkbox" name="langages" value="Python">Python
<input type="checkbox" name="langages" value="C++">C++
<input type="submit" value="Envoyer">
</form>
```

Exemple JSP :

```
<%
String[] langages = request.getParameterValues("langages");
if(langages != null){
for(String lang : langages){
out.println(lang + "<br>");
}
} else {
out.println("Aucun langage sélectionné");
}
%>
```

5. Listes déroulantes (select) Les listes peuvent être single ou multiple.

Exemple HTML :

```
<form action="traitement.jsp" method="post">
Ville :
<select name="ville">
<option value="Paris">Paris</option>
<option value="Lyon">Lyon</option>
<option value="Marseille">Marseille</option>
</select>
<br>
```

```
Langues parlées :
<select name="langues" multiple>
<option value="Français">Français</option>
<option value="Anglais">Anglais</option>
<option value="Espagnol">Espagnol</option>
</select>
<input type="submit" value="Envoyer">
</form>
```

Exemple JSP :

```
<%
String ville = request.getParameter("ville");
out.println("Ville : " + ville + "<br>");

String[] langues = request.getParameterValues("langues");
if(langues != null){
out.println("Langues parlées : <br>");
for(String l : langues){
out.println(l + "<br>");
}
}
%>
```

6. Fichiers (upload) Pour traiter un fichier, il faut utiliser `enctype="multipart/form-data"` et une API comme Apache Commons FileUpload.

Exemple HTML :

```
<form action="upload.jsp" method="post" enctype="multipart/form-data">
Sélectionner un fichier : <input type="file" name="monFichier"><br>
<input type="submit" value="Envoyer">
</form>
```

Exemple JSP avec Commons FileUpload :

```
<%@ page import="org.apache.commons.fileupload.*, org.apache.commons.fileupload.disk.*, java
<%
boolean isMultipart = ServletFileUpload.isMultipartContent(request);
if(isMultipart){
DiskFileItemFactory factory = new DiskFileItemFactory();
ServletFileUpload upload = new ServletFileUpload(factory);
try{
List<FileItem> items = upload.parseRequest(request);
for(FileItem item : items){
if(!item.isFormField()){
```

```
String fileName = item.getName();
out.println("Nom du fichier : " + fileName);
item.write(new java.io.File("C:/uploads/" + fileName));
}
}
} catch(Exception e){
out.println("Erreur : " + e.getMessage());
}
}
%>
```

7. Champs cachés (hidden) Les champs cachés transmettent des informations invisibles à l'utilisateur.

Exemple HTML :

```
<input type="hidden" name="id" value="12345">
```

Exemple JSP :

```
String id = request.getParameter("id");
out.println("ID caché : " + id);
```

8. Dates, emails, URLs, nombres HTML5 Les types `<input type="date|email|url|number">` renvoient aussi des chaînes, donc une conversion peut être nécessaire pour les nombres ou dates.

Exemple pour un nombre :

```
String montantStr = request.getParameter("montant");
double montant = Double.parseDouble(montantStr);
```

Résumé des méthodes JSP pour les formulaires

- **Texte / hidden** : `request.getParameter("name")`
- **Nombres** : `Integer.parseInt(request.getParameter("name"))`
- **Radio / Select** : `request.getParameter("name")`
- **Checkbox / multi-select** : `request.getParameterValues("name")`
- **Fichier** : utilisation de l'API `FileUpload` ou de `request.getPart()`

Formulaire complet JSP avec tous les types de champs

```
<%@ page import="org.apache.commons.fileupload.*, org.apache.commons.fileupload.disk.*, java
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<html>
<head>
<title>Traitement Formulaire JSP Complet</title>
</head>
```

```
<body>
<h2>Formulaire Complet</h2>

<form action="" method="post" enctype="multipart/form-data">
<!-- Champs texte -->
Nom : <input type="text" name="nom"><br>
Prénom : <input type="text" name="prenom"><br>

<!-- Champ nombre -->
Âge : <input type="number" name="age"><br>

<!-- Boutons radio -->
Sexe :
<input type="radio" name="sexe" value="Homme">Homme
<input type="radio" name="sexe" value="Femme">Femme<br>

<!-- Cases à cocher -->
Langages connus :
<input type="checkbox" name="langages" value="Java">Java
<input type="checkbox" name="langages" value="Python">Python
<input type="checkbox" name="langages" value="C++">C++<br>

<!-- Listes déroulantes -->
Ville :
<select name="ville">
<option value="Paris">Paris</option>
<option value="Lyon">Lyon</option>
<option value="Marseille">Marseille</option>
</select><br>

Langues parlées :
<select name="langues" multiple>
<option value="Français">Français</option>
<option value="Anglais">Anglais</option>
<option value="Espagnol">Espagnol</option>
</select><br>

<!-- Champ caché -->
<input type="hidden" name="id" value="12345">

<!-- Fichier -->
Sélectionner un fichier : <input type="file" name="monFichier"><br>
```

```
<input type="submit" value="Envoyer">
</form>

<hr>

<h2>Résultats :</h2>
<%
// Vérifier si c'est un formulaire multipart (fichier)
boolean isMultipart = ServletFileUpload.isMultipartContent(request);
if(isMultipart){
DiskFileItemFactory factory = new DiskFileItemFactory();
ServletFileUpload upload = new ServletFileUpload(factory);
try{
List<FileItem> items = upload.parseRequest(request);
for(FileItem item : items){
if(item.isFormField()){
// Champs normaux
out.println(item.getFieldName() + " : " + item.getString() + "<br>");
} else {
// Fichier
String fileName = item.getName();
out.println("Fichier uploadé : " + fileName + "<br>");
if(fileName != null && !fileName.isEmpty()){
item.write(new java.io.File("C:/uploads/" + fileName));
}
}
}
} catch(Exception e){
out.println("Erreur : " + e.getMessage());
}
} else if(request.getParameter("nom") != null){
// Si pas de fichier, récupérer les autres champs
String nom = request.getParameter("nom");
String prenom = request.getParameter("prenom");
String ageStr = request.getParameter("age");
int age = 0;
try{
if(ageStr != null && !ageStr.isEmpty()){
age = Integer.parseInt(ageStr);
}
} catch(NumberFormatException e){
```

```
out.println("Âge invalide !<br>");
}

String sexe = request.getParameter("sexe");
String[] langages = request.getParameterValues("langages");
String ville = request.getParameter("ville");
String[] langues = request.getParameterValues("langues");
String id = request.getParameter("id");

out.println("Nom : " + nom + "<br>");
out.println("Prénom : " + prenom + "<br>");
out.println("Âge : " + age + "<br>");
out.println("Sexe : " + (sexe != null ? sexe : "Non spécifié") + "<br>");

out.print("Langages connus : ");
if(langages != null){
for(String l : langages) out.print(l + " ");
} else out.print("Aucun");
out.println("<br>");

out.println("Ville : " + ville + "<br>");

out.print("Langues parlées : ");
if(langues != null){
for(String l : langues) out.print(l + " ");
} else out.print("Aucune");
out.println("<br>");

out.println("ID caché : " + id + "<br>");
}
%>
</body>
</html>
```

Filtres Servlet et JSP Un **filtre** est un objet spécial en Java EE (Jakarta EE) qui agit comme un **intercepteur** entre le client (navigateur) et une ressource web (Servlet, JSP, ou fichier statique). Il permet de **pré-traiter** ou **post-traiter** les requêtes et réponses HTTP. Contrairement aux servlets, les filtres ne génèrent pas directement une réponse, mais modifient ou contrôlent le flux avant qu'il atteigne sa destination.

Rôles typiques d'un filtre Les filtres sont souvent utilisés pour :

- **Authentification / Autorisation** : vérifier si l'utilisateur est connecté avant d'accéder à

une ressource.

- **Journalisation (Logging)** : enregistrer les détails de chaque requête (IP, URL, temps d'accès...).
- **Compression** : compresser la réponse HTTP (par ex. en GZIP).
- **Encodage** : définir le charset (`UTF-8`).
- **Filtrage de contenu** : supprimer des mots interdits dans une réponse HTML.

Cycle de vie d'un filtre Un filtre suit un cycle semblable à celui des servlets :

1. **Chargement** : le serveur d'application charge le filtre et appelle `init(FilterConfig config)`.
2. **Exécution** : pour chaque requête concernée, la méthode `doFilter(ServletRequest req, ServletResponse res, FilterChain chain)` est appelée.
   - Pré-traiter la requête.
   - Transmettre au filtre suivant ou à la ressource via `chain.doFilter(req, res)`.
   - Post-traiter la réponse.
3. **Destruction** : lorsque l'application est arrêtée, `destroy()` est appelée.

Exemple simple d'un filtre

```
import java.io.IOException;
import jakarta.servlet.*;

public class LoggingFilter implements Filter {
public void init(FilterConfig config) throws ServletException {
System.out.println("LoggingFilter initialisé !");
}

public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
throws IOException, ServletException {
System.out.println("Requête reçue à : " + new java.util.Date());

// Transmission au filtre suivant ou à la ressource
chain.doFilter(req, res);

System.out.println("Réponse envoyée à : " + new java.util.Date());
}

public void destroy() {
System.out.println("LoggingFilter détruit !");
}
}
```

Déclaration d'un filtre a) Via `web.xml` :

```
<filter>
<filter-name>LoggingFilter</filter-name>
<filter-class>com.example.LoggingFilter</filter-class>
</filter>

<filter-mapping>
<filter-name>LoggingFilter</filter-name>
<url-pattern>/*</url-pattern>
</filter-mapping>
```

b) Via annotations :

```
@WebFilter("/*")
public class LoggingFilter implements Filter {
...
}
```

Filtres et JSP Un JSP est compilé en Servlet par le conteneur. Ainsi, un filtre peut aussi intercepter une requête vers un JSP exactement comme pour une servlet.

Exemples concrets :

- Imposer une authentification avant d'afficher `dashboard.jsp`.
- Forcer l'encodage UTF-8 avant de traiter un `formulaire.jsp`.

Chaînage des filtres Plusieurs filtres peuvent être appliqués à la même ressource. L'ordre de déclaration (dans `web.xml` ou via annotations) définit la séquence d'exécution.

Exemple :

1. `AuthFilter` : vérifie si l'utilisateur est connecté.
2. `LoggingFilter` : enregistre l'accès.
3. `CompressionFilter` : compresse la réponse.

Résumé

- Les **servlets** traitent la logique métier et génèrent la réponse.
- Les **JSP** affichent la vue (HTML dynamique).
- Les **filtres** ajoutent une couche de contrôle transversal (sécurité, logging, encodage...) avant ou après le traitement.

Filtres d'authentification Ces filtres servent à contrôler l'accès aux ressources sécurisées. Ils vérifient si un utilisateur est connecté ou si un jeton (token) est valide.

Exemple (session) :

```
@WebFilter("/secure/*")
public class AuthFilter implements Filter {
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
throws IOException, ServletException {
HttpServletRequest request  = (HttpServletRequest) req;
HttpServletResponse response = (HttpServletResponse) res;

HttpSession session = request.getSession(false);
if (session != null && session.getAttribute("user") != null) {
chain.doFilter(req, res); // accès autorisé
} else {
response.sendRedirect(request.getContextPath() + "/login.jsp");
}
}
}
```

Filtres de compression de données Ces filtres réduisent la taille des réponses HTTP (HTML, JSON, CSS, JS) via GZIP.

Exemple :

```
@WebFilter("/*")
public class GzipFilter implements Filter {
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
throws IOException, ServletException {
HttpServletRequest request = (HttpServletRequest) req;
HttpServletResponse response = (HttpServletResponse) res;

String ae = request.getHeader("Accept-Encoding");
if (ae != null && ae.contains("gzip")) {
GzipResponseWrapper gzipResp = new GzipResponseWrapper(response);
chain.doFilter(req, gzipResp);
gzipResp.finish();
} else {
chain.doFilter(req, res);
}
}
}
```

Filtres de cryptage Ils chiffrent ou déchiffrent les données sensibles. Utiles en plus du HTTPS pour protéger certaines informations applicatives.

Exemple (schéma) :

```
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
```

```
throws IOException, ServletException {
byte[] encrypted = req.getInputStream().readAllBytes();
byte[] plain = decryptAesGcm(encrypted, key);
HttpServletRequest wrapped = new BufferedRequestWrapper(
(HttpServletRequest) req, plain);
chain.doFilter(wrapped, res);
}
```

Filtres qui déclenchent des événements d'accès aux ressources Ils publient un événement (audit, log, métrique) à chaque requête.

Exemple :

```
@WebFilter("/*")
public class AccessEventFilter implements Filter {
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
throws IOException, ServletException {
HttpServletRequest r = (HttpServletRequest) req;
long start = System.currentTimeMillis();
chain.doFilter(req, res);
long duration = System.currentTimeMillis() - start;
System.out.println("Accès à " + r.getRequestURI() + " en " + duration + "ms");
}
}
```

Filtres de conversion d'image Ils transforment des images (redimensionnement, conversion WebP, etc.).

Exemple :

```
@WebFilter("/images/*")
public class ImageResizeFilter implements Filter {
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
throws IOException, ServletException {
int width = Integer.parseInt(req.getParameter("w"));
BufferedImage src = ImageIO.read(req.getInputStream());
BufferedImage dest = new BufferedImage(width,
width * src.getHeight()/src.getWidth(), BufferedImage.TYPE_INT_RGB);
Graphics2D g = dest.createGraphics();
g.drawImage(src, 0, 0, dest.getWidth(), dest.getHeight(), null);
g.dispose();
res.setContentType("image/jpeg");
ImageIO.write(dest, "jpg", res.getOutputStream());
}
}
```

Filtres de journalisation et d'audit Ils enregistrent les requêtes et réponses pour le suivi et la traçabilité.

Exemple :

```
@WebFilter("/*")
public class LoggingFilter implements Filter {
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
throws IOException, ServletException {
HttpServletRequest r = (HttpServletRequest) req;
System.out.println("Requête " + r.getMethod() + " : " + r.getRequestURI());
chain.doFilter(req,res);
}
}
```

Filtres de chaîne de type MIME Ils imposent ou modifient le type MIME des réponses (Content-Type).

Exemple :

```
@WebFilter("/*")
public class CharsetFilter implements Filter {
public void doFilter(ServletRequest req, ServletResponse res, FilterChain chain)
throws IOException, ServletException {
req.setCharacterEncoding("UTF-8");
res.setCharacterEncoding("UTF-8");
res.setContentType("text/html; charset=UTF-8");
chain.doFilter(req,res);
}
}
```

Filtres de tokenisation Ils remplacent des données sensibles (ex. numéro de carte) par un token.

Exemple (schéma) :

```
String body = new String(req.getInputStream().readAllBytes(), StandardCharsets.UTF_8);
String tokenized = body.replaceAll("\\d{16}", "****TOKEN****");
HttpServletRequest wrapped = new BufferedRequestWrapper(
(HttpServletRequest) req, tokenized.getBytes());
chain.doFilter(wrapped, res);
```

Filtres XSL/T Ils transforment le contenu XML en un autre format (HTML, texte, etc.) via XSLT.

Exemple :

```
TransformerFactory tf = TransformerFactory.newInstance();
Templates template = tf.newTemplates(new StreamSource("view.xsl"));
Transformer t = template.newTransformer();
Source xmlSource = new StreamSource(new StringReader(xmlString));
t.transform(xmlSource, new StreamResult(res.getWriter()));
```

Les Cookies en JSP Un **cookie** est un petit fichier texte stocké côté navigateur client. Il contient des informations sous forme de paires *nom/valeur*, par exemple : `nom=Zey; age=25; theme=dark`. Ils sont utilisés pour conserver des préférences, garder un utilisateur connecté ou encore suivre la navigation.

Création d'un cookie En JSP ou Servlet, on peut créer un cookie comme suit :

```
<%
Cookie ck = new Cookie("user", "Alice");
ck.setMaxAge(60*60); // expire dans 1 heure
response.addCookie(ck);
%>
```

Lecture d'un cookie Pour lire les cookies envoyés par le client :

```
<%
Cookie[] cookies = request.getCookies();
if (cookies != null) {
for (Cookie c : cookies) {
if ("user".equals(c.getName())) {
out.println("Bonjour " + c.getValue());
}
}
}
%>
```

Suppression d'un cookie On ne peut pas directement supprimer un cookie, mais on peut le rendre expiré en fixant sa durée de vie à zéro :

```
<%
Cookie ck = new Cookie("user", "");
ck.setMaxAge(0); // expire immédiatement
response.addCookie(ck);
%>
```

Limites et sécurité

- Taille limitée à environ 4 Ko par cookie.
- Chaque domaine peut stocker environ 20 cookies.

- Les cookies sont accessibles côté client, donc il ne faut jamais y mettre des données sensibles.
- Pour plus de sécurité, privilégier les sessions côté serveur.

Exemple concret : Préférence de langue

```
<%
String lang = request.getParameter("lang");
if (lang != null) {
Cookie ck = new Cookie("lang", lang);
ck.setMaxAge(24*60*60); // 1 jour
response.addCookie(ck);
}
%>

<%
String langue = "fr"; // valeur par défaut
Cookie[] cookies = request.getCookies();
if (cookies != null) {
for (Cookie c : cookies) {
if ("lang".equals(c.getName())) {
langue = c.getValue();
}
}
}
out.println("Langue actuelle : " + langue);
%>
```

Les méthodes de la classe Cookie La classe `javax.servlet.http.Cookie` (ou `jakarta.servlet.http.Coo` selon la version) fournit plusieurs méthodes pour manipuler les cookies.

- **Constructeur**
  - `Cookie(String name, String value)` : crée un cookie avec un nom et une valeur.
- **Méthodes d'accès et de modification**
  - `String getName()` : retourne le nom du cookie (non modifiable après création).
  - `String getValue()` : retourne la valeur du cookie.
  - `void setValue(String newValue)` : modifie la valeur du cookie.
- **Durée de vie**
  - `void setMaxAge(int expiry)` : définit la durée de vie (en secondes).
    - > 0 : persiste pour le nombre de secondes défini.
    - 0 : supprime immédiatement le cookie.
    - < 0 : cookie de session (supprimé à la fermeture du navigateur).
  - `int getMaxAge()` : retourne la durée de vie en secondes.

- **Domaine**
  - `void setDomain(String pattern)` : définit le domaine du cookie (ex. `.example.com`).
  - `String getDomain()` : retourne le domaine associé.
- **Chemin**
  - `void setPath(String uri)` : définit le chemin où le cookie est valide.
  - `String getPath()` : retourne le chemin défini.
- **Sécurité et protocole**
  - `void setSecure(boolean flag)` : indique si le cookie doit être envoyé uniquement en HTTPS.
  - `boolean getSecure()` : retourne `true` si le cookie est sécurisé.
- **Version**
  - `void setVersion(int v)` : définit la version (0 = Netscape, 1 = RFC 2109).
  - `int getVersion()` : retourne la version.
- **Commentaire**
  - `void setComment(String purpose)` : ajoute une description du cookie.
  - `String getComment()` : retourne le commentaire.

Exemple d'utilisation

```
Cookie c = new Cookie("username", "Zey");
c.setMaxAge(3600);          // expire après 1 heure
c.setPath("/app");          // valable pour le chemin /app
c.setDomain("example.com"); // valable pour ce domaine
c.setSecure(true);          // transmis uniquement en HTTPS
response.addCookie(c);
```

L'objet de session (HttpSession) En **Java EE/Jakarta EE**, l'objet `HttpSession` permet de **conserver des données spécifiques à un utilisateur entre plusieurs requêtes HTTP**.

1. Pourquoi avons-nous besoin de la session ? Le protocole **HTTP est sans état** (stateless), ce qui signifie que chaque requête envoyée par le navigateur vers le serveur est **indépendante**. La session permet de **stocker temporairement ces informations côté serveur** pour chaque utilisateur.

2. Création et récupération d'une session On utilise l'objet `HttpServletRequest` pour obtenir la session :

```
HttpSession session = request.getSession();       // Crée si inexistante
HttpSession sessionExist = request.getSession(false); // Ne crée pas, retourne null si aucur
```

3. Stocker et récupérer des données dans la session La session fonctionne comme une **map clé-valeur** :

```
// Stocker un objet
session.setAttribute("username", "Zey");

// Récupérer un objet
String user = (String) session.getAttribute("username");

// Supprimer un objet
session.removeAttribute("username");
```

4. Durée de vie de la session Chaque session possède une durée de vie définie par le serveur (souvent 30 minutes) :

```
// Définir le timeout à 10 minutes
session.setMaxInactiveInterval(10*60); // en secondes
```

5. Supprimer une session Pour terminer une session :

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

6. Informations disponibles dans HttpSession

- `session.getId()` : ID unique de la session
- `session.getCreationTime()` : date de création
- `session.getLastAccessedTime()` : dernière requête
- `session.getMaxInactiveInterval()` : durée maximale d'inactivité

7. Bonnes pratiques

1. Ne pas stocker de gros objets dans la session.
2. Ne pas stocker d'informations sensibles sans sécurisation.
3. Invalider la session après un logout.
4. Limiter la durée de session pour économiser la mémoire serveur.

En résumé, l'objet `HttpSession` permet de **conserver l'état d'un utilisateur entre plusieurs requêtes**, fonctionne comme un dictionnaire côté serveur et est identifié par un ID unique (`JSESSIONID`).

Méthodes principales de l'objet HttpSession L'objet `HttpSession` fournit plusieurs méthodes pour manipuler les données de session, gérer la durée de vie et obtenir des informations sur la session.

1. getId()

- **Description :** Retourne l'ID unique de la session (`JSESSIONID`).

- **Usage concret :** Identifier une session particulière dans les logs, déboguer des problèmes liés aux sessions multiples.

```
String sessionId = session.getId();
System.out.println("Session ID : " + sessionId);
```

2. getCreationTime()

- **Description :** Retourne le timestamp de création de la session.
- **Usage concret :** Savoir depuis combien de temps l'utilisateur est connecté, afficher "Vous êtes connecté depuis X minutes".

```
long creationTime = session.getCreationTime();
System.out.println("Session créée à : " + new Date(creationTime));
```

3. getLastAccessedTime()

- **Description :** Retourne le timestamp de la dernière requête de l'utilisateur.
- **Usage concret :** Détecter les utilisateurs inactifs pour les déconnecter automatiquement, analyser l'activité sur le site.

```
long lastAccess = session.getLastAccessedTime();
System.out.println("Dernière activité : " + new Date(lastAccess));
```

4. setMaxInactiveInterval(int seconds)

- **Description :** Définit le temps maximum d'inactivité avant expiration de la session.
- **Usage concret :** Sécuriser les comptes sensibles, éviter de surcharger le serveur avec des sessions inutilisées.

```
session.setMaxInactiveInterval(15*60); // 15 minutes
```

5. getMaxInactiveInterval()

- **Description :** Retourne le timeout configuré pour la session.
- **Usage concret :** Afficher un message de warning avant expiration automatique.

```
int timeout = session.getMaxInactiveInterval();
System.out.println("Timeout session : " + timeout + " secondes");
```

6. setAttribute(String name, Object value)

- **Description :** Stocke un objet dans la session avec une clé unique.
- **Usage concret :** Conserver des informations utilisateur entre les pages, comme le panier ou le nom.

```
session.setAttribute("username", "Zey");
session.setAttribute("cart", cartObject);
```

7. getAttribute(String name)

- **Description :** Récupère un objet stocké dans la session.
- **Usage concret :** Lire les informations précédemment stockées pour personnaliser l'interface.

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

8. removeAttribute(String name)

- **Description :** Supprime un attribut de la session.
- **Usage concret :** Supprimer un panier ou un rôle utilisateur lors d'un logout partiel.

```
session.removeAttribute("cart");
```

9. invalidate()

- **Description :** Termine la session et supprime toutes ses données.
- **Usage concret :** Déconnexion d'un utilisateur, effacer les informations sensibles.

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

10. isNew()

- **Description :** Indique si la session vient juste d'être créée.
- **Usage concret :** Afficher un message de bienvenue pour un nouvel utilisateur.

```
if (session.isNew()) {
System.out.println("Bienvenue sur le site !");
}
```

Utilisations concrètes de HttpSession

1. **Gestion d'authentification :** Stocker l'identifiant de l'utilisateur après login et vérifier la session à chaque page.
2. **Panier d'achat :** Ajouter, modifier ou supprimer des articles dans la session pour conserver le panier entre pages.
3. **Personnalisation :** Sauvegarder la langue ou les préférences de l'utilisateur.
4. **Sécurité et timeout :** Déconnecter automatiquement un utilisateur après inactivité et supprimer les données sensibles après logout.
5. **Tracking et statistiques :** Suivre le nombre d'utilisateurs actifs et identifier les sessions inactives pour nettoyage.

Principe du téléchargement de fichiers En JSP/Servlet, un téléchargement de fichier consiste à envoyer un fichier du serveur vers le client via HTTP. Pour que le navigateur du client comprenne qu'il doit télécharger le fichier plutôt que l'afficher, il faut manipuler correctement les en-têtes HTTP.

Les étapes principales sont :

1. Identifier le fichier sur le serveur (chemin physique ou fichier dans `WEB-INF`).
2. Définir le type MIME du fichier (`application/pdf`, `image/png`, `application/octet-stream` pour binaire générique, etc.).
3. Spécifier l'en-tête `Content-Disposition` pour forcer le téléchargement.
4. Lire le fichier et écrire son contenu dans le flux de sortie de la réponse (`response.getOutputStream()`).

Exemple pratique avec JSP et Servlet Structure du projet :

```
WebContent/
 WEB-INF/
    files/
        exemple.txt
 download.jsp
 DownloadServlet.java
```

JSP avec un lien de téléchargement

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<html>
<head>
<title>Téléchargement de fichier</title>
</head>
<body>
<h2>Télécharger le fichier exemple.txt</h2>
<a href="DownloadServlet?file=exemple.txt">Télécharger</a>
</body>
</html>
```

Servlet pour gérer le téléchargement

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

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

String fileName = request.getParameter("file"); // Nom du fichier à télécharger
if (fileName == null || fileName.isEmpty()) {
response.getWriter().println("Nom de fichier invalide");
return;
}
```

```
// Chemin du fichier dans WEB-INF/files
String filePath = getServletContext().getRealPath("/WEB-INF/files/" + fileName);
File file = new File(filePath);

if (!file.exists()) {
response.getWriter().println("Fichier non trouvé");
return;
}

// Définir le type MIME
response.setContentType("application/octet-stream");
// Indiquer au navigateur que c'est un téléchargement
response.setHeader("Content-Disposition", "attachment;filename=" + fileName);
response.setContentLength((int) file.length());

// Lire le fichier et l'envoyer
FileInputStream in = new FileInputStream(file);
OutputStream out = response.getOutputStream();

byte[] buffer = new byte[4096];
int bytesRead = -1;
while ((bytesRead = in.read(buffer)) != -1) {
out.write(buffer, 0, bytesRead);
}

in.close();
out.close();
}
}
```

Points importants

- **Sécurité :** Ne jamais permettre aux utilisateurs de télécharger n'importe quel fichier du serveur (`../` peut poser problème). Toujours restreindre à un dossier spécifique.
- **WEB-INF :** Placer les fichiers dans `WEB-INF` les protège d'un accès direct via URL, ce qui est sécurisé.
- **MIME type :** `application/octet-stream` fonctionne pour tout type de fichier si vous voulez forcer le téléchargement.
- **Buffering :** Lire et écrire en blocs (ex: 4 KB) pour les gros fichiers afin d'éviter les problèmes de mémoire.

Téléchargement amélioré de fichiers en JSP/Servlet Cette version permet de :

- Gérer des fichiers volumineux sans saturer la mémoire.
- Détecter automatiquement le type MIME du fichier.
- Maintenir la sécurité en limitant l'accès aux fichiers dans un dossier spécifique.

Servlet améliorée pour téléchargement

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

public class DownloadServlet extends HttpServlet {
private static final int BUFFER_SIZE = 8192; // 8 KB

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

String fileName = request.getParameter("file");
if (fileName == null || fileName.isEmpty()) {
response.getWriter().println("Nom de fichier invalide");
return;
}

// Chemin sécurisé du fichier dans WEB-INF/files
String filePath = getServletContext().getRealPath("/WEB-INF/files/" + fileName);
File file = new File(filePath);

if (!file.exists()) {
response.getWriter().println("Fichier non trouvé");
return;
}

// Détection automatique du type MIME
String mimeType = getServletContext().getMimeType(filePath);
if (mimeType == null) {
mimeType = "application/octet-stream";
}
response.setContentType(mimeType);

// Indiquer au navigateur que c'est un téléchargement
response.setHeader("Content-Disposition", "attachment;filename=" + fileName);
response.setContentLengthLong(file.length());

// Lecture et écriture en flux tamponné pour gros fichiers
```

```
try (BufferedInputStream in = new BufferedInputStream(new FileInputStream(file));
BufferedOutputStream out = new BufferedOutputStream(response.getOutputStream())) {

byte[] buffer = new byte[BUFFER_SIZE];
int bytesRead = -1;
while ((bytesRead = in.read(buffer)) != -1) {
out.write(buffer, 0, bytesRead);
}
}
}
}
```

Points clés de cette version

- **BUFFER_SIZE :** Utilisation d'un buffer de 8 KB pour traiter les fichiers volumineux.
- **Détection MIME :** Utilisation de `getServletContext().getMimeType()` pour définir le type correct du fichier.
- **Sécurité :** Les fichiers sont placés dans `WEB-INF/files`, inaccessibles directement par URL.
- **Flux tamponné :** Utilisation de `BufferedInputStream` et `BufferedOutputStream` pour améliorer les performances et réduire la consommation mémoire.

Redirection des pages en JSP En JSP, il existe principalement deux façons de naviguer d'une page à une autre : **forward** (côté serveur) et **redirect** (côté client).

1. Redirection côté serveur (Forward) Le `forward` est effectué côté serveur avec l'objet `RequestDispatcher`. La page source reste la même pour le client, et les attributs de la requête sont conservés.

```
<%
String destination = "page2.jsp";
RequestDispatcher rd = request.getRequestDispatcher(destination);
rd.forward(request, response);
%>
```

2. Redirection côté client (Redirect) Le `redirect` est effectué côté client via `response.sendRedirect()`. Le navigateur reçoit un code HTTP 302 et recharge une nouvelle URL.

```
<%
response.sendRedirect("page2.jsp");
%>
```

3. Exemple complet avec formulaire **page1.jsp** : formulaire de saisie

```
<html>
```

```
<body>
<form action="page1.jsp" method="post">
Nom: <input type="text" name="nom">
<input type="submit" value="Envoyer">
</form>

<%
String nom = request.getParameter("nom");
if (nom != null) {
// Forward vers page2.jsp
// RequestDispatcher rd = request.getRequestDispatcher("page2.jsp");
// rd.forward(request, response);

// Redirect vers page2.jsp
response.sendRedirect("page2.jsp?nom=" + nom);
}
%>
</body>
</html>
```

page2.jsp : affichage du nom

```
<html>
<body>
<h2>Bienvenue</h2>
Nom reçu: <%= request.getParameter("nom") %>
</body>
</html>
```

Actualisation automatique des pages en JSP L'actualisation automatique des pages en JSP peut se faire de plusieurs manières selon le contexte : via HTTP, JSP ou JavaScript côté client.

1. Via la directive meta HTML C'est la méthode la plus simple et la plus courante. On utilise la balise `<meta>` pour demander au navigateur de recharger la page automatiquement après un certain intervalle.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>Actualisation automatique</title>
<!-- Actualisation toutes les 5 secondes -->
<meta http-equiv="refresh" content="5">
```

```
</head>
<body>
<h1>Heure actuelle : <%= new java.util.Date() %></h1>
</body>
</html>
```

Explications :

- `http-equiv="refresh"` indique que la page doit être rafraîchie.
- `content="5"` signifie que la page se rechargera toutes les 5 secondes.
- Ici, `<%= new java.util.Date() %>` montre la date et l'heure actuelle pour vérifier l'actualisation.

2. Via JavaScript On peut aussi utiliser JavaScript pour plus de flexibilité (ex. rafraîchissement conditionnel ou dynamique).

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html>
<head>
<title>Actualisation automatique</title>
<script>
// Rechargement toutes les 5 secondes
setTimeout(function(){
location.reload();
}, 5000);
</script>
</head>
<body>
<h1>Heure actuelle : <%= new java.util.Date() %></h1>
</body>
</html>
```

Avantages par rapport à la balise meta :

- Plus flexible : on peut conditionner le rafraîchissement.
- On peut stopper le rechargement ou le modifier dynamiquement.

3. Via l'en-tête HTTP JSP permet d'envoyer un en-tête HTTP pour demander un rafraîchissement automatique.

```
<%@ page language="java" contentType="text/html; charset=UTF-8" pageEncoding="UTF-8"%>
<%
// Rafraîchissement toutes les 5 secondes
response.setHeader("Refresh", "5");
%>
```

```
<!DOCTYPE html>
<html>
<head>
<title>Actualisation automatique</title>
</head>
<body>
<h1>Heure actuelle : <%= new java.util.Date() %></h1>
</body>
</html>
```

Remarque :

- Cette méthode fonctionne côté serveur.
- Elle est équivalente à la balise `<meta http-equiv="refresh">`, mais plus contrôlable dans le code JSP.

Points importants

1. **Performance** : rafraîchir trop souvent peut surcharger le serveur.
2. **Expérience utilisateur** : une actualisation constante peut être dérangeante pour l'utilisateur.
3. **Alternative moderne** : pour des mises à jour fréquentes de contenu, on préfère **AJAX** ou **WebSocket**, qui ne recharge pas toute la page, mais seulement les parties nécessaires.

Gestion des dates en JSP La gestion des dates en JSP peut se faire de plusieurs manières : en utilisant **Java standard** ou l'**Expression Language (EL)**.

1. Utilisation de `java.util.Date` et `java.text.SimpleDateFormat` C'est la méthode classique pour obtenir et formater une date en Java.

```
<%@ page import="java.util.Date, java.text.SimpleDateFormat" %>
<html>
<head>
<title>Exemple de date en JSP</title>
</head>
<body>
<%
Date dateActuelle = new Date();
SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy HH:mm:ss");
String dateFormatee = sdf.format(dateActuelle);
%>

<p>Date et heure actuelles : <%= dateFormatee %></p>
```

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

Explications :

- `Date dateActuelle = new Date();` crée un objet date avec la date et l'heure actuelles.
- `SimpleDateFormat` permet de définir le format d'affichage.
- `<%= dateFormatee %>` injecte la valeur dans la page HTML.

2. Utilisation de JSTL (`fmt:formatDate`) Avec JSTL, on peut formater les dates directement dans la JSP sans code Java.

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

<fmt:formatDate value="<%= new java.util.Date() %>"
pattern="dd/MM/yyyy HH:mm:ss" var="dateFormatee"/>
<p>Date et heure actuelles : ${dateFormatee}</p>

</body>
</html>
```

3. Avec Expression Language (EL) et JSTL Si la date est stockée dans un attribut request/session, EL permet de l'afficher facilement :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<%
request.setAttribute("maDate", new java.util.Date());
%>

<p>Date actuelle : <fmt:formatDate value="${maDate}" pattern="yyyy-MM-dd HH:mm:ss"/></p>
```

Envoi d'e-mails en JSP/Java EE Parlons en détail de l'envoi d'e-mails en JSP/Java EE. Comme JSP est une technologie côté serveur, pour envoyer des e-mails, on utilise JavaMail API, qui fait partie de l'écosystème Java EE. Je vais te détailler pas à pas.

1. Pré-requis

1. **Bibliothèques nécessaires** :
   - `javax.mail.jar` ou `jakarta.mail.jar` (selon la version de Java EE/Jakarta EE)
   - `activation.jar` (pour les pièces jointes et types MIME, si nécessaire)

2. **Serveur SMTP** : Tu auras besoin d'un serveur SMTP pour envoyer les e-mails. Par exemple :
   - Gmail SMTP : `smtp.gmail.com`, port 587 (TLS) ou 465 (SSL)
   - Serveur local comme `localhost:25` si tu testes en local
3. **JSP/Servlet Setup** : Idéalement, tu ne fais pas directement l'envoi dans le JSP mais dans une Servlet ou une classe utilitaire, et tu appelles cette classe depuis le JSP pour une meilleure séparation MVC.

Étapes pour télécharger et ajouter jakarta.mail.jar à un projet Java EE/JSP

1. **Aller sur le site officiel de Jakarta Mail :** Ouvre ton navigateur et va sur le site officiel de Jakarta EE pour Jakarta Mail : https://repo1.maven.org/maven2/jakarta/mail/jakarta.mail-api/2.1.3/?utm_source=chatgpt.com
2. **Accéder à la section Downloads :** Clique sur **Download** ou sur **Jakarta Mail Downloads**. Tu verras plusieurs options : versions stables, sources et jars compilés.
3. **Choisir la version stable :** Par exemple, la dernière version stable au moment est **2.x.x**. Télécharge le fichier `jakarta.mail-x.x.x.jar` (le JAR binaire compilé).
4. **Ajouter le JAR à ton projet Eclipse :**
   (a) Place le fichier `jakarta.mail-x.x.x.jar` dans un dossier de ton projet, par exemple `WebContent/WEB-INF/lib`.
   (b) Clique droit sur ton projet → **Properties** → **Java Build Path** → **Libraries** → **Add JARs** → sélectionne ton JAR.
5. **Optionnel : Utiliser Maven :** Si ton projet utilise Maven, ajoute cette dépendance dans le `pom.xml` :
   ```
   <dependency>
   <groupId>com.sun.mail</groupId>
   <artifactId>jakarta.mail</artifactId>
   <version>2.1.2</version>
   </dependency>
   ```
   Maven téléchargera automatiquement le JAR et le mettra dans ton classpath.
6. **Vérifier l'installation :** Crée un petit programme Java ou JSP qui importe `jakarta.mail.*`. Si Eclipse ne montre pas d'erreur, le JAR est correctement pris en compte.

2. Exemple simple avec JavaMail Voici un exemple de classe utilitaire pour envoyer un e-mail :

```
import java.util.Properties;
import javax.mail.*;
import javax.mail.internet.*;

public class EmailUtility {

public static void sendEmail(String to, String subject, String messageText) throws Messaging
```

```
// Configurer les propriétés SMTP
Properties props = new Properties();
props.put("mail.smtp.host", "smtp.gmail.com");
props.put("mail.smtp.port", "587");
props.put("mail.smtp.auth", "true");
props.put("mail.smtp.starttls.enable", "true");

// Authentification
Authenticator auth = new Authenticator() {
protected PasswordAuthentication getPasswordAuthentication() {
return new PasswordAuthentication("tonEmail@gmail.com", "tonMotDePasse");
}
};

// Créer la session
Session session = Session.getInstance(props, auth);

// Créer le message
Message message = new MimeMessage(session);
message.setFrom(new InternetAddress("tonEmail@gmail.com"));
message.setRecipients(Message.RecipientType.TO, InternetAddress.parse(to));
message.setSubject(subject);
message.setText(messageText);

// Envoyer l'e-mail
Transport.send(message);

System.out.println("E-mail envoyé avec succès !");
}
}
```

Remarque : Pour Gmail, il faut activer les applications moins sécurisées ou utiliser OAuth2 pour un envoi sécurisé.

3. Appeler cette classe depuis un JSP Même si on peut le faire directement dans JSP, la bonne pratique est de créer une Servlet, mais pour l'exemple simple :

```
<%@ page import="java.util.*,javax.mail.*,javax.mail.internet.*,yourpackage.EmailUtility" %>
<%
try {
EmailUtility.sendEmail("destinataire@example.com", "Test JSP Email", "Bonjour depuis JSP !")
out.println("E-mail envoyé !");
} catch(Exception e) {
```

```
out.println("Erreur : " + e.getMessage());
}
%>
```

4. Exemple complet JSP + Servlet + Formulaire HTML

Servlet pour traiter le formulaire :

```
import java.io.IOException;
import javax.servlet.ServletException;
import javax.servlet.annotation.WebServlet;
import javax.servlet.http.*;
import javax.mail.MessagingException;

@WebServlet("/SendEmailServlet")
public class SendEmailServlet extends HttpServlet {
protected void doPost(HttpServletRequest request, HttpServletResponse response)
throws ServletException, IOException {

String to = request.getParameter("to");
String subject = request.getParameter("subject");
String message = request.getParameter("message");

try {
EmailUtility.sendEmail(to, subject, message);
response.getWriter().println("E-mail envoyé avec succès !");
} catch (MessagingException e) {
response.getWriter().println("Erreur lors de l'envoi : " + e.getMessage());
}
}
}
```

Formulaire HTML (JSP) :

```
<!DOCTYPE html>
<html>
<head>
<meta charset="UTF-8">
<title>Envoyer un e-mail</title>
</head>
<body>
<h2>Formulaire d'envoi d'e-mail</h2>
<form action="SendEmailServlet" method="post">
<label>Destinataire : </label>
```

```
<input type="email" name="to" required><br><br>

<label>Sujet : </label>
<input type="text" name="subject" required><br><br>

<label>Message : </label><br>
<textarea name="message" rows="5" cols="30" required></textarea><br><br>

<input type="submit" value="Envoyer">
</form>
</body>
</html>
```

5. Points importants

- Ne jamais mettre le mot de passe en clair dans le code ; utiliser des variables d'environnement ou un fichier de configuration sécurisé.
- Séparer la logique d'envoi dans une classe utilitaire pour respecter le modèle MVC.
- Pour envoyer des pièces jointes, utiliser `MimeBodyPart` et `Multipart`.
- Activer le mode debug de JavaMail pour le diagnostic : `session.setDebug(true);`

Cet exemple est prêt à être testé sur un serveur Tomcat.

Compter des visiteurs en JSP Il est possible de compter les visiteurs d'une application JSP soit au niveau de l'application (total de visites), soit au niveau de la session (visiteurs uniques).

1. Compter le nombre total de visiteurs (application-wide) On stocke le compteur dans le `ServletContext` pour qu'il soit partagé par toutes les sessions et JSP.

```
<%@ page import="javax.servlet.*" %>
<%
// Récupérer le contexte de l'application
ServletContext application = getServletContext();

// Vérifier si le compteur existe déjà
Integer compteur = (Integer) application.getAttribute("visiteurCompteur");
if (compteur == null) {
compteur = 1; // premier visiteur
} else {
compteur = compteur + 1; // incrémenter le compteur
}

// Enregistrer le nouveau compteur dans le contexte
application.setAttribute("visiteurCompteur", compteur);
```

```
%>

<html>
<head>
<title>Compteur de visiteurs</title>
</head>
<body>
<h1>Bienvenue sur le site !</h1>
<p>Nombre total de visiteurs : <%= compteur %></p>
</body>
</html>
```

2. Compter les visiteurs uniques par session On utilise `HttpSession` pour compter un visiteur une seule fois par session.

```
<%
// Récupérer la session
HttpSession session = request.getSession();

// Vérifier si la session est nouvelle
Boolean visiteurExistant = (Boolean) session.getAttribute("visiteurExistant");

if (visiteurExistant == null) {
// Nouvelle session → incrémenter le compteur global
ServletContext application = getServletContext();
Integer compteur = (Integer) application.getAttribute("visiteurCompteur");
if (compteur == null) {
compteur = 1;
} else {
compteur = compteur + 1;
}
application.setAttribute("visiteurCompteur", compteur);

// Marquer la session comme déjà comptée
session.setAttribute("visiteurExistant", true);
}

// Récupérer le compteur pour affichage
Integer compteur = (Integer) getServletContext().getAttribute("visiteurCompteur");
%>

<html>
<head>
```

```
<title>Compteur de visiteurs uniques</title>
</head>
<body>
<h1>Bienvenue sur le site !</h1>
<p>Nombre total de visiteurs uniques : <%= compteur %></p>
</body>
</html>
```

Remarques

- `ServletContext` est partagé par toutes les sessions et JSP de l'application.
- `HttpSession` est unique par utilisateur pour une durée de session.
- Pour une application en production, il est conseillé de stocker le compteur dans une base de données afin de ne pas le perdre après un redémarrage du serveur.

Accès à une base de données PostgreSQL en JSP

Pré-requis :

- PostgreSQL installé et configuré.
- Base de données créée (exemple : `testdb`).
- Table créée :
  `users(id SERIAL PRIMARY KEY, name VARCHAR(50), email VARCHAR(50))`
- Pilote JDBC pour PostgreSQL (`postgresql-<version>.jar`) ajouté dans le dossier `WEB-INF/lib` du projet.

Connexion à PostgreSQL en JSP :

```
<%@ page import="java.sql.*" %>
<%
String url = "jdbc:postgresql://localhost:5432/testdb";
String user = "postgres";
String password = "12345";

Connection conn = null;
Statement stmt = null;
ResultSet rs = null;

try {
Class.forName("org.postgresql.Driver");
conn = DriverManager.getConnection(url, user, password);
stmt = conn.createStatement();
String sql = "SELECT * FROM users";
rs = stmt.executeQuery(sql);
```

```
out.println("<table border='1'>");
out.println("<tr><th>ID</th><th>Name</th><th>Email</th></tr>");
while(rs.next()){
int id = rs.getInt("id");
String name = rs.getString("name");
String email = rs.getString("email");
out.println("<tr><td>" + id + "</td><td>" + name + "</td><td>" + email + "</td></tr>");
}
out.println("</table>");
} catch(Exception e) {
out.println("Erreur : " + e.getMessage());
} finally {
try { if(rs != null) rs.close(); } catch(Exception e){}
try { if(stmt != null) stmt.close(); } catch(Exception e){}
try { if(conn != null) conn.close(); } catch(Exception e){}
}
%>
```

Insertion de données en JSP :

```
<%
String name = request.getParameter("name");
String email = request.getParameter("email");

try {
Class.forName("org.postgresql.Driver");
conn = DriverManager.getConnection(url, user, password);
String insertSQL = "INSERT INTO users(name, email) VALUES(?, ?)";
PreparedStatement ps = conn.prepareStatement(insertSQL);
ps.setString(1, name);
ps.setString(2, email);
int result = ps.executeUpdate();
if(result > 0) {
out.println("Utilisateur ajouté avec succès !");
}
ps.close();
} catch(Exception e) {
out.println("Erreur : " + e.getMessage());
} finally {
try { if(conn != null) conn.close(); } catch(Exception e){}
}
%>
```

Formulaire HTML pour l'insertion :

```
<form action="insertUser.jsp" method="post">
Nom : <input type="text" name="name"><br>
Email : <input type="text" name="email"><br>
<input type="submit" value="Ajouter">
</form>
```

Petit projet JSP complet avec PostgreSQL et MVC simplifié

Structure du projet :

```
webapp/

 WEB-INF/
    lib/                  # Bibliothèques JDBC
    web.xml               # Configuration web
    jsp/                  # JSP accessibles via le contrôleur
        listUsers.jsp
        addUser.jsp
        deleteUser.jsp

 index.jsp                 # Page d'accueil
src/
 com/example/controller/
 UserServlet.java     # Contrôleur
```

Contrôleur UserServlet.java :

```
package com.example.controller;

import java.io.*;
import java.sql.*;
import jakarta.servlet.*;
import jakarta.servlet.http.*;

public class UserServlet extends HttpServlet {
String url = "jdbc:postgresql://localhost:5432/testdb";
String user = "postgres";
String password = "12345";

protected void doGet(HttpServletRequest request, HttpServletResponse response) throws Servle
String action = request.getParameter("action");
request.setAttribute("users", getUsers());
```

```
if("delete".equals(action)) {
int id = Integer.parseInt(request.getParameter("id"));
deleteUser(id);
}
request.getRequestDispatcher("/WEB-INF/jsp/listUsers.jsp").forward(request, response);
}

protected void doPost(HttpServletRequest request, HttpServletResponse response) throws Servl
String name = request.getParameter("name");
String email = request.getParameter("email");
addUser(name, email);
response.sendRedirect("UserServlet");
}

private void addUser(String name, String email){
try(Connection conn = DriverManager.getConnection(url,user,password);
PreparedStatement ps = conn.prepareStatement("INSERT INTO users(name,email) VALUES(?,?)")) {
ps.setString(1,name);
ps.setString(2,email);
ps.executeUpdate();
} catch(Exception e) { e.printStackTrace(); }
}

private void deleteUser(int id){
try(Connection conn = DriverManager.getConnection(url,user,password);
PreparedStatement ps = conn.prepareStatement("DELETE FROM users WHERE id=?")) {
ps.setInt(1,id);
ps.executeUpdate();
} catch(Exception e) { e.printStackTrace(); }
}

private ResultSet getUsers(){
try{
Connection conn = DriverManager.getConnection(url,user,password);
Statement stmt = conn.createStatement();
return stmt.executeQuery("SELECT * FROM users");
} catch(Exception e) { e.printStackTrace(); return null; }
}
}
```

Liste des utilisateurs - listUsers.jsp :

```
<%@ page import="java.sql.*" %>
```

```
<%
ResultSet rs = (ResultSet) request.getAttribute("users");
%>
<h2>Liste des utilisateurs</h2>
<table border="1">
<tr><th>ID</th><th>Nom</th><th>Email</th><th>Action</th></tr>
<%
while(rs != null && rs.next()) {
int id = rs.getInt("id");
String name = rs.getString("name");
String email = rs.getString("email");
%>
<tr>
<td><%=id%></td>
<td><%=name%></td>
<td><%=email%></td>
<td><a href="UserServlet?action=delete&id=<%=id%>">Supprimer</a></td>
</tr>
<% } %>
</table>

<h3>Ajouter un utilisateur</h3>
<form action="UserServlet" method="post">
Nom: <input type="text" name="name"><br>
Email: <input type="text" name="email"><br>
<input type="submit" value="Ajouter">
</form>
```

Check-list de sécurité pour JSP

1. Entrées utilisateur

- Toujours valider les données côté **serveur**.
- Utiliser des expressions JSTL (`<c:out>`) ou des bibliothèques d'échappement pour éviter le **XSS**.
  `<c:out value="${param.name}" />`
- Ne jamais afficher directement une entrée utilisateur avec `<%= ... %>` sans échappement.

2. Base de données

- Utiliser **PreparedStatement** pour toutes les requêtes SQL.

```
PreparedStatement ps = conn.prepareStatement("SELECT * FROM users WHERE name=?");
ps.setString(1, request.getParameter("user"));
```

- Ne jamais concaténer les paramètres dans une requête SQL.

3. Sessions

- Régénérer l'ID de session après connexion (`session.invalidate()` puis `request.getSession(true)`).
- Configurer un **timeout de session** dans `web.xml`.

```
<session-config>
<session-timeout>20</session-timeout>
</session-config>
```

- Activer les cookies de session en mode **HttpOnly** et **Secure**.

4. Authentification & mots de passe

- Toujours utiliser **HTTPS**.
- Stocker les mots de passe avec un hachage sécurisé (ex : BCrypt).
- Ne jamais afficher les informations sensibles dans les JSP (ex : mot de passe en clair).

5. Protection contre CSRF

- Ajouter un **token CSRF** dans chaque formulaire sensible.

```
<input type="hidden" name="csrfToken" value="${sessionScope.csrfToken}">
```

- Vérifier le token côté serveur avant traitement.

6. Configuration et déploiement

- Ne jamais placer de JSP sensible hors de `/WEB-INF`.
- Désactiver le **directory listing** du serveur.
- Masquer les messages d'erreur (pas de stack trace en production).

7. Bonnes pratiques JSP

- Éviter le code Java direct (`<% ... %>`), préférer EL + JSTL.
- Limiter les inclusions dynamiques (`<jsp:include>`) et jamais avec des valeurs utilisateur.
- Mettre en place des filtres de sécurité (`jakarta.servlet.Filter`) pour centraliser la protection.

Gestion des exceptions en JSP En JSP (JavaServer Pages), la gestion des exceptions repose sur les mêmes mécanismes que ceux utilisés en Java classique, avec quelques particularités liées au contexte web.

1. Gestion des exceptions avec `try-catch-finally` Une JSP étant traduite en servlet Java, il est possible d'utiliser directement les blocs `try-catch-finally`.

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

<html>
<head><title>Gestion des exceptions</title></head>
<body>
<%
try {
int a = 10;
int b = 0;
int resultat = a / b; // Exception : division par zéro
out.println("Résultat : " + resultat);
} catch (ArithmeticException e) {
out.println("Erreur arithmétique : " + e.getMessage());
} finally {
out.println("<br>Bloc finally exécuté !");
}
%>
</body>
</html>
```

2. Utilisation de la directive `errorPage` On peut définir une page JSP dédiée à la gestion des erreurs à l'aide de la directive `page`.

Dans la JSP principale :

```
<%@ page errorPage="erreur.jsp" %>
<%
int a = 10 / 0; // Provoque une exception
%>
```

Dans la page d'erreur `erreur.jsp` :

```
<%@ page isErrorPage="true" %>

<html>
<head><title>Erreur détectée</title></head>
<body>
<h2>Une erreur est survenue :</h2>
<p><%= exception.getMessage() %></p>
<p>Type : <%= exception.getClass().getName() %></p>
</body>
</html>
```

Ici, l'objet implicite `exception` est accessible uniquement lorsque `isErrorPage="true"` est défini.

3. Gestion des erreurs au niveau du fichier `web.xml` Il est possible de configurer des pages d'erreur globales au niveau de l'application.

```
 <error-page> <exception-type>java.lang.ArithmeticException</exception-type> <location>/erre
```

```
<error-page>
<error-code>404</error-code>
<location>/notfound.jsp</location>
</error-page>
```

Ainsi :

- Toute exception de type `ArithmeticException` est redirigée vers `erreur.jsp`.
- Les erreurs HTTP 404 (page non trouvée) sont redirigées vers `notfound.jsp`.

4. Bonnes pratiques

- Limiter la logique Java dans les JSP, privilégier les Servlets et la JSTL.
- Centraliser la gestion des erreurs dans `web.xml` ou via les annotations.
- Éviter d'afficher les détails techniques des exceptions aux utilisateurs, mais les consigner dans les logs.

JavaBeans en JSP Un **JavaBean** en JSP est une classe Java réutilisable qui suit certaines conventions de codage, et qui est souvent utilisée pour **transporter des données** entre les couches d'une application (modèle MVC). Dans le contexte des JSP, les JavaBeans permettent de séparer la logique métier de la présentation.

Caractéristiques d'un JavaBean Un JavaBean doit respecter les règles suivantes :

- Posséder un **constructeur par défaut** (sans paramètre).
- Avoir des **attributs privés** (principe d'encapsulation).
- Fournir des **méthodes getters et setters** publics pour accéder et modifier les propriétés.
- Être **sérialisable** (implémenter l'interface `Serializable`, souvent facultatif si non distribué).

Exemple de JavaBean

```
package models;

import java.io.Serializable;

public class Etudiant implements Serializable {
private String nom;
private int age;

‘‘‘
```

```
// Constructeur par défaut
public Etudiant() {}

// Getter et Setter
public String getNom() {
return nom;
}
public void setNom(String nom) {
this.nom = nom;
}

public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
‘‘‘

}
```

Utilisation dans JSP Dans une page JSP, on manipule un bean grâce aux **actions standards JSP** :

1. **Déclaration d'un bean** avec `[jsp\:useBean](jsp:useBean)` :
   `\<jsp\:useBean id="et" class="models.Etudiant" scope="session" />`
2. **Affectation de propriétés** :
   `\<jsp\:setProperty name="et" property="nom" value="Ali" />`
   `\<jsp\:setProperty name="et" property="age" value="21" />`
3. **Récupération des propriétés** :
   `Nom : \<jsp\:getProperty name="et" property="nom" /><br>`
   `Âge : \<jsp\:getProperty name="et" property="age" />`

Portées disponibles (`scope`)

- **page** : accessible uniquement dans la page JSP courante.
- **request** : accessible pendant toute la requête HTTP.
- **session** : accessible pendant toute la session utilisateur.
- **application** : accessible par toute l'application (partagée).

L'internationalisation (i18n) en JSP L'internationalisation (i18n) est le mécanisme qui permet de développer une application capable de s'adapter à différentes langues et cultures sans modifier son code source. Dans les JSP, l'internationalisation repose sur l'utilisation des **fichiers**

de ressources (`ResourceBundle`) et des balises JSTL du `fmt` (Formatting Tag Library).

Principe

- On définit des **fichiers `.properties`** qui contiennent les traductions de textes dans différentes langues.
- On configure une **locale** (langue + pays).
- La JSP charge dynamiquement la ressource appropriée en fonction de la locale.

Exemple de fichiers de ressources

```
messages_en.properties
welcome=Welcome
bye=Goodbye

messages_fr.properties
welcome=Bienvenue
bye=Au revoir
```

Utilisation dans JSP avec JSTL

1. Importer la librairie JSTL :
   `<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>`
2. Définir la locale :
   `<fmt:setLocale value="fr_FR" />`
3. Charger le bundle de ressources :
   `<fmt:setBundle basename="messages" />`
4. Afficher un message traduit :
   `<fmt:message key="welcome" /><br>`
   `<fmt:message key="bye" />`

Résultat attendu Si la locale est `fr_FR`, la JSP affichera :

```
Bienvenue
Au revoir
```

Si la locale est `en_US`, la JSP affichera :

```
Welcome
Goodbye
```

Remarque Il est possible de détecter automatiquement la langue du navigateur avec :

```
<fmt:setLocale value="${pageContext.request.locale}" />
```

Mini-exemple complet avec un formulaire de sélection de langue (fr/en)

```
<%@ page contentType="text/html;charset=UTF-8" language="java" %>
<%@ taglib uri="http://java.sun.com/jsp/jstl/fmt" prefix="fmt" %>
<html>
<head>
<title>Internationalisation JSP</title>
</head>
<body>

<!-- Formulaire de sélection de langue -->
<form method="get" action="index.jsp">
<label>Choisissez la langue :</label>
<select name="lang">
<option value="fr_FR">Français</option>
<option value="en_US">English</option>
</select>
<input type="submit" value="Changer">
</form>

<!-- Définir la locale en fonction du choix utilisateur -->
<fmt:setLocale value="${param.lang}" />
<fmt:setBundle basename="messages" />

<!-- Affichage des messages traduits -->
<h2><fmt:message key="welcome" /></h2>
<p><fmt:message key="bye" /></p>

</body>
</html>
```

2.2.13 Expression Language (EL)

Définition. L'**Expression Language (EL)** est un langage intégré aux JSP qui simplifie l'accès aux objets, attributs et propriétés sans avoir recours aux scriptlets (`<% ... %>`). La syntaxe repose sur `${ ... }` et rend les pages JSP plus lisibles et maintenables.

Accès aux portées. EL permet d'accéder aux attributs stockés dans les différentes portées :

- **pageScope** : attributs de la portée page.
- **requestScope** : attributs de la requête.
- **sessionScope** : attributs de la session.
- **applicationScope** : attributs de l'application.

Exemple :

```
<p>Nom : ${requestScope.nom}</p>
<p>Utilisateur : ${sessionScope.utilisateur}</p>
```

Si aucune portée n'est précisée, EL recherche dans l'ordre : *page* → *request* → *session* → *application*.

Accès aux propriétés de JavaBeans. Si un objet JavaBean est disponible, EL permet d'accéder directement à ses propriétés.

```
<jsp:useBean id="personne" class="com.exemple.Personne" scope="session" />
<jsp:setProperty name="personne" property="nom" value="Alice" />
<jsp:setProperty name="personne" property="age" value="25" />

<p>Nom : ${personne.nom}</p>
<p>Âge : ${personne.age}</p>
```

Objets implicites EL. EL met à disposition plusieurs objets implicites facilitant l'accès aux données.

Objet EL	Description
`param`	Accède aux paramètres de requête (chaîne unique).
`paramValues`	Accède aux paramètres multiples (tableau).
`header`	Accède aux en-têtes HTTP.
`cookie`	Accède aux cookies.
`initParam`	Accède aux paramètres d'initialisation du contexte.

Exemple :

```
<p>Paramètre "id" : ${param.id}</p>
<p>User-Agent : ${header["User-Agent"]}</p>
<p>Cookie JSESSIONID : ${cookie.JSESSIONID.value}</p>
```

Opérateurs en EL.

- **Arithmétiques** : `+`, `-`, `*`, `/`, `\%`.
- **Logiques** : `&&`, `|`, `!` (ou `and`, `or`, `not`).
- **Comparaison** : `==`, `!=`, `<`, `>`, `<=`, `>=` (ou `eq`, `ne`, `lt`, `gt`, `le`, `ge`).
- **Test de vide / null** : `empty`.

Exemple :

```
<p>Somme : ${10 + 5}</p>
<p>Est majeur ? ${personne.age >= 18}</p>
<p>Nom vide ? ${empty personne.nom}</p>
```

Exemple complet (HTML5 + EL).

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html lang="fr">
<head>
<meta charset="UTF-8">
<title>Exemple EL</title>
</head>
<body>
<h1>Exemple Expression Language (EL)</h1>

<%-- Définition d'attributs --%>
<%
request.setAttribute("nom", "Zey");
session.setAttribute("utilisateur", "Alice");
%>

<p>Nom (requestScope) : ${requestScope.nom}</p>
<p>Utilisateur (sessionScope) : ${sessionScope.utilisateur}</p>

<%-- Utilisation d'un JavaBean --%>
<jsp:useBean id="personne" class="com.exemple.Personne" scope="request" />
<jsp:setProperty name="personne" property="nom" value="Bob" />
<jsp:setProperty name="personne" property="age" value="30" />

<p>Nom (Bean) : ${personne.nom}</p>
<p>Âge (Bean) : ${personne.age}</p>

<%-- Paramètres de requête --%>
<p>Paramètre "id" : ${param.id}</p>

<%-- Opérateurs --%>
<p>Âge + 5 = ${personne.age + 5}</p>
<p>Est-il majeur ? ${personne.age ge 18}</p>
<p>Nom vide ? ${empty personne.nom}</p>

</body>
</html>
```

2.2.14 JavaBean

Définition. Un **JavaBean** est une classe Java simple (*POJO – Plain Old Java Object*) qui suit des conventions précises. Il est utilisé dans les applications JSP pour représenter et manipuler

des données, et peut être combiné avec les balises JSP (`<jsp:useBean>`, `<jsp:setProperty>`, `<jsp:getProperty>`) ou avec EL.

Conventions d'un JavaBean. Un JavaBean doit respecter les règles suivantes :

- La classe doit être **publique**.
- Elle doit posséder un **constructeur public sans argument**.
- Les **attributs** doivent être déclarés **privés**.
- Chaque propriété doit avoir des **méthodes d'accès (getters)** et de **modification (setters)** publiques.

Exemple de JavaBean.

```
package com.exemple;

public class Personne {
private String nom;
private int age;

// Constructeur sans argument
public Personne() {}

// Getter et Setter pour nom
public String getNom() {
return nom;
}
public void setNom(String nom) {
this.nom = nom;
}

// Getter et Setter pour age
public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
}
```

Utilisation dans une JSP.

```
<jsp:useBean id="personne" class="com.exemple.Personne" scope="session" />
<jsp:setProperty name="personne" property="nom" value="Alice" />
<jsp:setProperty name="personne" property="age" value="25" />
```

```
<p>Nom : <jsp:getProperty name="personne" property="nom" /></p>
<p>Âge : <jsp:getProperty name="personne" property="age" /></p>
```

Avec EL, l'affichage est plus simple :

```
<p>Nom : ${personne.nom}</p>
<p>Âge : ${personne.age}</p>
```

Portées possibles d'un JavaBean. Lorsqu'on utilise `<jsp:useBean>`, il est possible de préciser la portée de l'objet.

Portée	Description
`page`	Objet lié uniquement à la page courante (par défaut).
`request`	Objet lié à une requête HTTP (partagé entre JSP et Servlet via l'objet `request`).
`session`	Objet lié à la session utilisateur (partagé sur plusieurs requêtes).
`application`	Objet lié au contexte global de l'application (partagé entre tous les utilisateurs).

Exemple complet (HTML5 + JSP + JavaBean).

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html lang="fr">
<head>
<meta charset="UTF-8">
<title>Exemple JavaBean</title>
</head>
<body>
<h1>Utilisation d'un JavaBean</h1>

<!-- Création ou récupération du bean -->
<jsp:useBean id="personne" class="com.exemple.Personne" scope="session" />

<!-- Initialisation des propriétés -->
<jsp:setProperty name="personne" property="nom" value="Zey" />
<jsp:setProperty name="personne" property="age" value="20" />

<!-- Affichage avec EL -->
<p>Nom : ${personne.nom}</p>
<p>Âge : ${personne.age}</p>

</body>
```

```
</html>
```

2.2.15 JavaBean

Définition. Un **JavaBean** est une classe Java simple (*POJO – Plain Old Java Object*) qui suit des conventions précises. Il est utilisé dans les applications JSP pour représenter et manipuler des données, et peut être combiné avec les balises JSP (`<jsp:useBean>`, `<jsp:setProperty>`, `<jsp:getProperty>`) ou avec EL.

Conventions d'un JavaBean. Un JavaBean doit respecter les règles suivantes :

- La classe doit être **publique**.
- Elle doit posséder un **constructeur public sans argument**.
- Les **attributs** doivent être déclarés **privés**.
- Chaque propriété doit avoir des **méthodes d'accès (getters)** et de **modification (setters)** publiques.

Exemple de JavaBean.

```
package com.exemple;

public class Personne {
private String nom;
private int age;

// Constructeur sans argument
public Personne() {}

// Getter et Setter pour nom
public String getNom() {
return nom;
}
public void setNom(String nom) {
this.nom = nom;
}

// Getter et Setter pour age
public int getAge() {
return age;
}
public void setAge(int age) {
this.age = age;
}
}
```

Utilisation dans une JSP.

```
<jsp:useBean id="personne" class="com.exemple.Personne" scope="session" />
<jsp:setProperty name="personne" property="nom" value="Alice" />
<jsp:setProperty name="personne" property="age" value="25" />

<p>Nom : <jsp:getProperty name="personne" property="nom" /></p>
<p>Âge : <jsp:getProperty name="personne" property="age" /></p>
```

Avec EL, l'affichage est plus simple :

```
<p>Nom : ${personne.nom}</p>
<p>Âge : ${personne.age}</p>
```

Portées possibles d'un JavaBean. Lorsqu'on utilise `<jsp:useBean>`, il est possible de préciser la portée de l'objet.

Portée	Description
`page`	Objet lié uniquement à la page courante (par défaut).
`request`	Objet lié à une requête HTTP (partagé entre JSP et Servlet via l'objet `request`).
`session`	Objet lié à la session utilisateur (partagé sur plusieurs requêtes).
`application`	Objet lié au contexte global de l'application (partagé entre tous les utilisateurs).

Exemple complet (HTML5 + JSP + JavaBean).

```
<%@ page contentType="text/html; charset=UTF-8" language="java" %>
<!DOCTYPE html>
<html lang="fr">
<head>
<meta charset="UTF-8">
<title>Exemple JavaBean</title>
</head>
<body>
<h1>Utilisation d'un JavaBean</h1>

<!-- Création ou récupération du bean -->
<jsp:useBean id="personne" class="com.exemple.Personne" scope="session" />

<!-- Initialisation des propriétés -->
<jsp:setProperty name="personne" property="nom" value="Zey" />
<jsp:setProperty name="personne" property="age" value="20" />
```

```
<!-- Affichage avec EL -->
<p>Nom : ${personne.nom}</p>
<p>Âge : ${personne.age}</p>

</body>
</html>
```

2.3 TP : Mise en œuvre du modèle MVC avec Java EE

2.3.1 Objectifs pédagogiques

- Comprendre et mettre en place le modèle **MVC (Model–View–Controller)**.
- Manipuler les **Servlets** comme contrôleurs.
- Créer et utiliser des **JavaBeans** comme modèles.
- Gérer les **JSP** comme vues.
- Utiliser correctement le répertoire **WEB-INF** pour sécuriser les vues.

2.3.2 Sujet

Vous allez développer une petite application web de gestion des étudiants. Cette application doit permettre :

1. D'accéder à une **page d'accueil** (`index.jsp`).
2. D'afficher un **formulaire d'ajout d'étudiant**.
3. De saisir **nom et prénom** puis d'enregistrer l'étudiant.
4. D'afficher la **liste des étudiants saisis**.

2.3.3 Étapes du TP

Étape 1 : Création du projet

- Créez un projet **Dynamic Web Project** dans Eclipse (ou IntelliJ) appelé `MVCApp`.
- Structurez le projet comme suit :

```
MVCApp

 src/
    model/          → classes métiers (JavaBeans)
    controller/     → servlets (contrôleurs)

 WebContent/
    index.jsp       → accessible directement
    WEB-INF/
          web.xml   → descripteur de déploiement
          views/    → pages JSP internes
                ajoutEtudiant.jsp
```

`listeEtudiants.jsp`

Étape 2 : Création du Model (JavaBean)

- Créez un package `model`.
- Ajoutez une classe `Etudiant` avec deux attributs `nom` et `prenom`.
- Fournissez :
  - un constructeur vide,
  - un constructeur avec paramètres,
  - des **getters** et **setters**.

Étape 3 : Création du Controller (Servlet)

- Créez un package `controller`.
- Ajoutez une Servlet `EtudiantServlet` qui jouera le rôle de contrôleur.
- Fonctionnalités attendues :
  1. **En GET** :
     - si `action=form` → afficher le formulaire `ajoutEtudiant.jsp` (via `RequestDispatcher`),
     - sinon → afficher la liste des étudiants.
  2. **En POST** :
     - récupérer les données envoyées par le formulaire (`nom`, `prenom`),
     - créer un objet `Etudiant`,
     - l'ajouter à une liste en mémoire,
     - renvoyer vers `listeEtudiants.jsp`.

Étape 4 : Création des Vues (JSP dans WEB-INF)

1. **ajoutEtudiant.jsp** : Contient un formulaire HTML avec deux champs `nom`, `prenom`. Envoi des données en POST vers la Servlet.
2. **listeEtudiants.jsp** : Affiche la liste des étudiants sous forme de liste `<ul>`. Les données seront récupérées depuis l'attribut de requête passé par la Servlet. Propose un lien permettant d'ajouter un autre étudiant.

Étape 5 : La page d'accueil

- Créez un fichier `index.jsp` accessible directement.
- Cette page doit contenir :
  - un message de bienvenue,
  - un lien permettant d'aller au formulaire via l'URL : `EtudiantServlet?action=form`

Étape 6 : Configuration du déploiement

- Dans le fichier `WEB-INF/web.xml`, déclarez la Servlet si nécessaire (même si l'annotation `@WebServlet` est utilisée).
- Vérifiez que le projet est bien déployé sur **Tomcat** (ou tout autre serveur).

Étape 7 : Test de l'application

1. Lancez le serveur Tomcat.
2. Ouvrez l'URL :
   `http://localhost:8080/MVCApp/index.jsp`
3. Vérifiez les enchaînements :
   - accueil → formulaire → ajout → liste.

2.3.4 Résultat attendu

- L'utilisateur n'accède qu'à `index.jsp`.
- Les pages internes (`ajoutEtudiant.jsp` et `listeEtudiants.jsp`) ne sont pas directement accessibles par URL car elles sont dans `WEB-INF`.
- Toute la navigation passe par le **contrôleur (Servlet)**.

2.3.5 Travail à rendre

1. Capture d'écran de la **page d'accueil**.
2. Capture d'écran du **formulaire rempli**.
3. Capture d'écran de la **liste affichée après ajout**.
4. Le projet complet compressé (`MVCApp.zip`).

3 Partie 3: Introduction à JSTL (JavaServer Pages Standard Tag Library)

Dans une application **Java EE**, les **JSP (JavaServer Pages)** permettent de mélanger du code Java avec du code HTML afin de générer des pages dynamiques. Cependant, écrire directement du code Java dans une JSP à l'aide de *scriptlets* (`<% ... %>`) pose des problèmes de lisibilité, de maintenance et de séparation des responsabilités.

Pour résoudre ce problème, **JSTL (JavaServer Pages Standard Tag Library)** a été introduit. C'est une **bibliothèque de balises standardisées** qui permet :

- de remplacer la logique Java dans les JSP par des **balises XML-like**,
- de rendre les pages plus **claires, maintenables et lisibles**,
- de favoriser une meilleure séparation entre la **logique de présentation** (vue) et la **logique métier** (servlets/beans).

3.1 Objectifs de JSTL

- Réduire l'utilisation des scriptlets dans les JSP.

- Standardiser des tâches courantes (boucles, conditions, manipulation de chaînes, formats, accès aux données, etc.).
- Faciliter le développement internationalisé (i18n).
- Offrir une solution portable et réutilisable entre serveurs d'applications Java EE/Jakarta EE.

3.2 Principaux groupes de balises JSTL

JSTL est divisé en plusieurs bibliothèques, chacune spécialisée :

1. **Core Tags** (`c:`) : pour la logique de base (conditions, boucles, variables, etc.).
   - Exemple : `<c:if>`, `<c:forEach>`, `<c:set>`, `<c:choose>`.
2. **Formatting Tags** (`fmt:`) : pour l'internationalisation et le formatage (dates, nombres, messages).
   - Exemple : `<fmt:formatNumber>`, `<fmt:formatDate>`.
3. **SQL Tags** (`sql:`) : pour exécuter des requêtes SQL directement dans la JSP (peu recommandé en pratique car cela viole MVC).
4. **XML Tags** (`x:`) : pour manipuler et transformer des documents XML (XPath, XSLT).
5. **Functions Tags** (`fn:`) : pour utiliser des fonctions utilitaires (longueur de chaînes, recherche, etc.).

3.3 Résumé

JSTL est une boîte à outils standard pour JSP, qui permet de remplacer du code Java par des balises propres, expressives et maintenables.

3.4 Différentes versions de JSTL

JSTL a évolué avec les versions de **Java EE** et de **Jakarta EE**.

1. **JSTL 1.0 (2002)**
   - Sortie avec J2EE 1.3.
   - Compatible avec JSP 1.2.
   - Anciennes bibliothèques séparées en plusieurs JAR (`jstl.jar`, `standard.jar`).
2. **JSTL 1.1 (2003)**
   - Sortie avec J2EE 1.4.
   - Compatible avec JSP 2.0.
   - Toujours avec les deux JARs (`jstl.jar` et `standard.jar`).
3. **JSTL 1.2 (2006)**
   - Inclus avec Java EE 5.
   - Compatible avec JSP 2.1.
   - Fusionné en un seul JAR (`javax.servlet.jsp.jstl-1.2.x.jar`).
   - Très largement utilisé, encore courant aujourd'hui.

4. **Jakarta JSTL 2.0+ (2020 et après)**
    - Migration vers Jakarta EE.
    - Les packages changent de `javax.*` à `jakarta.*`.
    - Dépendances disponibles via Maven Central sous `jakarta.servlet.jsp.jstl`.

Remarque : pour un projet académique avec **Tomcat** ou **GlassFish**, la version la plus utilisée reste **JSTL 1.2**.

3.5 Téléchargement de JSTL

3.5.1 Depuis Maven (recommandé)

Si l'on utilise Maven, il suffit d'ajouter la dépendance suivante dans le fichier `pom.xml` :

```
<dependency>
<groupId>javax.servlet</groupId>
<artifactId>jstl</artifactId>
<version>1.2</version>
</dependency>
```

3.5.2 Téléchargement et installation manuelle de JSTL dans Eclipse Java EE

1. Télécharger JSTL

- Aller sur le site **Maven Central** ou un dépôt de bibliothèques Java.
- Rechercher **JSTL 1.2** (version la plus courante pour Tomcat).
- Télécharger le fichier `jstl-1.2.jar`.
- Exemple de lien : https://mvnrepository.com/artifact/javax.servlet/jstl/1.2.
- Si tu es sur Tomcat 8 ou 9 (Java EE, JSTL 1.2)
- Télécharge jstl-1.2.jar depuis Maven Central.
- https://mvnrepository.com/artifact/javax.servlet/jstl/1.2
- Mets le JAR dans WEB-INF/lib.
- Utilise l'URI classique :
  `<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>`

- Si tu es sur Tomcat 10+ (Jakarta EE)
- Télécharge jakarta.servlet.jsp.jstl-3.0.x.jar et jakarta.servlet.jsp.jstl-api-3.0.x.jar.
- https://mvnrepository.com/artifact/org.glassfish.web/jakarta.servlet.jsp.jstl
- Mets-les dans WEB-INF/lib.
- Utilise les nouveaux URI Jakarta :
- <%@ taglib uri="jakarta.tags.core" prefix="c" %>

Remarque Dans Jakarta EE (comme auparavant avec Java EE), les librairies sont généralement séparées en deux parties :

* **jakarta.servlet.jsp.jstl-api-3.0.x.jar** : ce fichier contient uniquement les *interfaces (API)*,

c'est-à-dire la définition des classes, méthodes et balises JSTL (par exemple `\<c\:if>`, `\<c\:forEach>`, etc.). Il ne fournit aucune implémentation réelle. Il sert principalement à écrire et compiler le code JSP qui utilise JSTL, mais il ne permet pas d'exécuter les balises au runtime.

* **jakarta.servlet.jsp.jstl-3.0.x.jar** : ce fichier contient l'*implémentation effective* des API définies ci-dessus. C'est lui que le serveur d'applications (Tomcat, Payara, WildFly, etc.) utilise pour exécuter les balises JSTL. Sans ce fichier, les classes seraient bien déclarées par l'API, mais il n'existerait rien pour les exécuter, ce qui provoquerait des erreurs (`NoClassDefFoundError`, `Missing implementation`).

Ainsi, il est nécessaire de télécharger et d'utiliser les deux fichiers :

- L'API (`-api`) décrit les fonctionnalités.
- L'implémentation (`sans -api`) réalise concrètement ces fonctionnalités.

Remarque importante :

- Sur certains serveurs complets (comme GlassFish, Payara, ou WildFly), JSTL est déjà intégré, donc il n'est pas nécessaire d'ajouter ces fichiers manuellement.
- Sur Tomcat, en revanche, il faut absolument placer ces deux `.jar` dans le dossier `WEB-INF/lib/` du projet, sans quoi JSTL ne sera pas reconnu.

2. Ajouter JSTL dans le projet

1. Copier le fichier `jstl-1.2.jar` téléchargé.
2. Dans le projet Eclipse, ouvrir le dossier :
   `Webapp/WEB-INF/lib`
3. Coller le fichier `.jar` dans ce dossier.

Remarque : Eclipse ajoute automatiquement ce JAR dans le *classpath* via **Web App Libraries**.

3. Vérifier dans Eclipse

- Clic droit sur le projet → **Properties** → **Java Build Path** → **Libraries**.
- Vérifier que `jstl-1.2.jar` apparaît sous **Web App Libraries**.

4. Utiliser JSTL dans une JSP Exemple simple d'utilisation de JSTL dans une page JSP :

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>
<html>
<head><title>Exemple JSTL</title></head>
<body>
<c:set var="nom" value="Alice" />
<c:if test="${not empty nom}">
Bonjour, ${nom} !
</c:if>
```

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

Conclusion de l'installation Après ces étapes, le projet est prêt à utiliser **JSTL** dans Eclipse avec Tomcat (ou un autre conteneur web).

3.6 Résumé

Une fois la configuration terminée, il est possible d'utiliser directement les balises JSTL (`<c:forEach>`, `<c:if>`, etc.) dans les pages JSP.