**Shutaf-In Messenger Documentation**

* + - 1. **Common 2**
      2. **Web Socket Protocol 3**
      3. **STOMP over SockJs 4**
      4. **Backend Structure 5**
      5. **Frontend Structure 7**
      6. **Authentication Flow 10**

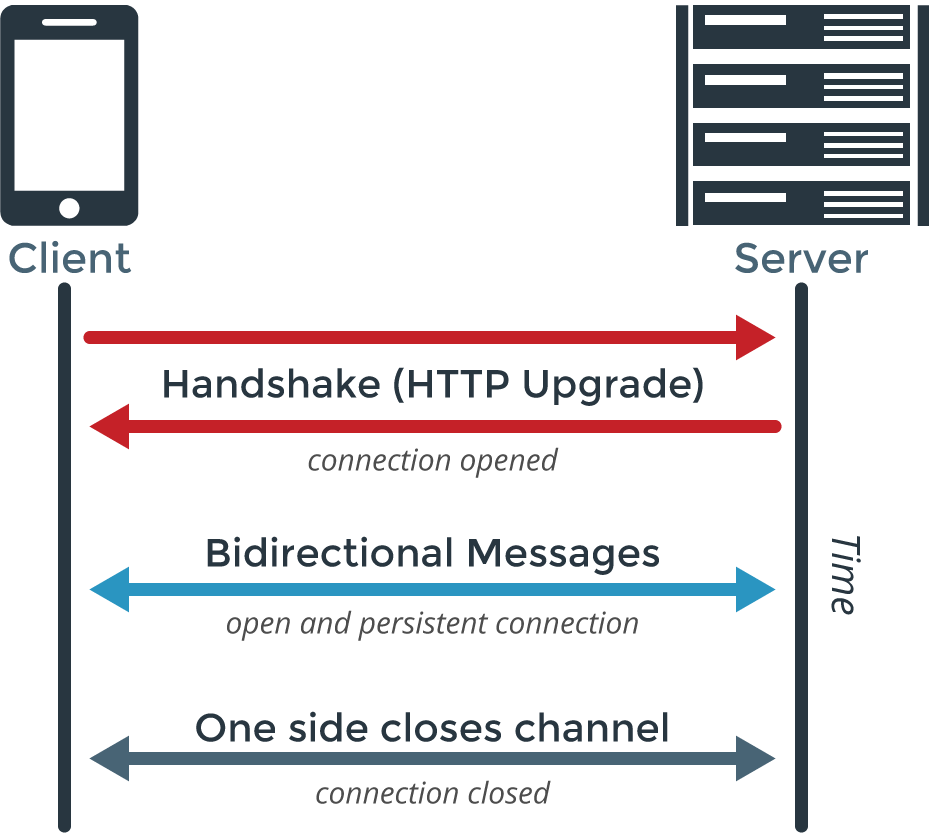
1. **Common:**

Basis: Each user is able to start conversation with another user. But all users need to pass the required matching, because the list of compatible chat-users will be requested from matching service. The infrastructure support creation of group-chats by default. So it is allowed to add any number of users in any already created chat conversation. Please notice, that it’s not possible to start conversation with the same user twice. In current version each user can delete every user from chat, there is no concept of “chat owner” realized. If user will be deleted from chat, he will see already received messages, but new messages will be unreachable for him. If somebody will add this user to chat again, he will be not able to see messages, that were sent in time he wasn’t in chat. The same is true for users that were added after the chat was created. If user deletes himself from chat, the backend will set flag isDeletedFromChat to true. That means, that he will lost access to chat and new messages. By default all messages will be stored on server without possibility for users to delete them. New chats don’t have title because of UI restrictions, so instead of title they will have concatenation of chat members names used. It is possible to change the title at any time by any chat member. All changes such as chat renaming, adding or deleting user from chat will be shown in “real-time” for all chat members. There was extra notification system for this purpose created. All chat data are fetched by initialization service, so there is no need to make extra queries on backend. Please notice, we are using multi-subscriptions model, so each chat-room has his own Websocket subscription handled by STOMP.

1. **Web Socket Protocol:**

The whole messaging system is based on Web Socket, that allows us to send

and receive messages in real-time. Each connection works over TCP after HTTP upgrade:

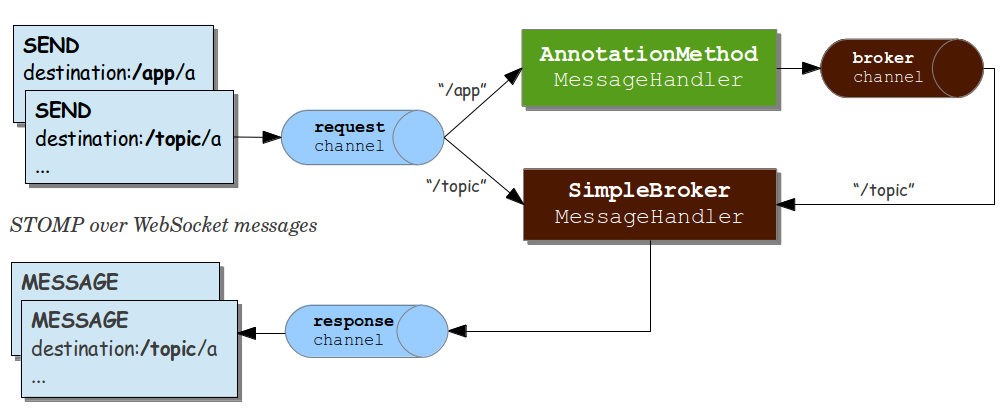


More here: <https://medium.com/platform-engineer/web-api-design-35df8167460>

<https://en.wikipedia.org/wiki/WebSocket>

1. **STOMP over SockJs:**

STOMP = Simple (or Streaming) Text Orientated Messaging Protocol. STOMP is a simple interoperable protocol designed for asynchronous message passing between clients via mediating servers. It defines a text based wire-format for messages passed between these clients and servers.



More here: <https://stomp.github.io/stomp-specification-1.2.html>

Example: <https://www.concretepage.com/spring-4/spring-4-websocket-sockjs-stomp-tomcat-example>

Example on Github:

<https://github.com/rstoyanchev/spring-websocket-portfolio> (very recommended)

1. **Backend Structure:**

In our project we are using abilities provided by Spring Boot Starter Websocket.

* **com.shutafin.controller.ChatController** - controller for external communication with client in browser is here.
* **com.shutafin.configuration.WebSocketEndpointBrokerAndInterceptorConfigurer** – here we are setting up the configuration of Message Broker, Stomp Endpoint, Client Inbound Channel Interceptor and Argument Resolvers.
* **com.shutafin.configuration.HttpSessionIdHandshakeInterceptor** – this class is used to intercept all websocket handshakes to verify a sessionId that contains in path variable inside ServletServerHttpRequest (we putting sessionId on frontend at si-gateway/si-gateway-webapp/src/client/src/config/**websocket.service.js** # function **getSocket**) and to put userId according to his sessionId to socket attributes to check it later
* **com.shutafin.configuration.ClientSessionInboundChannelInterceptor** – checks every message over websocket on correct sessionId and according it to correct userId in header (STOMP is used to pass headers over websocket). Saves websocketsessions by **WebSocketSessionService** on CONNECTand deletes them on DISCONNECT.
* **com.shutafin.configuration.AuthenticatedUserAnnotationMethodArgumentResolver** – fixes the bug with **AuthenticatedUser** annotation that happens sometimes by authentication of incoming messages over websocket.
* **com.shutafin.processors.AbstractAuthenticationProtocolTypeResolver** – bean post processor for authentication annotations, **WebSocketAuthentication** also.
* **com.shutafin.processors.AbstractAuthenticationProtocolTypeResolver** – decides which protocol will be authenticated and calls handlers, **WebSocketProtocolHandler** also.
* **com.shutafin.processors.WebSocketProtocolHandler** – verifies sessionId in STOMP header.
* **com.shutafin.service.impl.chat.ChatInfoServiceImpl** – handles all get requests and returns chats and messages.
* **com.shutafin.service.impl.chat.ChatAuthorizationServiceImpl** – used to find authorized chats for user and authorized users in chat.
* **com.shutafin.service.impl.chat.ChatManagementServiceImpl** – manages all changes on chats. Creating chat, adding and deleting of users, persisting messages, renaming chat and sending notifications to chat users.
* **com.shutafin.service.impl.chat.WebSocketSessionServiceImpl** – contains ConcurrentHashMap, where the key is userId and value is his WebSocketSessionId. Adds websocketsessions to map, finds them in map by userId and deletes key<>value pairs from map.
* **com.shutafin.model.web.notification.ChatNotificationWeb** – wrapper for chat notifications.
* **com.shutafin.model.web.notification.NotificationReason** – chat notification reasons.
* **com.shutafin.model.web.chat.ChatWithUsersListDTO** – contains all information about chat, his title, id, is the user active in this chat, list with id of chat users.
* **com.shutafin.handlers.ResponseExceptionHandler** – prevents overload of server logs with ClientAbortException that happens when user close client application without logout. In debug mode shows full stack trace.

**Models:**

* **com.shutafin.model.entities.Chat** – chat entity
* **com.shutafin.model.entities.ChatMessage** – chat message entity
* **com.shutafin.model.entities.ChatUser** – chat user entity
* **com.shutafin.model.entities.types.ChatMessageType** – enum for chat message type
* **com.shutafin.model.entities.types.ChatMessageTypeConverter** – converter for chat message type
* **com.shutafin.model.entities.types.ChatUserListConverter** – converter for chat user id list, that is persist as string in database

**Persistence**:

* **com.shutafin.repository.common.ChatMessageRepository** – persistence of chat messages**.**
* **com.shutafin.repository.common.ChatRepository** – persistence of chats.
* **com.shutafin.repository.common.ChatUserRepository** – persistence of chat users.

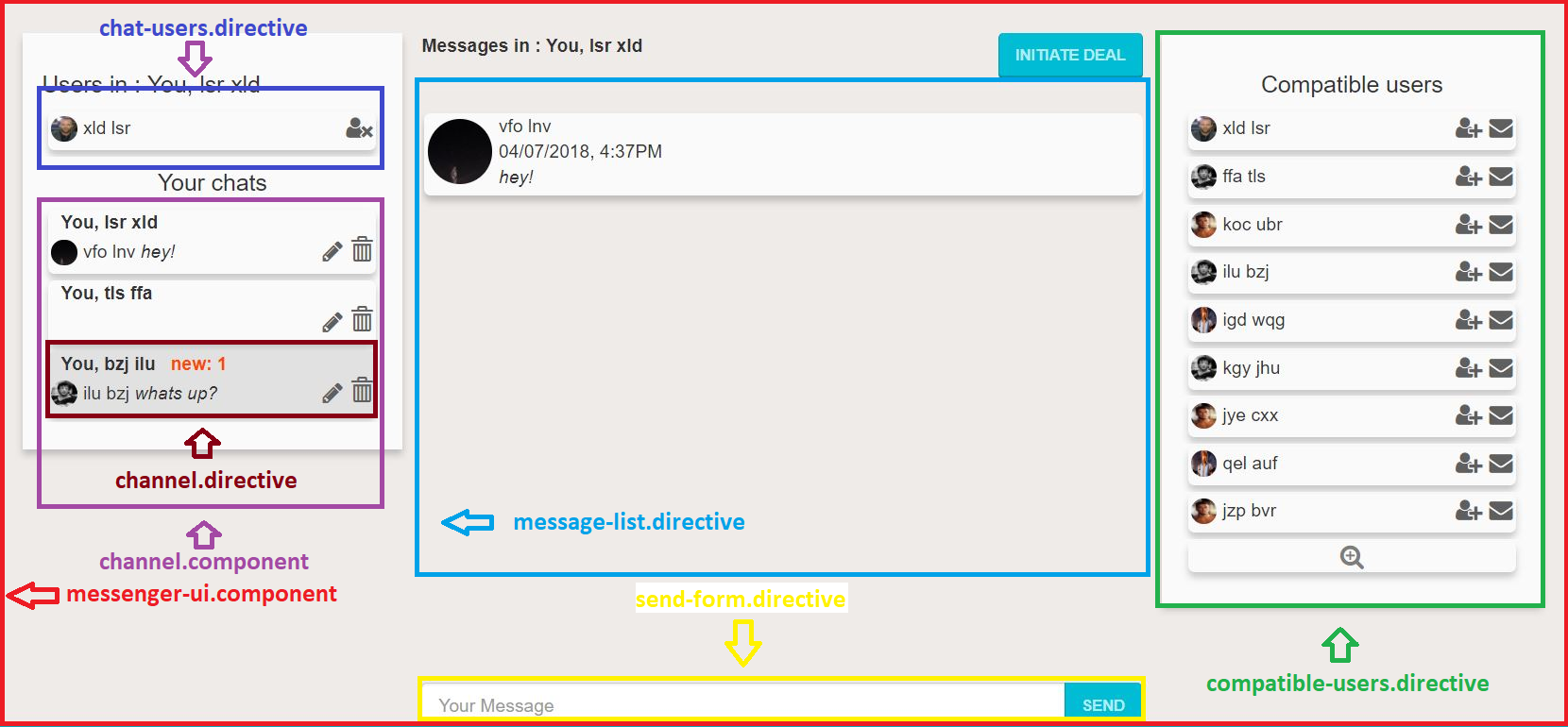
1. **Frontend Structure:**

**Services:**

* **si-gateway/si-gateway-webapp/src/client/src/config/websocket.service.js** – abstract service for creating socket, establishing websocket connection and subscribing channels
* **si-gateway/si-gateway-webapp/src/client/src/data/messenger-channel.service.js** – contains state for subscriptions, handles callbacks and subscribes channels.
* **si-gateway/si-gateway-webapp/src/client/src/data/messenger-current-data.service.js** – contains state of current chat, handles current chat callbacks and messages list
* **si-gateway/si-gateway-webapp/src/client/src/data/messenger-management.service.js** – contains state of all chats, activates messenger, deletes chats, send messages, adds chats, renames chats, adds users, deletes users.
* **si-gateway/si-gateway-webapp/src/client/src/data/notification-service.js** – chat notification system, subscribes internal channel that gets update messages from backend.

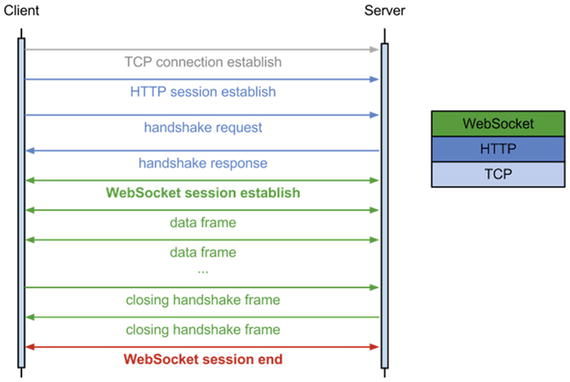
**Controllers and Templates:**

* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/messenger.model.js** – model for working with chat API.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/messenger-ui.component.js** – main component, contains information about current chat is registered as observer in messenger current data service.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/messenger-ui.component.html** – template for main component, contains all other components and directives.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/channel-component/channel.component.js** – contains all channels information, changes chat titles to user names if there is no chat title, is registered as observer in messenger channel service, updates list of chats when services notifies observers.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/channel-component/channel.component.html** – component for channel component, contains directive for each channel.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/channel-component/channel-directive/channel.directive.js** – directive for channel, registered as observer in messenger channel service. Activates channel when entered, marks messages as unread, queries list with message for current chat from backend, deletes chat and changes chat title in ng-dialog window.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/channel-component/channel-directive/channel.html** – template for channel directive.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/channel-component/channel-directive/chat-rename.popup.html** – template for ng-dialog window for renaming chat.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/chat-users/chat-users.directive.js** – registered as observer in messenger current data service, shows information about users in chat.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/chat-users/chat-users.html** – template for chat user directive.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/compatible-users/compatible-users.directive.js** – shows information about compatible users, creates new chat, adds new user to existing chat and changes state to user search if user search button pressed.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/compatible-users/compatible-users.html** – template for compatible users directive.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/message-list/message-list.directive.js** – shows message block with all messages for current chat, registered as observer in messenger current data service, marks messages as read, when they was shown and marks new messages as unread, when they was get from websocket.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/message-list/message-list.html** – template for message list directive.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/send-form/send-form.directive.js** – directive for ending messages over websocket, registered in messenger current data service and gets information about current chat.
* **si-gateway/si-gateway-webapp/src/client/src/partials/messenger/components/send-form/send-form.html** – template for send for directive.

****

1. **Authentication Flow:**

To understand following documentation part you need to have knowledge about bean post processors in spring boot applications and establishing WebSockets connections.



To secure handshake flow and upgrade from HTTP to WebSocket we need to pass SessionId. Unfortunately STOMP don’t allow us to pass any information to handshake because of browser restrictions. We decided to use path variable for this purpose:

**function** getSocket() {  
 **…**  
 **var url = '/api/socket?SESSION\_ID='+$sessionStorage.sessionId;**  
 **return new** SockJS(url, **null**, {transports: protocols, server: 'shutaf-in'});  
}

*(si-gateway/si-gateway-webapp/src/client/src/config/websocket.service.js)*

On backend side we are using implementation of HandshakeInterceptor:

public boolean beforeHandshake(ServerHttpRequest request, ServerHttpResponse response, WebSocketHandler wsHandler, Map<String, Object> attributes) throws Exception {  
 …  
 String sessionId = **servletRequest.getServletRequest().getParameter(*SESSION\_ID*)**;

*(com.shutafin.configuration.HttpSessionIdHandshakeInterceptor)*

If the sessionId is not correct interceptor throws an error in other way it adds userId as Socket attribute for later checks.

After upgrade process STOMP sends message with CONNECT header and Websocket service starts with subscriptions. Subscriptions are using SUBSCRIBE header. STOMP allows us to add custom headers, so we are able to pass sessionId as custom header:

**function** subscribe(destination) {  
 **…** vm.stompClient.subscribe(destination, **function** (message) {  
 deferred.notify(JSON.parse(message.body));  
 }, {**'session\_id': $sessionStorage.sessionId});**

*(si-gateway/si-gateway-webapp/src/client/src/config/websocket.service.js)*

Due to Spring Boot Messaging logic, we are able to intercept all inbound messages, CONNECT messages also:

public Message<?> preSend(Message<?> message, MessageChannel channel) {  
 …  
 if (accessor.getCommand() == **StompCommand.*CONNECT***||  
 accessor.getCommand() == **StompCommand.*SUBSCRIBE***) {

*(com.shutafin.configuration.ClientSessionInboundChannelInterceptor)*

Here we can check the sessionId and relationship between user and sessionId, so it shouldn’t be possible to use another’s sessionId to send messages.

After connection is established and subscriptions created, it is possible to send messages.

Each message should contain header with sessionId:

**function** sendMessage(message, address) {  
 vm.stompClient.send(address, {**'session\_id': $sessionStorage.sessionId**}, JSON.stringify(message));  
}

*(si-gateway/si-gateway-webapp/src/client/src/config/websocket.service.js)*

To get the userId in accordance with sessionId we are using annotation **@WebSocketAuthentication** to authenticate incoming chat messages.

This annotation is handled by: **com.shutafin.processors.AuthenticationAnnotationsBeanPostProcessor**

Details of authentication flow is not a part of this documentation, feel free to dive deeper into authentication with bean post processor.

Due to this annotation we will get userId inside parameters:

**@WebSocketAuthentication**

public ChatMessageResponse send(@DestinationVariable("chat\_id") Long chatId,  
 Message<ChatMessageRequest> message,  
 **@AuthenticatedUser Long userId**)

*(com.shutafin.controller.ChatController)*