REPUBLIQUE DU SENEGAL

Un Peuple _ Un But _ Une Foi

Ministère de L'Enseignement Supérieur de la Recherche et de L'Innovation

UNIVERSITE ALIOUNE DIOP DE BAMBEY

L'excellence ma constance, L'éthique ma vertu

UFR : Sciences Appliquées et Technologie s de L'Information et de la Communication (SATIC)

DEPARTEMENT : MATHEMATIQUES

SPECIALITE : Statistique et Informatique Décisionnelle (SID)

NIVEAU : Master 1 (M1)

Projet : **WebSocket**(avec JSP/Servlet)

Membres :

Pape Ngor Tine

Adama Sarr

Mouhamadou Ndiaye

Astou Wade

Assime Diakhaté

Ousmane Ndiaye

Professeur chargé du cours :

Mr. Maodo DIOP

Année Académique 2024-2025

Table des matières

Table des figures

I. Introduction :

Dans un monde de plus en plus connecté, la communication en temps réel est devenue une nécessité pour de nombreuses applications modernes, qu'il s'agisse de messagerie instantanée, de notifications en direct ou de collaboration en temps réel. Pour répondre à ce besoin, les **WebSockets** offrent une solution efficace en permettant une communication bidirectionnelle et instantanée entre le client et le serveur, sans avoir à rafraîchir la page ou à effectuer des requêtes HTTP répétées.

Dans ce projet, nous avons développé une **application de chat en temps réel** en utilisant les technologies **WebSocket**, **JSP (JavaServer Pages)** et **Servlet**, déployée sur **Apache Tomcat 10.1.36**. Cette application permet aux utilisateurs de se connecter, de discuter en temps réel et de se déconnecter, tout en exploitant la puissance de **Jakarta EE** (anciennement Java EE) pour gérer les connexions et les échanges de messages.

L'objectif de ce projet est de démontrer comment les WebSockets peuvent être intégrés dans une application web traditionnelle basée sur JSP et Servlet, tout en offrant une expérience utilisateur fluide et réactive. À travers cette présentation, nous allons explorer l'architecture du projet.

II. Technologies Utilisées :

- **WebSocket** : Protocole de communication full-duplex pour des échanges en temps réel.
- **JSP (JavaServer Pages)** : Pour créer des pages web dynamiques.
- **Servlet** : Pour gérer les requêtes HTTP et les sessions utilisateur.
- **Jakarta EE** : Pour les API WebSocket et Servlet.
- **Apache Tomcat** : Serveur d'application pour déployer l'application.

III. Structure du projet :

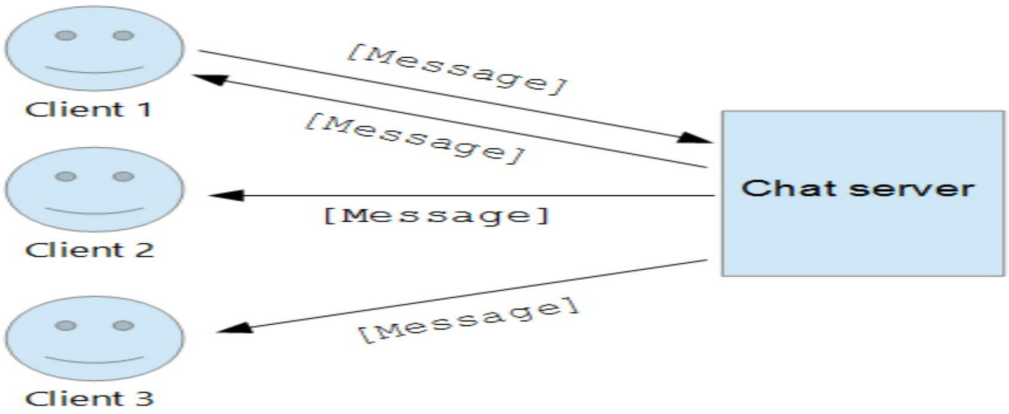

Figure 1 : Architecture du projet

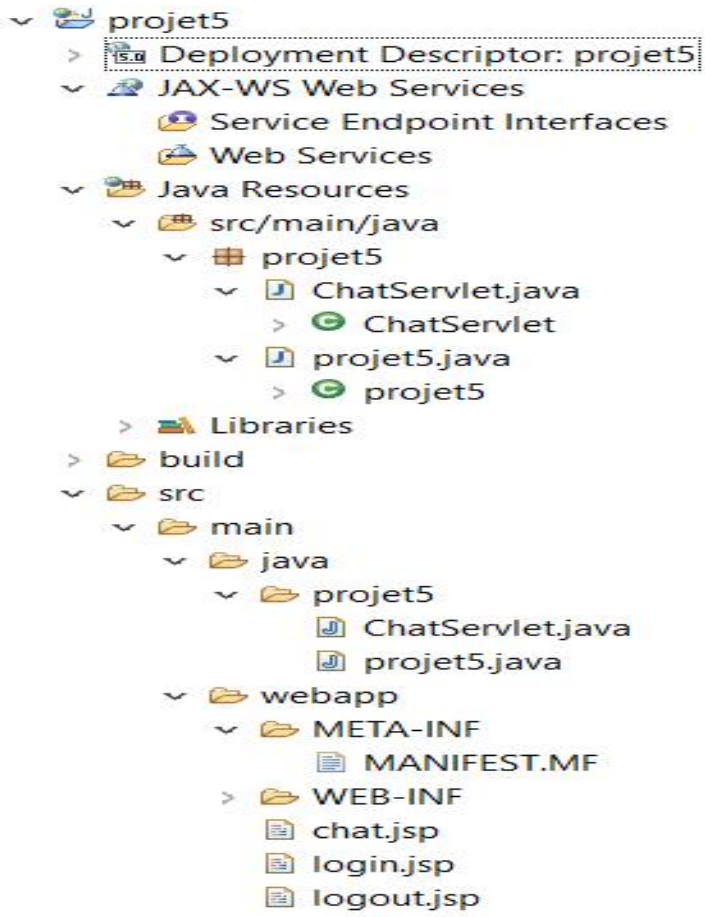

Figure 2 : Structure du Projet

1. Deployment Descriptor

Il est le descripteur de déploiement pour les applications web Java. Il configure les servlets, les filtres, les listeners, etc.

2. JAX-WS Web Services

Service Endpoint Interfaces : Définit les interfaces pour les services web.

Web Services : Contient les implémentations des services web.

3. Java Resources

src/main/java/projet5/ChatServlet.java : Une servlet pour gérer les fonctionnalités de chat.

src/main/java/projet5/projet5.java : Le point d'entrée principal de l'application (peut-être une classe utilitaire ou de configuration).

4. webapp

META-INF/MANIFEST.MF : Fichier de métadonnées pour l'application.

WEB-INF/chat.jsp : Page JSP pour l'interface de chat.

WEB-INF/login.jsp : Page JSP pour la connexion des utilisateurs.

WEB-INF/logout.jsp : Page JSP pour la déconnexion des utilisateurs.

IV. Implémentation :

1. Serveur WebSocket

Le serveur WebSocket est implémenté avec l'annotation @ServerEndpoint.

```
package projet5;


import java.util.Collections;

import java.util.Set;

import java.util.concurrent.ConcurrentHashMap;


import jakarta.websocket.OnClose;

import jakarta.websocket.OnMessage;

import jakarta.websocket.OnOpen;

import jakarta.websocket.Session;

import jakarta.websocket.server.ServerEndpoint;


@ServerEndpoint("/chat")

public class ChatServlet {


   private static Set<Session> userSessions = Collections.newSetFromMap(new
ConcurrentHashMap<Session, Boolean>());


   @OnOpen
```

```
    public void onOpen(Session curSession) {
        userSessions.add(curSession);
        System.out.println("Nouvelle connexion : " + curSession.getId());
    }

    @OnClose
    public void onClose(Session curSession) {
        userSessions.remove(curSession);
        System.out.println("Connexion fermée : " + curSession.getId());
    }

    @OnMessage
    public void onMessage(String message, Session userSession) {
        System.out.println("Message reçu de " + userSession.getId() + " : " + message);
        for (Session ses : userSessions) {
            ses.getAsyncRemote().sendText(message);
        }
    }
}
```

@ServerEndpoint("/chat") : Définit l'URL du point de terminaison WebSocket. Les clients se connectent à cette URL (par exemple, ws://localhost:8080/projet5/chat).

@OnOpen : Méthode appelée lorsqu'un client se connecte. La session du client est stockée dans un ensemble (Set) pour une gestion ultérieure.

@OnMessage : Méthode appelée lorsqu'un message est reçu d'un client. Le message est diffusé à tous les clients connectés.

@OnClose : Méthode appelée lorsqu'un client se déconnecte. La session est supprimée de l'ensemble.

@OnError : Méthode appelée en cas d'erreur lors de la communication WebSocket.

2. Interface Utilisateur (JSP)

La page chat.jsp permet aux utilisateurs de voir les messages et d'en envoyer de nouveaux.

```
<%@page contentType="text/html" pageEncoding="UTF-8"%>
<!DOCTYPE html>
<html lang="fr">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title>Chat en Temps Réel</title>
    <style>
        body {
            font-family: Arial, sans-serif;
            margin: 0;
            padding: 0;
            display: flex;
            flex-direction: column;
            align-items: center;
            justify-content: center;
            height: 100vh;
            background-color: #f4f4f9;
        }
        h1 {
            color: #333;
        }
        .chat-container {
            width: 400px;
            height: 500px;
            border: 1px solid #ccc;
            background-color: #fff;
```

```
    display: flex;
    flex-direction: column;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
}
#chat {
    flex: 1;
    padding: 10px;
    overflow-y: auto;
    border-bottom: 1px solid #ccc;
}
#msg {
    width: calc(100% - 20px);
    padding: 10px;
    border: none;
    border-top: 1px solid #ccc;
    outline: none;
}
button {
    padding: 10px;
    background-color: #28a745;
    color: white;
    border: none;
    cursor: pointer;
}
button:hover {
    background-color: #218838;
}
.logout-button {
    background-color: #dc3545;
```

```
        }
        .logout-button:hover {
            background-color: #c82333;
        }
    </style>
</head>
<body>
    <h1>Chat en Temps Réel - Bienvenue, ${sessionScope.username}</h1>
    <div class="chat-container">
        <div id="chat"></div>
        <input type="text" id="msg" placeholder="Tapez votre message ici..." />
        <button onclick="sendMsg()">Envoyer</button>
        <button class="logout-button" onclick="window.location.href='logout.jsp'">Déconnexion</button>
    </div>

    <script type="text/javascript">
        var wsUrl = (window.location.protocol === 'http:') ? 'ws://' : 'wss://';
        var ws = new WebSocket(wsUrl + window.location.host + "/projet5/chat");

        ws.onopen = function(event) {
            console.log("Connexion WebSocket établie.");
        };

        ws.onmessage = function(event) {
            console.log("Message reçu : ", event.data);
            var chatDiv = document.getElementById("chat");
            chatDiv.innerHTML += "<p>" + event.data + "</p>";
            chatDiv.scrollTop = chatDiv.scrollHeight;
        };
```

```
    ws.onerror = function(event) {
        console.error("Erreur WebSocket : ", event);
    };

    ws.onclose = function(event) {
        console.log("Connexion WebSocket fermée.");
    };

    function sendMsg() {
        var msgInput = document.getElementById("msg");
        var message = msgInput.value.trim();
        if (message) {
            console.log("Envoi du message : ", message);
            ws.send(message);
            msgInput.value = "";
        }
    }

    document.getElementById("msg").addEventListener("keyup", function(event) {
        if (event.key === "Enter") {
            sendMsg();
        }
    });
  </script>
</body>
</html>
```

a. En-tête de la Page JSP

contentType="text/html" : Indique que la page génère du contenu HTML.

pageEncoding="UTF-8" : Définit l'encodage des caractères en UTF-8.

b. Structure HTML

<meta charset="UTF-8"> : Assure que les caractères spéciaux sont correctement affichés.

<meta name="viewport"> : Rend la page responsive pour les appareils mobiles.

<title> : Titre de la page affiché dans l'onglet du navigateur.

c. Styles CSS

Les styles CSS sont définis dans la balise <style> pour :

Centrer le contenu de la page.

Donner une apparence moderne au conteneur de chat.

Styler les boutons et les zones de texte.

d. Contenu de la Page

${sessionScope.username} : Affiche le nom d'utilisateur stocké dans la session.

 : Zone où les messages du chat sont affichés.

<input id="msg"> : Champ de saisie pour écrire un message.

Boutons :

Envoyer : Envoie le message via WebSocket.

Déconnexion : Redirige l'utilisateur vers logout.jsp.

e. Script JavaScript

Le script JavaScript gère la connexion WebSocket et les interactions avec le serveur.

wsUrl : Détermine le protocole WebSocket (ws:// pour HTTP, wss:// pour HTTPS).

new WebSocket(...) : Établit une connexion WebSocket avec le serveur.

onopen : Appelé lorsque la connexion WebSocket est établie.

onmessage : Appelé lorsqu'un message est reçu du serveur. Le message est ajouté à la zone de chat.

onerror : Appelé en cas d'erreur WebSocket.

onclose : Appelé lorsque la connexion WebSocket est fermée.

3. Gestion des Sessions (Servlet)

Le ChatServlet gère les requêtes HTTP pour afficher la page de chat.

```
package projet5;

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

@WebServlet("/projet5")
public class projet5 extends HttpServlet {
   private static final long serialVersionUID = 1L;

   public projet5() {
      super();
   }

   protected void doGet(HttpServletRequest request, HttpServletResponse response) throws
ServletException, IOException {
      String action = request.getParameter("action");

      if ("logout".equals(action)) {
         HttpSession session = request.getSession(false);
         if (session != null) {
            session.invalidate();
```

```
        }
        response.sendRedirect("login.jsp");
    } else {
        response.sendRedirect("login.jsp");
    }
}

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

    if ("logout".equals(action)) {
        HttpSession session = request.getSession(false);
        if (session != null) {
            session.invalidate();
        }
        response.sendRedirect("login.jsp");
    } else {
        // Simuler une authentification simple
        if ("admin".equals(username) && "admin".equals(password)) {
            HttpSession session = request.getSession();
            session.setAttribute("username", username);
            response.sendRedirect("chat.jsp");
        } else {
            response.sendRedirect("login.jsp");
        }
    }
}
```

}

a. Annotations et Déclarations

@WebServlet("/projet5") : Définit l'URL de la Servlet (/projet5).

HttpServlet : La classe étend HttpServlet pour gérer les requêtes HTTP.

serialVersionUID : Identifiant de version pour la sérialisation.

b. Méthode doGet

Fonctionnalité : Gère les requêtes HTTP GET.

Paramètre action : Détermine l'action à effectuer.

Si action est "logout", la session de l'utilisateur est invalidée (déconnexion), et l'utilisateur est redirigé vers login.jsp.

Sinon, l'utilisateur est redirigé vers login.jsp.

c. Méthode doPost

Fonctionnalité : Gère les requêtes HTTP POST.

Paramètres :

username : Nom d'utilisateur saisi.

password : Mot de passe saisi.

action : Action à effectuer.

Logique :

Si action est "logout", la session est invalidée, et l'utilisateur est redirigé vers login.jsp.

Sinon, une authentification simple est simulée :

Si le nom d'utilisateur et le mot de passe sont "admin", une session est créée, et l'utilisateur est redirigé vers chat.jsp. Sinon, l'utilisateur est redirigé vers login.jsp.

V. Démonstration :

1. Connexion

L'utilisateur accède à login.jsp et saisit ses identifiants, le formulaire est envoyé à la servlet projet5.java pour vérification et si les identifiants sont valides, l'utilisateur est redirigé vers chat.jsp.

Voici le code et la sortie de la page connexion :

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

```
<!DOCTYPE html>
<html lang="fr">
<head>
  <meta charset="UTF-8">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <title>Connexion</title>
  <style>
    body {
      font-family: Arial, sans-serif;
      margin: 0;
      padding: 0;
      display: flex;
      flex-direction: column;
      align-items: center;
      justify-content: center;
      height: 100vh;
      background-color: #f4f4f9;
    }
    h1 {
      color: #333;
    }
    .login-container {
      width: 300px;
      padding: 20px;
      border: 1px solid #ccc;
      background-color: #fff;
      box-shadow: 0 4px 8px rgba(0, 0, 0, 0.1);
    }
    input[type="text"], input[type="password"] {
```

```
            width: 100%;
            padding: 10px;
            margin: 10px 0;
            border: 1px solid #ccc;
            outline: none;
        }
        button {
            width: 100%;
            padding: 10px;
            background-color: #28a745;
            color: white;
            border: none;
            cursor: pointer;
        }
        button:hover {
            background-color: #218838;
        }
    </style>
</head>
<body>
    <h1>Connexion</h1>
    <div class="login-container">
        <form action="projet5" method="post">
            <input type="text" name="username" placeholder="Nom d'utilisateur" required>
            <input type="password" name="password" placeholder="Mot de passe" required>
            <button type="submit">Se connecter</button>
        </form>
    </div>
</body>
```

</html>

Figure 3 : Interface de Connexion pour l'application de Chat en Temps Réel

a. Affichage de la Page :

L'utilisateur accède à la page login.jsp.

La page affiche un formulaire de connexion avec des champs pour le nom d'utilisateur et le mot de passe.

b. Soumission du Formulaire :

L'utilisateur saisit ses identifiants et clique sur "Se connecter".

Le formulaire est soumis via une requête POST à l'URL projet5 (la Servlet).

c. Traitement par la Servlet :

La Servlet projet5 reçoit les données du formulaire.

Elle vérifie les identifiants (dans cet exemple, admin/admin).

Si les identifiants sont valides, une session est créée, et l'utilisateur est redirigé vers chat.jsp.

Sinon, l'utilisateur est redirigé vers login.jsp.

2. Envoi de Messages et Affichage en Temps Réel

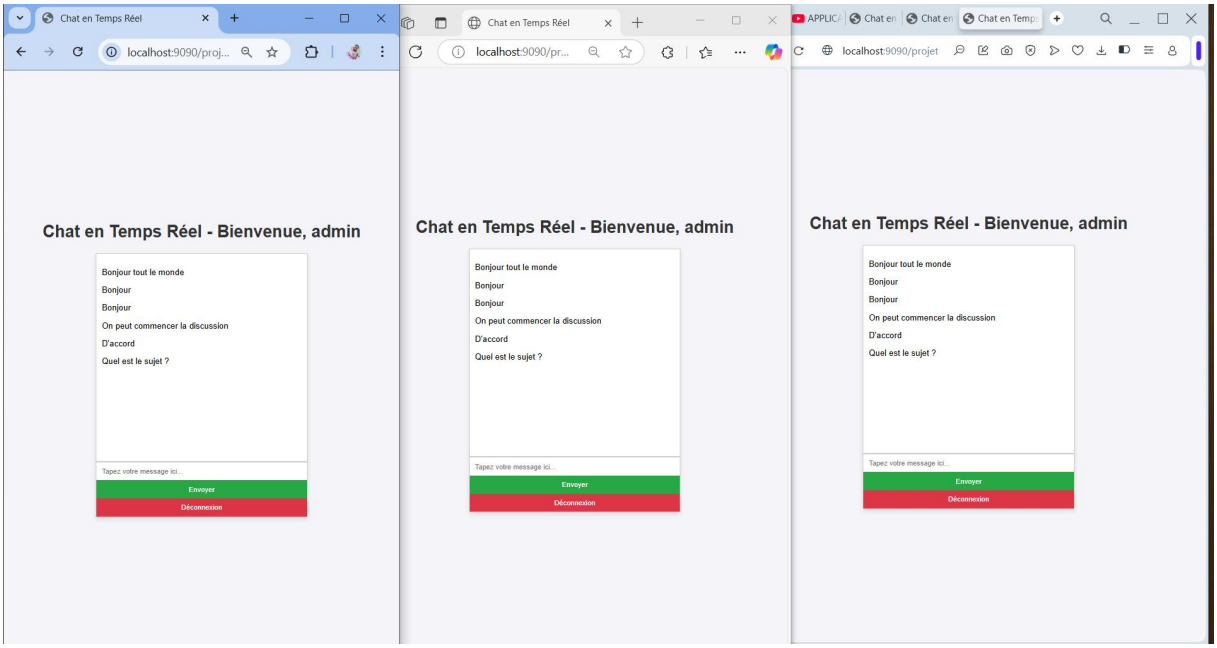

Figure 4 : Interface de Chat en Temps Réel avec Messages Échangés

a. Envoi de Messages :

L'envoi de messages dans l'application de chat en temps réel utilisant WebSocket se déroule en plusieurs étapes clés :

Saisie du Message :

L'utilisateur saisit un message dans un champ de texte sur l'interface utilisateur (<input>).

Transmission au Serveur :

Le message est envoyé au serveur via la connexion WebSocket établie entre le client (navigateur) et le serveur.

Le client utilise JavaScript pour envoyer le message avec la méthode send() de l'objet WebSocket.

Réception par le Serveur :

Le serveur WebSocket reçoit le message via la méthode annotée @OnMessage.

Le serveur traite le message (le diffuser à tous les clients connectés).

Diffusion aux Clients :

Le serveur envoie le message à tous les clients connectés en utilisant leurs sessions WebSocket.

b. Réception de Messages :

La réception de messages dans une application de chat avec WebSocket est un processus efficace et réactif, permettant une communication instantanée et une mise à jour dynamique de l'interface

3. Lecture des Messages :

Le client (navigateur) lit les messages entrants via l'événement onmessage de l'objet WebSocket.

Réception des Messages :

Lorsque le serveur envoie un message, le client le reçoit via la fonction onmessage.

Le message est ensuite extrait de l'objet event.data.

Affichage des Messages :

Le message reçu est ajouté dynamiquement à la zone de chat (par exemple, une avec l'ID chat).

Chaque message est généralement affiché sous forme de paragraphe (<p>).

Mise à Jour en Temps Réel :

Grâce à WebSocket, les messages sont affichés instantanément sans avoir à recharger la page.

La zone de chat est automatiquement mise à jour pour montrer les messages les plus récents.

VI. Avantages et Limites :

4. Avantages :

a. Communication en Temps Réel :

Instantanéité : Les messages sont envoyés et reçus en temps réel, sans délai.

Bidirectionnelle : WebSocket permet une communication full-duplex, où le client et le serveur peuvent envoyer et recevoir des données simultanément.

Fluidité : Pas de latence due à l'établissement de nouvelles connexions, rendant l'expérience utilisateur plus fluide.

b. Simplicité :

Intégration Facile : Utilisation de technologies standard : JSP, Servlet, et WebSocket, bien intégrées dans l'écosystème Java EE.

Code Minimal : Réduction de la complexité du code par rapport aux méthodes traditionnelles comme les requêtes AJAX.

Documentation et Support : Technologies largement documentées et supportées, facilitant la résolution des problèmes et l'apprentissage.

c. Scalabilité :

Gestion des Sessions : WebSocket permet de gérer plusieurs sessions utilisateur simultanément.

Support de Milliers de Connexions : Conçu pour gérer un grand nombre de connexions simultanées, essentiel pour les applications à grande échelle.

Efficacité : Une seule connexion est utilisée par utilisateur, réduisant la charge sur le serveur.

Extensions Possibles : Facilement extensible pour inclure des fonctionnalités supplémentaires comme les notifications, l'historique des messages, et la gestion des utilisateurs.

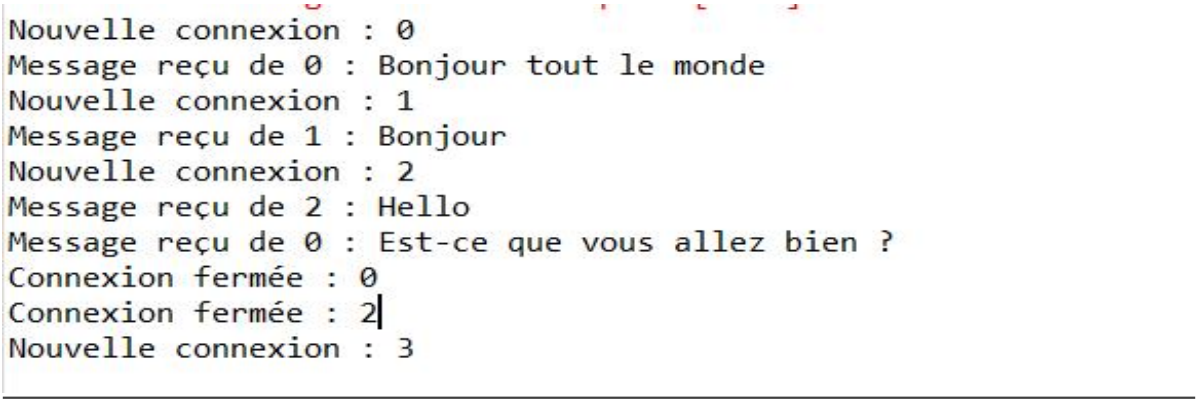

Figure 5 : Journal des Connexions et Messages dans une Application de Chat en Temps Réel

Ces avantages en font un choix idéal pour les applications nécessitant des échanges en temps réel, comme les chats, les jeux en ligne, ou les tableaux de bord interactifs.

5. Limites

a. Compatibilité :

Navigateurs Supportés : WebSocket nécessite des navigateurs modernes (Chrome, Firefox, Safari, Edge). Les anciens navigateurs comme Internet Explorer 10 et versions antérieures ne sont pas supportés.

Solutions de Rechange : Pour assurer une compatibilité maximale, il faut implémenter des alternatives comme les requêtes HTTP longues (long polling) ou Server-Sent Events (SSE).

b. Scalabilité Avancée :

Limites du Serveur Unique : Un serveur unique peut gérer un nombre limité de connexions WebSocket simultanées. Pour les applications à grande échelle, des solutions supplémentaires sont nécessaires.

Solutions : Utilisation de Redis pour stocker les sessions et les messages, de clusters pour répartir la charge, et de message brokers comme RabbitMQ ou Kafka pour gérer les messages entre serveurs.

Ces limites peuvent être surmontées avec une planification et une architecture appropriées, mais elles ajoutent de la complexité et des coûts au projet. La compatibilité et la scalabilité avancée sont des défis importants, mais des solutions existent pour les relever et garantir une expérience utilisateur fluide et performante.

Conclusion :

Ce projet illustre comment les technologies **WebSocket**, **JSP**, et **Servlet** peuvent être combinées pour créer une application de chat en temps réel performante et interactive. Bien qu'il présente certaines limites, notamment en termes de compatibilité et de scalabilité, ces défis peuvent être surmontés avec des solutions appropriées. Ce projet ouvre la voie à des applications plus complexes et évolutives, tout en offrant une base solide pour explorer davantage les possibilités des WebSockets et des technologies Jakarta EE.