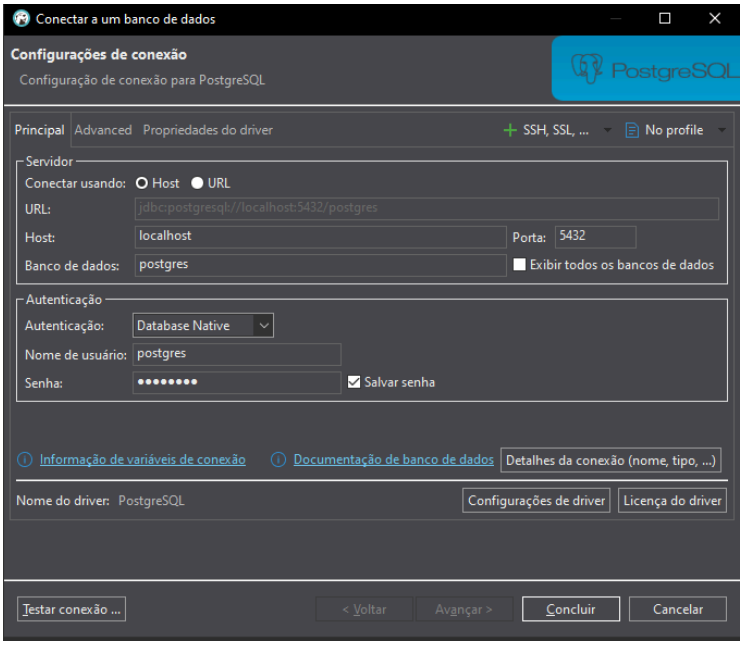
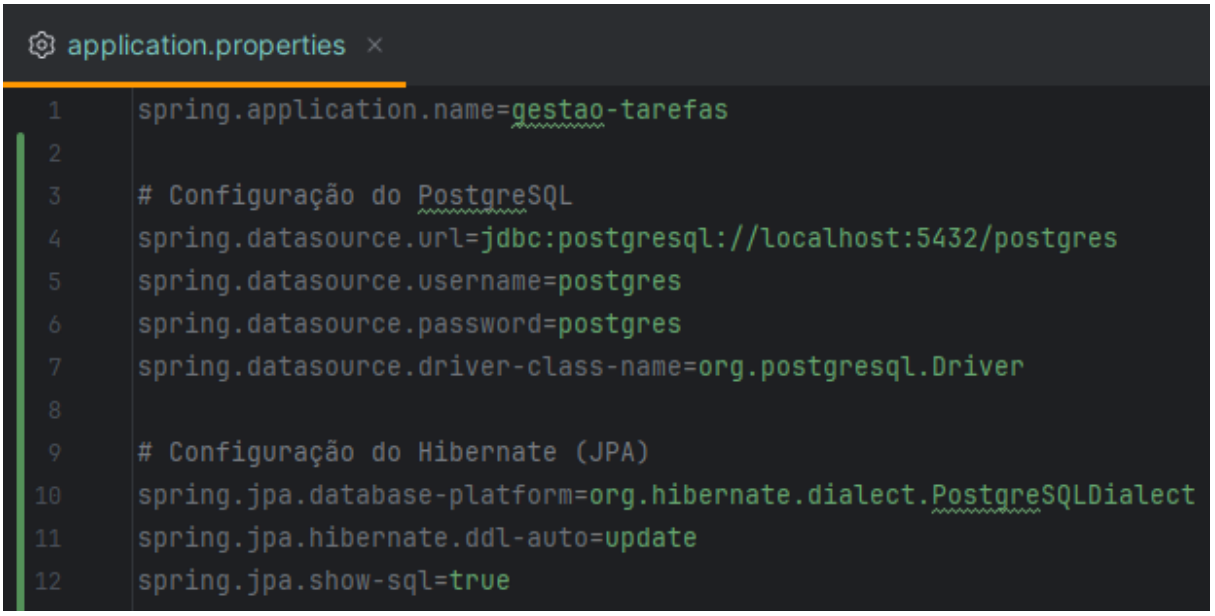
Sistema de Gestão de Tarefas - PASSO A PASSO

1 - Criei o projeto no Spring Initializr utilizando PostgreSQL, Jpa, Lombok, Spring Web, Spring Security, DevTools.

2 - Caso tenha que utilizar um banco de dados, configure esse banco, criando a conexão com o banco local (localhost), nome do banco, nome de usuário e senha (postgres em ambos), porta 5432.



3 - Configuramos o application.properties com a Configuração do PostgreSQL + Hibernate(JPA)



4 - Comecar criando uma entidade, nesse caso, foi a entidade User.

```
@Entity
@Table(name = "tb_users")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true) // Diz quem vai utiliza
r EqualsAndHash
public class User {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Setter(AccessLevel.NONE) // Não permite edição no ID
    @EqualsAndHashCode.Include // Seleciona o ID pra utilizar EqualsAndH
ash
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true) // username unico
```

```
    private String username;

    @Column(nullable = false)
    private String password;

    @Column(nullable = false, unique = true) // email único
    private String email;

    @Column(name = "created_at", nullable = false, updatable = false) //upd
atable não altera, pq ele já foi criado e não muda
    @CreationTimestamp
    private LocalDateTime createdAt;

    @Column(name = "updated_at", nullable = false)
    @UpdateTimestamp
    private LocalDateTime updatedAt;
}
```

5 - Criar o repositório dessa entidade, no caso, UsuarioRepository

```
@Repository
public interface UserRepository extends JpaRepository<User, Long> {}

Optional<User> findByUsername(String username);
```

💡 OBS: Após criar a Entidade e o Repositório, uma ideia de commit seguindo as convenções pode ser: git commit -m "chore(usuario): create entity and repository for user"

6 - Criar um DTO da classe `User` para transportar apenas os dados necessários entre controller e service, evitando expor campos sensíveis ou gerados automaticamente pelo banco, como `id`, `createdAt` e `updatedAt`.

```
package project.study.gestao_tarefas.dto;

import jakarta.validation.constraints.Email;
import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.Size;

public record UserDTO (
        @NotBlank(message = "Nome é obrigatório.")
        String name,

        @NotBlank(message = "Usuário é obrigatório.")
        @Size(min = 3, max = 20, message = "Usuário deve ter entre 3 e 20 c
aracteres")
        String username,

        @NotBlank(message = "Senha é obrigatória")
        @Size(min = 6, max = 20, message = "Senha deve ter entre 6 e 20 car
acteres")
        String password,

        @NotBlank(message = "Email é obrigatório")
        @Email(message = "Email informado é inválido")
        String email) {}
```

7 - Como criamos o `UserDTO` para expor apenas os dados necessários, precisamos de um `UserMapper` para **converter entre a entidade `User` e o DTO**. Dessa forma, o service e o controller trabalham apenas com DTOs, mantendo a entidade desacoplada e protegendo campos sensíveis ou gerados automaticamente pelo banco.

```
package project.study.gestao_tarefas.mapper;

import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import project.study.gestao_tarefas.dto.UserDTO;
import project.study.gestao_tarefas.entity.User;
```

```
@Mapper(componentModel = "spring")
public interface UserMapper {

    UserDTO toDTO(User user); // Converte a entidade User pra UserDTO, s
em expor campos sensíveis

    @Mapping(target = "id", ignore = true) // Ignora campos controlados pel
o banco
    @Mapping(target = "createdAt", ignore = true) // Ignora campos controla
dos pelo banco
    @Mapping(target = "updatedAt", ignore = true) // Ignora campos control
ados pelo banco
    User toEntity(UserDTO dto); // de DTO para persistência
}
```

💡 **Resumo prático:**

- **GET** → sempre `toDTO`
- **POST/PUT/PATCH** → `toEntity` para persistir, e depois `toDTO` se precisar retornar resultado

8 - Depois disso, começamos a criar os métodos principais do nosso usuário (registro e edição com save, procurar por id pro usuário ver seu próprio perfil, deletar o próprio perfil).

```
@Service
@RequiredArgsConstructor
public class UserService {
    private final UserRepository userRepository;
    private final UserMapper userMapper;

        public List<UserDTO> findAll() {
            return userRepository.findAll() // Busca TODOS os usuários no banc
o
```

```
                .stream() // Converte a lista em Stream (para usar operações fu
ncionais)
                .map(userMapper::toDTO) // Transforma cada User em UserDT
O usando o mapper
                .collect(Collectors.toList()); // Coleta tudo de volta numa List<U
serDTO>
        }

    public Optional<UserDTO> findById(Long id) {
       return userRepository.findById(id) // Busca usuário por ID (retorna Opti
onal<User>)
              .map(userMapper::toDTO);  // SE encontrou, converte User para
UserDTO
           // SE não encontrou, continua Optional.empty()
    }

public Optional<UserDTO> update(Long id, UserDTO userDTO) {
    return userRepository.findById(id) // Busca usuário existente por ID
           .map(existingUser -> { // SE encontrou, executa o bloco abaixo
              existingUser.setName(userDTO.name());         // Atualiza nome
              existingUser.setUsername(userDTO.username()); // Atualiza user
name
              existingUser.setEmail(userDTO.email());       // Atualiza email
              existingUser.setPassword(userDTO.password()); // Atualiza senha

              User updatedUser = userRepository.save(existingUser); // Salva n
o banco
              return userMapper.toDTO(updatedUser); // Converte para DTO e r
etorna
           });
    // SE não encontrou o usuário, retorna Optional.empty()
}

public UserDTO save(UserDTO userDTO) {
    User user = userMapper.toEntity(userDTO); // Converte UserDTO para U
ser (entidade)
    User savedUser = userRepository.save(user); // Salva no banco (gera ID,
timestamps)
```

```
    return userMapper.toDTO(savedUser); // Converte User salvo de volta pa
ra DTO
}

public void deleteById(Long id) {
    userRepository.deleteById(id); // Deleta usuário do banco pelo ID
}
```

9 - Pra finalizar nosso CRUD básico da entidade User, finalizamos implementando o controller baseado nas regras de negócio do nosso service

```
@RestController
@RequestMapping("/api/users")
@RequiredArgsConstructor
public class UserController {

// Injeção de dependência - Spring injeta automaticamente
private final UserService userService;

// ENDPOINT: GET /api/users - Lista todos os usuários
@GetMapping // Mapeia requisições GET para este método
@PreAuthorize("hasRole('ADMIN')")  // Só usuários ADMIN podem acessar
(Spring Security)
public ResponseEntity<List<UserDTO>> findAll() {
    // Chama service para buscar todos usuários
    // ResponseEntity.ok() = status 200 + dados no body
    return ResponseEntity.ok(userService.findAll());
}

// ENDPOINT: GET /api/users/1 - Busca usuário por ID
@GetMapping("/{id}") // {id} é variável na URL
public ResponseEntity<UserDTO> findById(@PathVariable Long id) { //@Pa
thVariable captura {id} da URL
    return userService.findById(id)  // Service retorna Optional<UserDTO>
            .map(ResponseEntity::ok)  // Se encontrou: status 200 + dados
            .orElse(ResponseEntity.notFound().build()); // Se não encontrou: stat
us 404
```

```
}

// ENDPOINT: PUT /api/users/1 - Atualiza usuário por ID
@PutMapping("/{id}") // Mapeia requisições PUT
public ResponseEntity<UserDTO> update(@PathVariable Long id, // ID da
URL
                                      @Valid @RequestBody UserDTO userDTO) { // @Vali
d valida DTO, @RequestBody pega JSON do body
   return userService.update(id, userDTO)  // Service tenta atualizar
          .map(ResponseEntity::ok)         // Se atualizou: status 200 + dados at
ualizados
          .orElse(ResponseEntity.notFound().build()); // Se não encontrou: stat
us 404
}

// ENDPOINT: POST /api/users/register - Cadastra novo usuário
@PostMapping("/register") // Mapeia requisições POST para "/register"
public ResponseEntity<UserDTO> save(@Valid @RequestBody UserDTO u
serDTO) { // Valida e pega dados do JSON
   UserDTO savedUser = userService.save(userDTO); // Service salva no b
anco
   return ResponseEntity // Monta resposta customizada
          .created(URI.create("/users/" + savedUser.username())) // Status 20
1 (Created) + Location header
          .body(savedUser); // Retorna dados do usuário criado
}

// ENDPOINT: DELETE /api/users/123 - Deleta usuário por ID
@DeleteMapping("/{id}") // Mapeia requisições DELETE
public ResponseEntity<Void> deleteById(@PathVariable Long id) { // Void =
não retorna dados no body
   userService.deleteById(id); // Service deleta do banco
   return ResponseEntity.noContent().build(); // Status 204 (No Content) =
"deletado com sucesso"
}
}
```

10 - Finalizado o básico da entidade User, vamos criar o básico da entidade Task e ir aprimorando os dois aos poucos. Primeiro, criamos a Entidade Task com os seguintes atributos:

```
@Entity
@Table(name = "tb_tasks")
@NoArgsConstructor
@AllArgsConstructor
@Getter
@Setter
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
public class Task {

        // Define o relacionamento Many-to-One entre Task e User.
        // Cada Task pertence a um User, e várias Tasks podem compartilhar o mesmo User.
        // No banco, isso cria uma coluna 'user_id' na tabela Task apontando para o User.
        @ManyToOne
        private User user;

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Setter(AccessLevel.NONE)
    @EqualsAndHashCode.Include
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false)
    private String description;

    @Column(nullable = false)
    @Enumerated(EnumType.STRING) // Banco salva Enum em texto puro, em vez de utilizar números
    private TaskStatus status;
```

```
    @Column(name = "created_at", nullable = false, updatable = false)
    @CreationTimestamp
    private LocalDateTime createdAt;

    @Column(name = "updated_at", nullable = false)
    @UpdateTimestamp
    private LocalDateTime updatedAt;
}
```

Na tabela `User`, utilizamos a anotação `@OneToMany(mappedBy = "user")` para representar que um usuário pode ter várias tarefas associadas. O uso do `mappedBy` indica que o lado dono do relacionamento é a entidade `Task`, que já contém a referência para o usuário via `@ManyToOne`. Com isso, não é necessário criar uma coluna extra na tabela `User`, evitando duplicação de dados.

```
// Define o relacionamento One-to-Many entre User e Task.
// Um User pode ter várias Tasks associadas, mas a tabela User não precis
a de coluna extra.
// O atributo 'mappedBy = "user"' indica que o lado dono do relacionament
o é Task,
// que já possui a referência para User (via @ManyToOne), evitando criar F
K duplicada.
    @OneToMany(mappedBy = "user")
    private List<Task> tasks;
```

11 - Conforme vimos na Entidade, nós vamos criar um ENUM chamado TaskStatus pra facilitar nossa vida:

```
public enum TaskStatus {
    TO_DO,
    DOING,
    DONE
}
```

12 - Finalizada a entidade, adicionamos o Repository:

```
@Repository
public interface TaskRepository extends JpaRepository<Task, Long> {
}
```

13 - Nosso DTO:

```
package project.study.gestao_tarefas.dto;

import jakarta.validation.constraints.NotBlank;
import jakarta.validation.constraints.NotNull;
import jakarta.validation.constraints.Size;
import project.study.gestao_tarefas.enums.TaskStatus;

/**
 * DTO para representar uma tarefa.
 * Inclui validações e mapeamento para a entidade Task.
 */
public record TaskDTO(
        @NotBlank(message = "O nome da tarefa é obrigatório")
        @Size(max = 100, message = "O nome da tarefa deve ter no máximo 1
00 caracteres")
        String name,

        @NotBlank(message = "A descrição é obrigatória")
        @Size(max = 1000, message = "A descrição deve ter no máximo 1000
caracteres")
        String description,

        @NotNull(message = "O status da tarefa é obrigatório")
        TaskStatus status,

        // ID do usuário responsável pela tarefa
        Long userId
) {
    // Construtor adicional sem userId para facilitar a criação
    public TaskDTO(String name, String description, TaskStatus status) {
```

```
        this(name, description, status, null);
    }
}
```

14 - Nosso mapper:

```
package project.study.gestao_tarefas.mapper;

import org.mapstruct.Mapper;
import org.mapstruct.Mapping;
import org.mapstruct.Named;
import org.mapstruct.ReportingPolicy;
import project.study.gestao_tarefas.dto.TaskDTO;
import project.study.gestao_tarefas.entity.Task;
import project.study.gestao_tarefas.entity.User;
import project.study.gestao_tarefas.exception.ResourceNotFoundExceptio
n;
import project.study.gestao_tarefas.repository.UserRepository;

/**
 * Mapper para conversão entre Task e TaskDTO.
 * Configurado para ignorar campos não mapeados e usar o Spring como p
rovedor de injeção.
 */
@Mapper(
    componentModel = "spring",
    unmappedTargetPolicy = ReportingPolicy.IGNORE,
    uses = {UserRepository.class}
)
public interface TaskMapper {

    /**
     * Converte uma entidade Task para TaskDTO.
     * @param task A entidade Task a ser convertida
     * @return O DTO correspondente
     */
    @Mapping(source = "user.id", target = "userId")
```

```
    @Mapping(source = "name", target = "name")
    @Mapping(source = "description", target = "description")
    @Mapping(source = "status", target = "status")
    TaskDTO toDTO(Task task);

    /**
     * Converte um TaskDTO para entidade Task.
     * @param taskDTO O DTO a ser convertido
     * @return A entidade correspondente
     */
    @Mapping(target = "id", ignore = true)
    @Mapping(target = "createdAt", ignore = true)
    @Mapping(target = "updatedAt", ignore = true)
    @Mapping(target = "user", expression = "java(getUser(taskDTO.userId
()))")
    Task toEntity(TaskDTO taskDTO);

    /**
     * Método auxiliar para obter um usuário pelo ID.
     * @param userId ID do usuário
     * @return A entidade User correspondente
     * @throws ResourceNotFoundException se o usuário não for encontrad
o
     */
    default User getUser(Long userId) {
        if (userId == null) {
            return null;
        }
        return UserRepository
            .findById(userId)
            .orElseThrow(() -> new ResourceNotFoundException("Usuário não e
ncontrado com o ID: " + userId));
    }
}
```

15 - Nosso service:

```
@Service
@RequiredArgsConstructor
public class TaskService {

    private final TaskRepository taskRepository;
    private final TaskMapper taskMapper;

    public TaskDTO save (TaskDTO taskDTO) {
        Task task = taskMapper.toEntity(taskDTO);
        Task taskSaved = taskRepository.save(task);
        return taskMapper.toDTO(taskSaved);
    }

    public TaskDTO update (Long id, TaskDTO taskDTO) {
        Task existingTask = taskRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Task não encontrad
a."));

        existingTask.setName(taskDTO.name());
        existingTask.setDescription(taskDTO.description());
        existingTask.setStatus(taskDTO.status());

        Task updatedTask = taskRepository.save(existingTask);
        return taskMapper.toDTO(updatedTask);
    }

    public TaskDTO findById (Long id) {
        Task task = taskRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Task não encontrad
a."));
        return taskMapper.toDTO(task);
    }

    public List<TaskDTO> findAll () {
        return taskRepository.findAll().stream().map(taskMapper::toDTO).colle
ct(Collectors.toList());
    }
```

```
    public void delete (Long id) {
        Task task = taskRepository.findById(id)
                .orElseThrow(() -> new RuntimeException("Task não encontrad
a."));
        taskRepository.delete(task);
    }
}
```

16 - E o nosso controller:

```
@RestController
@RequestMapping("/api/tasks")
@RequiredArgsConstructor
public class TaskController {

    private final TaskService taskService;

    @PostMapping
    public ResponseEntity<TaskDTO> createTask (@RequestBody TaskDTO t
askDTO) {
        TaskDTO taskSaved = taskService.save(taskDTO);
        return ResponseEntity.status(HttpStatus.CREATED).body(taskSaved);
    }

    @PutMapping("/{id}")
    public ResponseEntity<TaskDTO> updateTask(@PathVariable Long id, @
RequestBody TaskDTO taskDTO) {
        TaskDTO updatedTask = taskService.update(id, taskDTO);
        return ResponseEntity.ok(updatedTask);
    }

    @GetMapping("/{id}")
    public ResponseEntity<TaskDTO> getTasKById(@PathVariable Long id) {
        TaskDTO taskDTO = taskService.findById(id);
        return ResponseEntity.ok(taskDTO);
    }
```

```
    @GetMapping
    public ResponseEntity<List<TaskDTO>> getAllTasks () {
        List<TaskDTO> tasks = taskService.findAll();
        return ResponseEntity.ok(tasks);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteTask(@PathVariable Long id) {
      taskService.delete(id);
      return ResponseEntity.noContent().build();

    }
}
```

É importante lembrar que essas classes são o esqueleto do nosso CRUD, muitas delas sofrerão alterações no decorrer do código, com a inserção de validações, exceções, autenticação e etc.

17 - Finalizado o CRUD básico de Task e User, nós vamos começar a implementar Autenticação de Segurança com Spring Security e JWT. Começaremos adicionando essas dependências no POM.XML:

```
<!-- Spring Security -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-security</artifactId>
</dependency>

<!-- JWT Dependencies -->
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-api</artifactId>
    <version>0.11.5</version>
</dependency>
<dependency>
```

```
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-impl</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>io.jsonwebtoken</groupId>
    <artifactId>jjwt-jackson</artifactId>
    <version>0.11.5</version>
    <scope>runtime</scope>
</dependency>

<!-- Jakarta Servlet API (necessário para Spring Boot 3.x) -->
<dependency>
    <groupId>jakarta.servlet</groupId>
    <artifactId>jakarta.servlet-api</artifactId>
    <version>6.0.0</version>
    <scope>provided</scope>
</dependency>

<!-- Java Validation para os DTOS -->
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
</dependency>
```

18 - Pra começar nossa segurança, vamos implementar criptografia para salvar senhas no banco de dados. A criptografia escolhida foi o BCrypt, sua implementação é simples, criamos essa classe no pacote config:

```
@Configuration
public class BCryptPasswordConfig {

    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
```

```
    }
}
```

19 - Modificaremos o application.properties para a utilização do jwt

```
# JWT Configuration
app.jwt.secret-key=SegredoSuperSecretoMuitoLongoQueDeveSerMudado
EmAmbienteDeProducao12345
app.jwt.expiration-ms=86400000 # 24 horas em milissegundos
app.jwt.issuer=gestao-tarefas-api

# H2 Database (para desenvolvimento)
spring.datasource.url=jdbc:h2:mem:gestaotarefasdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

# JPA/Hibernate
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
spring.jpa.hibernate.ddl-auto=update
spring.jpa.show-sql=true

# Logging
logging.level.org.springframework.security=DEBUG
logging.level.project.study.gestao_tarefas=DEBUG
```

20 - Vamos criar o Enum de Papéis (UserRole)

```
package project.study.gestao_tarefas.enums;

import org.springframework.security.core.GrantedAuthority;

/**
 * Define os papéis de usuário no sistema.
```

```
 * Implementa GrantedAuthority para integração com o Spring Security.
 * O prefixo 'ROLE_' é necessário para o Spring Security.
 */
public enum UserRole implements GrantedAuthority {
    ROLE_USER,   // Usuário comum
    ROLE_ADMIN;  // Administrador do sistema

    @Override
    public String getAuthority() {
        return name(); // Retorna o nome do enum (ex: "ROLE_USER")
    }
}
```

21 - Modificamos a entidade User para implementar UserDetails do Spring Security, permitindo integração direta com o mecanismo de autenticação e autorização da aplicação.
Além disso, adicionamos o campo roles para gerenciar permissões de forma estruturada e um relacionamento @OneToMany com Task, facilitando o acesso às tarefas de cada usuário.
Ajustamos Lombok e equals/hashCode para manter consistência

```
package project.study.gestao_tarefas.entity;

import jakarta.persistence.*;
import lombok.*;
import org.hibernate.annotations.CreationTimestamp;
import org.hibernate.annotations.UpdateTimestamp;
import org.springframework.security.core.GrantedAuthority;
import org.springframework.security.core.userdetails.UserDetails;
import project.study.gestao_tarefas.enums.UserRole;

import java.time.LocalDateTime;
import java.util.Collection;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
```

```
/**
 * Entidade que representa um usuário no sistema.
 * Implementa UserDetails para integração com o Spring Security.
 */
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table(name = "tb_users")
public class User implements UserDetails {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @Column(nullable = false)
    private String name;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    // Relacionamento muitos-para-muitos com UserRole
    @ElementCollection(fetch = FetchType.EAGER) // EAGER para carregar a
s roles junto com o usuário
    @Enumerated(EnumType.STRING) // Armazena o nome da enumeração
como string
    private Set<UserRole> roles = new HashSet<>();

    @OneToMany(mappedBy = "user")
    private List<Task> tasks;
```

```
    @CreationTimestamp
    @Column(name = "created_at", nullable = false, updatable = false)
    private LocalDateTime createdAt;

    @UpdateTimestamp
    @Column(name = "updated_at", nullable = false)
    private LocalDateTime updatedAt;

    // Métodos obrigatórios do UserDetails
    @Override
    public Collection<? extends GrantedAuthority> getAuthorities() {
        return this.roles;
    }

    @Override
    public boolean isAccountNonExpired() {
        return true; // Conta nunca expira
    }

    @Override
    public boolean isAccountNonLocked() {
        return true; // Conta nunca é bloqueada
    }

    @Override
    public boolean isCredentialsNonExpired() {
        return true; // Credenciais nunca expiram
    }

    @Override
    public boolean isEnabled() {
        return true; // Conta sempre está ativa
    }
}
```

Com essa modificação, precisaremos modificar o UserMapper e o TaskMapper,

22 - Depois, vamos criar o JwtProperties para configurações de JWT

```
package project.study.gestao_tarefas.config;

import org.springframework.boot.context.properties.ConfigurationPropertie
s;
import org.springframework.context.annotation.Configuration;

/**
 * Classe de configuração para as propriedades do JWT.
 * As propriedades são carregadas do arquivo application.properties com o
prefixo 'app.jwt'.
 */
@Configuration
@ConfigurationProperties(prefix = "app.jwt")
public class JwtProperties {
    private String secretKey;      // Chave secreta para assinar o token
    private long expirationMs;     // Tempo de expiração do token em milisse
gundos
    private String issuer;         // Emissor do token

    // Getters e Setters
    public String getSecretKey() {
        return secretKey;
    }

    public void setSecretKey(String secretKey) {
        this.secretKey = secretKey;
    }

    public long getExpirationMs() {
        return expirationMs;
    }

    public void setExpirationMs(long expirationMs) {
        this.expirationMs = expirationMs;
    }

    public String getIssuer() {
        return issuer;
```

```
    }

    public void setIssuer(String issuer) {
        this.issuer = issuer;
    }
}
```

23 - Depois, criaremos o JwtService

```
package project.study.gestao_tarefas.security.jwt;

import io.jsonwebtoken.*;
import io.jsonwebtoken.security.Keys;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.stereotype.Service;
import project.study.gestao_tarefas.config.JwtProperties;

import java.security.Key;
import java.util.Date;
import java.util.HashMap;
import java.util.Map;
import java.util.function.Function;

/**
 * Serviço responsável por gerar, validar e extrair informações de tokens J
WT.
 */
@Slf4j
@Service
@RequiredArgsConstructor
public class JwtService {

    private final JwtProperties jwtProperties;
```

```
    /**
     * Gera um token JWT para o usuário.
     */
    public String generateToken(UserDetails userDetails) {
        return generateToken(new HashMap<>(), userDetails);
    }

    /**
     * Gera um token JWT com claims adicionais.
     */
    public String generateToken(Map<String, Object> extraClaims, UserDeta
ils userDetails) {
        return Jwts.builder()
                .setClaims(extraClaims)
                .setSubject(userDetails.getUsername())
                .setIssuer(jwtProperties.getIssuer())
                .setIssuedAt(new Date(System.currentTimeMillis()))
                .setExpiration(new Date(System.currentTimeMillis() + jwtProperti
es.getExpirationMs()))
                .signWith(getSignInKey(), SignatureAlgorithm.HS256)
                .compact();
    }

    /**
     * Verifica se um token JWT é válido para o usuário.
     */
    public boolean isTokenValid(String token, UserDetails userDetails) {
        final String username = extractUsername(token);
        return (username.equals(userDetails.getUsername())) && !isTokenExpi
red(token);
    }

    /**
     * Verifica se o token JWT está expirado.
     */
    public boolean isTokenExpired(String token) {
        return extractExpiration(token).before(new Date());
    }
```

```
    /**
     * Extrai o nome de usuário do token JWT.
     */
    public String extractUsername(String token) {
        return extractClaim(token, Claims::getSubject);
    }

    /**
     * Extrai a data de expiração do token JWT.
     */
    public Date extractExpiration(String token) {
        return extractClaim(token, Claims::getExpiration);
    }

    /**
     * Extrai uma claim específica do token JWT.
     */
    public <T> T extractClaim(String token, Function<Claims, T> claimsReso
lver) {
        final Claims claims = extractAllClaims(token);
        return claimsResolver.apply(claims);
    }

    /**
     * Extrai todas as claims do token JWT.
     */
    private Claims extractAllClaims(String token) {
        try {
            return Jwts.parserBuilder()
                    .setSigningKey(getSignInKey())
                    .build()
                    .parseClaimsJws(token)
                    .getBody();
        } catch (ExpiredJwtException ex) {
            log.error("Token expirado: {}", ex.getMessage());
            throw new JwtException("Token expirado", ex);
        } catch (UnsupportedJwtException | MalformedJwtException ex) {
```

```
            log.error("Token inválido: {}", ex.getMessage());
            throw new JwtException("Token inválido", ex);
        } catch (Exception ex) {
            log.error("Erro ao processar o token: {}", ex.getMessage());
            throw new JwtException("Erro ao processar o token", ex);
        }
    }

    /**
     * Obtém a chave de assinatura para o token JWT.
     */
    private Key getSignInKey() {
        byte[] keyBytes = jwtProperties.getSecretKey().getBytes();
        return Keys.hmacShaKeyFor(keyBytes);
    }
}
```

24 - Depois, criaremos os filtros de autenticação com o JwtAuthenticationFilter

```
package project.study.gestao_tarefas.security.jwt;

import jakarta.servlet.FilterChain;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import lombok.RequiredArgsConstructor;
import lombok.extern.slf4j.Slf4j;
import org.springframework.security.authentication.UsernamePasswordAu
thenticationToken;
import org.springframework.security.core.context.SecurityContextHolder;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.security.core.userdetails.UserDetailsService;
import org.springframework.security.web.authentication.WebAuthenticatio
nDetailsSource;
import org.springframework.stereotype.Component;
import org.springframework.util.StringUtils;
```

```
import org.springframework.web.filter.OncePerRequestFilter;

import java.io.IOException;

/**
 * Filtro que processa as requisições HTTP e valida os tokens JWT.
 * É executado uma vez por requisição.
 */
@Slf4j
@Component
@RequiredArgsConstructor
public class JwtAuthenticationFilter extends OncePerRequestFilter {

    private static final String AUTHORIZATION_HEADER = "Authorization";
    private static final String BEARER_PREFIX = "Bearer ";

    private final JwtService jwtService;
    private final UserDetailsService userDetailsService;

    /**
     * Método principal do filtro que processa cada requisição.
     */
    @Override
    protected void doFilterInternal(
            HttpServletRequest request,
            HttpServletResponse response,
            FilterChain filterChain
    ) throws ServletException, IOException {
        try {
            // Extrai o token JWT do cabeçalho da requisição
            String jwt = getJwtFromRequest(request);

            // Verifica se o token é válido e se o usuário está autenticado
            if (StringUtils.hasText(jwt) && jwtService.isTokenValid(jwt, getCurren
tUserDetails())) {
                // Extrai o nome de usuário do token
                String username = jwtService.extractUsername(jwt);
```

```
            // Carrega os detalhes do usuário do banco de dados
            UserDetails userDetails = userDetailsService.loadUserByUserna
me(username);

            // Cria um objeto de autenticação
            UsernamePasswordAuthenticationToken authentication = new Us
ernamePasswordAuthenticationToken(
                    userDetails,
                    null,
                    userDetails.getAuthorities()
            );

            // Define os detalhes da autenticação
            authentication.setDetails(
                new WebAuthenticationDetailsSource().buildDetails(request)
            );

            // Define a autenticação no contexto de segurança
            SecurityContextHolder.getContext().setAuthentication(authenticat
ion);
        }
    } catch (Exception ex) {
        log.error("Não foi possível autenticar o usuário: {}", ex.getMessage
());
    }

    // Continua o processamento da requisição
    filterChain.doFilter(request, response);
  }

  /**
   * Extrai o token JWT do cabeçalho da requisição.
   */
  private String getJwtFromRequest(HttpServletRequest request) {
    String bearerToken = request.getHeader(AUTHORIZATION_HEADER);
    if (StringUtils.hasText(bearerToken) && bearerToken.startsWith(BEARE
R_PREFIX)) {
        return bearerToken.substring(BEARER_PREFIX.length());
```

```
        }
        return null;
    }

    /**
     * Obtém os detalhes do usuário atual do contexto de segurança.
     */
    private UserDetails getCurrentUserDetails() {
        return (UserDetails) SecurityContextHolder.getContext()
                .getAuthentication()
                .getPrincipal();
    }
}
```

25 - Logo em seguida, vamos criar a classe que vai tratar os erros de autenticação, a JwtAuthenticationEntryPoint

```
package project.study.gestao_tarefas.security.jwt;

import com.fasterxml.jackson.databind.ObjectMapper;
import jakarta.servlet.ServletException;
import jakarta.servlet.http.HttpServletRequest;
import jakarta.servlet.http.HttpServletResponse;
import org.springframework.http.MediaType;
import org.springframework.security.core.AuthenticationException;
import org.springframework.security.web.AuthenticationEntryPoint;
import org.springframework.stereotype.Component;

import java.io.IOException;
import java.util.HashMap;
import java.util.Map;

/**
 * Classe que trata erros de autenticação.
 * É chamada quando um usuário não autenticado tenta acessar um recurs
o protegido.
```

```
 */
@Component
public class JwtAuthenticationEntryPoint implements AuthenticationEntryP
oint {

    /**
     * Método chamado quando ocorre um erro de autenticação.
     */
    @Override
    public void commence(
            HttpServletRequest request,
            HttpServletResponse response,
            AuthenticationException authException
    ) throws IOException, ServletException {
        // Configura o tipo de conteúdo da resposta como JSON
        response.setContentType(MediaType.APPLICATION_JSON_VALUE);
        // Define o status da resposta como 401 (Não Autorizado)
        response.setStatus(HttpServletResponse.SC_UNAUTHORIZED);

        // Cria um mapa com os detalhes do erro
        final Map<String, Object> body = new HashMap<>();
        body.put("status", HttpServletResponse.SC_UNAUTHORIZED);
        body.put("error", "Não autorizado");
        body.put("message", authException.getMessage());
        body.put("path", request.getServletPath());

        // Converte o mapa para JSON e envia na resposta
        final ObjectMapper mapper = new ObjectMapper();
        mapper.writeValue(response.getOutputStream(), body);
    }
}
```

26 - Atualizaremos o SecurityConfig

```
package project.study.gestao_tarefas.config;

import lombok.RequiredArgsConstructor;
```

```
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.security.authentication.AuthenticationManage
r;
import org.springframework.security.config.annotation.authentication.confi
guration.AuthenticationConfiguration;
import org.springframework.security.config.annotation.web.builders.HttpS
ecurity;
import org.springframework.security.config.annotation.web.configuration.E
nableWebSecurity;
import org.springframework.security.config.annotation.web.configurers.Ab
stractHttpConfigurer;
import org.springframework.security.config.http.SessionCreationPolicy;
import org.springframework.security.crypto.bcrypt.BCryptPasswordEncod
er;
import org.springframework.security.crypto.password.PasswordEncoder;
import org.springframework.security.web.SecurityFilterChain;
import org.springframework.security.web.authentication.UsernamePasswo
rdAuthenticationFilter;
import org.springframework.web.cors.CorsConfiguration;
import org.springframework.web.cors.CorsConfigurationSource;
import org.springframework.web.cors.UrlBasedCorsConfigurationSource;
import project.study.gestao_tarefas.security.jwt.JwtAuthenticationEntryPoi
nt;
import project.study.gestao_tarefas.security.jwt.JwtAuthenticationFilter;

import java.util.Arrays;
import java.util.List;

@Configuration
@EnableWebSecurity
@RequiredArgsConstructor
public class SecurityConfig {

    private final JwtAuthenticationEntryPoint unauthorizedHandler;
    private final JwtAuthenticationFilter jwtAuthenticationFilter;

    /**
```

```
    * Configura a cadeia de filtros de segurança (SecurityFilterChain).
    *
    * - Desabilita CSRF (não necessário em APIs stateless).
    * - Define CORS customizado.
    * - Define tratamento para autenticação não autorizada.
    * - Define sessão como STATELESS (sem sessões HTTP).
    * - Configura permissões para endpoints públicos (auth, swagger, h2-c
onsole).
    * - Requer autenticação para qualquer outro endpoint.
    * - Adiciona filtro JWT antes do UsernamePasswordAuthenticationFilter.
    *
    * @param http objeto de configuração de segurança do Spring
    * @return a cadeia de filtros configurada
    * @throws Exception em caso de falha na configuração
    */
   @Bean
   public SecurityFilterChain filterChain(HttpSecurity http) throws Exception
{
       http
           // Desabilita CSRF (não é necessário para APIs stateless)
           .csrf(AbstractHttpConfigurer::disable)

           // Configura o CORS
           .cors(cors → cors.configurationSource(corsConfigurationSource()))

           // Configura o tratamento de erros de autenticação
           .exceptionHandling(exception → exception
               .authenticationEntryPoint(unauthorizedHandler)
           )

           // Configura a política de sessão como stateless
           .sessionManagement(session → session
               .sessionCreationPolicy(SessionCreationPolicy.STATELESS)
           )

           // Define as regras de autorização
           .authorizeHttpRequests(auth → auth
               // Permite acesso público aos endpoints de autenticação
```

```
                .requestMatchers("/api/auth/**").permitAll()
                // Permite acesso público à documentação da API
                .requestMatchers(
                    "/v3/api-docs/**",
                    "/swagger-ui/**",
                    "/swagger-ui.html",
                    "/swagger-resources/**",
                    "/webjars/**"
                ).permitAll()
                // Permite acesso ao console do H2 (apenas para desenvolviment
o)
                .requestMatchers("/h2-console/**").permitAll()
                // Exige autenticação para todos os outros endpoints
                .anyRequest().authenticated()
            );

        // Configuração específica para o console do H2 (apenas para desenv
olvimento)
        http.headers(headers -> headers
            .frameOptions(frame -> frame.sameOrigin())
        );

        // Adiciona o filtro JWT antes do filtro de autenticação de usuário e se
nha
        http.addFilterBefore(jwtAuthenticationFilter, UsernamePasswordAuthe
nticationFilter.class);

        return http.build();
    }

    /**
     * Define a configuração de CORS da aplicação.
     *
     * - Permite origem específica (ex.: frontend React em localhost:3000).
     * - Permite métodos HTTP comuns (GET, POST, PUT, DELETE, OPTION
S).
     * - Permite todos os cabeçalhos.
     * - Permite envio de credenciais (cookies, authorization header).
```

```
     *
     * @return configuração de CORS
     */
    @Bean
    public CorsConfigurationSource corsConfigurationSource() {
        CorsConfiguration configuration = new CorsConfiguration();
        configuration.setAllowedOrigins(List.of("http://localhost:3000")); // Ad
icione suas origens permitidas
        configuration.setAllowedMethods(Arrays.asList("GET", "POST", "PU
T", "DELETE", "OPTIONS"));
        configuration.setAllowedHeaders(List.of("*"));
        configuration.setAllowCredentials(true);

        UrlBasedCorsConfigurationSource source = new UrlBasedCorsConfig
urationSource();
        source.registerCorsConfiguration("/**", configuration);
        return source;
    }

    /**
     * Bean responsável por expor o AuthenticationManager.
     * O AuthenticationManager será usado pelo Spring Security para autenti
car usuários.
     *
     * @param authenticationConfiguration configuração de autenticação do
Spring
     * @return AuthenticationManager configurado
     * @throws Exception em caso de falha na configuração
     */
    @Bean
    public AuthenticationManager authenticationManager(
            AuthenticationConfiguration authenticationConfiguration) throws Ex
ception {
        return authenticationConfiguration.getAuthenticationManager();
    }

    /**
     * Bean responsável por definir o algoritmo de hashing de senhas.
```

```
     *
     * Neste caso, utiliza BCrypt, que é o padrão recomendado pelo Spring S
ecurity
     * para armazenamento seguro de senhas.
     *
     * @return PasswordEncoder baseado em BCrypt
     */
    @Bean
    public PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
}
```

27 - Vamos começar a implementar os controladores, começando pelo famoso AuthController, que permite que os usuários façam login

```
package project.study.gestao_tarefas.controller;

import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.authentication.AuthenticationManage
r;
import org.springframework.security.authentication.UsernamePasswordAu
thenticationToken;
import org.springframework.security.core.Authentication;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;
import project.study.gestao_tarefas.dto.auth.AuthenticationRequest;
import project.study.gestao_tarefas.dto.auth.AuthenticationResponse;
import project.study.gestao_tarefas.security.jwt.JwtService;

/**
 * Controlador responsável pelos endpoints de autenticação.
```

```
 * Permite que os usuários façam login e recebam um token JWT.
 */
@RestController
@RequestMapping("/api/auth")
@RequiredArgsConstructor
public class AuthController {

    private final AuthenticationManager authenticationManager;
    private final JwtService jwtService;

    /**
     * Endpoint para autenticação de usuário.
     * Recebe um nome de usuário e senha, autentica o usuário e retorna um
token JWT.
     */
    @PostMapping("/login")
    public ResponseEntity<AuthenticationResponse> authenticate(
            @Valid @RequestBody AuthenticationRequest request
    ) {
        // Autentica o usuário usando o gerenciador de autenticação do Sprin
g Security
        Authentication authentication = authenticationManager.authenticate(
                new UsernamePasswordAuthenticationToken(
                        request.username(),
                        request.password()
                )
        );

        // Gera um token JWT para o usuário autenticado
                String jwt = jwtService.generateToken((UserDetails) authenticatio
n.getPrincipal());

        // Retorna o token na resposta
        return ResponseEntity.ok(new AuthenticationResponse(jwt));
    }
}
```

28 - Depois vamos pro segundo controlador, dessa vez o TestController

```
package project.study.gestao_tarefas.controller;

import org.springframework.http.ResponseEntity;
import org.springframework.security.access.prepost.PreAuthorize;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

/**
 * Controlador de teste para demonstrar os diferentes níveis de acesso.
 * Contém endpoints públicos, protegidos e restritos a administradores.
 */
@RestController
@RequestMapping("/api/test")
public class TestController {

    /**
     * Endpoint público - acessível por qualquer pessoa, mesmo sem autenti
cação.
     */
    @GetMapping("/public")
    public ResponseEntity<String> publicEndpoint() {
        return ResponseEntity.ok("Este é um endpoint público. Todos podem a
cessar.");
    }

    /**
     * Endpoint protegido - requer autenticação e a role ROLE_USER.
     */
    @GetMapping("/private")
    @PreAuthorize("hasRole('USER')")
    public ResponseEntity<String> privateEndpoint() {
        return ResponseEntity.ok("Este é um endpoint privado. Apenas usuári
os autenticados podem acessar.");
    }
```

```
    /**
     * Endpoint restrito - requer autenticação e a role ROLE_ADMIN.
     */
    @GetMapping("/admin")
    @PreAuthorize("hasRole('ADMIN')")
    public ResponseEntity<String> adminEndpoint() {
        return ResponseEntity.ok("Este é um endpoint de administrador. Apen
as administradores podem acessar.");
    }
}
```

29 - Vamos finalizar a implementação do Spring Security com Jwt adicionando dois DTOS, requisição de autenticação e resposta de autenticação

```
package project.study.gestao_tarefas.dto.auth;

import jakarta.validation.constraints.NotBlank;

/**
 * DTO que representa uma requisição de autenticação.
 * Contém as credenciais do usuário (nome de usuário e senha).
 */
public record AuthenticationRequest(
    @NotBlank(message = "O nome de usuário é obrigatório")
    String username,

    @NotBlank(message = "A senha é obrigatória")
    String password
) {}
```

```
package project.study.gestao_tarefas.dto.auth;

/**
 * DTO que representa uma resposta de autenticação bem-sucedida.
```

```
 * Contém o token JWT que o cliente pode usar para autenticar requisições futuras.
 */
public record AuthenticationResponse(
    String token,
    String type
) {
    public AuthenticationResponse(String token) {
        this(token, "Bearer");
    }
}
```

30 - Agora modificaremos nosso TaskService e TaskController para utilizar o Spring Security com JWT

```
package project.study.gestao_tarefas.service;

import lombok.RequiredArgsConstructor;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import project.study.gestao_tarefas.dto.TaskDTO;
import project.study.gestao_tarefas.entity.Task;
import project.study.gestao_tarefas.exception.ResourceNotFoundExceptio
n;
import project.study.gestao_tarefas.mapper.TaskMapper;
import project.study.gestao_tarefas.repository.TaskRepository;
import project.study.gestao_tarefas.repository.UserRepository;

import java.util.List;
import java.util.stream.Collectors;

@Service
@RequiredArgsConstructor
public class TaskService {

    private final TaskRepository taskRepository;
    private final TaskMapper taskMapper;
```

```
    private final UserRepository userRepository;

    @Transactional
    public TaskDTO createTask(TaskDTO taskDTO, Long userId) {
        // Verifica se o usuário existe
        if (!userRepository.existsById(userId)) {
            throw new ResourceNotFoundException("Usuário não encontrado c
om o ID: " + userId);
        }

        // Cria uma nova tarefa
        Task task = taskMapper.toEntity(taskDTO);

        // Salva a tarefa
        Task savedTask = taskRepository.save(task);

        // Retorna o DTO da tarefa salva
        return taskMapper.toDTO(savedTask);
    }

    @Transactional(readOnly = true)
    public List<TaskDTO> findAllByUserId(Long userId) {
        return taskRepository.findByUserId(userId).stream()
                .map(taskMapper::toDTO)
                .collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public TaskDTO findByIdAndUserId(Long id, Long userId) {
        return taskRepository.findByIdAndUserId(id, userId)
                .map(taskMapper::toDTO)
                .orElseThrow(() -> new ResourceNotFoundException(
                        "Tarefa não encontrada com o ID: " + id + " para o usuário I
D: " + userId));
    }

    @Transactional
    public TaskDTO update(Long id, TaskDTO taskDTO, Long userId) {
```

```
        Task task = taskRepository.findByIdAndUserId(id, userId)
                .orElseThrow(() -> new ResourceNotFoundException(
                        "Tarefa não encontrada com o ID: " + id + " para o usuário I
D: " + userId));

        // Atualiza os campos permitidos
        task.setName(taskDTO.name());
        task.setDescription(taskDTO.description());
        task.setStatus(taskDTO.status());

        Task updatedTask = taskRepository.save(task);
        return taskMapper.toDTO(updatedTask);
    }

    @Transactional
    public void delete(Long id, Long userId) {
        if (!taskRepository.existsByIdAndUserId(id, userId)) {
            throw new ResourceNotFoundException(
                    "Tarefa não encontrada com o ID: " + id + " para o usuário ID: "
+ userId);
        }
        taskRepository.deleteById(id);
    }
}
```

```
package project.study.gestao_tarefas.controller;

import jakarta.validation.Valid;
import lombok.RequiredArgsConstructor;
import org.springframework.http.ResponseEntity;
import org.springframework.security.core.annotation.AuthenticationPrincip
al;
import org.springframework.security.core.userdetails.UserDetails;
import org.springframework.web.bind.annotation.*;
import project.study.gestao_tarefas.dto.TaskDTO;
import project.study.gestao_tarefas.service.TaskService;
```

```
import java.net.URI;
import java.util.List;

@RestController
@RequestMapping("/api/tasks")
@RequiredArgsConstructor
public class TaskController {

    private final TaskService taskService;

    @PostMapping
    public ResponseEntity<TaskDTO> createTask(
            @Valid @RequestBody TaskDTO taskDTO,
            @AuthenticationPrincipal UserDetails userDetails) {

        Long userId = Long.parseLong(userDetails.getUsername());
        TaskDTO createdTask = taskService.createTask(taskDTO, userId);

        return ResponseEntity
                .created(URI.create("/api/tasks/" + createdTask.id()))
                .body(createdTask);
    }

    @GetMapping
    public ResponseEntity<List<TaskDTO>> getAllUserTasks(
            @AuthenticationPrincipal UserDetails userDetails) {

        Long userId = Long.parseLong(userDetails.getUsername());
        return ResponseEntity.ok(taskService.findAllByUserId(userId));
    }

    @GetMapping("/{id}")
    public ResponseEntity<TaskDTO> getTaskById(
            @PathVariable Long id,
            @AuthenticationPrincipal UserDetails userDetails) {

        Long userId = Long.parseLong(userDetails.getUsername());
```

```
        return ResponseEntity.ok(taskService.findByIdAndUserId(id, userId));
    }

    @PutMapping("/{id}")
    public ResponseEntity<TaskDTO> updateTask(
            @PathVariable Long id,
            @Valid @RequestBody TaskDTO taskDTO,
            @AuthenticationPrincipal UserDetails userDetails) {

        Long userId = Long.parseLong(userDetails.getUsername());
        return ResponseEntity.ok(taskService.update(id, taskDTO, userId));
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> deleteTask(
            @PathVariable Long id,
            @AuthenticationPrincipal UserDetails userDetails) {

        Long userId = Long.parseLong(userDetails.getUsername());
        taskService.delete(id, userId);
        return ResponseEntity.noContent().build();
    }
}
```