SOFTWARE REQUIREMENTS

Software Requirements Specification for

"SimpleTalk" Online Chat

Version 1.0

Table of Contents

1. Introduction

1.1. Purpose

This document provides a detailed Software Requirements Specification (SRS) for a minor university project: a simple, real-time online chat application named "SimpleTalk". The purpose of this document is to define the functional and non-functional requirements, system architecture, and overall scope of the project. It serves as a foundational agreement between the project developer and evaluators.

1.2. Document Conventions

This document is written in Markdown.

1.3. Intended Audience

This SRS is intended for the project developer, university professors, and any evaluators who need to understand the system's functionality, constraints, and design.

1.4. Project Scope

The project's scope is to develop a proof-of-concept desktop chat application demonstrating core programming skills, client-server communication, and WebSocket technology. The application consists of two independent components: a Java Swing-based frontend client and a Java Spring-based backend server.

The system will:

- Allow multiple users to connect to a central server.
- Provide a single, public chat room for all connected users.
- Display messages in real-time.
- Show a list of currently online users.

The system will NOT (in this version):

- Provide user authentication or private messaging.
- Store chat history persistently.
- Support file transfers, voice, or video calls.

1.5. References

- Schildt, H., & Coward, D. (2025). *Java: The Complete Reference, 13th Edition*. McGraw Hill.
- Xu, A. (2020). *System Design Interview – An Insider's Guide*.

2. Overall Description

2.1. Product Perspective

SimpleTalk is a self-contained, client-server application. The backend is a microservice deployed independently on a cloud platform (Render.com), while the frontend is a desktop application that can run on any machine with a Java Runtime Environment (JRE). The two components are loosely coupled and communicate exclusively over the internet using the STOMP protocol over WebSockets.

2.2. Product Features

The main features of the application are:

- **User Identification:** Prompts for a username upon launch.
- **Group Chat:** A single chat room for all online users.
- **Real-time Communication:** Instantaneous message delivery.
- **Online User List:** A visible list of all active participants.

2.3. User Classes and Characteristics

There is a single class of user: a **General User**. This user can connect to the chat, send messages to the public group, and view messages from others. No technical expertise is required to use the application.

2.4. Operating Environment

- **Backend:** Deployed on the Render.com cloud platform within a Docker container. It requires a Java environment and is accessible via a public URL.
- **Frontend:** A desktop application built with Java Swing. It requires a machine running a compatible operating system (Windows, macOS, Linux) with Java Runtime Environment (JRE) 8 or newer installed.

2.5. Design and Implementation Constraints

- **Backend Language/Framework:** Java, Spring Boot, Spring WebSocket.
- **Frontend Language/Framework:** Java, Swing (GUI built programmatically).
- **Communication Protocol:** STOMP over WebSocket.
- **Data Interchange Format:** JSON.
- **Deployment:** The backend must be containerized using Docker and deployed on Render.com.

3. System Features

3.1. User Connection and Identification

- **Description:** Before entering the main chat, a user must provide a nickname for identification within the session.
- **Functional Requirements:**
  - **FR-1:** On launch, the application shall display a modal dialog box with a text field and a "Connect" button.
  - **FR-2:** The user must enter a non-empty string as their username.
  - **FR-3:** This username is used for display purposes only and does not involve authentication or validation against a database.

- **FR-4:** Upon providing a username, the application will attempt to establish a WebSocket connection with the backend server.

3.2. Real-Time Messaging

- **Description:** Users can send text messages that are broadcast to all other connected users in real-time.
- **Functional Requirements:**
  - **FR-5:** The main chat window shall contain a text input area and a "Send" button.
  - **FR-6:** When a user types a message and clicks "Send", a Message object (containing the username and message text) is created.
  - **FR-7:** This Message object shall be serialized into a JSON string.
  - **FR-8:** The JSON payload shall be sent to the backend via the /app/chat.sendMessage STOMP destination.
  - **FR-9:** The client shall subscribe to the /topic/public STOMP destination to receive messages.
  - **FR-10:** Incoming JSON payloads from the topic shall be deserialized into Message objects.
  - **FR-11:** The received message, prefixed with the sender's username, shall be appended to the central message display area (e.g., [Username]: Hello world!).

3.3. Online User Roster

- **Description:** The client interface displays a list of all users currently connected to the chat server.
- **Functional Requirements:**
  - **FR-12:** A dedicated UI component (e.g., a list or text area) on the left side of the main window shall display usernames.
  - **FR-13:** When a new user connects, the server shall broadcast an "add user" event to all clients via the /topic/public destination. Clients will update their user list accordingly.
  - **FR-14:** When a user disconnects, the server shall broadcast a "remove user" event, and clients will remove the username from their list.

4. External Interface Requirements

4.1. User Interfaces

The application will have two main UI screens:

1. **Username Entry Dialog:** A simple modal window with one input field for the username and one button to proceed.
2. **Main Chat Window:**
   - **Theme:** Dark theme with light-colored text for readability.
   - **Layout:**
     - **Left Pane:** A vertical list displaying the usernames of online users.
     - **Center Pane:** A non-editable, scrollable area displaying the chat message history for the current session.
     - **Bottom Pane:** A single-line text input field for composing messages and a "Send" button.

4.2. Hardware Interfaces

No specific hardware interfaces are required.

4.3. Software Interfaces

- **Frontend:** Requires a Java Runtime Environment (JRE) to be installed on the user's machine.
- **Backend:** Interacts with the Docker engine for containerization and the Render.com platform APIs for deployment and networking.

4.4. Communications Interfaces

- **Protocol:** The client communicates with the server using the **STOMP** messaging protocol layered on top of **WebSocket**. The initial connection is established via an HTTP upgrade request.
- **Data Format:** All data payloads exchanged between the client and server are in **JSON** format.
- **Endpoints:**

- **Client-to-Server (Sending Messages):** Clients send messages to a broker-mapped destination, typically /app/{action} (e.g., /app/chat.sendMessage).
- **Server-to-Client (Broadcasting Messages):** The server pushes messages to clients subscribed to a topic, typically /topic/{channel} (e.g., /topic/public).

5. Non-Functional Requirements

5.1. Performance

The system should handle message delivery with latency under 500ms for up to 20 concurrent users. The UI should remain responsive and not freeze during message sending or receiving.

5.2. Security

This initial prototype has no security requirements. Communication is not encrypted beyond the standard WebSocket protocol (ws://). Usernames are not validated, and there is no protection against impersonation or spam.

5.3. Reliability

The backend service, hosted on Render.com, should aim for high availability. The client application should not crash on unexpected input. Graceful handling of connection loss is a goal for future versions.

5.4. Usability

The user interface must be intuitive and simple. A new user should understand how to enter a name, send a message, and read the chat without instructions.

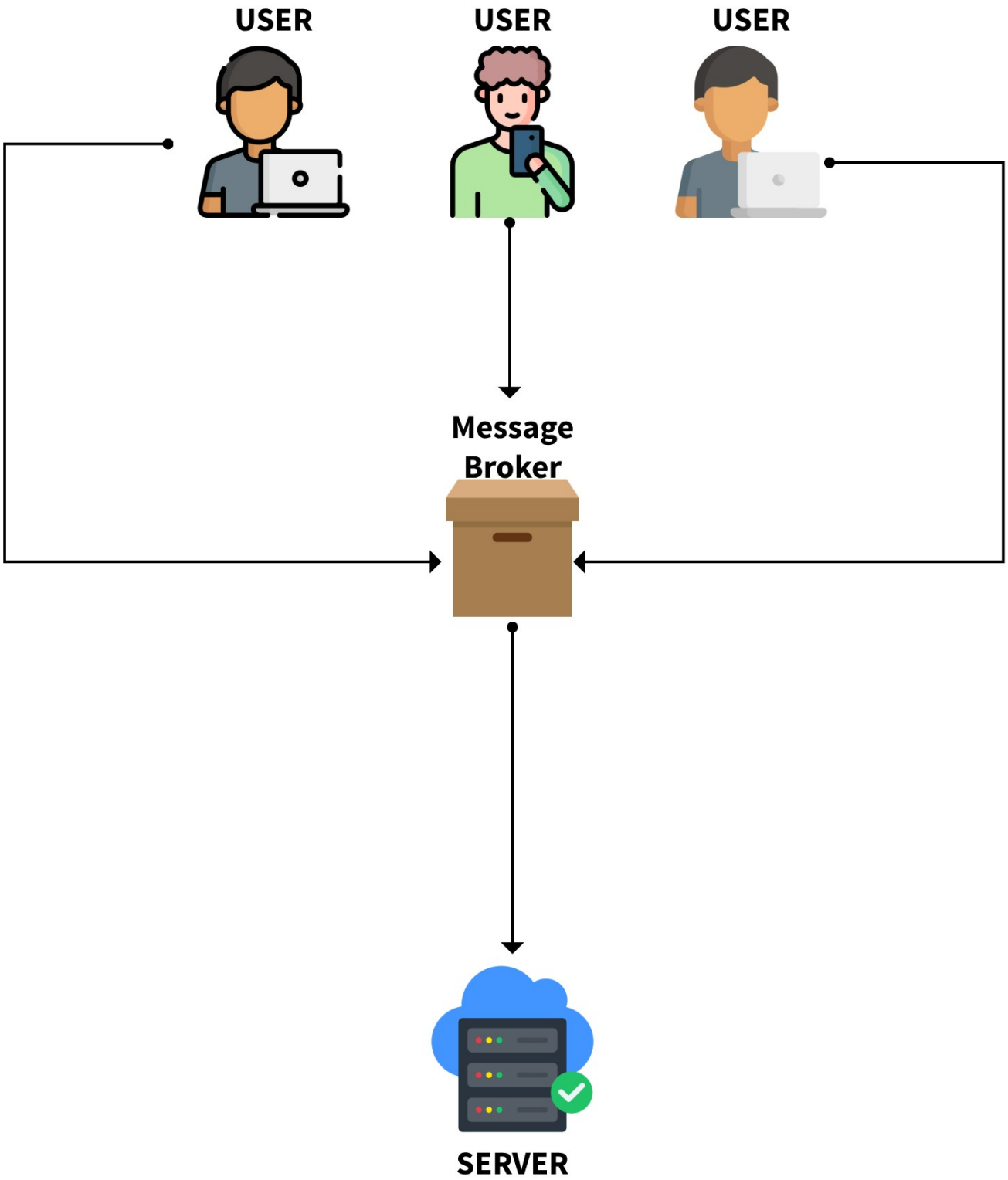
6. System Architecture

6.1. Backend Architecture

The backend follows a simple microservice architecture built with Spring Boot.

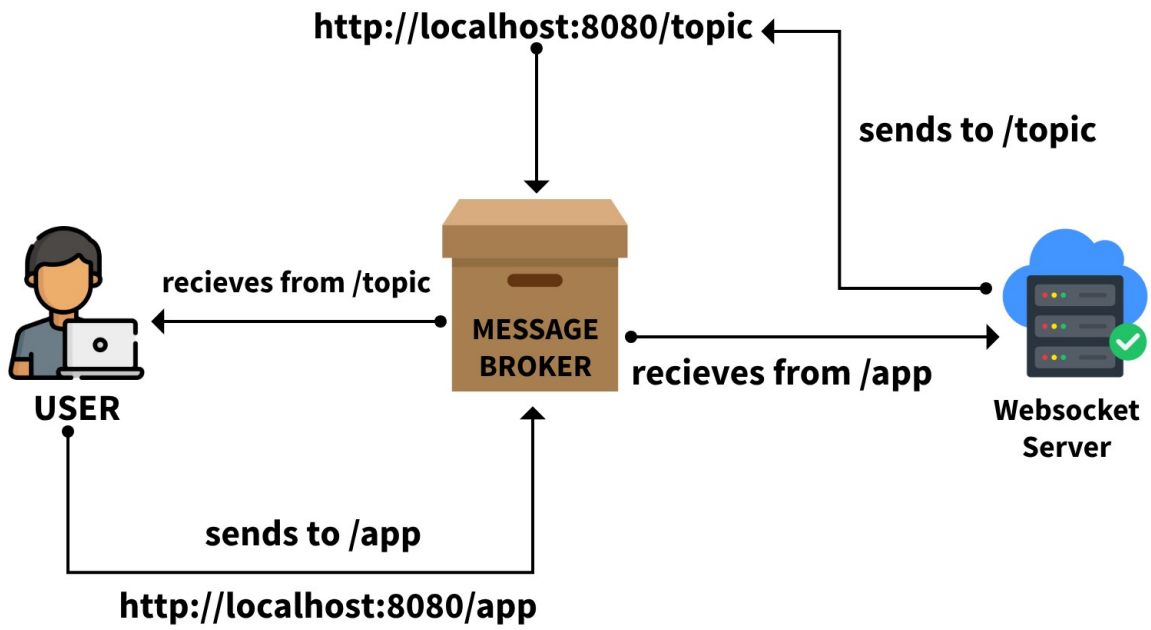
- **WebSocket Controller:** A Spring @Controller class handles incoming STOMP messages from clients.

- **Message Broker:** An in-memory STOMP message broker is configured to route messages between clients and the controller. It manages subscriptions to topics.



A diagram showing the data flow from user to server through Message Broker.

- **Containerization:** The application is packaged as a Docker image using a Dockerfile and managed with docker-compose.yml for easy deployment and dependency management.



A diagram showing the Spring Boot application, the WebSocket endpoint, the STOMP broker, and its interaction with the controller.

6.2. Frontend Architecture

The frontend is a monolithic desktop application built with the Java Swing toolkit.

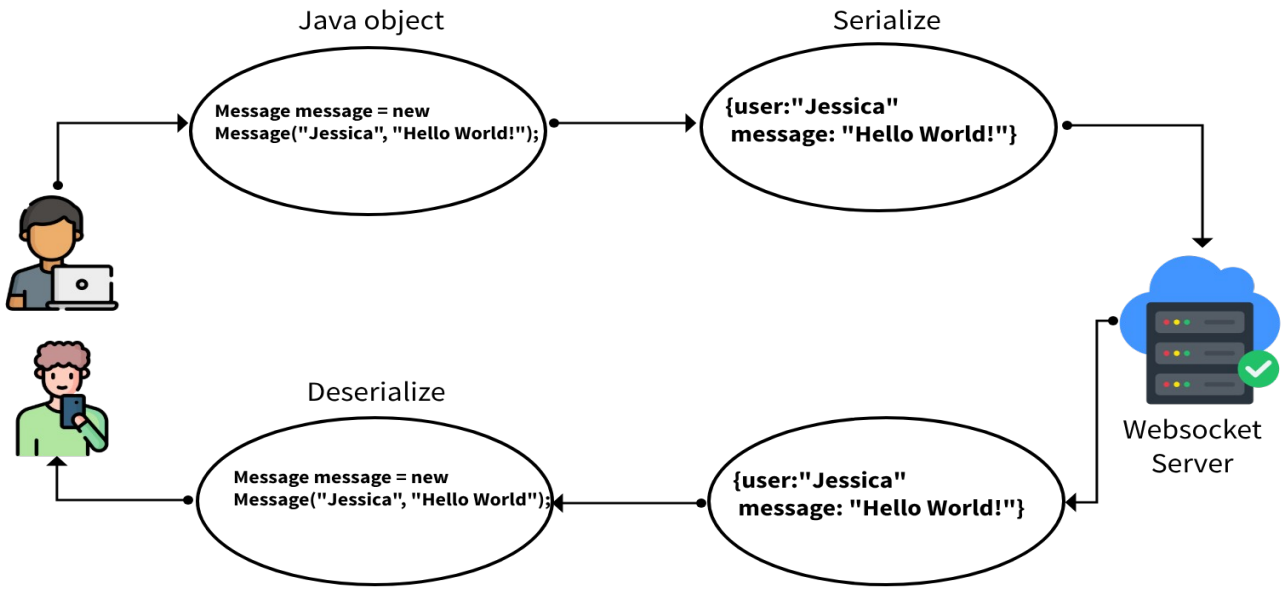
- **View:** The UI is constructed programmatically (without a GUI builder). It consists of JFrame, JPanel, JTextArea, JTextField, and JButton components.
- **Controller/Logic:** Event listeners (ActionListener) handle user input. A dedicated class manages the WebSocket connection and message handling logic, updating the UI components as needed.

6.3. Data and Communication Flow

The communication flow is central to the application's design.

1. Client A establishes a WebSocket connection and subscribes to /topic/public.

2. Client A sends a message. The Message object is serialized to JSON.
3. The JSON payload is sent as a STOMP frame to the /app/chat.sendMessage destination.
4. The server's message broker routes this to the @MessageMapping method in the WebSocket controller.
5. The controller processes the message and sends it back to the broker, targeting the /topic/public destination.
6. The broker broadcasts the message to all clients subscribed to /topic/public, including Client A.
7. Clients receive the STOMP frame, deserialize the JSON payload, and display the message.



A sequence diagram illustrating the end-to-end journey of a message from one client to another through the backend server.

7. Future Scope and Scalability

This project serves as a foundation for a more feature-rich application. Future enhancements could include:

- **Authentication & Authorization:** Integrate Spring Security for user login and role-based access.
- **Web Version:** Develop a parallel web client using a modern JavaScript framework (e.g., React, Vue).
- **Improved GUI:** Redesign the desktop and web UIs for a better user experience.
- **Chat History:** Integrate a database (e.g., PostgreSQL, MongoDB) to persist conversations.
- **Scalability:** Introduce a load balancer (e.g., Nginx) and a more robust message queue (e.g., RabbitMQ, Kafka) to handle a larger number of users.
- **User Profiles:** Allow users to have profiles with avatars and status messages.
- **Notifications:** Implement desktop or browser notifications for new messages.
- **Advanced Features:** Add support for sending files, voice messages, and initiating voice/video calls (using WebRTC).
- **Caching:** Use a cache like Redis for quick access to recent messages or user session data.
- **Settings:** Provide user-configurable settings for themes, notifications, and profile management.

8. Glossary

- **Broker:** An intermediary component that routes messages from senders to the correct receivers.
- **Client:** The frontend application (Java Swing GUI) that the user interacts with.
- **JSON (JavaScript Object Notation):** A lightweight data-interchange format.
- **Server:** The backend application (Java Spring) that manages connections and message broadcasting.
- **STOMP (Simple Text Oriented Messaging Protocol):** A messaging protocol designed for simplicity, operating on top of other transport protocols like WebSocket.
- **WebSocket:** A communication protocol that provides full-duplex communication channels over a single TCP connection.