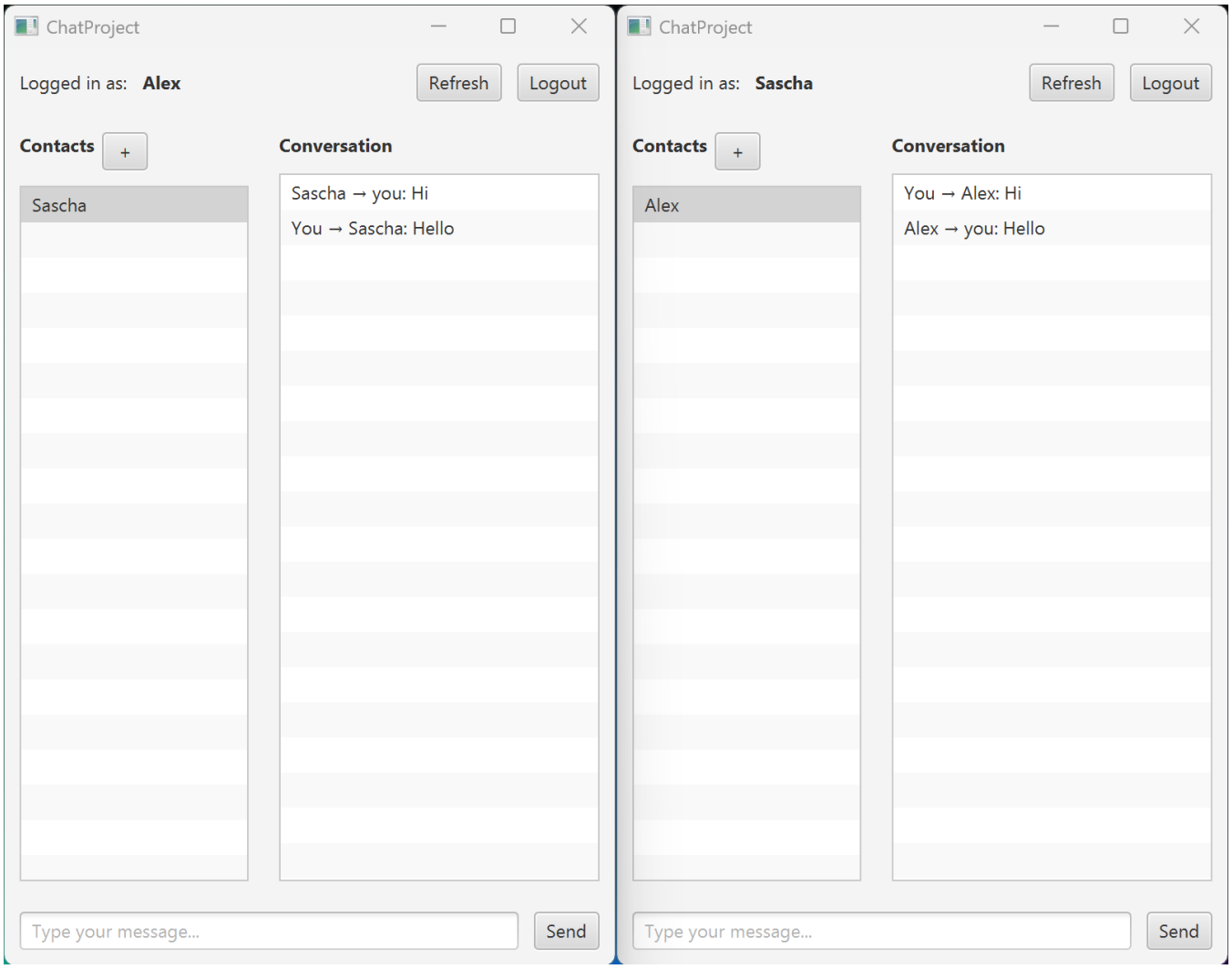
Chat Client Project Report

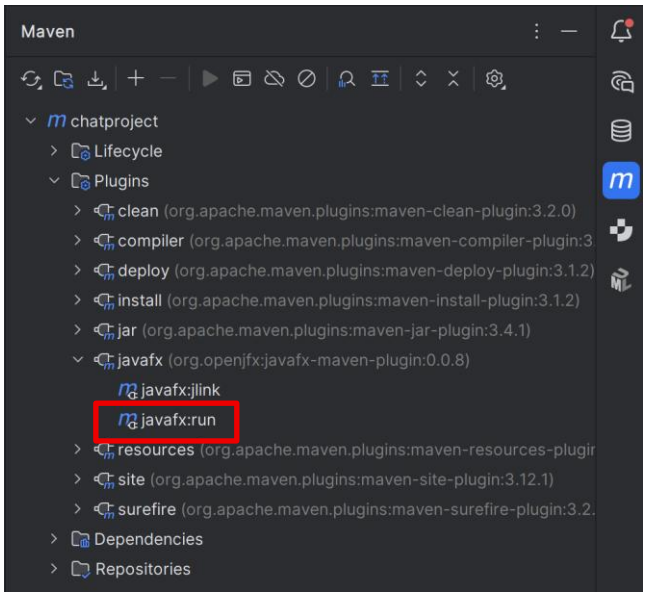
Author: Sascha Niederhauser & Alexander Burri
Course: 25HS Software Engineering (SEL)
Instructor: Prof. Dr. Bradley Richards



1. Usage Instructions

To get started with the project, first clone the repository from GitHub using git clone https://github.com/AlexanderBurri1/ChatProject.git and navigate into the project directory.

After the build completes successfully, start the application with Maven-chatproject-Plugins-javafx-javafx:run.



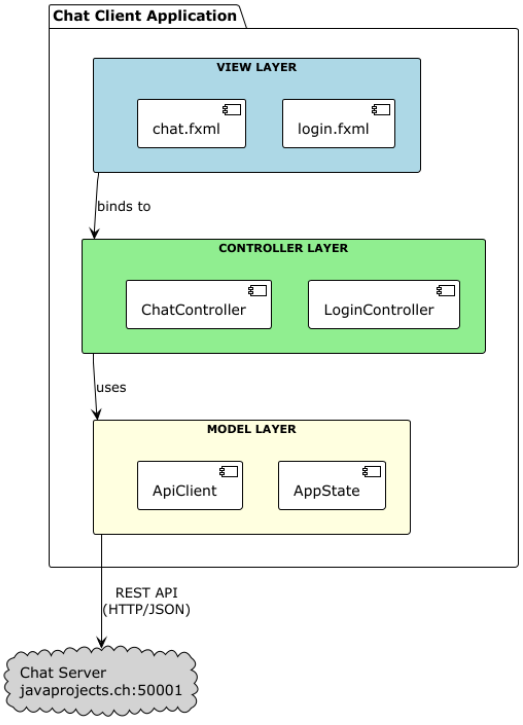
When the application starts, you'll see the login screen with the default server address already configured as http://javaprojects.ch:50001. You can change this if you want to connect to a different server.

To use the chat, you first need to register a new account or login with existing credentials. After logging in, you can add contacts by clicking the "+" button next to the contact list. Select any contact to view your conversation history with them. You can send messages either by clicking the Send button or by pressing Enter. The application automatically checks for new messages every 3 seconds, so you don't need to manually refresh however you can.

2. Design

2.1 Architecture Overview

The application follows the Model-View-Controller (MVC) architectural pattern using JavaFX as the UI framework. The model layer consists of the ApiClient class for REST communication and the AppState class for managing global application state. The view layer is implemented using FXML files that define the UI layouts. The controller layer contains LoginController and ChatController, which handle user interactions and coordinate between the model and view.



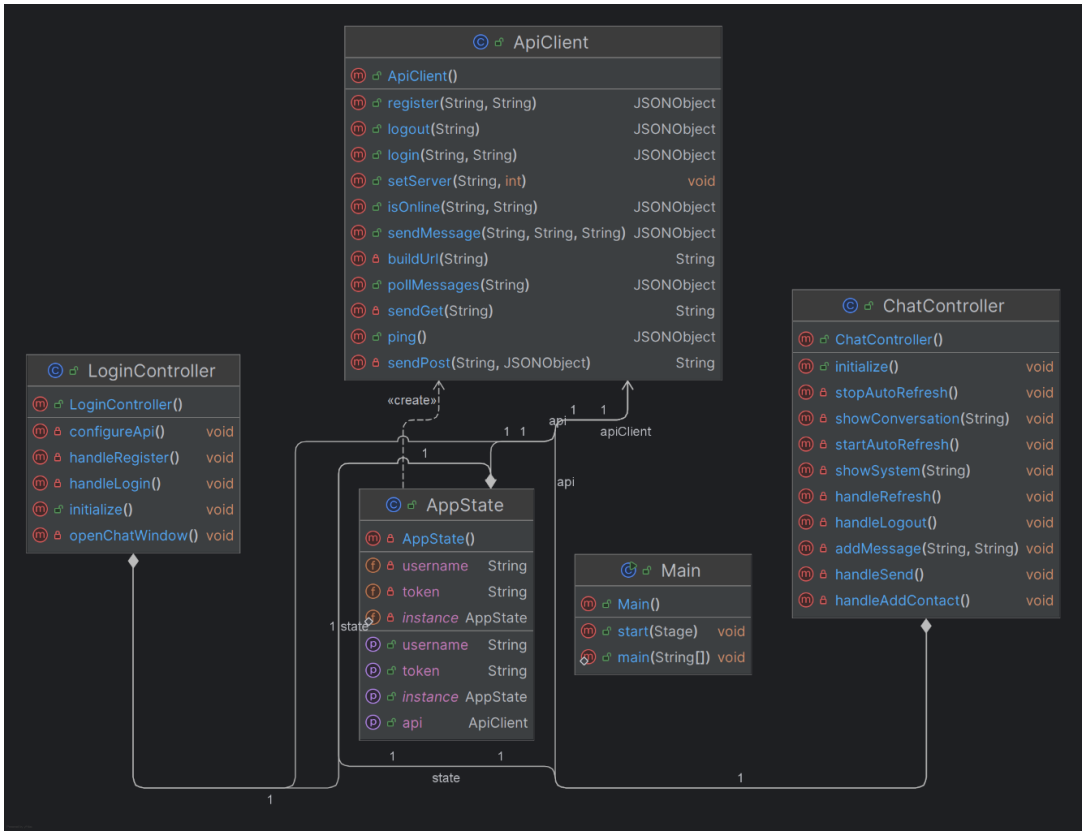
2.2 Key Classes and Their Design

The **ApiClient** class encapsulates all REST API communication with the server. Every API method follows the same pattern: it constructs the appropriate JSON request, sends it to the server, and returns the response as a JSONObject. An important design decision was to handle all exceptions internally and return error messages as JSON objects with an "Error" field. This approach provides consistent error handling throughout the application since controllers can always check for the presence of an error field rather than dealing with different types of exceptions.

The **AppState** class implements the Singleton pattern to maintain global application state. It stores the currently logged-in username, the authentication token, and a shared instance of ApiClient. Using a Singleton makes sense here because only one user can be logged in at a time, and these values need to be accessible from multiple controllers. The alternative would be passing the token through every method call or using dependency injection, which would add unnecessary complexity for this project scope.

The **LoginController** manages the authentication flow. It allows users to configure the server address and port, handles both registration and login requests, and transitions to the chat interface upon successful authentication. The controller validates user input before making API calls and displays appropriate error messages in the status label. One practical detail is that the controller calls a configuration method before each API request, ensuring that any changes to server settings are applied immediately.

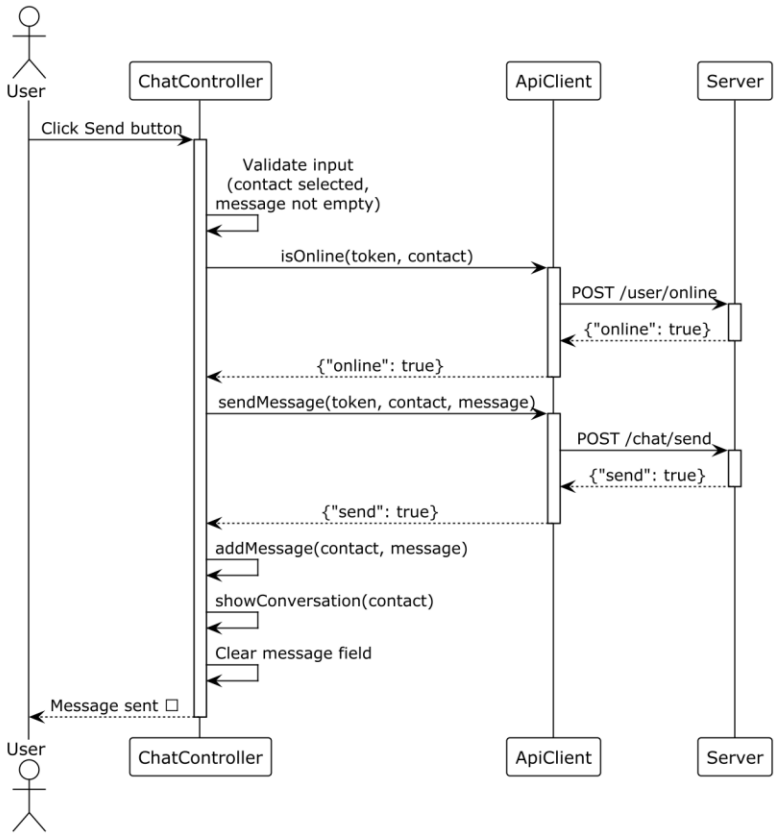
The **ChatController** is the most complex part of the application as it manages the entire chat interface. Conversations are stored in a HashMap where each contact name maps to a list of message strings. This data structure provides fast lookups and naturally separates different conversations. The controller uses a JavaFX Timeline to automatically poll the server for new messages every 3 seconds. When new messages arrive, the controller adds them to the appropriate conversation and updates the UI if that contact is currently selected. Before sending a message, the controller checks whether the recipient is online using the isOnline API call, providing immediate feedback to the user rather than attempting to send to offline users.



2.3 Communication Flow

When a user wants to send a message, several steps happen in sequence. First, the controller verifies that a contact is selected and that the message field isn't empty. Then it checks if the recipient is currently online by calling the user/online endpoint. Only if the recipient is online does it proceed to send the message through the chat/send endpoint. After successfully sending, the message is added to the local conversation history and displayed in the UI. This approach prevents confusion by immediately telling users if their message couldn't be delivered.

For receiving messages, the Timeline in ChatController triggers the handleRefresh method every 3 seconds. This method calls the chat/poll endpoint which returns an array of new messages since the last poll. For each incoming message, the controller adds it to the appropriate conversation history and updates the contact list if the sender is new. If the sender is the currently selected contact, the message appears immediately in the conversation view. This polling approach is necessary because the project requires REST API usage rather than WebSocket connections.



2.4 User Interface Design

The **login** screen uses a simple VBox layout that vertically stacks the server configuration fields, username and password inputs, and action buttons. This straightforward layout makes the login process intuitive and focuses attention on the essential inputs.

The **chat interface** uses a BorderPane layout to organize different functional areas. The top section displays the logged-in username along with refresh and logout buttons. The left side contains the contact list with an add button. The center area shows the conversation history for the selected contact. The bottom section has the message input field and send button. This layout follows familiar messaging app conventions, making it immediately recognizable to users.

3. Testing

The application was thoroughly tested against the reference server running at javaprojects.ch on port 50001. All required API endpoints were verified to work correctly including user registration, login, sending messages to online users, polling for new messages, checking user online status, and logout functionality.

Testing covered both successful operations and error cases such as attempting to login with incorrect passwords, sending messages to offline users, and handling network connectivity issues. The application correctly displays error messages and maintains a consistent user experience even when API calls fail.

In addition, the chat client was tested on multiple fresh devices to ensure its functionality.