

Projeto Final

RGM 27588254 - Jonas Durau

RGM 25990403 - Kaio Mauricio Lima Fernandes

Códigos de import não devem ser adicionados ao projeto

Descrição do que o código executa: Texto (Arial 10) - Margens de: 1.54 cm, Alinhamento a esquerda, Espaçamento de 1,15.

1 Receive-Send-API

Pasta clients:

AuthCliente.java

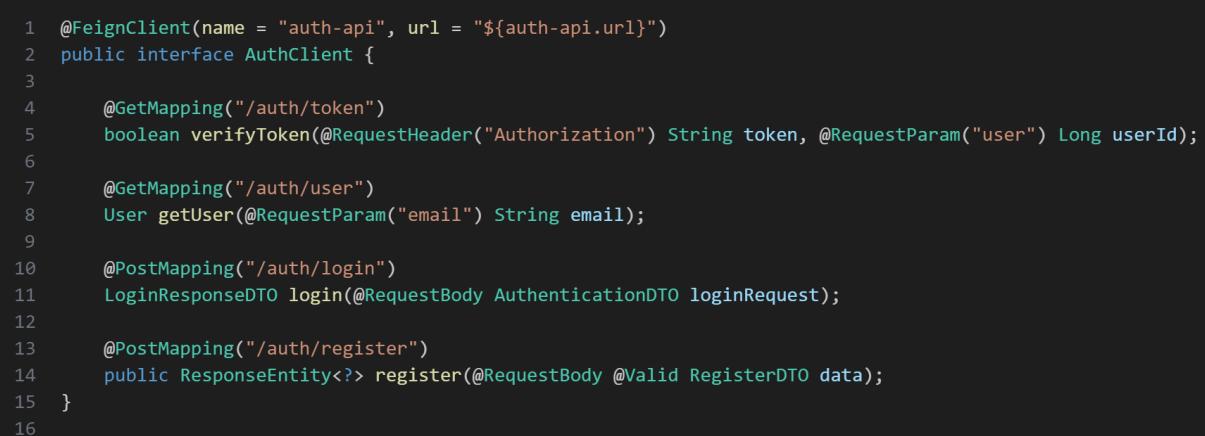
Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\clients\AuthClient.java

Descrição:

O código abaixo define um cliente Feign para se comunicar com a API de autenticação (auth-api). Este cliente Feign fornece métodos para verificar tokens de autenticação, obter detalhes do usuário com base no e-mail, efetuar login e registrar novos usuários. A interface AuthClient utiliza anotações do Spring para mapear endpoints HTTP na API de autenticação.

- **verifyToken:** Verifica a validade de um token de autenticação para um usuário específico.
- **getUser:** Obtém as informações do usuário com base no e-mail fornecido.
- **login:** Efetua o login de um usuário utilizando suas credenciais.
- **register:** Registra um novo usuário na API de autenticação.



```
1  @FeignClient(name = "auth-api", url = "${auth-api.url}")
2  public interface AuthClient {
3
4      @GetMapping("/auth/token")
5      boolean verifyToken(@RequestHeader("Authorization") String token, @RequestParam("user") Long userId);
6
7      @GetMapping("/auth/user")
8      User getUser(@RequestParam("email") String email);
9
10     @PostMapping("/auth/login")
11     LoginResponseDTO login(@RequestBody AuthenticationDTO loginRequest);
12
13     @PostMapping("/auth/register")
14     public ResponseEntity<?> register(@RequestBody @Valid RegisterDTO data);
15  }
16
```

RecordCliente.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\clients\RecordClient.java

Descrição:

O código abaixo define um cliente Feign para se comunicar com a API de registro (record-api). Este cliente Feign fornece métodos para salvar mensagens e obter mensagens para um usuário específico. A interface RecordClient utiliza anotações do Spring para mapear endpoints HTTP na API de registro.

- **saveMessage**: Envia uma solicitação POST para salvar uma mensagem no sistema de registro.
- **getMessages**: Envia uma solicitação GET para obter todas as mensagens recebidas por um usuário específico.

```

1  @FeignClient(name = "record-api", url = "${record-api.url}")
2  public interface RecordClient {
3
4      @PostMapping("/record/message")
5      void saveMessage(@RequestBody MessageDTO messageDTO);
6
7      @GetMapping("/record/messages")
8      List<Message> getMessages(@RequestParam("user") Long userId);
9  }

```

Pasta config:

RabbitMQConfig.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\config\RabbitMQConfig.java

Descrição:

O código abaixo configura o RabbitMQ para ser utilizado em um projeto Spring. Ele define uma fila chamada "messageQueue" e configura a conversão de mensagens para JSON usando o Jackson. Além disso, ele cria um template RabbitTemplate para enviar mensagens ao RabbitMQ.

- **messageQueue**: Cria uma fila durável chamada "messageQueue".
- **jackson2JsonMessageConverter**: Configura a conversão de mensagens para JSON usando Jackson.
- **rabbitTemplate**: Cria e configura um template RabbitTemplate para enviar mensagens ao RabbitMQ, utilizando o conversor de mensagens JSON configurado.

```

1  @Configuration
2  public class RabbitMQConfig {
3
4      @Bean
5      public Queue messageQueue() {
6          return new Queue("messageQueue", true);
7      }
8
9      @Bean
10     public MessageConverter jackson2JsonMessageConverter() {
11         return new Jackson2JsonMessageConverter();
12     }
13
14     @Bean
15     public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
16         RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
17         rabbitTemplate.setMessageConverter(jackson2JsonMessageConverter());
18         return rabbitTemplate;
19     }
20 }

```

Pasta controllers:

MessageController.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\config\RabbitMQConfig.java

Descrição:

O código abaixo define o controlador MessageController que gerencia endpoints relacionados a mensagens, registro e login de usuários. Ele utiliza clientes Feign para comunicação com outras APIs e RabbitTemplate para enviar mensagens para a fila do RabbitMQ.

- **AuthClient e RecordClient:** Usados para comunicação com APIs de autenticação e de registro de mensagens.
- **RabbitTemplate e Queue:** Usados para enviar e processar mensagens através do RabbitMQ.

Endpoints:

- **/register:** Registra um novo usuário.
- **/login:** Realiza o login de um usuário.
- **/message:** Envia uma mensagem para a fila do RabbitMQ.
- **/worker:** Processa mensagens da fila e as salva no banco de dados.
- **/messages:** Retorna todas as mensagens recebidas por um usuário específico.

```
1  @RestController
2  @RequestMapping("/message")
3  public class MessageController {
4
5      @Autowired
6      private AuthClient authClient;
7
8      @Autowired
9      private RecordClient recordClient;
10
11     @Autowired
12     private RabbitTemplate rabbitTemplate;
13
14     @Autowired
15     private Queue messageQueue;
16
17     @PostMapping("/register")
18     public ResponseEntity<?> register(@RequestBody RegisterDTO registerRequest) {
19         ResponseEntity<?> registerResponse = authClient.register(registerRequest);
20         return registerResponse;
21     }
22
23     @PostMapping("/login")
24     public ResponseEntity<?> login(@RequestBody AuthenticationDTO loginRequest) {
25         LoginResponseDTO loginResponse = authClient.login(loginRequest);
26         return ResponseEntity.ok(loginResponse);
27     }
28
29     @PostMapping
30     public ResponseEntity<?> sendMessage(@RequestHeader("Authorization") String token, @RequestBody MessageDTO messageDTO) {
31         boolean isAuthenticated = authClient.verifyToken(token, messageDTO.getUserIdSend());
32         if (!isAuthenticated) {
33             return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("not auth");
34         }
35         // Envia mensagem para a fila
36         rabbitTemplate.convertAndSend("messageQueue", messageDTO);
37         return ResponseEntity.ok("message sent with success");
38     }
39
40     @PostMapping("/worker")
41     public ResponseEntity<?> processMessages(@RequestHeader("Authorization") String token, @RequestBody WorkerDTO workerDTO) {
42         boolean isAuthenticated = authClient.verifyToken(token, workerDTO.getUserIdSend());
43         if (!isAuthenticated) {
44             return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("not auth");
45         }
46
47         // Processa mensagens da fila
48         MessageDTO messageDTO = (MessageDTO) rabbitTemplate.receiveAndConvert(messageQueue.getName());
49         if (messageDTO != null) {
50             recordClient.saveMessage(messageDTO);
51             return ResponseEntity.ok("Message processed and saved");
52         } else {
53             return ResponseEntity.status(HttpStatus.NO_CONTENT).body("No messages in the queue");
54         }
55     }
56
57     @GetMapping
58     public ResponseEntity<?> getMessages(@RequestHeader("Authorization") String token, @RequestParam("user") Long userId) {
59         boolean isAuthenticated = authClient.verifyToken(token, userId);
60         if (!isAuthenticated) {
61             return ResponseEntity.status(HttpStatus.UNAUTHORIZED).body("not auth");
62         }
63         List<Message> messages = recordClient.getMessages(userId);
64         return ResponseEntity.ok(messages);
65     }
66 }
```

Pasta DTOs:

AuthenticationDTO.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\DTOs\AuthenticationDTO.java

Descrição:

A classe AuthenticationDTO é um objeto de transferência de dados (DTO) utilizado para encapsular as informações de autenticação de um usuário. Ela implementa a interface Serializable para permitir que suas instâncias sejam convertidas em um fluxo de bytes, facilitando o envio de dados através da rede ou a persistência em um armazenamento. Esta classe possui dois campos: login e password, representando as credenciais do usuário. Abaixo está o código completo da classe:

- **Construtor sem argumentos:** Necessário para a deserialização e frameworks de injeção de dependências.
- **Construtor com argumentos:** Facilita a criação de instâncias com valores iniciais para login e password.
- **Getters e Setters:** Métodos de acesso e modificação para os campos login e password.

```
 1  public class AuthenticationDTO implements Serializable {
 2      private static final long serialVersionUID = 1L;
 3
 4      private String login;
 5      private String password;
 6
 7      public AuthenticationDTO() {
 8
 9
10      public AuthenticationDTO(String login, String password) {
11          this.login = login;
12          this.password = password;
13      }
14
15      public String getLogin() {
16          return login;
17      }
18
19      public void setLogin(String login) {
20          this.login = login;
21      }
22
23      public String getPassword() {
24          return password;
25      }
26
27      public void setPassword(String password) {
28          this.password = password;
29      }
30  }
```

MessageDTO.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\DTOs\MessageDTO.java

Descrição:

A classe MessageDTO é um objeto de transferência de dados (DTO) usado para encapsular as informações de uma mensagem trocada entre usuários na aplicação. Implementa a interface Serializable para permitir que suas instâncias sejam convertidas em um fluxo de bytes, facilitando o transporte de dados através da rede ou a persistência em um armazenamento. Abaixo está o código completo da classe:

Campos:

- **userIdSend**: ID do usuário que enviou a mensagem.
- **userIdReceive**: ID do usuário que recebeu a mensagem.
- **message**: Conteúdo da mensagem.
- **Construtor sem argumentos**: Necessário para a deserialização e frameworks de injecão de dependências.
- **Construtor com argumentos**: Facilita a criação de instâncias com valores iniciais para userIdSend, userIdReceive e message.
- **Getters e Setters**: Métodos de acesso e modificação para os campos userIdSend, userIdReceive e message.



```
1  public class MessageDTO implements Serializable {
2      private static final long serialVersionUID = 1L;
3
4      private Long userIdSend;
5      private Long userIdReceive;
6      private String message;
7
8      public MessageDTO() {
9
10
11     public MessageDTO(Long userIdSend, Long userIdReceive, String message) {
12         this.userIdSend = userIdSend;
13         this.userIdReceive = userIdReceive;
14         this.message = message;
15     }
16
17     public Long getUserIdSend() {
18         return userIdSend;
19     }
20     public void setUserIdSend(Long userIdSend) {
21         this.userIdSend = userIdSend;
22     }
23     public Long getUserIdReceive() {
24         return userIdReceive;
25     }
26     public void setIdReceive(Long userIdReceive) {
27         this.userIdReceive = userIdReceive;
28     }
29     public String getMessage() {
30         return message;
31     }
32     public void setMessage(String message) {
33         this.message = message;
34     }
35 }
```

RegisterDTO.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\DTOS\RegisterDTO.java

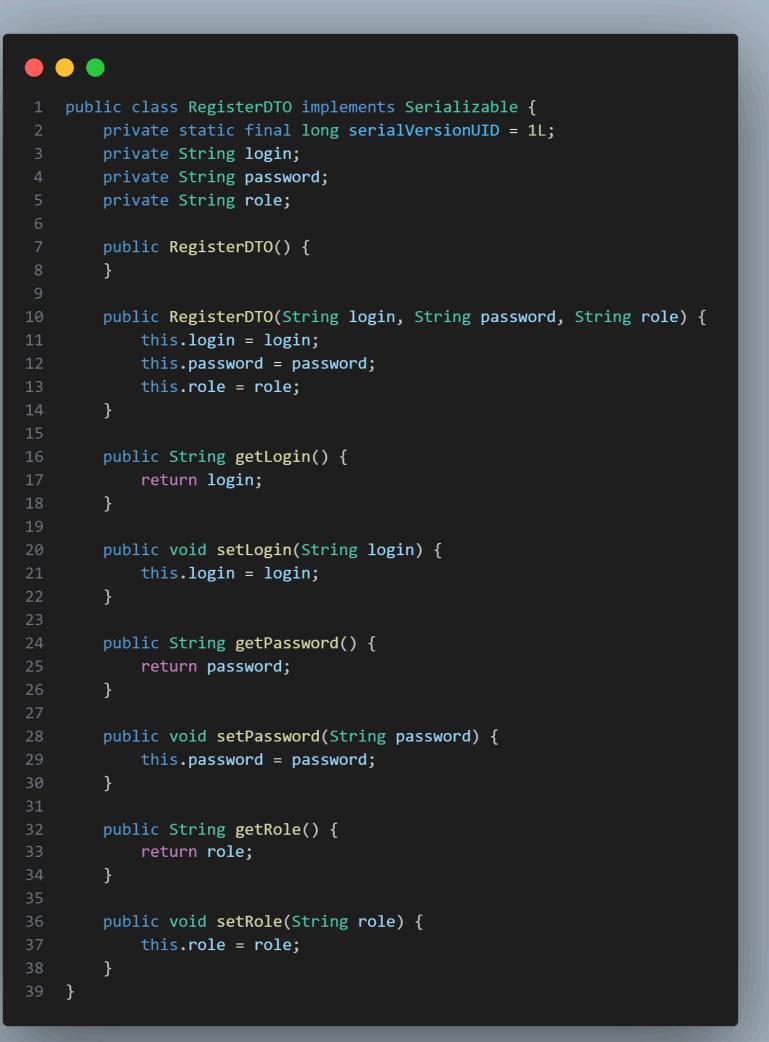
Descrição:

A classe RegisterDTO é um objeto de transferência de dados (DTO) utilizado para encapsular as informações necessárias para o registro de um novo usuário no sistema. Implementa a interface Serializable, permitindo que suas instâncias sejam convertidas em um fluxo de bytes, o que facilita o transporte de dados através da rede ou a persistência em um armazenamento. Abaixo está o código completo da classe:

Campos:

- **login:** Nome de usuário ou email usado para o registro.
- **password:** Senha do usuário.
- **role:** Papel ou função do usuário no sistema (por exemplo, ROLE_USER, ROLE_ADMIN).
- **Construtor sem argumentos:** Necessário para a deserialização e frameworks de injecão de dependências.
- **Construtor com argumentos:** Facilita a criação de instâncias com valores iniciais para login, password e role.
- **Getters e Setters:** Métodos de acesso e modificação para os campos login, password e role.

Esses métodos permitem a manipulação segura dos dados encapsulados na instância da classe, promovendo boas práticas de encapsulamento e abstração na programação orientada a objetos.



```
1  public class RegisterDTO implements Serializable {
2      private static final long serialVersionUID = 1L;
3      private String login;
4      private String password;
5      private String role;
6
7      public RegisterDTO() {
8      }
9
10     public RegisterDTO(String login, String password, String role) {
11         this.login = login;
12         this.password = password;
13         this.role = role;
14     }
15
16     public String getLogin() {
17         return login;
18     }
19
20     public void setLogin(String login) {
21         this.login = login;
22     }
23
24     public String getPassword() {
25         return password;
26     }
27
28     public void setPassword(String password) {
29         this.password = password;
30     }
31
32     public String getRole() {
33         return role;
34     }
35
36     public void setRole(String role) {
37         this.role = role;
38     }
39 }
```

WorkerDTO.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\DTOS\WorkerDTO.java

Descrição:

A classe WorkerDTO é um objeto de transferência de dados (DTO) utilizado para encapsular as informações necessárias para identificar as mensagens enviadas e recebidas por um usuário específico no sistema. Implementa a interface Serializable, permitindo que suas instâncias sejam convertidas em um fluxo de bytes, o que facilita o transporte de dados através da rede ou a persistência em um armazenamento. Abaixo está o código completo da classe:

Campos:

- **userIdSend**: Identificador do usuário que enviou a mensagem.
- **userIdReceive**: Identificador do usuário que recebeu a mensagem.
- **Construtor sem argumentos**: Necessário para a deserialização e frameworks de injecção de dependências.
- **Construtor com argumentos**: Facilita a criação de instâncias com valores iniciais para userIdSend e userIdReceive.
- **Getters e Setters**: Métodos de acesso e modificação para os campos userIdSend e userIdReceive.

Esses métodos permitem a manipulação segura dos dados encapsulados na instância da classe, promovendo boas práticas de encapsulamento e abstração na programação orientada a objetos

```
1  public class WorkerDTO implements Serializable {
2      private static final long serialVersionUID = 1L;
3
4      private Long userIdSend;
5      private Long userIdReceive;
6
7      public WorkerDTO() {
8      }
9
10     public WorkerDTO(Long userIdSend, Long userIdReceive) {
11         this.userIdSend = userIdSend;
12         this.userIdReceive = userIdReceive;
13     }
14
15     public Long getUserIdSend() {
16         return userIdSend;
17     }
18
19     public void setUserIdSend(Long userIdSend) {
20         this.userIdSend = userIdSend;
21     }
22
23     public Long getUserIdReceive() {
24         return userIdReceive;
25     }
26
27     public void setUserIdReceive(Long userIdReceive) {
28         this.userIdReceive = userIdReceive;
29     }
30 }
31
```

Pasta models:

Message.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\models\Message.java

Descrição:

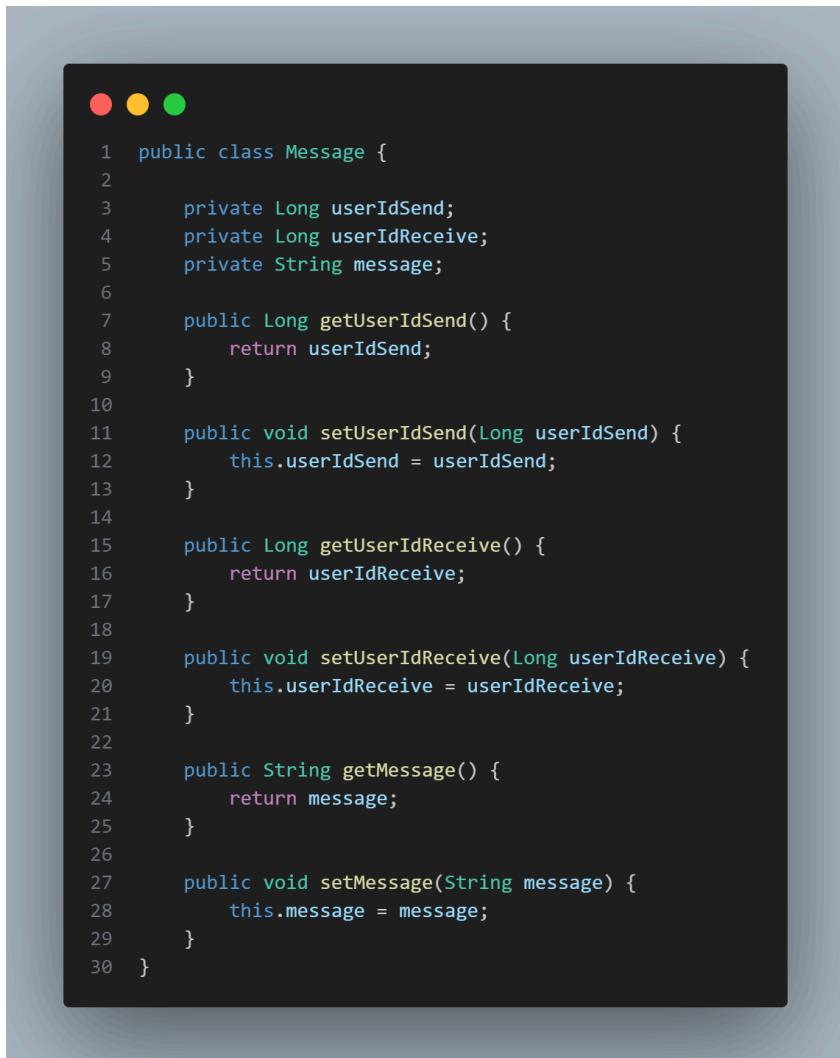
A classe Message representa uma entidade que encapsula os dados de uma mensagem no sistema de comunicação. Esta classe não só armazena o conteúdo da mensagem, mas também os identificadores dos usuários que enviaram e receberam a mensagem. Abaixo está o código completo da classe:

Campos:

- **userIdSend**: Identificador do usuário que enviou a mensagem.
- **userIdReceive**: Identificador do usuário que recebeu a mensagem.
- **message**: Conteúdo da mensagem.

Métodos Getters e Setters:

- **getUserIdSend()**: Retorna o identificador do usuário que enviou a mensagem.
- **setUserIdSend(Long userIdSend)**: Define o identificador do usuário que enviou a mensagem.
- **getUserIdReceive()**: Retorna o identificador do usuário que recebeu a mensagem.
- **setUserIdReceive(Long userIdReceive)**: Define o identificador do usuário que recebeu a mensagem.
- **getMessage()**: Retorna o conteúdo da mensagem.
- **setMessage(String message)**: Define o conteúdo da mensagem.



```
1 public class Message {  
2  
3     private Long userIdSend;  
4     private Long userIdReceive;  
5     private String message;  
6  
7     public Long getUserIdSend() {  
8         return userIdSend;  
9     }  
10    public void setUserIdSend(Long userIdSend) {  
11        this.userIdSend = userIdSend;  
12    }  
13    public Long getUserIdReceive() {  
14        return userIdReceive;  
15    }  
16    public void setUserIdReceive(Long userIdReceive) {  
17        this.userIdReceive = userIdReceive;  
18    }  
19    public String getMessage() {  
20        return message;  
21    }  
22    public void setMessage(String message) {  
23        this.message = message;  
24    }  
25 }  
26  
27 }  
28 }
```

User.java

Endereço do Arquivo:

receive-send-api\src\main\java\com\integracao_de_sistemas\receive_send_api\models\User.java

Descrição:

A classe User representa um usuário no sistema de comunicação, encapsulando informações pessoais e credenciais. Esta classe é essencial para gerenciar a identificação e autenticação dos usuários. Abaixo está o código completo da classe:

Campos:

- **id**: Identificador único do usuário.
- **name**: Nome do usuário.
- **lastName**: Sobrenome do usuário.
- **email**: Endereço de e-mail do usuário, usado para login e comunicação.
- **password**: Senha do usuário, usada para autenticação.

Construtores:

- **User()**: Construtor padrão sem parâmetros.
- **User(Long id, String name, String lastName, String email, String password)**: Construtor com parâmetros para inicializar todos os campos da classe.

Métodos:

- Métodos Getters e Setters.

```
1  public class User {  
2  
3      private Long id;  
4      private String name;  
5      private String lastName;  
6      private String email;  
7      private String password;  
8  
9      public User() {  
10     }  
11  
12     public User(Long id, String name, String lastName, String email, String password) {  
13         this.id = id;  
14         this.name = name;  
15         this.lastName = lastName;  
16         this.email = email;  
17         this.password = password;  
18     }  
19  
20     public Long getId() {  
21         return id;  
22     }  
23  
24     public void setId(Long id) {  
25         this.id = id;  
26     }  
27  
28     public String getName() {  
29         return name;  
30     }  
31  
32     public void setName(String name) {  
33         this.name = name;  
34     }  
35  
36     public String getLastname() {  
37         return lastName;  
38     }  
39  
40     public void setLastName(String lastName) {  
41         this.lastName = lastName;  
42     }  
43  
44     public String getEmail() {  
45         return email;  
46     }  
47  
48     public void setEmail(String email) {  
49         this.email = email;  
50     }  
51  
52     public String getPassword() {  
53         return password;  
54     }  
55  
56     public void setPassword(String password) {  
57         this.password = password;  
58     }  
59 }
```

2 Auth-API

Pasta config:

SecurityConfigurations.java

Endereço do Arquivo:

auth-api\src\main\java\com\integracao_de_sistemas\auth_api\config\SecurityConfigurations.java

Descrição:

Esta classe é responsável pela configuração de segurança do Spring Security na aplicação. Inclui várias configurações importantes, tais como:

- **Configuração do SecurityFilterChain:** Define uma política de segurança para as requisições HTTP. Desabilita a proteção CSRF, define a política de gerenciamento de sessões como STATELESS (sem estado) e permite requisições HTTP POST para os endpoints /auth/login e /auth/register sem autenticação. Todas as outras requisições precisam ser autenticadas.
- **Autenticação:** Configura um gerenciador de autenticação (AuthenticationManager) utilizando a configuração de autenticação padrão do Spring.
- **Codificador de Senhas:** Define um codificador de senhas usando BCryptPasswordEncoder para armazenar e validar senhas de forma segura.



```
1  @Configuration
2  @EnableWebSecurity
3  public class SecurityConfigurations {
4
5      @Autowired
6      private SecurityFilter securityFilter;
7
8      @Bean
9      public SecurityFilterChain securityFilterChain(HttpSecurity httpSecurity) throws Exception {
10         return httpSecurity
11             .csrf(csrf -> csrf.disable())
12             .sessionManagement(session -> session.sessionCreationPolicy(SessionCreationPolicy.STATELESS))
13             .authorizeHttpRequests(authorize -> authorize
14                 .requestMatchers(HttpMethod.POST, "/auth/login").permitAll()
15                 .requestMatchers(HttpMethod.POST, "/auth/register").permitAll()
16                 .anyRequest().authenticated()
17             )
18             .addFilterBefore(securityFilter, UsernamePasswordAuthenticationFilter.class)
19             .build();
20     }
21
22     @Bean
23     public AuthenticationManager authenticationManager(AuthenticationConfiguration authenticationConfiguration) throws Exception {
24         return authenticationConfiguration.getAuthenticationManager();
25     }
26
27     @Bean
28     public PasswordEncoder encoder() {
29         return new BCryptPasswordEncoder();
30     }
31 }
```

- **@Configuration:** Indica que esta classe é uma classe de configuração Spring.
- **@EnableWebSecurity:** Habilita a segurança web no projeto.
- **securityFilterChain(HttpSecurity httpSecurity):** Método que define a configuração do filtro de segurança.
- **authenticationManager(AuthenticationConfiguration authenticationConfiguration):** Método que fornece o gerenciador de autenticação.
- **encoder():** Método que define o codificador de senhas usando BCrypt.

SecurityFilter.java

Endereço do Arquivo: auth-api\src\main\java\com\integracao_de_sistemas\auth_api\config\SecurityFilter.java

Descrição:

A classe SecurityFilter é um filtro de segurança personalizado que estende OncePerRequestFilter para interceptar todas as requisições HTTP e validar o token JWT. Esta classe é um componente Spring e está configurada para executar uma vez por requisição:

- **Intercepção de Requisições:** O método doFilterInternal intercepta todas as requisições HTTP, extrai o token JWT do cabeçalho de autorização, valida o token e autentica o usuário no contexto de segurança do Spring.
- **Recuperação do Token:** O método recoverToken extrai o token JWT do cabeçalho de autorização da requisição.

```
1 @Component
2 public class SecurityFilter extends OncePerRequestFilter {
3     @Autowired
4     TokenService tokenService;
5     @Autowired
6     UserRepository userRepository;
7
8     @Override
9     protected void doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain) throws ServletException, IOException {
10         var token = this.recoverToken(request);
11         if(token != null){
12             var email = tokenService.getEmailFromToken(token);
13             UserDetails user = userRepository.findByEmail(email);
14
15             var authentication = new UsernamePasswordAuthenticationToken(user, null, user.getAuthorities());
16             SecurityContextHolder.getContext().setAuthentication(authentication);
17         }
18         filterChain.doFilter(request, response);
19     }
20
21     private String recoverToken(HttpServletRequest request){
22         var authHeader = request.getHeader("Authorization");
23         if(authHeader == null) return null;
24         return authHeader.replace("Bearer ", "");
25     }
26 }
```

- **@Component:** Indica que esta classe é um componente gerenciado pelo Spring.
- **doFilterInternal(HttpServletRequest request, HttpServletResponse response, FilterChain filterChain):** Método que executa o filtro de segurança. Se o token JWT for válido, ele define a autenticação no contexto de segurança.
- **recoverToken(HttpServletRequest request):** Método que recupera o token JWT do cabeçalho de autorização da requisição HTTP.

TokenService.java

Endereço do Arquivo: auth-api\src\main\java\com\integracao_de_sistemas\auth_api\config\TokenService.java

Descrição:

A classe TokenService é um serviço Spring responsável por gerar, validar e extrair informações de tokens JWT. Ela utiliza a biblioteca java-jwt para manipulação dos tokens e é configurada com uma chave secreta definida no arquivo de propriedades da aplicação.

Geração de Token: Gera tokens JWT assinados com HMAC utilizando uma chave secreta.

Validação de Token: Valida tokens JWT, verificando a assinatura e comparando o e-mail do usuário com o ID do usuário no banco de dados.

Extração de E-mail: Extrai o e-mail do usuário a partir de um token JWT válido.

Geração de Data de Expiração: Define a data de expiração dos tokens JWT para duas horas após a geração.

generateToken(User user):

- Utiliza a biblioteca java-jwt para criar um token JWT assinado.
- Define o emissor (issuer) do token como "auth-api" e o assunto (subject) como o e-mail do usuário.
- Define a data de expiração do token para duas horas após a criação.
- Assina o token utilizando a chave secreta.

validateToken(String token, Long userId):

- Verifica a assinatura do token JWT utilizando a chave secreta.
- Extrai o e-mail do usuário do token e busca o usuário no banco de dados.
- Compara o ID do usuário no banco de dados com o ID fornecido para validação.

getEmailFromToken(String token):

- Verifica a assinatura do token JWT utilizando a chave secreta.
- Extrai e retorna o e-mail do usuário do token.

generateExpirationDate():

- Gera a data de expiração para o token, definida para duas horas após o momento da geração.

Tratamento de Exceções

- JWTCreationException: Lançada durante a geração do token caso ocorra algum erro.
- JWTVerificationException: Lançada durante a verificação do token caso ele seja inválido.



```
1  @Service
2  public class TokenService {
3
4      @Value("${api.security.token.secret}")
5      private String secret;
6
7      private final UserRepository userRepository;
8
9      public TokenService(UserRepository userRepository) {
10          this.userRepository = userRepository;
11      }
12
13      public String generateToken(User user){
14          try {
15              Algorithm algorithm = Algorithm.HMAC256(secret);
16              return JWT.create()
17                  .withIssuer("auth-api")
18                  .withSubject(user.getEmail())
19                  .withExpiresAt(generateExpirationDate())
20                  .sign(algorithm);
21          } catch (JWTCreationException exception) {
22              throw new RuntimeException("Error while generating token", exception);
23          }
24      }
25
26      public boolean validateToken(String token, Long userId) {
27          try {
28              Algorithm algorithm = Algorithm.HMAC256(secret);
29              String email = JWT.require(algorithm)
30                  .withIssuer("auth-api")
31                  .build()
32                  .verify(token.replace("Bearer ", ""))
33                  .getSubject();
34              User user = userRepository.findByEmail(email);
35              return user != null && user.getId().equals(userId);
36          } catch (JWTVerificationException exception){
37              return false;
38          }
39      }
40
41      public String getEmailFromToken(String token) {
42          try {
43              Algorithm algorithm = Algorithm.HMAC256(secret);
44              return JWT.require(algorithm)
45                  .withIssuer("auth-api")
46                  .build()
47                  .verify(token.replace("Bearer ", ""))
48                  .getSubject();
49          } catch (JWTVerificationException exception) {
50              throw new RuntimeException("Invalid token", exception);
51          }
52      }
53
54      private Instant generateExpirationDate(){
55          return LocalDateTime.now().plusHours(2).toInstant(ZoneOffset.of("-03:00"));
56      }
57  }
```

Pasta controllers:

AuthenticationController.java

Endereço do Arquivo:

auth-api\src\main\java\com\integracao_de_sistemas\auth_api\controllers\AuthenticationController.java

Descrição:

A classe AuthenticationController é um controlador REST do Spring responsável por gerenciar a autenticação de usuários. Este controlador expõe endpoints para login, registro de novos usuários e verificação de tokens JWT.

login(@RequestBody @Valid AuthenticationDTO data):

- **Endpoint:** POST /auth/login
- **Descrição:** Autentica o usuário com base nas credenciais fornecidas.
- **Processo:**
 - Busca o usuário pelo e-mail.
 - Verifica se a senha fornecida corresponde à senha armazenada, utilizando o PasswordEncoder.
 - Gera um token JWT em caso de sucesso e retorna no corpo da resposta.
 - Retorna um erro 403 se a autenticação falhar.

register(@RequestBody @Valid RegisterDTO data):

- **Endpoint:** POST /auth/register
- **Descrição:** Registra um novo usuário no sistema.
- **Processo:**
 - Verifica se o e-mail já está registrado.
 - Criptografa a senha utilizando o PasswordEncoder.
 - Cria e salva o novo usuário no banco de dados.
 - Retorna uma resposta de sucesso ou de erro em caso de falha.

verifyToken(@RequestHeader("Authorization") String token, @RequestParam("user") Long userId):

- **Endpoint:** GET /auth/token
- **Descrição:** Verifica a validade de um token JWT para um usuário específico.
- **Processo:**
 - Valida o token utilizando o TokenService.
 - Retorna um valor booleano indicando a validade do token.



```
1  @RestController
2  @RequestMapping("auth")
3  public class AuthenticationController {
4
5      @Autowired
6      private AuthenticationManager authenticationManager;
7
8      @Autowired
9      private UserService userService;
10
11     @Autowired
12     private TokenService tokenService;
13
14     @Autowired
15     private PasswordEncoder passwordEncoder; // Adicione isso
16
17     private static final Logger logger = LoggerFactory.getLogger(AuthenticationController.class);
18
19     @PostMapping("/login")
20     public ResponseEntity<?> login(@RequestBody @Valid AuthenticationDTO data){
21         logger.info("Attempting login for user: " + data.getLogin());
22         try {
23             User user = userService.getUserByEmail(data.getLogin());
24             if (user != null) {
25                 logger.info("User found: " + user.getEmail());
26                 if (passwordEncoder.matches(data.getPassword(), user.getPassword())) {
27                     var token = tokenService.generateToken(user);
28                     logger.info("Login successful for user: " + data.getLogin());
29                     return ResponseEntity.ok(new LoginResponseDTO(token));
30                 } else {
31                     logger.error("Password mismatch for user: " + data.getLogin());
32                 }
33             } else {
34                 logger.error("User not found: " + data.getLogin());
35             }
36             return ResponseEntity.status(403).body("Authentication failed");
37         } catch (Exception e) {
38             logger.error("Login failed for user: " + data.getLogin(), e);
39             return ResponseEntity.status(403).body("Authentication failed");
40         }
41     }
42
43     @PostMapping("/register")
44     public ResponseEntity<?> register(@RequestBody @Valid RegisterDTO data){
45         if(this.userService.getUserByEmail(data.getLogin()) != null) return ResponseEntity.badRequest().build();
46
47         String encryptedPassword = passwordEncoder.encode(data.getPassword());
48         User newUser = new User(data.getLogin(), encryptedPassword, data.getRole());
49         this.userService.createUser(newUser);
50
51         logger.info("User created: " + newUser.getEmail() + " with encrypted password: " + newUser.getPassword());
52         return ResponseEntity.ok().build();
53     }
54
55     @GetMapping("/token")
56     public ResponseEntity<Boolean> verifyToken(@RequestHeader("Authorization") String token,
57                                                 @RequestParam("user") Long userId) {
58         boolean isValid = tokenService.validateToken(token, userId);
59         return ResponseEntity.ok(isValid);
60     }
61 }
```

Pasta DTOs:

AuthenticationDTO.java

Endereço do Arquivo:

auth-api\src\main\java\com\integracao_de_sistemas\auth_api\controllers\AuthenticationController.java

Descrição:

A classe AuthenticationDTO é um objeto de transferência de dados (DTO) utilizado para encapsular as informações de autenticação de um usuário. Esta classe é serializável, o que permite que seus objetos sejam convertidos em um fluxo de bytes, facilitando a transmissão entre diferentes camadas da aplicação, como do cliente para o servidor em uma API REST.

Armazenamento de Credenciais: A classe armazena as credenciais de login do usuário, especificamente o e-mail (ou nome de usuário) e a senha.

Encapsulamento de Dados: Fornece métodos getters e setters para acessar e modificar as credenciais de forma controlada.

Construtores

- **AuthenticationDTO():** Construtor padrão que inicializa um objeto AuthenticationDTO vazio.
- **AuthenticationDTO(String login, String password):** Construtor que inicializa um objeto AuthenticationDTO com o e-mail e a senha fornecidos.

Métodos

- **getLogin():** Retorna o e-mail ou nome de usuário.
- **setLogin(String login):** Define o e-mail ou nome de usuário.
- **getPassword():** Retorna a senha.
- **setPassword(String password):** Define a senha.

```
1  public class AuthenticationDTO implements Serializable {
2      private static final long serialVersionUID = 1L;
3
4      private String login;
5      private String password;
6
7      public AuthenticationDTO() {
8
9
10     public AuthenticationDTO(String login, String password) {
11         this.login = login;
12         this.password = password;
13     }
14
15     public String getLogin() {
16         return login;
17     }
18
19     public void setLogin(String login) {
20         this.login = login;
21     }
22
23     public String getPassword() {
24         return password;
25     }
26
27     public void setPassword(String password) {
28         this.password = password;
29     }
30 }
```

LoginResponseDTO.java

Endereço do Arquivo:

auth-api\src\main\java\com\integracao_de_sistemas\auth_api\DTOs>LoginResponseDTO.java

Descrição:

LoginResponseDTO é utilizada principalmente nos endpoints de login da aplicação. Quando um usuário faz login com sucesso, um objeto LoginResponseDTO é criado com o token JWT gerado e enviado ao cliente, que utilizará este token para autorizar futuras requisições:

- **LoginResponseDTO(String token):** Construtor que inicializa um objeto LoginResponseDTO com o token fornecido.
- **getToken():** Retorna o token JWT.
- **setToken(String token):** Define o token JWT.



```
1 public class LoginResponseDTO implements Serializable {
2     private static final long serialVersionUID = 1L;
3     private String token;
4
5     public LoginResponseDTO(String token) {
6         this.token = token;
7     }
8
9     public String getToken() {
10        return token;
11    }
12
13    public void setToken(String token) {
14        this.token = token;
15    }
16 }
```

LoginResponseDTO.java

Endereço do Arquivo: auth-api\src\main\java\com\integracao_de_sistemas\auth_api\DTOs\RegisterDTO.java

Descrição:

A RegisterDTO é utilizada principalmente no endpoint de registro da aplicação. Quando um novo usuário se registra, um objeto RegisterDTO é criado com os dados fornecidos pelo usuário. Este objeto é então enviado para o serviço de registro, que processa e armazena os dados no banco de dados.

Essa classe garante que os dados de registro sejam transmitidos de forma segura e organizada, facilitando a manutenção e expansão da funcionalidade de registro da aplicação.

Atributos

- **login:** Armazena o nome de usuário ou endereço de e-mail usado para login.
- **password:** Armazena a senha do usuário, que será posteriormente criptografada antes de ser armazenada no banco de dados.
- **role:** Define o papel do usuário na aplicação (por exemplo, "ROLE_USER", "ROLE_ADMIN").

Construtores

- **RegisterDTO():** Construtor padrão sem argumentos, necessário para frameworks que criam instâncias da classe via reflexão.
- **RegisterDTO(String login, String password, String role):** Construtor completo que inicializa todos os campos com os valores fornecidos.

Métodos

- `getLogin():` Retorna o login do usuário.
- `setLogin(String login):` Define o login do usuário.
- `getPassword():` Retorna a senha do usuário.
- `setPassword(String password):` Define a senha do usuário.
- `getRole():` Retorna o papel do usuário.
- `setRole(String role):` Define o papel do usuário



The screenshot shows a code editor window with a dark theme. At the top left, there are three circular icons: red, yellow, and green. The code editor displays the following Java class definition:

```
1 public class RegisterDTO implements Serializable {
2     private static final long serialVersionUID = 1L;
3     private String login;
4     private String password;
5     private String role;
6
7     public RegisterDTO() {
8
9
10    public RegisterDTO(String login, String password, String role) {
11        this.login = login;
12        this.password = password;
13        this.role = role;
14    }
15
16    public String getLogin() {
17        return login;
18    }
19
20    public void setLogin(String login) {
21        this.login = login;
22    }
23
24    public String getPassword() {
25        return password;
26    }
27
28    public void setPassword(String password) {
29        this.password = password;
30    }
31
32    public String getRole() {
33        return role;
34    }
35
36    public void setRole(String role) {
37        this.role = role;
38    }
39 }
40 }
```

Pasta models:

User.java

Endereço do Arquivo: auth-api\src\main\java\com\integracao_de_sistemas\auth_api\models\User.java

Descrição:

A classe User é usada para autenticação e autorização no Spring Security, bem como para persistir informações de usuários no banco de dados. Ela encapsula todas as informações necessárias sobre um usuário, incluindo credenciais e permissões, e fornece métodos para integrar essas informações com o sistema de segurança da aplicação.

Construtores

- **User():** Construtor padrão sem argumentos, necessário para JPA.
- **User(String email, String password, String role):** Construtor que inicializa os campos email, password e role com os valores fornecidos.

Métodos

- Getters e Setters
- **getId():** Retorna o ID do usuário.
- **setId(Long id):** Define o ID do usuário.
- **getEmail():** Retorna o e-mail do usuário.
- **setEmail(String email):** Define o e-mail do usuário.
- **getPassword():** Retorna a senha do usuário.
- **setPassword(String password):** Define a senha do usuário.
- **getRole():** Retorna o papel do usuário.
- **setRole(String role):** Define o papel do usuário.

Métodos de UserDetails

- **getAuthorities():** Retorna uma coleção de autoridades concedidas ao usuário. Neste caso, retorna uma lista contendo uma única autoridade baseada no papel do usuário.
- **getUsername():** Retorna o e-mail do usuário como o nome de usuário.
- **isAccountNonExpired():** Indica se a conta do usuário está expirada. Sempre retorna true, significando que a conta nunca expira.
- **isAccountNonLocked():** Indica se a conta do usuário está bloqueada. Sempre retorna true, significando que a conta nunca está bloqueada.
- **isCredentialsNonExpired():** Indica se as credenciais do usuário (senha) estão expiradas. Sempre retorna true, significando que as credenciais nunca expiram.
- **isEnabled():** Indica se a conta do usuário está habilitada. Sempre retorna true, significando que a conta sempre está habilitada.

```
1  @Entity
2  @Table(name = "tb_users")
3  public class User implements UserDetails {
4
5      @Id
6      @GeneratedValue(strategy = GenerationType.IDENTITY)
7      private Long id;
8
9      private String email;
10     private String password;
11     private String role;
12
13     public User() {}
14
15     public User(String email, String password, String role) {
16         this.email = email;
17         this.password = password;
18         this.role = role;
19     }
20
21     // Getters and setters
22
23     public Long getId() {
24         return id;
25     }
26
27     public void setId(Long id) {
28         this.id = id;
29     }
30
31     public String getEmail() {
32         return email;
33     }
34
35     public void setEmail(String email) {
36         this.email = email;
37     }
38
39     public String getPassword() {
40         return password;
41     }
42
43     public void setPassword(String password) {
44         this.password = password;
45     }
46
47     public String getRole() {
48         return role;
49     }
50
51     public void setRole(String role) {
52         this.role = role;
53     }
54
55     @Override
56     public Collection<? extends GrantedAuthority> getAuthorities() {
57         return Collections.singletonList(new SimpleGrantedAuthority(role));
58     }
59
60     @Override
61     public String getUsername() {
62         return email;
63     }
64
65     @Override
66     public boolean isAccountNonExpired() {
67         return true;
68     }
69
70     @Override
71     public boolean isAccountNonLocked() {
72         return true;
73     }
74
75     @Override
76     public boolean isCredentialsNonExpired() {
77         return true;
78     }
79
80     @Override
81     public boolean isEnabled() {
82         return true;
83     }
84 }
```

Pasta repositories:

UserUserRepository.java

Endereço do Arquivo:

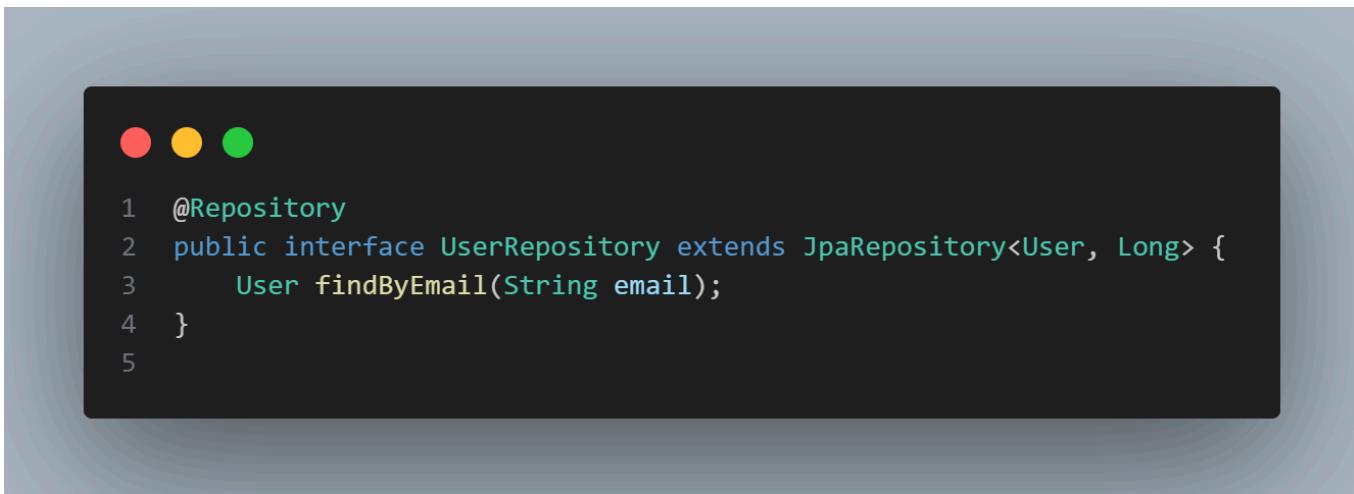
auth-api\src\main\java\com\integracao_de_sistemas\auth_api\repositories\UserRepository.java

Descrição:

A interface UserRepository é um repositório Spring Data JPA que fornece operações de persistência para a entidade User. Ele estende a interface JpaRepository, que inclui métodos CRUD (Create, Read, Update, Delete) prontos para uso e funcionalidades adicionais para trabalhar com a entidade User.

Método

findByEmail(): Este método é usado para encontrar um usuário com base no endereço de e-mail. O Spring Data JPA gera automaticamente a implementação desse método com base na assinatura e no nome do método, fornecendo uma maneira conveniente de realizar consultas personalizadas.



Pasta services:

UserService.java

Endereço do Arquivo: auth-api\src\main\java\com\integracao_de_sistemas\auth_api\services\UserService.java

Descrição:

A classe UserService é um serviço responsável pela lógica de negócios relacionada à entidade User. Implementa a interface UserDetailsService do Spring Security, permitindo a integração com o mecanismo de autenticação do Spring. O principal propósito do UserService é gerenciar operações de usuário, incluindo validação de credenciais, criação de novos usuários e recuperação de usuários por e-mail. Além disso, ele implementa o método loadUserByUsername para fornecer detalhes do usuário ao Spring Security durante o processo de autenticação.

Métodos

- **public User validateUser(String email, String password)**: Este método verifica se o usuário existe e se a senha fornecida corresponde à senha armazenada. Ele retorna o objeto User se as credenciais forem válidas; caso contrário, retorna null.
- **public User createUser(User user)**: Este método salva um novo usuário no banco de dados e registra a criação do usuário no log. Ele retorna o objeto User salvo.
- **public User getUserByEmail(String email)**: Este método recupera um usuário com base no endereço de e-mail fornecido. Ele retorna o objeto User correspondente.
- **public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException**: Este método é utilizado pelo Spring Security para carregar os detalhes do usuário durante o processo de autenticação. Ele lança uma exceção UsernameNotFoundException se o usuário não for encontrado.

```

1  @Service
2  public class UserService implements UserDetailsService {
3
4      @Autowired
5      private UserRepository userRepository;
6
7      @Autowired
8      private PasswordEncoder passwordEncoder;
9
10     private static final Logger logger = LoggerFactory.getLogger(AuthenticationController.class);
11
12     public User validateUser(String email, String password) {
13         User user = userRepository.findByEmail(email);
14         if (user != null && passwordEncoder.matches(password, user.getPassword())) {
15             return user;
16         }
17         return null;
18     }
19
20     public User createUser(User user) {
21         User savedUser = userRepository.save(user);
22         logger.info("User created: " + savedUser.getEmail() + " with encrypted password: " + savedUser.getPassword());
23         return savedUser;
24     }
25
26     public User getUserByEmail(String email) {
27         return userRepository.findByEmail(email);
28     }
29
30     @Override
31     public UserDetails loadUserByUsername(String email) throws UsernameNotFoundException {
32         User user = userRepository.findByEmail(email);
33         if (user == null) {
34             throw new UsernameNotFoundException("User not found");
35         }
36         return user;
37     }
38 }

```

3 Record-API

Pasta config:

RabbitMQConfig.java

Endereço do Arquivo:

record-api\src\main\java\com\integracao_de_sistemas\record_api\config\RabbitMQConfig.java

Descrição:

Esta classe RabbitMQConfig configura o RabbitMQ para a aplicação. Define uma fila chamada "messageQueue" que é durável e persiste após reinicializações do servidor. Também configura um conversor de mensagens utilizando Jackson para converter mensagens JSON e o RabbitTemplate para interagir com o RabbitMQ usando este conversor.

- **messageQueue():** Cria uma fila chamada "messageQueue" que é durável, garantindo que as mensagens não sejam perdidas em caso de reinicializações do servidor.
- **jackson2JsonMessageConverter():** Cria um conversor de mensagens que utiliza Jackson para converter mensagens para e de JSON, facilitando a serialização e desserialização de objetos JSON.
-
- **rabbitTemplate(ConnectionFactory connectionFactory):** Cria um RabbitTemplate, configurando-o com o connectionFactory e o conversor de mensagens Jackson. O RabbitTemplate é usado para enviar e receber mensagens do RabbitMQ, utilizando o conversor de JSON definido.



```
1  @Configuration
2  @EnableRabbit
3  public class RabbitMQConfig {
4
5      @Bean
6      public Queue messageQueue() {
7          return new Queue("messageQueue", true);
8      }
9
10     @Bean
11     public MessageConverter jackson2JsonMessageConverter() {
12         return new Jackson2JsonMessageConverter();
13     }
14
15     @Bean
16     public RabbitTemplate rabbitTemplate(ConnectionFactory connectionFactory) {
17         RabbitTemplate rabbitTemplate = new RabbitTemplate(connectionFactory);
18         rabbitTemplate.setMessageConverter(jackson2JsonMessageConverter());
19         return rabbitTemplate;
20     }
21 }
```

Pasta controllers:

RecordController.java

Endereço do Arquivo:

record-api\src\main\java\com\integracao_de_sistemas\record_api\controllers\RecordController.java

Descrição:

A classe RecordController é responsável por gerenciar as operações relacionadas às mensagens recebidas e enviadas. Contém os seguintes endpoints:

- **saveMessage(@RequestBody MessageDTO messageDTO):** Um endpoint POST que recebe um MessageDTO no corpo da requisição e salva a mensagem usando o serviço MessageService. Retorna um status HTTP 200 com a mensagem "ok" após salvar a mensagem.
- **getMessages(@RequestParam("user") Long userId):** Um endpoint GET que recebe um ID de usuário como parâmetro e retorna todas as mensagens associadas a esse usuário usando o serviço MessageService. Retorna um status HTTP 200 com a lista de mensagens.

```
1  @RestController
2  @RequestMapping("/record")
3  public class RecordController {
4
5      @Autowired
6      private MessageService messageService;
7
8      @PostMapping("/message")
9      public ResponseEntity<?> saveMessage(@RequestBody MessageDTO messageDTO) {
10          messageService.saveMessage(messageDTO);
11          return ResponseEntity.ok("ok");
12      }
13
14      @GetMapping("/messages")
15      public ResponseEntity<?> getMessages(@RequestParam("user") Long userId) {
16          List<Message> messages = messageService.getMessagesByUserId(userId);
17          return ResponseEntity.ok(messages);
18      }
19  }
20
```

Pasta DTOs:

MessageDTO.java

Endereço do Arquivo:

record-api\src\main\java\com\integracao_de_sistemas\record_api\DTOs\MessageDTO.java

Descrição:

A classe MessageDTO serve como um objeto de transferência de dados entre diferentes camadas da aplicação, particularmente entre os serviços que manipulam mensagens. Sua principal função é encapsular os dados relevantes de uma mensagem (como os IDs do remetente e destinatário, e o conteúdo da mensagem) em um formato que pode ser facilmente transmitido e processado por diferentes partes do sistema. Isso facilita a comunicação entre serviços, especialmente ao trabalhar com frameworks de serialização e desserialização, como o Jackson, e ao enviar mensagens via RabbitMQ.

Construtores:

- Um construtor padrão sem parâmetros.
- Um construtor que aceita userIdSend, userIdReceive e message para inicializar os atributos da classe.

Métodos Getter e Setter:

- Métodos para acessar e modificar os atributos userIdSend, userIdReceive e message.

```
 1  public class MessageDTO implements Serializable {
 2      private static final long serialVersionUID = 1L;
 3
 4      private Long userIdSend;
 5      private Long userIdReceive;
 6      private String message;
 7
 8      public MessageDTO() {
 9
10
11      public MessageDTO(Long userIdSend, Long userIdReceive, String message) {
12          this.userIdSend = userIdSend;
13          this.userIdReceive = userIdReceive;
14          this.message = message;
15      }
16
17      public Long getUserIdSend() {
18          return userIdSend;
19      }
20      public void setUserIdSend(Long userIdSend) {
21          this.userIdSend = userIdSend;
22      }
23      public Long getUserIdReceive() {
24          return userIdReceive;
25      }
26      public void setUserIdReceive(Long userIdReceive) {
27          this.userIdReceive = userIdReceive;
28      }
29      public String getMessage() {
30          return message;
31      }
32      public void setMessage(String message) {
33          this.message = message;
34      }
35  }
```

Pasta models:

Message.java

Endereço do Arquivo: record-api\src\main\java\com\integracao_de_sistemas\record_api\models\Message.java

Descrição:

Esta classe é uma entidade JPA que representa uma mensagem armazenada no banco de dados. Aqui estão os componentes e funcionalidades principais:

Construtores:

- **Construtor Padrão:** Necessário para o JPA criar instâncias da entidade.
- **Construtor Completo:** Permite a criação de uma instância com todos os atributos.

Métodos:

- **Getters e Setters:** Métodos de acesso e modificação para todos os atributos, permitindo que outros componentes do sistema interajam com os dados encapsulados pela entidade.



```
1  @Entity
2  public class Message {
3
4      @Id
5      @GeneratedValue(strategy = GenerationType.IDENTITY)
6      private Long id;
7
8      private Long userIdSend;
9      private Long userIdReceive;
10     private String message;
11
12     public Message() {
13     }
14
15     public Message(Long id, Long userIdSend, Long userIdReceive, String message) {
16         this.id = id;
17         this.userIdSend = userIdSend;
18         this.userIdReceive = userIdReceive;
19         this.message = message;
20     }
21
22     public Long getId() {
23         return id;
24     }
25
26     public void setId(Long id) {
27         this.id = id;
28     }
29
30     public Long getUserIdSend() {
31         return userIdSend;
32     }
33
34     public void setUserIdSend(Long userIdSend) {
35         this.userIdSend = userIdSend;
36     }
37
38     public Long getUserIdReceive() {
39         return userIdReceive;
40     }
41
42     public void setUserIdReceive(Long userIdReceive) {
43         this.userIdReceive = userIdReceive;
44     }
45
46     public String getMessage() {
47         return message;
48     }
49
50     public void setMessage(String message) {
51         this.message = message;
52     }
53 }
```

Pasta repositories:

MessageRepository.java

Endereço do Arquivo:

record-api\src\main\java\com\integracao_de_sistemas\record_api\repositories\MessageRepository.java

Descrição:

A função principal do MessageRepository é abstrair e simplificar a interação com o banco de dados, permitindo operações de persistência e consultas personalizadas para a entidade Message de forma eficiente e sem necessidade de implementar consultas SQL diretamente.

Métodos:

- **findByIdReceive**: Método personalizado que busca todas as mensagens recebidas por um usuário específico, identificado pelo userId.

Herança:

- **JpaRepository<Message, Long>**: Herda métodos de CRUD (Create, Read, Update, Delete) padrão do JPA, permitindo operações como salvar, buscar, atualizar e deletar mensagens.



```
1 public interface MessageRepository extends JpaRepository<Message, Long> {  
2     List<Message> findByIdReceive(Long userId);  
3 }  
4
```

Pasta services:

MessageService.java

Endereço do Arquivo:

record-api\src\main\java\com\integracao_de_sistemas\record_api\repositories\MessageRepository.java

Descrição:

A função principal do MessageService é fornecer métodos de alto nível para salvar e recuperar mensagens, abstraindo a lógica de persistência e delegando essa responsabilidade ao repositório. Isso promove uma arquitetura limpa, separando a lógica de negócios da lógica de acesso a dados.

Métodos:

- **saveMessage**: Converte o DTO em uma entidade Message e a salva no banco de dados utilizando o messageRepository.
- **getMessagesByUserId**: Utiliza o método findByIdReceive do messageRepository para buscar todas as mensagens recebidas pelo usuário.

```
1 @Service
2 public class MessageService {
3
4     @Autowired
5     private MessageRepository messageRepository;
6
7     public void saveMessage(MessageDTO messageDTO) {
8         Message message = new Message();
9         message.setUserIdSend(messageDTO.getUserIdSend());
10        message.setUserIdReceive(messageDTO.getUserIdReceive());
11        message.setMessage(messageDTO.getMessage());
12        messageRepository.save(message);
13    }
14
15    public List<Message> getMessagesByUserId(Long userId) {
16        return messageRepository.findByUserIdReceive(userId);
17    }
18 }
```

4 SQL(s)

Auth-api:

v1__create_tables.sql:

Endereço do Arquivo: auth-api\src\main\resources\db\migration\v1__create_tables.sql

Descrição:

Este script SQL cria uma tabela chamada tb_users no banco de dados, que armazena informações sobre os usuários, incluindo seus e-mails, senhas e papéis.

```
1 CREATE TABLE tb_users (
2     id BIGINT AUTO_INCREMENT PRIMARY KEY,
3     email VARCHAR(255) NOT NULL,
4     password VARCHAR(255) NOT NULL,
5     role VARCHAR(50) NOT NULL
6 );
7
```

Recieve-Send-api:

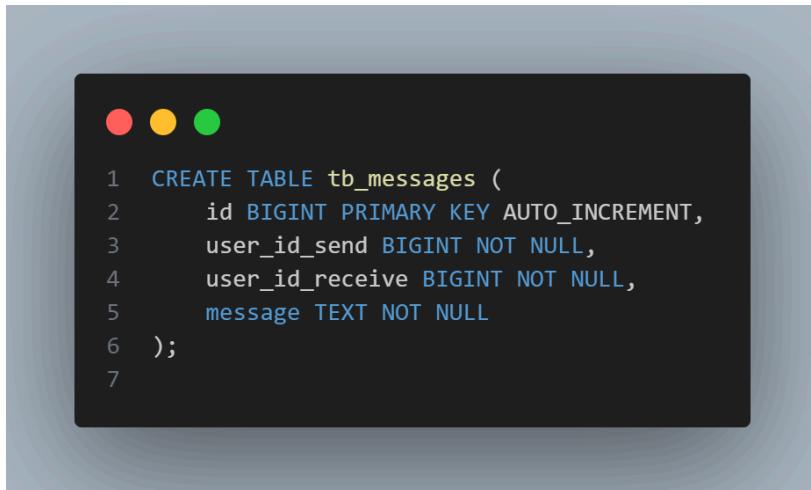
Record-api:

V1__crate_tb_messages.sql:

Endereço do Arquivo: record-api\src\main\resources\db\migration\V1__crate_tb_messages.sql

Descrição:

Este script SQL cria uma tabela chamada tb_messages no banco de dados, que armazena informações sobre mensagens enviadas e recebidas por usuários.



A screenshot of a terminal window with a dark background. At the top left are three colored dots: red, yellow, and green. The terminal displays the following SQL code:

```
1 CREATE TABLE tb_messages (
2     id BIGINT PRIMARY KEY AUTO_INCREMENT,
3     user_id_send BIGINT NOT NULL,
4     user_id_receive BIGINT NOT NULL,
5     message TEXT NOT NULL
6 );
7
```

5 Curl(s) do item: Mostrar funcionando (24/06) em

https://github.com/joneng2016/ensino/blob/master/integracao_de_sistemas/avaliacaofinal.md

Registro:

```
curl -X POST http://localhost:8080/message/register \
-H "Content-Type: application/json" \
-d '{
    "login": "usuario1@email.com",
    "password": "senha123",
    "role": "ROLE_USER"
}'
```

Login:

```
curl -X POST http://localhost:8080/message/login \
-H "Content-Type: application/json" \
-d '{
    "login": "usuario1@email.com",
    "password": "senha123"
}'
```

Enviar mensagem:

```
curl -X POST http://localhost:8080/message \
-H "Authorization: Bearer <TOKEN>" \
```

```
-H "Content-Type: application/json" \
-d '{
    "userIdSend": 1,
    "userIdReceive": 2,
    "message": "Hello, this is a test message."
}'
```

Processar mensagem (salvar no banco):

```
curl -X POST http://localhost:8080/message/worker \
-H "Authorization: Bearer <TOKEN>" \
-H "Content-Type: application/json" \
-d '{
    "userIdSend": 1,
    "userIdReceive": 2
}'
```

Ver mensagens enviadas para mim (deve estar logado no user que recebeu):

```
curl -X GET "http://localhost:8080/message?user=2" \
-H "Authorization: Bearer <TOKEN>"
```