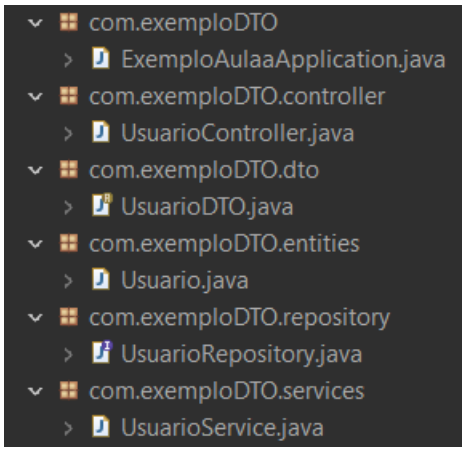
Descrição do projeto DTO

Criação dos pacotes:



Cada camada dentro de um pacote específico e ambos os pacotes são filhos do “com.exemploDTO”. Com uma sequência de criação, entities primeiro, depois DTO, logo em seguida repository, service e controller.

Criação entities:

```java
@Data
@NoArgsConstructor
@AllArgsConstructor
@Entity
@Table (name = "usuario")
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotNull
    private String nome;

    @NotNull
    private String senha;

    private String permissao;

    public Usuario(String nome, String senha) {
        this.nome = nome;
        this.senha = senha;
    }
}
```

Com a utilização do “lombook” nos 3 primeiros arrobas, o código se torna mais simplificado substituindo os “getters” e “setters” que deixam o código extenso.

Em seguida, tem a criação da camada entities com o “@Entity” e da tabela com o “@Table”.

A seguir a criação da classe “Usuario”, onde é armazenado as variáveis e construtor.

O “@Id” é a criação do id, um dado exclusivo para cada usuario/cliente, em conjunto do “@GeneratedValue” o id é criado pelo próprio banco de dados (o usuário não precisara informar essa camada).

“Private Long id”, sinaliza qual tipo de variável e seu encapsulamento (no caso o id é privado), e isso se estende para todas as variáveis da tabela.

O “@NotNull” sinaliza que não permite valores nulos.

O “private permissao” nesse código é um exemplo da utilização do DTO, onde essa variável vai estar fora do DTO e assim não aparecendo no teste do postman.

E por último tem o construtor:

```
public Usuario(String nome, String senha) {
    this.nome = nome;
    this.senha = senha;
}
```

Criação DTO:

Diferente do comum na criação do DTO, não se utiliza “class” e sim “record”, que é basicamente uma classe imutável, que promove ainda mais segurança, já que por si só, DTO tem como principal fundamento garantir um encapsulamento excelente, nesse código especificamente no métodos “POST” e “PUT”.

```
public record UsuarioDTO(Long id, String nome, String senha) {

}
```

Mais um construtor.

Criação Repository:

Em resumo, esse código declara uma interface que herda funcionalidades prontas de persistência de dados do Spring Data JPA para a entidade Usuario, como metodos de salvar, deletar e etc.

```
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {

}
```

Criação Service:

@Service serve para a criação dessa camada, onde ser armazenado os metodos, salvar, atualizar, deletar e buscar

METODOS

```
public UsuarioDTO salvar(UsuarioDTO usuarioDTO) {
    Usuario usuario = new Usuario(usuarioDTO.nome(),usuarioDTO.senha());
    Usuario salvarUsuario = usuarioRepository.save(usuario);
    return new UsuarioDTO(salvarUsuario.getId(),salvarUsuario.getNome(),salvarUsuario.getSenha());
}
public UsuarioDTO atualizar(Long id,UsuarioDTO usuarioDTO) {
    Usuario existeUsuario = usuarioRepository.findById(id).orElseThrow(()-> new RuntimeException("Usuario não e
    existeUsuario.setNome(usuarioDTO.nome());
    existeUsuario.setSenha(usuarioDTO.senha());

    Usuario updateUsuario = usuarioRepository.save(existeUsuario);
    return new UsuarioDTO(updateUsuario.getId(),updateUsuario.getNome(),updateUsuario.getSenha());
}

public boolean deleteUsuario(Long id) {
    Optional<Usuario> existingUsuario = usuarioRepository.findById(id);
    if (existingUsuario.isPresent()) {
        usuarioRepository.deleteById(id);
       return true;
    }
    return false;
}
public List<Usuario> buscarTodos(){
    return usuarioRepository.findAll();
}
public Usuario buscarPorId(Long id) {
    Optional <Usuario> usuario = usuarioRepository.findById(id);
    return usuario.orElse(null);
}
```

```
@Autowired
public UsuarioService(UsuarioRepository usuarioRepository) {
     this.usuarioRepository = usuarioRepository;
}
```

Mas antes disso tem o “@Autowired” (que realiza a implementação de dependências automaticas) em conjunto com um construtor ligando o a camada service e repository.

```
public UsuarioDTO salvar(UsuarioDTO usuarioDTO) {
    Usuario usuario = new Usuario(usuarioDTO.nome(),usuarioDTO.senha());
    Usuario salvarUsuario = usuarioRepository.save(usuario);
    return new UsuarioDTO(salvarUsuario.getId(),salvarUsuario.getNome(),salvarUsuario.getSenha());
}
```

Este método recebe um objeto UsuarioDTO.
Cria um novo objeto Usuario com base nos dados do UsuarioDTO.
Salva o novo usuário no banco de dados usando o usuarioRepository.save(usuario).
Retorna um novo objeto UsuarioDTO com as informações do usuário recém-salvo.

```
public UsuarioDTO atualizar(Long id,UsuarioDTO usuarioDTO) {
    Usuario existeUsuario = usuarioRepository.findById(id).orElseThrow(()-> new RuntimeException("Usuario não encontrado"));
    existeUsuario.setNome(usuarioDTO.nome());
    existeUsuario.setSenha(usuarioDTO.senha());

    Usuario updateUsuario = usuarioRepository.save(existeUsuario);
    return new UsuarioDTO(updateUsuario.getId(),updateUsuario.getNome(),updateUsuario.getSenha());
```

Este método atualiza um usuário existente no banco de dados.
Obtém o usuário existente pelo ID usando usuarioRepository.findById(id).
Se o usuário não existe, lança uma exceção.
Atualiza as informações do usuário existente com base nos dados do UsuarioDTO.
Salva as alterações no banco de dados usando usuarioRepository.save(existeUsuario).
Retorna um novo objeto UsuarioDTO com as informações atualizadas do usuário.

```java
public boolean deleteUsuario(Long id) {
    Optional<Usuario> existingUsuario = usuarioRepository.findById(id);
    if (existingUsuario.isPresent()) {
        usuarioRepository.deleteById(id);
       return true;
    }
    return false;
}
```

Este método exclui um usuário com base no ID fornecido.

Verifica se o usuário existe usando usuarioRepository.findById(id).

Se o usuário existir, ele é removido do banco de dados usando usuarioRepository.deleteById(id).

Retorna true se o usuário foi removido com sucesso, caso contrário, retorna false.

```java
public List<Usuario> buscarTodos(){
    return usuarioRepository.findAll();
}
public Usuario buscarPorId(Long id) {
    Optional <Usuario> usuario = usuarioRepository.findById(id);
    return usuario.orElse(null);
}
```

buscarTodos():

Retorna uma lista de todos os usuários no banco de dados usando usuarioRepository.findAll().

buscarPorId(Long id):

Retorna um usuário específico com base no ID fornecido.

Usa usuarioRepository.findById(id) para obter o usuário.

Se o usuário existe, retorna o objeto Usuario, caso contrário, retorna null.

Criação Controller:

O @RestController sinaliza a criação da camada controller.

O @RequestMapping é como será realizado a busca pelo postam ou Swagger

“@Autowired” (que realiza a implementação de dependências automaticas) em conjunto com um construtor ligando o a camada service e controller.

Em seguida é realizado as implementações do metodo get por id

```java
@GetMapping("/{id}")
@Operation(summary = "Localiza Usuario por ID")
public ResponseEntity<Usuario> getProductById(@PathVariable Long id) {
    Usuario usuario = usuarioService.buscarPorId(id);
    if (usuario != null) {
        return ResponseEntity.ok(usuario);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Mapeia a requisição GET.
Recupera um usuário pelo ID usando o serviço usuarioService.buscarPorId(id).
Se o usuário for encontrado, retorna uma resposta HTTP 200 (OK) com o usuário no corpo da resposta. Caso contrário, retorna uma resposta HTTP 404 (Not Found).

Get All e Post:

```java
@GetMapping("/")
@Operation(summary = "Apresenta todos os Usuarios")
public ResponseEntity<List<Usuario>> buscaTodos() {
    List<Usuario> usuarios = usuarioService.buscarTodos();
    return ResponseEntity.ok(usuarios);
}
@PostMapping("/")
@Operation(summary = "Cadastra um Usuario")
public ResponseEntity<UsuarioDTO> criar(@RequestBody @Valid UsuarioDTO usuarioDTO) {
    UsuarioDTO salvarUsuario = usuarioService.salvar(usuarioDTO);
    return ResponseEntity.status(HttpStatus.CREATED).body(salvarUsuario);
}
```

@GetMapping("/") - buscaTodos:

Mapeia a requisição GET.
Obtém todos os usuários usando usuarioService.buscarTodos().
Retorna uma resposta HTTP 200 (OK) com a lista de usuários no corpo da resposta.
@PostMapping("/") - criar:
Mapeia a requisição POST.
Recebe um objeto JSON representando um novo usuário no corpo da requisição (@RequestBody @Valid UsuarioDTO usuarioDTO).
Cria o usuário usando usuarioService.salvar(usuarioDTO).
Retorna uma resposta HTTP 201 (Created) com o usuário recém-criado no corpo da resposta.

Delete:

```java
@DeleteMapping("/{id}")
@Operation(summary = "Deleta o Usuario")
public ResponseEntity<Usuario> apagar(@PathVariable Long id) {
    boolean deleted = usuarioService.deleteUsuario(id);
    if (deleted) {
        return ResponseEntity.status(HttpStatus.NO_CONTENT).build();
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Mapeia a requisição DELETE.
Exclui o usuário pelo ID usando usuarioService.deleteUsuario(id).
Retorna uma resposta HTTP 204 (No Content) se a exclusão for bem-sucedida. Caso contrário, retorna uma resposta HTTP 404 (Not Found).

Put:

```java
@PutMapping("/{id}")
@Operation(summary = "Altera a Tarefa")
public ResponseEntity<UsuarioDTO> alterar(@PathVariable Long id, @RequestBody @Valid UsuarioDTO usuarioDTO) {
    UsuarioDTO alteraUsuarioDTO = usuarioService.salvar(usuarioDTO);
    if (alteraUsuarioDTO != null) {
        return ResponseEntity.ok(alteraUsuarioDTO);
    } else {
        return ResponseEntity.notFound().build();
    }
}
```

Mapeia a requisição PUT.
Recebe um objeto JSON representando as alterações no usuário no corpo da requisição (@RequestBody @Valid UsuarioDTO usuarioDTO).
Atualiza o usuário pelo ID usando usuarioService.salvar(usuarioDTO).
Retorna uma resposta HTTP 200 (OK) com o usuário atualizado no corpo da resposta se a atualização for bem-sucedida. Caso contrário, retorna uma resposta HTTP 404 (Not Found).

Criação Application:

```
spring.datasource.url=jdbc:mysql://localhost:3306/tarefas
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.datasource.username=root
spring.datasource.password=P@ssw0rd


spring.jpa.database=mysql
spring.jpa.show-sql=true
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.properties.hibernate.use_sql_comments=true
spring.jpa.defer-datasource-initialization=true
spring.jpa.hibernate.ddl-auto=update
```

Cria uma tabela no mysql onde sera armazenado e configurado o banco de dados.