Министерство науки и высшего образования Российской Федерации

федеральное государственное автономное

образовательное учреждение высшего образования

«Самарский национальный исследовательский университет имени академика С.П. Королева»

Институт информатики и кибернетики

Кафедра технической кибернетики

Финальный отчёт

Дисциплина: «Технологии сетевого программирования» Финализация приложения «Strollie - мессенджер»

Выполнили:

Ледяев Е.В.

Захаров Н.К.

Группа: 6301-010302D

6303-010302D

Самара, 2025

**ОПИСАНИЕ СТРУКТУРЫ КОНТЕЙНЕРИЗАЦИИ И НАСТРОЙКИ ОКРУЖЕНИЯ**

Контейнеризация приложения "Strollie - мессенджер" реализована с помощью инструмента Docker Compose, который обеспечивает удобную сборку и управление несколькими взаимосвязанными контейнерами. Для разработки использованы технологии Java Spring Boot, PostgreSQL, Keycloak, Maven (для сервиса сообщений) и Gradle (для клиентского сервиса).

Приложение включает следующие основные компоненты, определенные в файле docker-compose.yml:

1. **postgres-keycloak (PostgreSQL):**

* Используется стандартный образ PostgreSQL.
* Предназначен для хранения данных сервера аутентификации Keycloak.
* Переменные окружения (POSTGRES\_DB, POSTGRES\_USER, POSTGRES\_PASSWORD) определяют параметры базы данных для Keycloak.
* Порт на хосте: 5432.

2. **keycloak (Quay.io Keycloak):**

* Сервер аутентификации и авторизации.
* Использует базу данных postgres-keycloak для хранения своих данных.
* Настроен для работы с KC\_DB\_URL\_HOST: postgres-keycloak.
* Административные учетные данные (KEYCLOAK\_ADMIN, KEYCLOAK\_ADMIN\_PASSWORD) заданы как admin/admin.
* Доступен по адресу: http://localhost:8080 (благодаря network\_mode: "service:localhost").

3. **postgres-messenger (PostgreSQL):**

* Используется стандартный образ PostgreSQL.
* Хранит данные непосредственно самого мессенджера (чаты, сообщения, пользователи).
* Параметры подключения для сервиса messenger указаны в его конфигурационном файле (application.yaml) и ссылаются на этот контейнер.
* Порт на хосте: 5433.

4. **messenger (Spring Boot + Maven):**

* Основной API-сервис приложения.
* Реализует бизнес-логику мессенджера, включая управление чатами, отправку и получение сообщений.
* Сборка проекта осуществляется с помощью Maven и на основе Dockerfile.
* Подключается к postgres-messenger для хранения данных и к rabbitmq для обмена сообщениями.
* Защищен с помощью Keycloak (OAuth2 Resource Server).
* Доступен по адресу: http://localhost:8082

5. **client (Spring Boot + Gradle + Thymeleaf):**

* Клиентский веб-интерфейс пользователя.
* Предоставляет HTML-страницы для взаимодействия с мессенджером (список чатов, просмотр чата, создание чата).
* Сборка проекта осуществляется с помощью Gradle на основе Dockerfile.
* Взаимодействует с API сервиса messenger для получения данных и выполнения операций.
* Использует Thymeleaf для рендеринга HTML-страниц.
* Интегрирован с Keycloak для аутентификации пользователей (OAuth2 Client).
* Доступен по адресу: http://localhost:8090 (благодаря network\_mode: "service:localhost").

6. **localhost (Alpine):**

* Вспомогательный контейнер, используемый для настройки общего сетевого пространства localhost для сервисов keycloak и client, что упрощает их доступность с хост-машины по стандартным портам localhost:8080 и localhost:8090 соответственно.

**Процесс развертывания:**

Развертывание компонентов управляется механизмом depends\_on в файле docker-compose.yml, обеспечивая корректную последовательность запуска:

* Запускаются базы данных: postgres-keycloak и postgres-messenger.
* Запускается брокер сообщений rabbitmq.
* После успешного запуска postgres-keycloak, стартует сервис keycloak.
* После успешного запуска postgres-messenger и rabbitmq, стартует основной API-сервис messenger.
* После успешного запуска keycloak, стартует клиентский сервис client.

**АРХИТЕКТУРА ПРОЕКТА**

Проект "Strollie - мессенджер" представляет собой распределённую систему, состоящую из нескольких ключевых микросервисов и инфраструктурных компонентов. Каждый компонент выполняет свою специфическую роль и взаимодействует с другими для обеспечения полнофункциональной работы мессенджера.

**Компоненты архитектуры**

1. **База данных postgres-messenger**

* Роль: хранилище данных приложения.
* Описание: основное назначение — хранение информации о пользователях, чатах, сообщениях и участниках чатов. Использует реляционную модель данных. Схема базы данных управляется с помощью миграций Flyway.
* Технология: PostgreSQL.

2. **Сервис messenger (API-сервис и WebSocket)**

* Роль: бэкенд-сервис, реализующий основную бизнес-логику мессенджера и предоставляющий RESTful API и WebSocket для взаимодействия.
* Описание: обрабатывает запросы на создание чатов, отправку/получение сообщений, управление участниками. Интегрирован с RabbitMQ для асинхронной обработки сообщений и их доставки клиентам через WebSocket в реальном времени. Защищен с помощью Spring Security и Keycloak (OAuth2 Resource Server), принимая JWT токен.
* Технологии: Java, Spring Boot, Spring Data JPA (Hibernate), Spring Security, Spring WebSocket, Maven, Flyway.

3. **Сервис client (Клиентский веб-интерфейс)**

* Роль: пользовательский интерфейс мессенджера.
* Описание: предоставляет веб-страницы для входа, просмотра списка чатов, участия в чатах, отправки сообщений и создания новых чатов. Взаимодействует с сервисом messenger через REST API. Аутентификация пользователей осуществляется через Keycloak (OAuth2 Client).
* Технологии: Java, Spring Boot, Thymeleaf, Spring Security (OAuth2 Client), Gradle, HTML, CSS.

4. **Сервер аутентификации keycloak**

* Роль: централизованное управление пользователями, аутентификацией и авторизацией.
* Описание: отвечает за процессы входа пользователей, выдачу JWT токенов, которые затем используются для защиты API в сервисе messenger и аутентификации в client.
* Технология: Keycloak.

5. **База данных postgres-keycloak**

* Роль: хранилище данных для Keycloak.
* Описание: содержит информацию о пользователях, ролях, клиентах и других конфигурациях Keycloak.
* Технология: PostgreSQL.

Взаимодействие компонентов:

* Пользователь входит в систему через client, который перенаправляет его на keycloak для аутентификации.
* После успешной аутентификации keycloak выдает JWT токен.
* client использует этот токен для отправки запросов к messenger API.
* messenger валидирует токен с помощью keycloak и выполняет запрошенное действие, взаимодействуя с postgres-messenger для работы с данными.
* При отправке нового сообщения messenger публикует его в rabbitmq.
* messenger также использует WebSocket (через STOMP и RabbitMQ) для доставки сообщений клиентам в реальном времени. Клиенты (client через JavaScript в браузере) подписываются на соответствующие топики в rabbitmq через WebSocket соединение с messenger.

ПРИЛОЖЕНИЕ А

**Изображения веб интерфейса проекта и иных связанных программ с реализацией данного проекта**

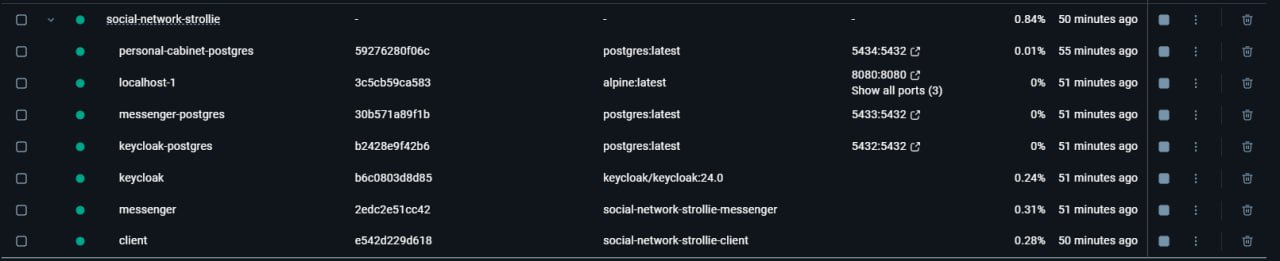
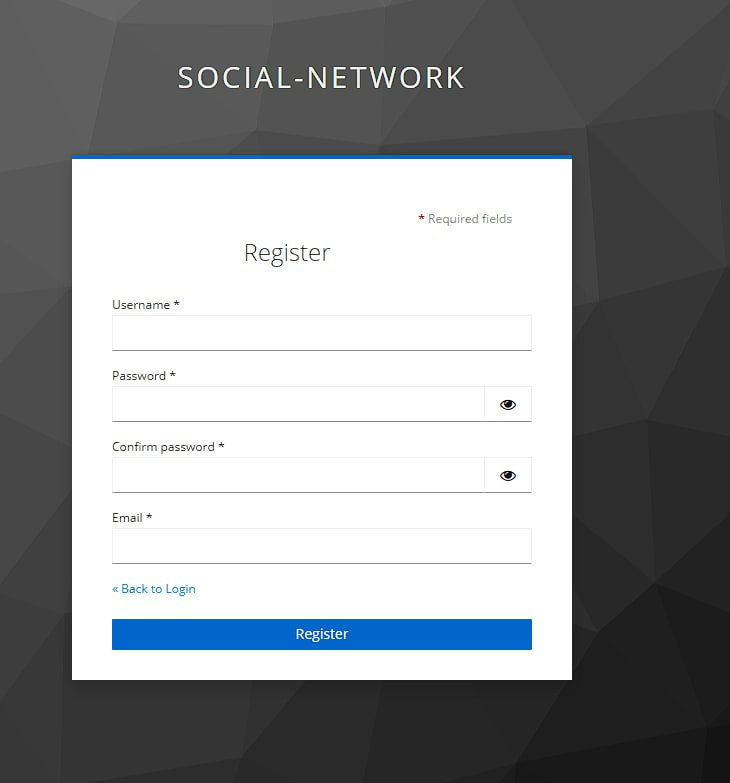
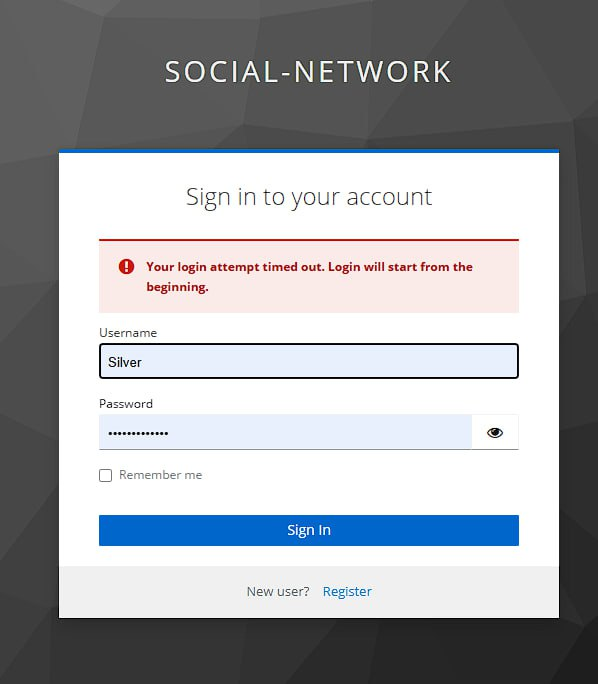
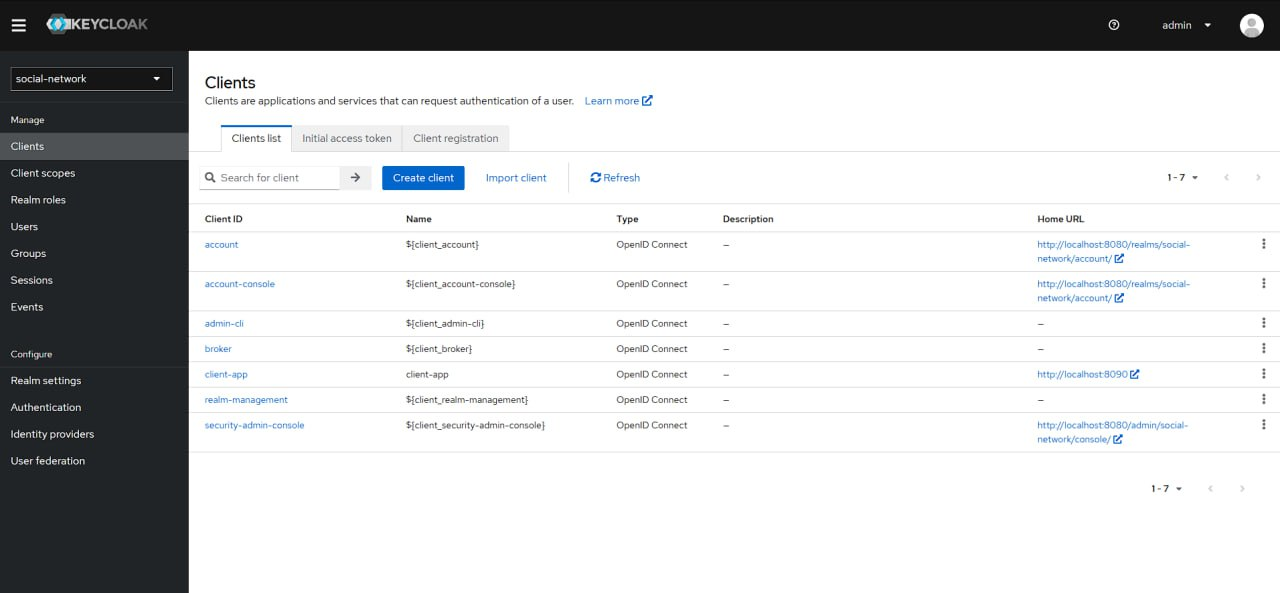
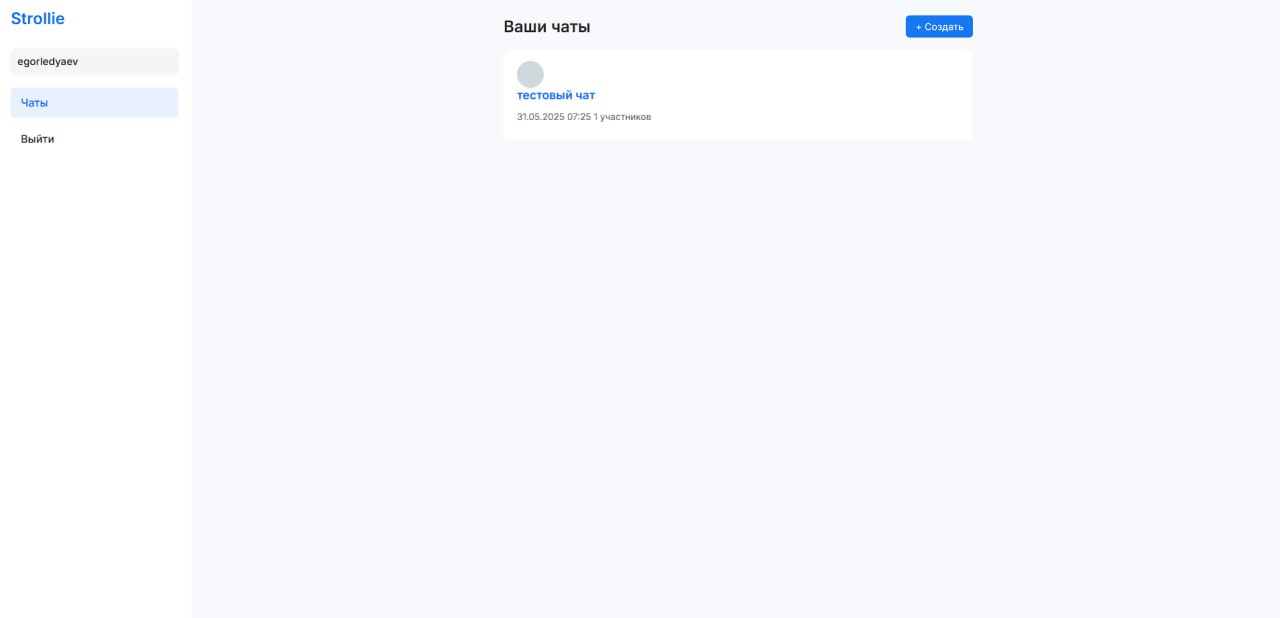


Рисунок 1 — Работающие контейнеры

Рисунок 2 — Страница регистрации аккаунта

Рисунок 3 — Страница входа в аккаунт с отображаемой ошибкой входа

Рисунок 4 — Страница настройки KeyCloak

Рисунок 5 — Страница главной страницы с чатами

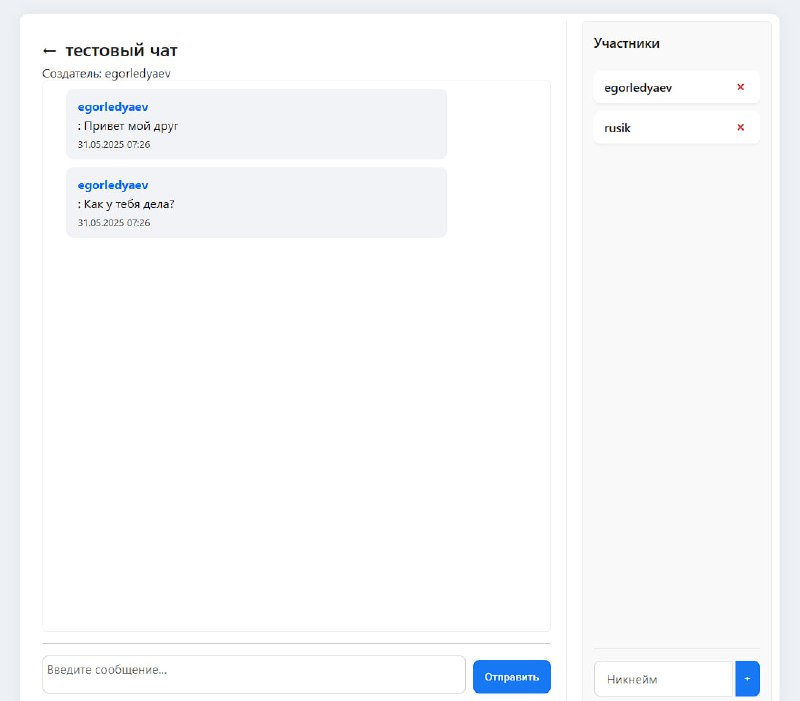


Рисунок 6 — Страница с открытым чатом и отправленным сообщениями

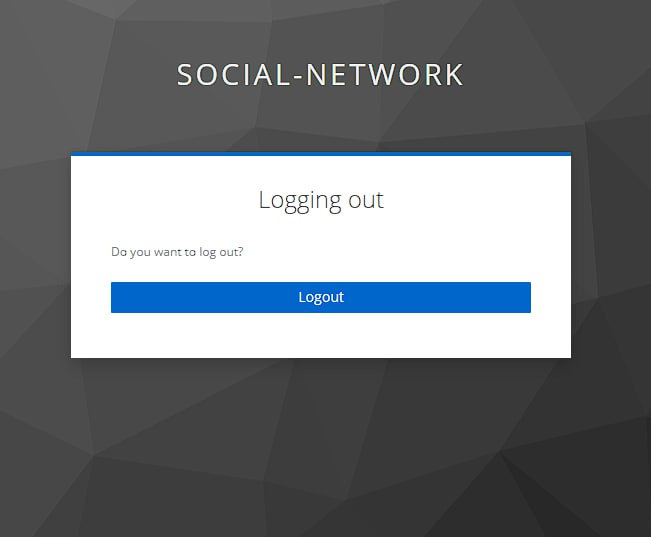


Рисунок 7 — Страница выхода

ПРИЛОЖЕНИЕ Б

**Экземпляры кода реализации проекта**

@Controller

@RequiredArgsConstructor

public class ChatsController {

private final RestClientMessenger restClientMessenger;

@GetMapping("/chats")

public String chats(Model model, Principal principal) {

String username = restClientMessenger.getUsernameById(UUID.fromString(principal.getName()));

model.addAttribute("username", username);

List<ChatDto> chats = restClientMessenger.getUserChats();

model.addAttribute("chats", chats);

return "chats";

}

@GetMapping("/chats/create")

public String createChatForm(Model model) {

model.addAttribute("chatForm", new CreateChatRequest());

return "create\_chat";

}

@PostMapping("/chats/create")

public String submitCreateChat(@ModelAttribute("chatForm") CreateChatRequest chatForm,

@RequestParam("participantUsernames") String participantUsernamesRaw) {

List<String> usernames = Arrays.stream(participantUsernamesRaw.split(","))

.map(String::trim)

.filter(s -> !s.isEmpty())

.collect(Collectors.toList());

chatForm.setParticipantUsernames(usernames);

restClientMessenger.createChat(chatForm);

return "redirect:/chats";

}

@GetMapping("/chats/{id}")

public String viewChat(@PathVariable UUID id, Model model) {

ChatDto chat = restClientMessenger.getChatById(id);

List<MessageDto> messages = restClientMessenger.getMessagesByChatId(id);

Collections.reverse(messages);

model.addAttribute("chat", chat);

model.addAttribute("author", restClientMessenger.getUsernameById(chat.getAuthorId()));

model.addAttribute("messages", messages);

return "chat";

}

@PostMapping("/chats/{id}/send")

public String sendMessage(@PathVariable UUID id, @RequestParam String content) {

SendMessageRequest request = new SendMessageRequest();

request.setChatId(id);

request.setContent(content);

restClientMessenger.sendMessage(request);

return "redirect:/chats/" + id;

}

@PostMapping("/chats/{id}/add-participant")

public String addParticipant(@PathVariable UUID id,

@RequestParam String username,

RedirectAttributes redirectAttributes) {

try {

restClientMessenger.addParticipant(id, username);

return "redirect:/chats/" + id;

} catch (Exception e) {

return "redirect:/errorAdd";

}

}

@PostMapping("/chats/{id}/remove-participant")

public String removeParticipant(@PathVariable UUID id,

@RequestParam String username,

RedirectAttributes redirectAttributes) {

try {

restClientMessenger.removeParticipant(id, username);

return "redirect:/chats/" + id;

} catch (Exception e) {

return "redirect:/errorRemove";

}

}

@GetMapping("/errorRemove")

public String errorRemovePage(@RequestParam(value = "error", required = false) String error, Model model) {

return "errorRemove";

}

@GetMapping("/errorAdd")

public String errorAddPage(@RequestParam(value = "error", required = false) String error, Model model) {

return "errorAdd";

}

@Configuration  
*public class* ClientBeans {  
  
 @Bean  
 *public* RestClientMessenger productsRestClient(  
 @Value("${messenger.service.base-url:http://localhost:8082}") String catalogueBaseUri,  
 *ClientRegistrationRepository* clientRegistrationRepository,  
 *OAuth2AuthorizedClientRepository* authorizedClientRepository,  
 @Value("${strollie.services.catalogue.registration-id:keycloak}") String registrationId) {  
 *return new* RestClientMessenger(*RestClient*.*builder*()  
 .baseUrl(catalogueBaseUri)  
 .requestInterceptor(  
 *new* OAuthClientHttpRequestInterceptor(  
 *new* DefaultOAuth2AuthorizedClientManager(clientRegistrationRepository,  
 authorizedClientRepository), registrationId))  
 .build());  
 }  
  
}  
  
@Configuration  
*public class* SecurityBeans {  
  
 @Bean  
 *public SecurityFilterChain* securityFilterChain(HttpSecurity http) *throws* Exception {  
 *return* http  
 .authorizeHttpRequests(authorize -> authorize  
 .requestMatchers("/", "/css/\*\*", "/js/\*\*").permitAll()  
 .anyRequest().authenticated()  
 ).logout(logout -> logout  
 .logoutSuccessUrl("/")  
 .invalidateHttpSession(*true*)  
 .clearAuthentication(*true*)  
 .deleteCookies("JSESSIONID")  
 .logoutSuccessHandler((request, response, authentication) -> {  
 String issuer = ((*OidcUser*) authentication.getPrincipal())  
 .getIssuer().toString();  
 String clientId = "client-app";  
 String redirectUri = "http://localhost:8090/";  
 String logoutUrl = issuer + "/protocol/openid-connect/logout" +  
 "?post\_logout\_redirect\_uri=" + redirectUri +  
 "&client\_id=" + clientId;  
 response.sendRedirect(logoutUrl);  
 })  
 )  
 .oauth2Login(*Customizer*.*withDefaults*())  
 .oauth2Client(*Customizer*.*withDefaults*())  
 .build();  
 }  
  
  
 @Bean  
 *public OAuth2UserService*<OidcUserRequest, *OidcUser*> oAuth2UserService() {  
 OidcUserService oidcUserService = *new* OidcUserService();  
 *return* userRequest -> {  
 *OidcUser* oidcUser = oidcUserService.loadUser(userRequest);  
 *List*<*GrantedAuthority*> authorities =  
 *Stream*.*concat*(oidcUser.getAuthorities().stream(),  
 Optional.*ofNullable*(oidcUser.getClaimAsStringList("groups"))  
 .orElseGet(*List*::*of*)  
 .stream()  
 .filter(role -> role.startsWith("ROLE\_"))  
 .map(SimpleGrantedAuthority::*new*)  
 .map(*GrantedAuthority*.*class*::cast))  
 .toList();  
 *return new* DefaultOidcUser(authorities, oidcUser.getIdToken(), oidcUser.getUserInfo());  
 };  
 }  
}

@RequiredArgsConstructor

public class OAuthClientHttpRequestInterceptor implements ClientHttpRequestInterceptor {

private final OAuth2AuthorizedClientManager authorizedClientManager;

private final String registrationId;

@Setter

private SecurityContextHolderStrategy securityContextHolderStrategy =

SecurityContextHolder.getContextHolderStrategy();

@Override

public ClientHttpResponse intercept(HttpRequest request, byte[] body, ClientHttpRequestExecution execution)

throws IOException {

if (!request.getHeaders().containsKey(HttpHeaders.AUTHORIZATION)) {

OAuth2AuthorizedClient authorizedClient = this.authorizedClientManager.authorize(

OAuth2AuthorizeRequest.withClientRegistrationId(this.registrationId)

.principal(this.securityContextHolderStrategy.getContext().getAuthentication())

.build());

request.getHeaders().setBearerAuth(authorizedClient.getAccessToken().getTokenValue());

}

return execution.execute(request, body);

}

}  
  
@Slf4j

@RestController

@RequestMapping("/api/v1/messenger")

@RequiredArgsConstructor

public class ChatController {

private final ChatService chatService;

@GetMapping("/users/{userId}/username")

public ResponseEntity<String> getUsernameById(@PathVariable UUID userId) {

try {

String username = chatService.getUsernameById(userId);

return ResponseEntity.ok(username);

} catch (IllegalArgumentException e) {

return ResponseEntity.status(HttpStatus.NOT\_FOUND).body(e.getMessage());

}

}

@PostMapping("/chats")

public ResponseEntity<ChatDto> createChat(@Valid @RequestBody CreateChatRequest request, Authentication authentication) {

UUID creatorId = PrincipalUtil.getUserId(authentication);

ChatDto chatDto = chatService.createChat(request, creatorId);

return ResponseEntity.status(HttpStatus.CREATED).body(chatDto);

}

@GetMapping("/chats/{chatId}")

public ResponseEntity<ChatDto> getChatById(@PathVariable UUID chatId, Authentication authentication) {

UUID userId = PrincipalUtil.getUserId(authentication);

ChatDto chatDto = chatService.getChatById(chatId, userId);

return ResponseEntity.ok(chatDto);

}

@GetMapping("/chats")

public ResponseEntity<List<ChatDto>> getUserChats(Authentication authentication) {

UUID userId = PrincipalUtil.getUserId(authentication);

List<ChatDto> chats = chatService.getUserChats(userId);

return ResponseEntity.ok(chats);

}

@GetMapping("/chats/{chatId}/messages")

public ResponseEntity<List<MessageDto>> getChatMessages(

@PathVariable UUID chatId,

Authentication authentication) {

UUID userId = PrincipalUtil.getUserId(authentication);

List<MessageDto> messages = chatService.getAllMessagesForChat(chatId, userId);

return ResponseEntity.ok(messages);

}

@PostMapping("/messages")

public ResponseEntity<MessageDto> sendHttpMessage(@Valid @RequestBody SendMessageRequest request, Authentication authentication) {

UUID senderId = PrincipalUtil.getUserId(authentication);

String senderUsername = PrincipalUtil.getUsername(authentication);

MessageDto messageDto = chatService.sendMessage(senderId, senderUsername, request);

return ResponseEntity.status(HttpStatus.CREATED).body(messageDto);

}

@PostMapping("/chats/{chatId}/participants")

public ResponseEntity<?> addParticipant(

@PathVariable UUID chatId,

@RequestParam String username,

Authentication authentication) {

try {

UUID requesterId = PrincipalUtil.getUserId(authentication);

ChatDto chatDto = chatService.addParticipant(chatId, username, requesterId);

return ResponseEntity.ok(chatDto);

} catch (IllegalArgumentException e) {

String message = e.getMessage() != null && !e.getMessage().isEmpty() ? e.getMessage() : "Неверные параметры запроса.";

return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body(message);

} catch (AccessDeniedException e) {

String message = e.getMessage() != null && !e.getMessage().isEmpty() ? e.getMessage() : "Доступ запрещен.";

return ResponseEntity.status(HttpStatus.FORBIDDEN).body(message);

} catch (Exception e) {

log.error("Error adding participant: chatId={}, username={}", chatId, username, e);

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body("Внутренняя ошибка сервера при добавлении участника.");

}

}

@DeleteMapping("/chats/{chatId}/participants")

public ResponseEntity<?> removeParticipant(

@PathVariable UUID chatId,

@RequestParam String username,

Authentication authentication) {

try {

UUID requesterId = PrincipalUtil.getUserId(authentication);

ChatDto chatDto = chatService.removeParticipant(chatId, username, requesterId);

return ResponseEntity.ok(chatDto);

} catch (IllegalArgumentException e) {

String message = e.getMessage() != null && !e.getMessage().isEmpty() ? e.getMessage() : "Неверные параметры запроса.";

return ResponseEntity.status(HttpStatus.BAD\_REQUEST).body(message);

} catch (AccessDeniedException e) {

String message = e.getMessage() != null && !e.getMessage().isEmpty() ? e.getMessage() : "Доступ запрещен.";

return ResponseEntity.status(HttpStatus.FORBIDDEN).body(message);

} catch (Exception e) {

log.error("Error removing participant: chatId={}, username={}", chatId, username, e);

return ResponseEntity.status(HttpStatus.INTERNAL\_SERVER\_ERROR).body("Внутренняя ошибка сервера при удалении участника.");

}

}

}  
  
@Component

@RequiredArgsConstructor

public class UserExistenceInterceptor implements HandlerInterceptor {

private final UserRepository userRepository;

@Override

public boolean preHandle(HttpServletRequest request, HttpServletResponse response, Object handler) throws Exception {

Authentication authentication = SecurityContextHolder.getContext().getAuthentication();

if (authentication != null && authentication.isAuthenticated()) {

UUID userId = PrincipalUtil.getUserId(authentication);

String username = PrincipalUtil.getUsername(authentication);

boolean exists = userRepository.existsById(userId);

if (!exists) {

UserEntity user = new UserEntity();

user.setId(userId);

user.setUsername(username);

userRepository.save(user);

}

}

return true;

}

}

@Service

@RequiredArgsConstructor

public class ChatService {

private final ChatRepository chatRepository;

private final MessageRepository messageRepository;

private final ParticipantsRepository participantsRepository;

private final UserRepository userRepository;

public String getUsernameById(UUID userId) {

return userRepository.findById(userId)

.map(UserEntity::getUsername)

.orElseThrow(() -> new IllegalArgumentException("User not found with id: " + userId));

}

@Transactional

public ChatDto createChat(CreateChatRequest request, UUID creatorId) {

ChatEntity chat = new ChatEntity();

chat.setAuthorId(creatorId);

chat.setChatName(request.getChatName());

chat.setLastActivityAt(Instant.now());

if (request.getParticipantUsernames().size() > 1 &&

(request.getChatName() == null || request.getChatName().isBlank())) {

throw new IllegalArgumentException("Chat name is required for group chats.");

}

ChatEntity savedChat = chatRepository.save(chat);

List<UUID> participantIds = userRepository.findAllByUsernameIn(request.getParticipantUsernames())

.stream()

.map(UserEntity::getId)

.collect(Collectors.toList());

if (!participantIds.contains(creatorId)) {

participantIds.add(creatorId);

}

List<ParticipantsEntity> participants = participantIds.stream()

.map(userId -> {

ParticipantsEntity participant = new ParticipantsEntity();

participant.setId(new ParticipantsEntityId(savedChat.getId(), userId));

return participant;

})

.collect(Collectors.toList());

participantsRepository.saveAll(participants);

return mapToChatDto(savedChat, request.getParticipantUsernames());

}

public List<ChatDto> getUserChats(UUID userId) {

List<ChatEntity> chats = chatRepository.findChatsByUserId(userId);

return chats.stream()

.map(chat -> {

List<UUID> participantIds = participantsRepository.findById\_ChatId(chat.getId())

.stream()

.map(p -> p.getId().getUserId())

.collect(Collectors.toList());

List<String> participantUsernames = userRepository.findAllById(participantIds)

.stream()

.map(UserEntity::getUsername)

.collect(Collectors.toList());

return mapToChatDto(chat, participantUsernames);

})

.collect(Collectors.toList());

}

@Transactional

public MessageDto sendMessage(UUID senderId, String senderUsername, SendMessageRequest request) {

if (!isUserParticipant(request.getChatId(), senderId)) {

throw new AccessDeniedException("User is not a participant of this chat.");

}

MessageEntity message = new MessageEntity();

message.setChatId(request.getChatId());

message.setSenderId(senderId);

message.setContent(request.getContent());

message.setTimestamp(Instant.now());

MessageEntity savedMessage = messageRepository.save(message);

chatRepository.findById(request.getChatId()).ifPresent(chat -> {

chat.setLastActivityAt(savedMessage.getTimestamp());

chatRepository.save(chat);

});

MessageDto messageDto = mapToMessageDto(savedMessage, senderUsername);

String routingKey = "chat.message." + request.getChatId().toString();

System.out.println("Sent message to RabbitMQ: " + routingKey + " Payload: " + messageDto);

return messageDto;

}

public List<MessageDto> getAllMessagesForChat(UUID chatId, UUID userId) {

if (!participantsRepository.existsById\_ChatIdAndId\_UserId(chatId, userId)) {

throw new AccessDeniedException("User does not have access to this chat");

}

List<MessageEntity> messages = messageRepository.findByChatIdOrderByTimestampDesc(chatId);

return messages.stream()

.map(this::mapEntityToDto)

.collect(Collectors.toList());

}

private MessageDto mapEntityToDto(MessageEntity entity) {

return MessageDto.builder()

.id(entity.getId())

.chatId(entity.getChatId())

.senderId(entity.getSenderId())

.senderUsername(userRepository.findById(entity.getSenderId()).get().getUsername())

.content(entity.getContent())

.timestamp(entity.getTimestamp())

.build();

}

public boolean isUserParticipant(UUID chatId, UUID userId) {

return participantsRepository.existsById\_ChatIdAndId\_UserId(chatId, userId);

}

private ChatDto mapToChatDto(ChatEntity chatEntity, List<String> participantName) {

return ChatDto.builder()

.id(chatEntity.getId())

.authorId(chatEntity.getAuthorId())

.chatName(chatEntity.getChatName())

.lastActivityAt(chatEntity.getLastActivityAt())

.participantsNames(participantName)

.build();

}

private MessageDto mapToMessageDto(MessageEntity messageEntity, String senderUsername) {

return MessageDto.builder()

.id(messageEntity.getId())

.chatId(messageEntity.getChatId())

.senderId(messageEntity.getSenderId())

.senderUsername(senderUsername) // Placeholder or passed in

.content(messageEntity.getContent())

.timestamp(messageEntity.getTimestamp())

.build();

}

public ChatDto getChatById(UUID chatId, UUID userUuid) {

UserEntity user = userRepository.findById(userUuid)

.orElseThrow(() -> new IllegalArgumentException("User not found"));

String username = user.getUsername();

ChatEntity chatEntity = chatRepository.findById(chatId)

.orElseThrow(() -> new IllegalArgumentException("Chat not found"));

List<String> participantUsernames = participantsRepository.findById\_ChatId(chatId)

.stream()

.map(p -> userRepository.findById(p.getId().getUserId()).get().getUsername()) // Предполагаем, что участник связан с UserEntity

.collect(Collectors.toList());

return mapToChatDto(chatEntity, participantUsernames);

}

@Transactional

public ChatDto addParticipant(UUID chatId, String username, UUID requesterId) {

ChatEntity chat = chatRepository.findById(chatId)

.orElseThrow(() -> new IllegalArgumentException("Chat not found"));

if (!isUserParticipant(chatId, requesterId)) {

throw new AccessDeniedException("User is not a participant of this chat");

}

UserEntity userToAdd = userRepository.findByUsername((username))

.orElseThrow(() -> new IllegalArgumentException("User not found"));

if (isUserParticipant(chatId, userToAdd.getId())) {

throw new IllegalArgumentException("User is already a participant");

}

ParticipantsEntity participant = new ParticipantsEntity();

participant.setId(new ParticipantsEntityId(chatId, userToAdd.getId()));

participantsRepository.save(participant);

chat.setLastActivityAt(Instant.now());

chatRepository.save(chat);

return getChatById(chatId, requesterId);

}

@Transactional

public ChatDto removeParticipant(UUID chatId, String username, UUID requesterId) {

ChatEntity chat = chatRepository.findById(chatId)

.orElseThrow(() -> new IllegalArgumentException("Chat not found"));

if (!chat.getAuthorId().equals(requesterId)) {

throw new AccessDeniedException("Only chat owner can remove participants");

}

UserEntity userToRemove = userRepository.findByUsername(username)

.orElseThrow(() -> new IllegalArgumentException("User not found"));

if (userToRemove.getId().equals(requesterId)) {

throw new IllegalArgumentException("Chat owner cannot remove themselves");

}

ParticipantsEntityId participantId = new ParticipantsEntityId(chatId, userToRemove.getId());

if (!participantsRepository.existsById(participantId)) {

throw new IllegalArgumentException("User is not a participant of this chat");

}

participantsRepository.deleteById(participantId);

chat.setLastActivityAt(Instant.now());

chatRepository.save(chat);

return getChatById(chatId, requesterId);

}

}