Chatting App
IstiChat

Team members: Jad Abdallah

Jerome Chaker

Toufic Al Mabsout

Cynthia Ibrahim

Table of Contents

Table of Figures

Abstract

This document describes the architecture and features of a comprehensive chatting application designed to facilitate real-time communication between users. The application includes essential functionalities such as user registration, login, and a dynamic chat interface, all while leveraging robust technologies to enhance user experience and message management.

The application begins with a streamlined sign-up and login system that ensures secure access to the platform. Users can register with unique credentials, including a username and password, which are securely stored in a database. This database forms the core of the authentication system, allowing users to access their accounts and maintain secure sessions.

Once logged in, users are presented with a user-friendly chatting interface. This interface allows users to select from a list of other users and initiate conversations. The application supports real-time messaging, enabling users to send and receive messages instantly. Each conversation is managed through a unique RabbitMQ queue, specifically created for every pair of users. This queuing system ensures that messages are routed efficiently and accurately between the intended recipients.

In addition to direct chat functionalities, the application integrates a SOAP-based web service. This service facilitates the ability for users to send messages to other users, providing additional flexibility and integration options. The SOAP interface allows external applications or systems to interact with the chatting platform, expanding its utility and accessibility.

The project is developed using a dynamic web project structure, ensuring scalability and maintainability. The application features a friendly interface, designed to enhance user engagement and ease of use. Its intuitive design supports a seamless user experience, making it accessible to both novice and experienced users alike.

Overall, this chatting application combines modern communication technology with a well-structured backend and a user-centric interface, creating a robust platform for real-time interaction and messaging.

Installation Guide

This project requires creating a Dynamic Web Project on java and adding the initial jars for activating spring framework. In addition to this, this project requires a Tomcat 7.0 server and activating the RabbitMQ server.

To activate the RabbitMQ server follows these steps:

- Install Erlang
- Install RabbitMQ after installing Erlang for it to work
- Open RabbitMQ command prompt
- Type rabbitmq-plugins.bat enable rabbitmq_management
- Open RabbitMQ Service – stop to stop the server
- Open RabbitMQ Service – start to start the server again
- Open localhost:15672 to access the administration page
- Login with default credentials (guest/guest)
- Go to the admin page and create a user where you can specify the privileges for that user
- Now this user can be used and recognized between other PCs since the default one only works on localhost

Project Structure

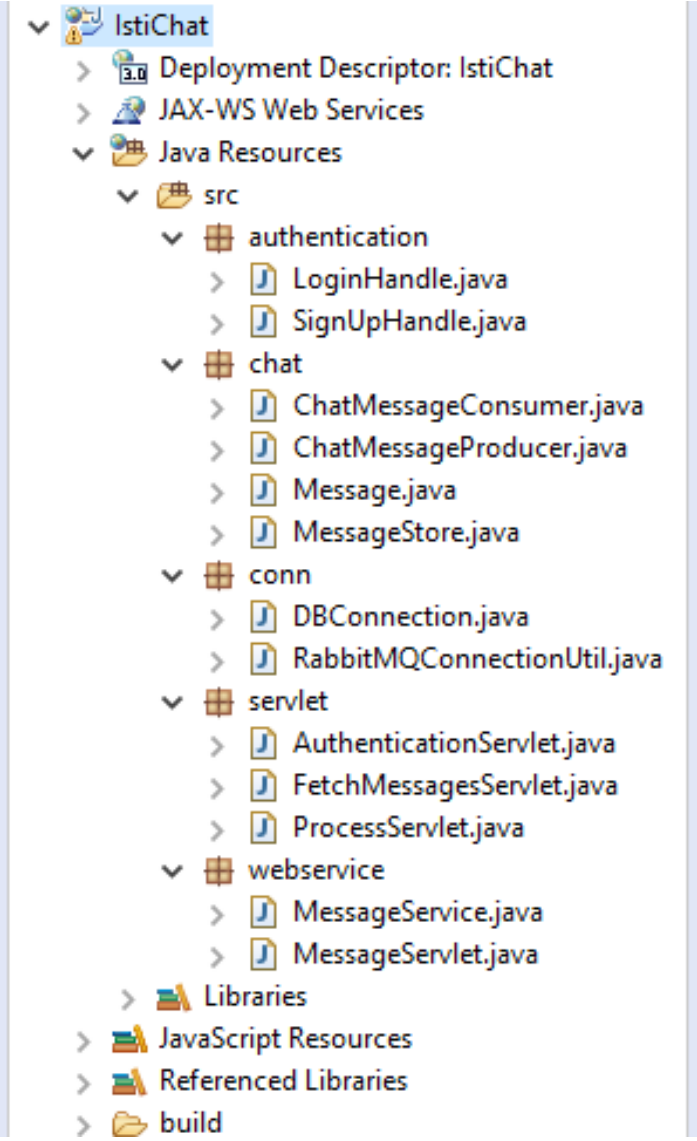

Figure 1 Project Structure Java files

The Java files in the project are organized into several packages, each serving a specific purpose. The conn package contains database connection utilities, facilitating interaction with the database. The servlet package includes servlets like AuthenticationServlet, FetchMessagesServlet, ProcessServlet, and MessageServlet, which handle various HTTP requests related to user authentication, message retrieval, and processing. The chat package manages chat-related logic with classes like Message and MessageStore, which handle message creation, storage, and retrieval. The webservice package provides a SOAP-based web service via MessageService for sending messages between users. Lastly, the authentication package includes LoginHandle and SignUpHandle for managing user authentication and registration processes. This structured approach ensures clear separation of concerns and efficient handling of different application functionalities.

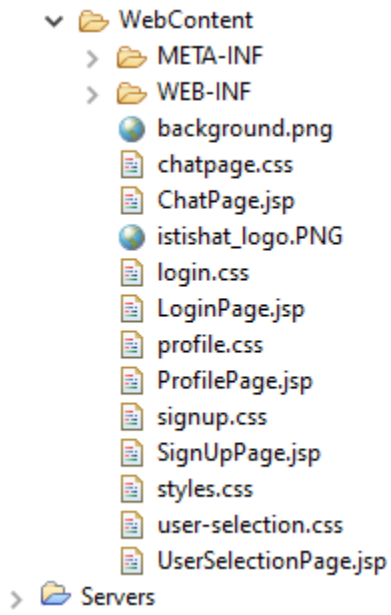

Figure 2 Project Structure jsp files

The JSP files in the project provide the user interface for various aspects of the web application. The LoginPage.jsp is used for user authentication, allowing users to log in with their credentials. The SignUpPage.jsp handles user registration by collecting new user details and validating them. Upon successful login or registration, users are redirected to UserSelectionPage.jsp, where they can select other users to chat with. The ChatPage.jsp serves as the main interface for real-time messaging, displaying chat histories and providing input fields for sending messages. The error handling pages, such as error.jsp, provide feedback and guidance if issues occur during login or sign-up processes.

Conn

DBConnection.java

```java
package conn;

import java.io.FileInputStream;

public class DBConnection {

    private static final String PROPERTIES_FILE_PATH = "C:\\Users\\User2\\Desktop\\config.properties";

    public static Connection getConnection() {
        Connection con = null;
        Properties properties = new Properties();

        try (FileInputStream fis = new FileInputStream(PROPERTIES_FILE_PATH)) {
            properties.load(fis);

            String uname = properties.getProperty("db.username");
            String pswrd = properties.getProperty("db.password");
            String url = properties.getProperty("db.url");

            Class.forName("oracle.jdbc.OracleDriver");
            con = DriverManager.getConnection(url, uname, pswrd);

        } catch (IOException | ClassNotFoundException | SQLException e) {
            e.printStackTrace();
        }

        return con;
    }
}
```

Figure 3 DBConnection.java

The DBConnection class is designed to manage database connections using JDBC in Java. It reads database connection details from a configuration file located at C:\\Users\\User2\\Desktop\\config.properties. This properties file should include three key entries: db.username for the database username, db.password for the password, and db.url for the JDBC connection URL. The class loads these properties, initializes the Oracle JDBC driver, and establishes a connection to the database using DriverManager.getConnection. If any issues arise during file reading, driver loading, or connection creation, they are caught and printed as error messages. The resulting Connection object allows interaction with the database.

RabbitMQConnectionUtil.java

```java
package conn;
import com.rabbitmq.client.Connection;
import com.rabbitmq.client.ConnectionFactory;
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Properties;

public class RabbitMQConnectionUtil {
    private static final String PROPERTIES_FILE_PATH = "C:\\Users\\User2\\Desktop\\config.properties"; // Default path

    public static Connection getConnection() throws Exception {
        Properties properties = new Properties();

        try (FileInputStream fis = new FileInputStream(PROPERTIES_FILE_PATH)) {
            properties.load(fis);

            String host = properties.getProperty("rabbitmq.host");
            String username = properties.getProperty("rabbitmq.username");
            String password = properties.getProperty("rabbitmq.password");

            ConnectionFactory factory = new ConnectionFactory();
            factory.setHost(host);
            factory.setUsername(username);
            factory.setPassword(password);
            return factory.newConnection();

        } catch (IOException e) {
            e.printStackTrace();
            throw new RuntimeException("Failed to load RabbitMQ properties file", e);
        }
    }
}
```

Figure 4 RabbitMQConnectionUtil.java

The RabbitMQConnectionUtil class is used to create a connection to a RabbitMQ message broker. It retrieves connection details from a configuration file located at C:\\Users\\User2\\Desktop\\config.properties. This file must include the properties rabbitmq.host for the RabbitMQ server address, rabbitmq.username for the username, and rabbitmq.password for the password. The class loads these properties, sets up a ConnectionFactory with the retrieved details, and establishes a connection to RabbitMQ using factory.newConnection(). If there are any issues loading the properties file, an error is printed and a runtime exception is thrown.

Authentication

LoginHandle.java

```java
package authentication;

import conn.DBConnection;


public class LoginHandle {


        // Method to check user credentials
    public boolean authenticateUser(String username, String password) {
            String query = "SELECT username FROM jad_users WHERE username = ? AND password = ?";
            //users_jad

            try (Connection connection = DBConnection.getConnection();
                 PreparedStatement statement = connection.prepareStatement(query)) {

                statement.setString(1, username);
                statement.setString(2, password);

                try (ResultSet resultSet = statement.executeQuery()) {
                    return resultSet.next(); // If a row is found, login is successful
                }

            } catch (SQLException e) {
                e.printStackTrace();
                // Handle database connection errors
            }

            return false; // Default to login failure
        }


}
```

Figure 5 LoginHandle.java

The LoginHandle class in the authentication package is designed to manage user authentication within a Java application. It contains a method called authenticateUser that verifies if the provided username and password match any record in the jad_users table of the database. The method establishes a connection using the DBConnection class, executes a SQL query with the given credentials, and returns true if a record is found, indicating successful login. If no matching record is found or if a database error occurs, the method returns false, indicating a failed login attempt. Any SQL exceptions encountered are printed for debugging purposes.

SignUpHandle.java

```
import conn.DBConnection;

public class SignUpHandle {

    // Method to register a new user
    public boolean registerUser(String username, String password) {
}
```

Figure 6 SignUpHandle.java

The SignUpHandle class in the authentication package is responsible for user registration within a Java application. It includes the registerUser method, which adds a new user to the jad_users table in the database. The method first retrieves the current maximum user ID from the table to determine the next available ID. It then inserts the new user's details, including the generated ID, username, and password, into the table. The method returns true if the insertion is successful, otherwise it returns false. The class uses the DBConnection class to manage database connections and prints stack traces for any SQL exceptions encountered during the process. Note that it is advisable to hash the password before storing it for improved security.

Chat

ChatMessageConsumer.java

```java
package chat;

import com.rabbitmq.client.Channel;

public class ChatMessageConsumer {
    private static final String EXCHANGE_NAME = "EXCHANGE_TWO";

    public void startConsuming(int user1Id, int user2Id) {
        String queueName = getQueueName(user1Id, user2Id);

        try (Channel channel = RabbitMQConnectionUtil.getConnection().createChannel()) {
            channel.exchangeDeclare(EXCHANGE_NAME, "direct", true);
            channel.queueDeclare(queueName, true, false, false, null);
            channel.queueBind(queueName, EXCHANGE_NAME, queueName);

            DeliverCallback deliverCallback = (consumerTag, delivery) -> {
                String message = new String(delivery.getBody(), "UTF-8");
                System.out.println("Received message: " + message);
                // Store the message in MessageStore
                MessageStore.addMessage(user1Id, user2Id, message);
            };

            channel.basicConsume(queueName, true, deliverCallback, consumerTag -> {
                System.out.println("Consumer cancelled: " + consumerTag);
            });
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private String getQueueName(int user1Id, int user2Id) {
}
```

Figure 7 ChatMessageConsumer.java

The ChatMessageConsumer class is designed to handle real-time chat message consumption from a RabbitMQ messaging broker. It defines an EXCHANGE_NAME for message routing and establishes a connection with RabbitMQ to consume messages. The startConsuming method initializes a channel, declares an exchange of type "direct", and creates a durable queue specific to the chat participants (identified by user1Id and user2Id). The queue is bound to the exchange using the queue's name to ensure messages are correctly routed. A DeliverCallback is set up to process incoming messages by printing them to the console and storing them in a MessageStore for further use. The queue name is dynamically generated to maintain consistency in ordering and avoid duplicates for the same user pair. If any exceptions occur during setup or message consumption, they are caught and logged.

ChatMessageProducer.java

```java
package chat;

import com.rabbitmq.client.Channel;

public class ChatMessageProducer {
    private static final String EXCHANGE_NAME = "EXCHANGE_TWO";

    public void sendMessage(int user1Id, int user2Id, String message) {
        String queueName = getQueueName(user1Id, user2Id);

        try (Channel channel = RabbitMQConnectionUtil.getConnection().createChannel()) {
            channel.exchangeDeclare(EXCHANGE_NAME, "direct", true);
            channel.queueDeclare(queueName, true, false, false, null); // Durable queue
            channel.queueBind(queueName, EXCHANGE_NAME, queueName); // Routing key same as queue name

            channel.basicPublish(EXCHANGE_NAME, queueName, MessageProperties.PERSISTENT_TEXT_PLAIN, message.getBytes());
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    private String getQueueName(int user1Id, int user2Id) {
        // Ensure consistent ordering to avoid having two queues for the same pair in reverse order
        if (user1Id > user2Id) {
            return "chatQueue_" + user1Id + "_" + user2Id;
        } else {
            return "chatQueue_" + user2Id + "_" + user1Id;
        }
    }
}
```

Figure 8 ChatMessageProducer.java

The ChatMessageProducer class is responsible for sending chat messages to a RabbitMQ messaging broker. It uses an EXCHANGE_NAME to route messages and handles the setup and publishing process. The sendMessage method creates a channel, declares an exchange of type "direct", and sets up a durable queue specific to the chat participants (identified by user1Id and user2Id). The queue is bound to the exchange with a routing key that matches the queue name to ensure accurate message delivery. The message is published with persistent properties to guarantee it is not lost even if the broker crashes. If any exceptions are encountered during the setup or publishing process, they are logged for debugging purposes. The getQueueName method ensures a consistent queue naming convention by ordering user IDs to avoid creating duplicate queues for the same pair in reverse order.

Message.java

```java
package chat;

public class Message {
    private final int senderId;
    private final String content;

    public Message(int senderId, String content) {
        this.senderId = senderId;
        this.content = content;

    }

    public int getSenderId() {
        return senderId;
    }

    public String getContent() {
        return content;
    }

}
```

Figure 9 Message.java

The Message class represents a chat message in the system. It encapsulates two primary attributes: senderId, an integer that identifies the sender of the message, and content, a string containing the message text. The class provides a constructor to initialize these attributes and includes getter methods (getSenderId and getContent) to retrieve their values. This design ensures that messages are immutable once created, providing a straightforward and secure way to manage message data within the chat application.

MessageStore.java

```java
package chat;

import java.util.ArrayList;

public class MessageStore {
    private static final Map<String, List<Message>> userMessages = new ConcurrentHashMap<>();

    public static synchronized void addMessage(int senderId, int receiverId, String messageContent) {
        String key = getConversationKey(senderId, receiverId);
        Message message = new Message(senderId, messageContent);
        userMessages.computeIfAbsent(key, k -> new ArrayList<>()).add(message);
    }

    public static synchronized List<Message> getMessagesForUser(int user1Id, int user2Id) {
        String key = getConversationKey(user1Id, user2Id);
        return userMessages.getOrDefault(key, new ArrayList<>());
    }

    private static String getConversationKey(int user1Id, int user2Id) {
        if (user1Id > user2Id) {
            return user1Id + "_" + user2Id;
        } else {
            return user2Id + "_" + user1Id;
        }
    }
}

```

Figure 10 MessageStore.java

The MessageStore class provides a thread-safe mechanism for managing chat messages between users using a ConcurrentHashMap. It maps conversation keys, which are consistently ordered based on user IDs, to lists of Message objects. The addMessage method stores messages by creating a Message object and adding it to the appropriate list for the conversation. The getMessagesForUser method retrieves all messages for a given pair of user IDs, returning an empty list if no messages are found. The class ensures consistent conversation key management to avoid duplicate entries and supports efficient message retrieval and storage.

Servlet

AuthenticationServlet.java

```
1  package servlet;
2
3⊕ import javax.servlet.ServletException;
19
20 @WebServlet("/authentication")
21 public class AuthenticationServlet extends HttpServlet {
22
23⊖     /**
24      *
25      */
26     private static final long serialVersionUID = 1L;
27     private LoginHandle loginHandle = new LoginHandle();
28     private SignUpHandle signupHandle = new SignUpHandle();
29
31⊕    protected void doPost(HttpServletRequest request, HttpServletResponse response)
45
46⊕    private void handleLogin(HttpServletRequest request, HttpServletResponse response)
70
71
72⊕    private void handleSignUp(HttpServletRequest request, HttpServletResponse response)
03
04⊕    public static ArrayList<String> fetchAllUsers() {
22
23⊕    public static ArrayList<Integer> fetchAllUsersExceptSender() {
41
42⊕    public static int retrieveId(String username) {
66
67⊕    public static String retrieveName(int id) {
91
92 }
93
```

Figure 11 AuthenticationServlet.java

The AuthenticationServlet class handles user authentication and registration in a web application. It processes HTTP POST requests to either log in or sign up users based on the "action" parameter. During login, it authenticates the user, retrieves their ID, and forwards them to a user selection page if successful, or redirects to the login page with an error if not. For sign-up, it validates the password, checks if the username already exists, and either creates a new user and forwards them to the user selection page or redirects back to the sign-up page with an error. The servlet also includes static methods for fetching all users, retrieving user IDs by username, and getting usernames by ID from the database. These methods interact with the database using SQL queries and handle potential SQL exceptions.

FetchMessageServlet.java

```
1
2  package servlet;
3
4⊕ import chat.Message;
15
16 @WebServlet("/FetchMessagesServlet")
17 public class FetchMessagesServlet extends HttpServlet {
18
19     private static final long serialVersionUID = 1L;
20
22⊕    protected void doGet(HttpServletRequest request, HttpServletResponse response)
48
49⊕    private String formatMessage(String text) {
62
63⊕    private String escapeHtml(String text) {
73
74⊕    private String convertUrlsToLinks(String text) {
06 }
07
```

Figure 12 FetchMessageServlet.java

The FetchMessagesServlet class is responsible for retrieving and displaying chat messages between users in a web application. Upon receiving a GET request, it first verifies that the user is logged in by checking the session for a username. It then retrieves the sender's ID from the session and the receiver's ID from the request parameters. If the receiver's ID is missing or empty, it responds with a bad request error. Messages are fetched from the `MessageStore` for the specified user pair and formatted for HTML display. Each message is wrapped in a with a class indicating whether it was sent by the current user or received. The formatMessage method ensures that message content is HTML-escaped, newlines are converted to
 tags, and URLs are turned into clickable links. This approach provides a secure and user-friendly way to display chat messages in a web interface.

ProcessServlet.java

```java
package servlet;

import javax.servlet.ServletException;

@WebServlet("/process")
public class ProcessServlet extends HttpServlet {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String receiverUsername = request.getParameter("receiver");
        int receiverId = AuthenticationServlet.retrieveId(receiverUsername);


        // Set the receiverId as a request attribute
        request.setAttribute("receiverId", receiverId);
        request.setAttribute("receiverUsername", receiverUsername);
        System.out.println("the name is" + receiverUsername);

        // Forward request to ChatPage.jsp
        request.getRequestDispatcher("ChatPage.jsp").forward(request, response);
    }
}
```

Figure 13 ProcessServlet.java

The ProcessServlet class handles HTTP POST requests to process and forward chat-related data. It retrieves the receiver's username from the request parameters, obtains the receiver's user ID using AuthenticationServlet.retrieveId(), and sets both the receiver's ID and username as request attributes. These attributes are then used to prepare for displaying the chat page. Finally, the servlet forwards the request and response to ChatPage.jsp for further processing or display. This setup ensures that the necessary data is available for rendering the chat interface or handling subsequent interactions.

WebService

MessageService.java

```
package webservice;
import javax.jws.WebMethod;
@WebService
public class MessageService {
    @WebMethod
      public String sendMessage(@WebParam(name = "username") String username,
              @WebParam(name = "password") String password,
              @WebParam(name = "usernameReceive")   String usernameReceive,
              @WebParam(name = "messageContent") String messageContent) {

        if (messageContent == null || messageContent.trim().isEmpty() || usernameReceive.isEmpty() ||
                username ==null || username.trim().isEmpty() ||password == null || password.trim().isEmpty()) {
                    return "error fill everything please";
        }
        LoginHandle login= new LoginHandle();
        if(login.authenticateUser(username, password)){
          int senderId=AuthenticationServlet.retrieveId(username);
          if (username.equals(usernameReceive)) {
              return "error cannot send message to yourself";}
   else{
            try{
                int receiverId = AuthenticationServlet.retrieveId(usernameReceive);
             MessageStore.addMessage(senderId, receiverId, messageContent); // Store message
            }
            catch (Exception e){
                e.printStackTrace();
            }

        return "success";
    }
          }
        else{
              return "username or password incorrect";
           }
        }
}
```

Figure 14 MessageService.java

The MessageService class provides a web service for sending chat messages between users. It defines a sendMessage method, which requires parameters for both the sender's and receiver's usernames, the sender's password, and the message content. The method first checks if any of these parameters are missing or empty and returns an error message if so. It then authenticates the sender using LoginHandle. If authentication succeeds and the receiver is not the sender, it retrieves the IDs for both the sender and receiver and stores the message in MessageStore. If any issues arise during this process, exceptions are caught and logged. The method returns a success message upon successful storage of the message, or an error message if authentication fails or if the sender attempts to message themselves.

MessageServlet.java

```java
package webservice;

import javax.servlet.ServletException;

@WebServlet("/messagesoap")
public class MessageServlet extends HttpServlet {

    /**
     *
     */
    private static final long serialVersionUID = 1L;

    @Override
    protected void doPost(HttpServletRequest request, HttpServletResponse response)
            throws ServletException, IOException {
        String username = (String) request.getSession().getAttribute("username");
        int receiverId = Integer.parseInt(request.getParameter("receiverId"));
        String message= request.getParameter("message");
        int senderId = AuthenticationServlet.retrieveId(username);

        System.out.println(username);
        System.out.println(senderId);
        System.out.println(receiverId);
        System.out.println(message);

        try{
            MessageStore.addMessage(senderId, receiverId, message); // Store message
           }
           catch (Exception e){
               e.printStackTrace();
           }


    }
}
```

Figure 15 MessageServlet.java

The MessageServlet class handles HTTP POST requests to process and store chat messages between users. When a POST request is received, it retrieves the sender's username from the session and parses the receiver's ID and message content from the request parameters. It then obtains the sender's ID using AuthenticationServlet.retrieveId() and attempts to store the message in MessageStore using the addMessage method. If any exceptions occur during message storage, they are caught and logged. This servlet ensures that chat messages are correctly recorded in the backend system for further processing or retrieval.

SignUpPage

SignUpPage.jsp

```jsp
<!DOCTYPE html>
<html>
<head>
    <title>Sign Up Page</title>
    <link rel="stylesheet" type="text/css" href="signup.css">
</head>
<body>
    <div class="signup-container">
        <div class="signup-card">
            <h2>Sign Up</h2>
            <form action="authentication" method="post">

             <input type="hidden" name="action" value="signup">

                <label for="username">Username:</label>
                <input type="text" id="username" name="username" required><br><br>
                <label for="password">Password:</label>
                <input type="password" id="password" name="password" required><br><br>
                <label for="confirm-password">Confirm Password:</label>
                <input type="password" id="confirm-password" name="confirm-password" required><br><br>
                <input type="submit" value="Sign Up">

                <%-- Display error message if present --%>
                <% String error = request.getParameter("error");
                if ("true".equals(error)) { %>
                    <p style="color: red;">Error with your signup. Please try again.</p>
                <% } %>
            </form>

            <!-- Link to redirect to Login Page -->
            <a href="LoginPage.jsp" class="redirect-link">
                Already have an account? Log In
            </a>
        </div>
    </div>
</body>
</html>
```

Figure 16 SignUpPage.jsp

The provided JSP file, SignUpPage.jsp, is a web page designed for user registration. It includes a form where users can enter their desired username, password, and confirm their password to create a new account. The form uses the POST method to submit data to the authentication servlet, with a hidden field specifying the action as "signup". If there is an error in the signup process (such as mismatched passwords or an existing username), an error message is displayed in red text. The page also includes a link to the login page for users who already have an account, allowing them to switch to the login interface easily. The styling for the page is handled by an external CSS file (signup.css), which is linked in the <head> section of the document.

LoginPage

LoginPage.jsp

```
<!DOCTYPE html>
<html>
<head>
    <title>Login Page</title>
    <link rel="stylesheet" type="text/css" href="login.css">
</head>
<body>
    <div class="login-container">
        <div class="login-card">
            <h2>Login</h2>
            <form action="authentication" method="post">

            <input type="hidden" name="action" value="login">

                <label for="username">Username:</label>
                <input type="text" id="username" name="username" required><br><br>

                <label for="password">Password:</label>
                <input type="password" id="password" name="password" required><br><br>

                <input type="submit" value="Login">

        <%-- Display error message if present --%>
           <% String error = request.getParameter("error");
   if ("true".equals(error)) { %>
       <p style="color: red;">Invalid username or password. Please try again.</p>
<% } %>
            </form>
 <!-- Link to redirect to SignUp Page -->
            <a href="SignUpPage.jsp" class="redirect-link">
                Don't have an account? Sign Up
            </a>
        </div>
    </div>
</body>
</html>

```

Figure 17 LoginPage.jsp

The LoginPage.jsp file is a web page designed for user authentication. It contains a form where users can enter their username and password to log in. The form submits the data using the POST method to the authentication servlet with a hidden field indicating the action as "login". If there is an authentication error, such as incorrect credentials, an error message is displayed in red text. The page also includes a link to the signup page for users who do not have an account, allowing them to navigate to the registration page easily. The page's appearance is styled by an external CSS file (login.css), which is referenced in the <head> section of the HTML document.

UserSelectionPage

UserSelectionPage.jsp

```jsp
<%@page import="servlet.AuthenticationServlet"%>
<%@ page import="java.util.ArrayList" %>
<%@ page import="java.util.List" %>

<!DOCTYPE html>
<html>
<head>
    <title>IstiChat</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
    <link rel="stylesheet" type="text/css" href="user-selection.css">
      <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css">
    <script>
        function startChat(receiverId, user) {
            var iframe = document.getElementById('chatFrame');
            iframe.src = 'ChatPage.jsp?receiverId=' + encodeURIComponent(receiverId) + '&receiverUsername=' + user;

            // Remove 'selected' class from all user items
            var userItems = document.querySelectorAll('.user-item');
            userItems.forEach(function(item) {
                item.classList.remove('selected');
            });

            // Add 'selected' class to the clicked user
            document.getElementById('user-' + user).classList.add('selected');
        }
        function filterUsers() {
            var input = document.getElementById('searchBar').value.toLowerCase();
            var userItems = document.querySelectorAll('.user-item');

            userItems.forEach(function(item) {
                var username = item.textContent || item.innerText;
                if (username.toLowerCase().indexOf(input) > -1) {
                    item.style.display = "";
                } else {
                    item.style.display = "none";
                }
            });
        }
```


Figure 18 UserSelectionPage.jsp (1)

```
        function clearSearch() {
            document.getElementById('searchBar').value = '';
            filterUsers();
            document.getElementById('searchBar').focus();
        }

    </script>
</head>
<body>

    <div class="user-list">
        <div id="logo">
            <img src="istishat_logo.PNG" alt="Logo" id="logoImage">
        </div>
        <h2>Users</h2>
         <div class="search-container">
            <input type="text" id="searchBar" onkeyup="filterUsers()" placeholder="Search users..." />
            <button id="clearButton" onclick="clearSearch()">X</button>
        </div>
        <div id="userList">
            <%
                Object userListObj = request.getAttribute("users");

                if (userListObj instanceof ArrayList<?>) {
                    @SuppressWarnings("unchecked")
                    ArrayList<String> users = (ArrayList<String>) userListObj;

                    if (users.isEmpty()) {
                        out.println("<p>No users available</p>");
                    } else {
                        String currentUsername = (String) request.getSession().getAttribute("username");
                        if (currentUsername == null) {
                            currentUsername = "";
                        }

                        for (String user : users) {
                            if (!user.equals(currentUsername)) {
            %>
```


Figure 19 UserSelectionPage.jsp (2)

```
            <div id="user-<%=user%>" class="user-item" onclick="startChat(<%=AuthenticationServlet.retrieveId(user) %>, '<%=user%>')">
                <%= user %>
              <a href="ProfilePage.jsp?user=<%=user%>" class="profile-button">
                    <i class="fas fa-user-circle"></i> <!-- Font Awesome icon -->
                </a>
            </div>
            <%
                        }
                    }
                }
            } else {
                out.println("<p>No users found or wrong attribute type</p>");
            }
            %>
        </div>
    </div>
    <div class="chat-area">
        <iframe id="chatFrame" src="" title="Chat Window"></iframe>
    </div>
</body>
</html>
```


Figure 20 UserSelectionPage.jsp (3)

The provided JSP file, UserSelectionPage.jsp, serves as the main interface for users to select and initiate chat conversations. Upon loading, it displays a list of users, excluding the currently logged-in user, which is fetched and rendered dynamically using server-side code. Each user is displayed with a clickable item that, when selected, updates an embedded iframe (chatFrame) to load the chat interface (ChatPage.js) with the selected user's ID and username.

The page also includes a search bar with live filtering functionality to help users quickly locate other users, along with a clear button to reset the search input. The design incorporates external CSS stylesheets (styles.css and user-selection.css) and uses Font Awesome icons for additional visual appeal. Users can also access their profile pages via profile buttons next to each username. The script manages dynamic interactions, such as updating the chat frame and filtering the user list based on search input.

ChatPage

ChatPage.jsp

```
<%@ page import="java.util.ArrayList" %>
<%@ page import="java.util.List" %>

<!DOCTYPE html>
<html>
<head>
    <title>Chat</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
    <link rel="stylesheet" type="text/css" href="chatpage.css">
    <script>
        document.addEventListener("DOMContentLoaded", function() {
            var urlParams = new URLSearchParams(window.location.search);
            var receiverId = urlParams.get('receiverId');
            var textarea = document.querySelector('textarea[name="message"]');


            function adjustHeight() {
                textarea.style.height = 'auto'; // Reset height to auto to calculate scrollHeight
                textarea.style.height = textarea.scrollHeight + 'px'; // Set height based on scrollHeight
            }
            textarea.addEventListener('input', adjustHeight);
            adjustHeight();

            function scrollToBottom() {
                var messagesContainer = document.getElementById('messages');
                messagesContainer.scrollTop = messagesContainer.scrollHeight; // Scroll to the bottom
            }

            function handleKeyPress(event) {

                if (event.key === 'Enter' && !event.shiftKey) {
                     event.preventDefault();
                    sendMessage(event); // Trigger the send message function
                    setTimeout(scrollToBottom, 2000);
                    }
                }
```

Figure 21 ChatPage.jsp (1)

```
        textarea.addEventListener('keydown', handleKeyPress); // Add event listener to handle key press

        function fetchMessages() {
            var xhr = new XMLHttpRequest();
            xhr.open('GET', 'FetchMessagesServlet?receiverId=' + encodeURIComponent(receiverId), true);
            xhr.onload = function() {
                if (xhr.status === 200) {
                    var messagesContainer = document.getElementById('messages');
                    var newContent = xhr.responseText;
                    // Check if the content has changed
                    if (messagesContainer.innerHTML !== newContent) {
                        messagesContainer.innerHTML = newContent;
                }
                } else {
                    console.error('Error fetching messages: ' + xhr.status);
                }
            };
            xhr.send();
        }

        function sendMessage(event) {
            event.preventDefault();
            var messageContent = document.querySelector('textarea[name="message"]').value.trim();

            // Prevent sending empty or whitespace-only messages
            if (messageContent === "") {
                return; // Do nothing if the message is empty or just whitespace
            }
            var form = document.getElementById('sendMessageForm');
            var formData = new URLSearchParams();

            formData.append('message', document.querySelector('textarea[name="message"]').value);
            formData.append('receiverId', receiverId);

            var xhr = new XMLHttpRequest();
            xhr.open('POST', 'messagesoap', true);
            xhr.setRequestHeader('Content-Type', 'application/x-www-form-urlencoded');
            xhr.onload = function() {
```


Figure 22 ChatPage.jsp (2)

```
                xhr.onload = function() {
                    if (xhr.status === 200) {
                        setTimeout(scrollToBottom, 1000);
                    } else {
                        console.error('Error sending message: ' + xhr.status);
                    }
                };
                document.querySelector('textarea[name="message"]').value = "";

                xhr.send(formData.toString());
            }

            document.getElementById('sendMessageForm').addEventListener('submit', sendMessage);

            // Fetch messages initially and then periodically
            fetchMessages();
            setInterval(fetchMessages, 1000); // Fetch every second
            setTimeout(scrollToBottom, 1000);

        });


    </script>
</head>
<body>
    <h2>Chat with <%= request.getParameter("receiverUsername") %></h2>
     <div id="messages">
        <!-- Messages will be loaded here -->
    </div>
    <form id="sendMessageForm" action="messagesoap" method="post">
        <textarea id="message" name="message" rows="4" cols="50" required></textarea><br>
        <input type="hidden" name="receiverId" value="<%= request.getParameter("receiverId") %>">
        <input type="submit" id="sendButton" value="Send">
    </form>

</body>
</html>
```


Figure 23 ChatPage.jsp (3)

The provided JSP file, ChatPage.jsp, is designed for real-time chat interactions between users. It initializes the chat interface by dynamically loading the chat history and providing an input area for composing messages. Upon page load, it sets up event listeners to handle user input, automatically adjust the textarea height, and manage scrolling to ensure the most recent messages are visible.

The script includes functions to handle sending messages via AJAX, fetch new messages periodically, and scroll the message view to the bottom. It also prevents sending empty messages and ensures that the message area is cleared after submission. The chat messages are fetched every second from the FetchMessagesServlet, and new messages are sent using the messagesoap endpoint. The page also provides a user-friendly experience by adjusting the textarea height dynamically and ensuring smooth scrolling.

ProfilePage

ProfilePage.jsp

```jsp
<%@ page import="java.util.*" %>
<!DOCTYPE html>
<html>
<head>
    <title>User Profile</title>
    <link rel="stylesheet" type="text/css" href="styles.css">
    <link rel="stylesheet" type="text/css" href="profile.css">
    <link rel="stylesheet" href="https://cdnjs.cloudflare.com/ajax/libs/font-awesome/6.0.0-beta3/css/all.min.css">
</head>
<body>
    <a href="javascript:history.back()" class="back-button">
        <i class="fas fa-arrow-left"></i> Back
    </a>
    <div class="profile-container">
        <header class="profile-header">
            <h1>User Profile</h1>
        </header>
        <form class="profile-details-form">
            <div class="profile-details">
                <%
                    String username = request.getParameter("user");
                    if (username != null && !username.isEmpty()) {
                        out.println("<h2>" + username + "</h2>");
                    } else {
                        out.println("<h2>User Not Found</h2>");
                    }
                %>
            </div>
            <input type="hidden" name="username" value="<%=username%>">
            <button type="submit" class="block-button" disabled>Block User</button>
        </form>
    </div>
</body>
</html>
```

Figure 24 ProfilePage.jsp

The ProfilePage.jsp file is used to display the profile information of a specific user. It starts with a link to navigate back to the previous page and includes styles for layout and design. The main content of the page is centered on a profile-container that displays the username of the user whose profile is being viewed.

The JSP script retrieves the username from the request parameters and displays it within a heading. If the username is missing or invalid, it shows a default message indicating that the user was not found. Additionally, the form includes a "Block User" button, which is currently disabled, suggesting that blocking functionality might be planned but is not yet implemented. The use of Font Awesome icons enhances the visual appeal of the back navigation button.

Dependencies

The project relies on a set of essential libraries to provide various functionalities and ensure compatibility. These include ampq-client-5.0.0.jar for AMQP protocol support, javax.servlet-3.1.jar and javax.servlet-api-4.0.1.jar for servlet API functionalities, json-20230618.jar for JSON processing, and ojdbc8.jar along with orai18n-19.3.0.0.jar for Oracle database connectivity and internationalization support. Additionally, slf4j-api-1.7.36.jar and slf4j-simple-1.7.36.jar are used for logging and debugging purposes. These dependencies collectively ensure robust communication, data handling, and operational capabilities within the application.

Limitations

The current implementation of the application is limited to handling only text-based messages, such as strings and hyperlinks. Support for multimedia content, including images and files, is not yet available. Additionally, the application is not optimized for scalability and may struggle under heavy loads, particularly with a large number of concurrent users. As user traffic increases, the system may experience performance degradation and higher latency, highlighting the need for further optimization and scaling strategies to accommodate more extensive use cases in the future.

Conclusion

In conclusion, this project presents a foundational chat application that facilitates user interaction through basic text messaging. Leveraging core technologies such as JSP for the front end, Java Servlets for server-side processing, and a range of essential libraries and dependencies, the system provides a straightforward user experience for authentication, messaging, and profile management. Despite its current limitations, such as the lack of support for multimedia messages and scalability concerns, the project serves as a solid basis for future enhancements. With additional development, including support for richer media content and improved performance optimizations, the application has the potential to evolve into a more robust and scalable communication platform.