

Desenvolvimento de APIs REST

08 - Services e DTOs

- Flyway
- Controller - Gerando URIs
- Exceptions “UnprocessableEntity”
- DTOs - Data Transfer Objects

Camadas

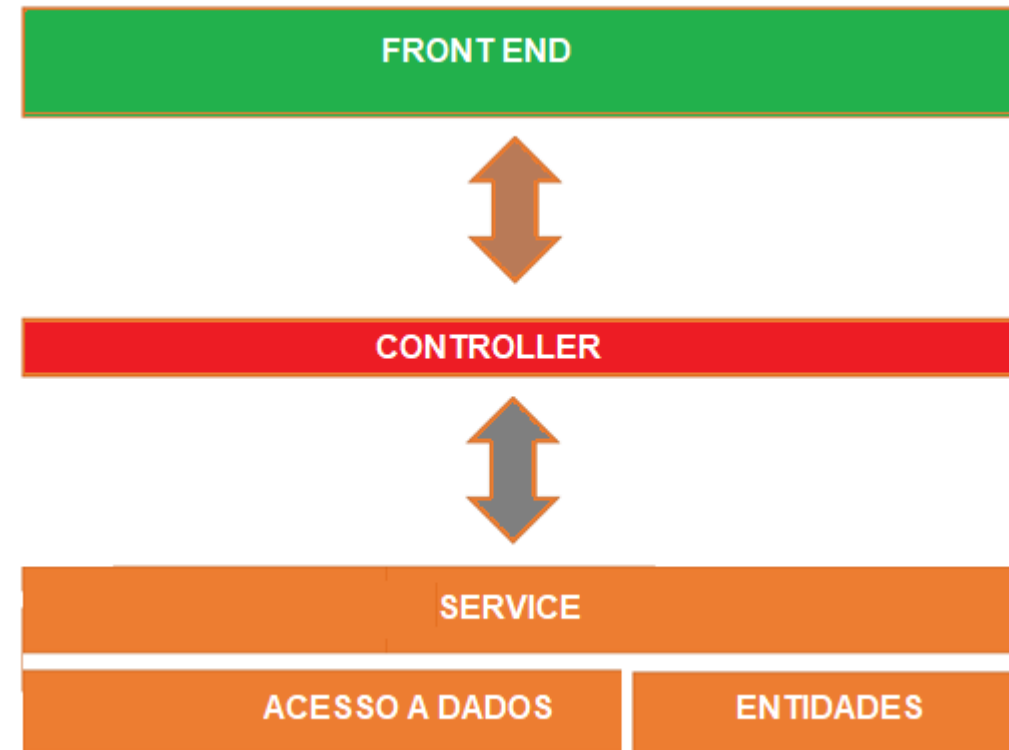


Padrão em Camadas

Cada camada conversa com sua camada adjacente

Vantagens

- Organização do código
- Desacoplamento
- Facilita Manutenção
- Cada camada tem sua responsabilidade
- O que é modificado em uma camada não interfere em outras camadas



Novo Projeto



Criar um novo projeto com o nome **projeto-service-dto**:

Name: projeto-service-dto

Group: org.serratec.backend

Artifact: projeto-service-dto

Version: 0.0.0-SNAPSHOT

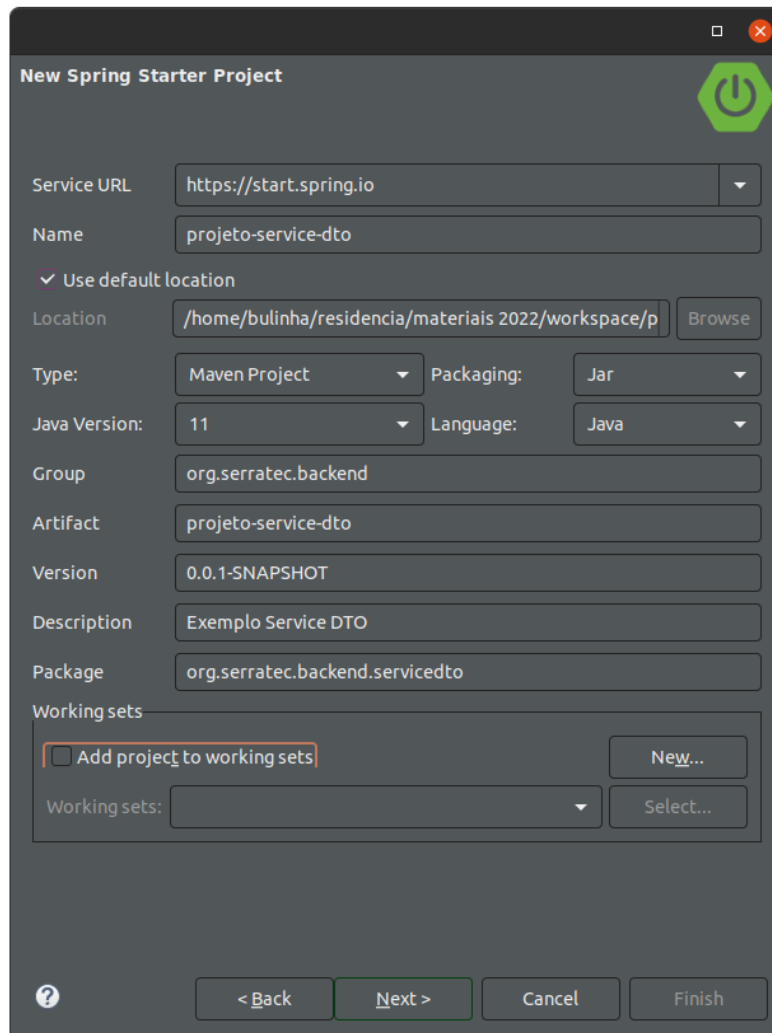
Description: Exemplo Service e DTO

Package: org.serratec.backend.servicedto

Spring Boot Version: 2.7.3

Dependencies:

- Spring Boot DevTools
- Spring Data JPA
- Flyway Migration
- PostgreSQL Driver
- Spring Web



New Spring Starter Project

Service URL:

Name:

☒ Use default location

Location:

Type: Packaging:

Java Version: Language:

Group:

Artifact:

Version:

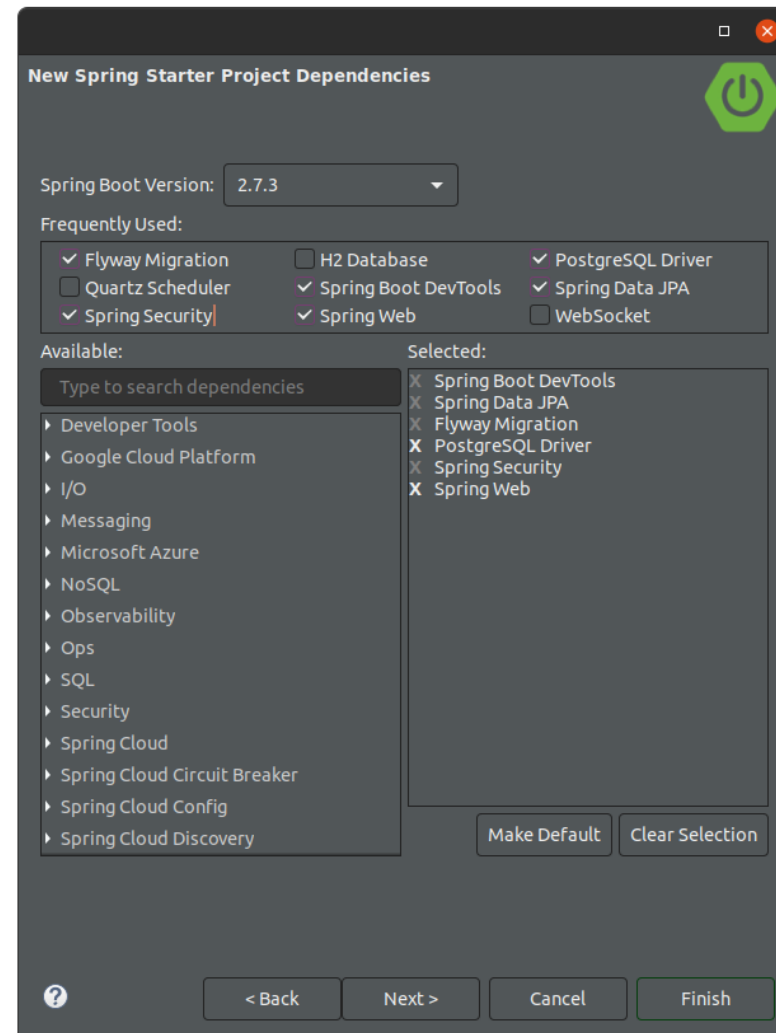
Description:

Package:

Working sets

☐ Add project to working sets

Working sets:



New Spring Starter Project Dependencies

Spring Boot Version:

Frequently Used:

<input checked="" type="checkbox"/> Flyway Migration	<input type="checkbox"/> H2 Database	<input checked="" type="checkbox"/> PostgreSQL Driver
<input type="checkbox"/> Quartz Scheduler	<input checked="" type="checkbox"/> Spring Boot DevTools	<input checked="" type="checkbox"/> Spring Data JPA
<input checked="" type="checkbox"/> Spring Security	<input checked="" type="checkbox"/> Spring Web	<input type="checkbox"/> WebSocket

Available:

Type to search dependencies

- Developer Tools
- Google Cloud Platform
- I/O
- Messaging
- Microsoft Azure
- NoSQL
- Observability
- Ops
- SQL
- Security
- Spring Cloud
- Spring Cloud Circuit Breaker
- Spring Cloud Config
- Spring Cloud Discovery

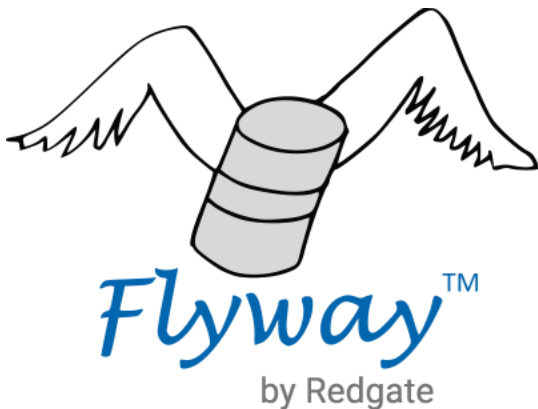
Selected:

- ☒ Spring Boot DevTools
- ☒ Spring Data JPA
- ☒ Flyway Migration
- ☒ PostgreSQL Driver
- ☒ Spring Security
- ☒ Spring Web

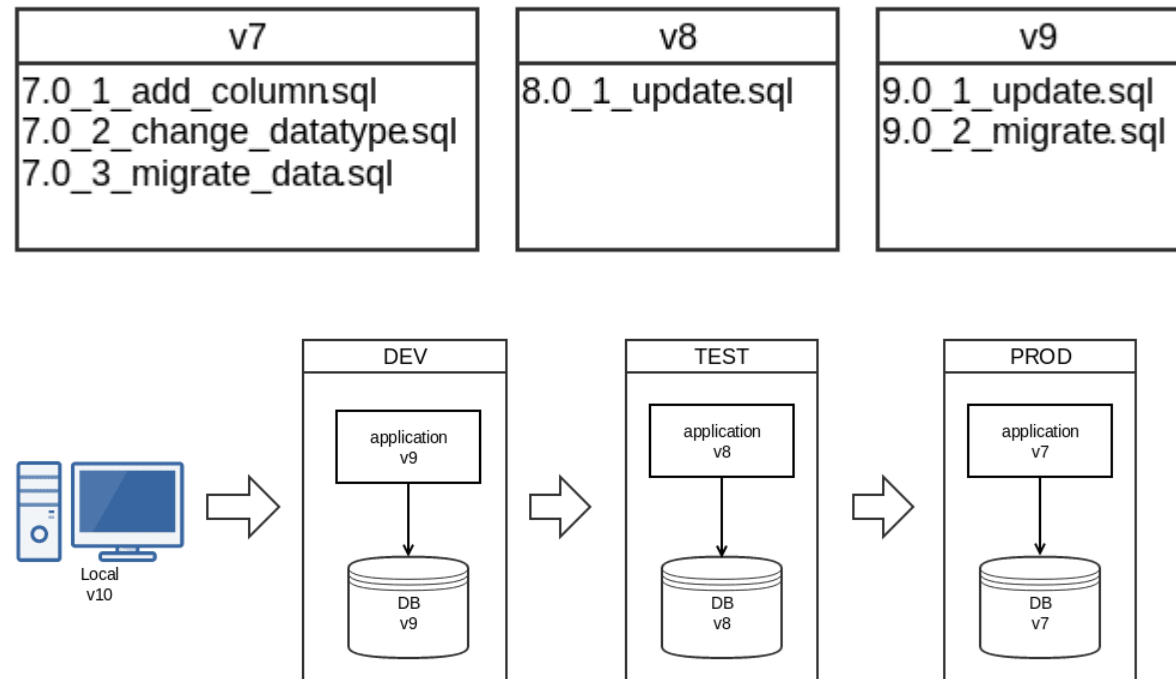
Flyway



Flyway é uma biblioteca de controle de versão para evoluir o banco de dados de uma aplicação de maneira fácil e confiável entre todas as suas instâncias (desenvolvimento, qualidade e produção)



Atualiza um banco de dados de uma versão para outra usando migrações. Podemos escrever migrações em SQL com sintaxe específica de banco de dados ou em Java para transformações avançadas de banco de dados.

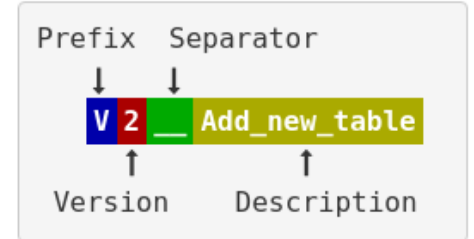


Mais informações:

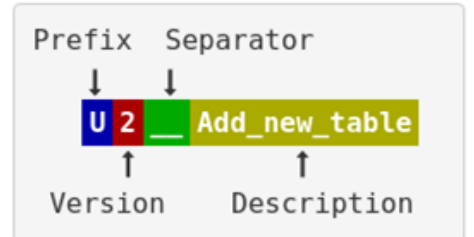
<https://www.baeldung.com/database-migrations-with-flyway>
https://www.tutorialspoint.com/spring_boot/spring_boot_flyway_database.htm
<https://blog.cvinicius.com.br/2018/02/versionamento-de-banco-dados-com-flyway.html>
<https://flywaydb.org/documentation/concepts/migrations.html>

ATENÇÃO: o nome do arquivo deve conter prefixo, número de versão, **2 e** caracteres "underscore" usado como separador e a descrição da migração

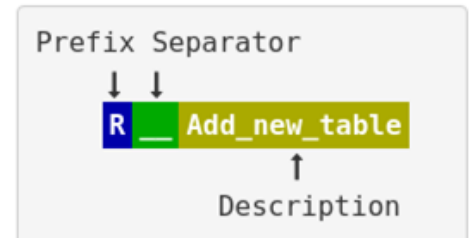
Versioned Migrations



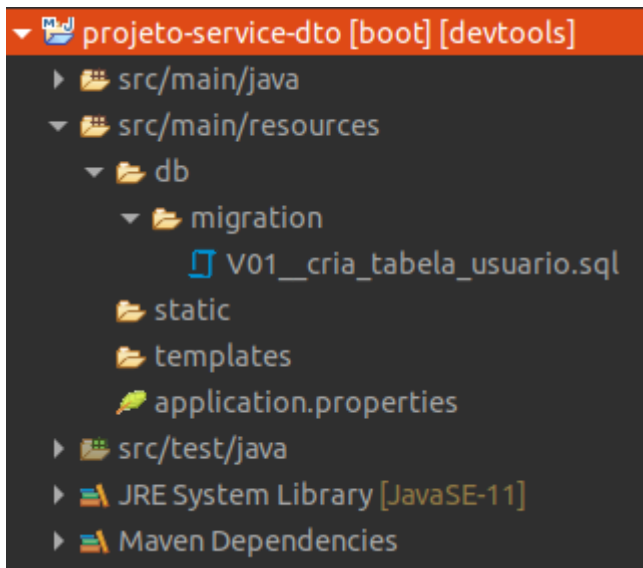
Undo Migrations



Repeatable Migrations



Configurando a Aplicação



Como incluímos no projeto a biblioteca **flyway**, o Wizard de projetos Spring Boot já criou uma pasta **db/migration** para incluir os arquivos de migração do banco de dados. Vamos criar o primeiro arquivo **V01__cria_tabela_usuario.sql** com o script para criar a tabela **usuario** no banco de dados.

```
CREATE TABLE usuario (  
    id_usuario serial primary key,  
    nome varchar(60),  
    email varchar(60),  
    senha varchar(255)  
);
```

Criar um banco de dados no **Postgres** com o nome **projeto**. Inserir o conteúdo abaixo no arquivo `application.properties`

```
spring.datasource.url=jdbc:postgresql://localhost:5432/projeto  
spring.datasource.username=postgres  
spring.datasource.password=postgres  
spring.jpa.show-sql=true  
spring.jpa.hibernate.ddl-auto=none  
spring.jackson.deserialization.fail-on-unknown-properties=false
```

Execução e criação das tabelas pelo Flyway



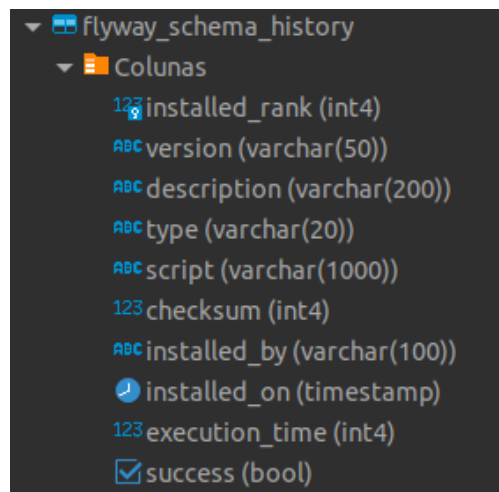
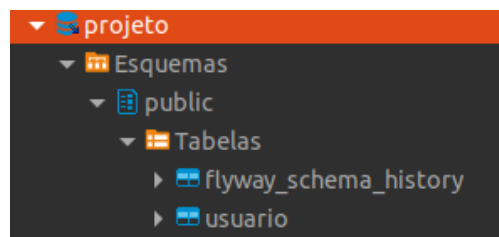
No log que aparece no console, é possível visualizar a execução do script pelo flyway

```
edMain] o.f.core.internal.command.DbValidate : Successfully validated 1 migration (execution time 00:00.017s)
edMain] o.f.c.i.s.JdbcTableSchemaHistory : Creating Schema History table "public"."flyway_schema_history"
edMain] o.f.core.internal.command.DbMigrate : Current version of schema "public": << Empty Schema >>
edMain] o.f.core.internal.command.DbMigrate : Migrating schema "public" to version "01 - cria tabela usuario"
edMain] o.f.core.internal.command.DbMigrate : Successfully applied 1 migration to schema "public", now at vers
```

Flyway cria a tabela **flyway_schema_history** para controlar a execução dos scripts indicando a versão, descrição, usuário de execução, tempo de execução e até um **checksum** do arquivo.

Em outras execuções, ele verifica se o arquivo de script sofreu alteração com base no checksum, e nesse caso ele lança uma exceção.

Para efetuar alterações no banco de dados, devemos adicionar mais scripts com estas alterações no futuro. Ex: Caso queiramos adicionar um novo campo na tabela, não devemos alterar este script, e sim criar um novo com o comando **alter table**.



installed_rank	1
version	01
description	cria tabela usuario
type	SQL
script	V01__cria_tabela_usuario.sql
checksum	1363435703
installed_by	postgres
installed_on	2022-09-06 11:18:32.821
execution_time	140
success	true

Entidade



Inserir a classe de modelo **Usuario** com atributos, getter, setter, toString, equals e hashCode

```
@Entity
public class Usuario {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name = "id_usuario")
    private Long id;

    private String nome;
    private String email;
    private String senha;

    public Usuario();
    public Usuario(Long id, String nome, String email, String senha) {
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.senha = senha;
    }

    public Usuario() {
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }
}
```

Repository - consultas por nome de método



Inserir a interface **UsuarioRepository** com a assinatura `findByEmail` para retornar o usuário a partir de um e-mail passado

```
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long> {
    Usuario findByEmail(String email);
}
```

O Spring data permite criar consultas personalizadas baseada nos nomes dos métodos do Repositório:

List<Usuario> findByNomels(String parametro) List<Usuario> findByNomeEquals(String parametro)	Select * from usuario where nome="Luis"	List<Usuario> findOrderByldadeDescTop3()	order by idade desc limit 3
List<Usuario> findByNomelsNot(String parametro)	where not nome="Luis"	List<Usuario> findByNomeldade(String parametro, Integer valor)	nome="parametro" and idade=valor
List<Usuario> findByNomelsNull()	where nome is null	List<Usuario> findByldadeBetween(Integer inicio, Integer fim) List<Usuario> findByldadeIn(Collection<Integer> valores)	idade between inicio and fim idade in (15, 29, 32, 56)
List<Usuario> findByNomeStartingWith(String prefixo) List<Usuario> findByNomeContaining(String texto) List<Usuario> findByNameLike(String padraoLike)	nome like "A%" nome like "%A%" nome like <padraoLike>	List<Usuario> findByldadeLessThan(Integer valor) List<Usuario> findByldadeLessThanEqual(Integer valor) List<Usuario> findByldadeGreaterThan(Integer valor) List<Usuario> findByldadeGreaterThanEqual(Integer valor)	idade < valor idade <= valor idade > valor idade >= valor

Mais exemplos e informações em <https://www.baeldung.com/spring-data-derived-queries>

Service



```
@Service
public class UsuarioService {
    @Autowired
    private UsuarioRepository usuarioRepository;

    public List<Usuario> findAll(){
        return usuarioRepository.findAll();
    }

    public Usuario inserir(Usuario user) throws EmailException {
        Usuario usuario = usuarioRepository.findByEmail(user.getEmail());
        if (usuario!=null) {
            throw new EmailException("Email já existente");
        }
        return usuarioRepository.save(user);
    }
}
```

Criar as classes **UsuarioService** no pacote service para gerenciar as regras de negócio relacionadas ao usuário.

Em um aplicativo, a lógica de negócios reside na camada de serviço, portanto, usamos a anotação `@Service` para indicar que uma classe pertence a essa camada.

Criar as classes `EmailException` e `ControllerExceptionHandler`. **UnprocessableEntity** ocorre quando, apesar da requisição estar correta (as validações foram atendidas), ela não pode ser inserida por outro motivo (e-mail já existente), retornando o código de status 422

```
public class EmailException extends RuntimeException {

    public EmailException(String message) {
        super(message);
    }
}
```

```
@ControllerAdvice
public class ControllerExceptionHandler extends ResponseEntityExceptionHandler{
    @ExceptionHandler(EmailException.class)
    protected ResponseEntity<Object> handleEmailExceptoin(EmailException ex) {
        return ResponseEntity.unprocessableEntity().body(ex.getMessage());
    }
}
```

Controller - Gerando URI



Inserir a classe **UsuarioController**

No padrão Rest devemos inserir a informação no **headers** do registro que acabou de ser inserido na tabela do banco de dados.

fromCurrentRequest.path - a partir da uri atual, vamos adicionar o valor do **id** no final da uri sendo assim quando um recurso for criado saberemos como localizar o novo recurso.

O método **inserir** recebe o objeto no formato json. Esse objeto é persistido na base de dados e é retornada na resposta a URL do registro que foi criado, acessando a URL podemos consultar o objeto que foi criado.

O código de resposta **422 Unprocessable Entity** indica que o servidor entende o tipo de conteúdo da entidade da requisição, e a sintaxe da requisição está correta, mas não foi possível processar as instruções presentes.

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioController {
    @Autowired
    UsuarioService usuarioService;

    @GetMapping
    public ResponseEntity<List<Usuario>> listar() {
        return ResponseEntity.ok(usuarioService.findAll());
    }

    @PostMapping
    public ResponseEntity<Usuario> inserir(@RequestBody Usuario usuario) {
        usuario = usuarioService.inserir(usuario);
        URI uri = ServletUriComponentsBuilder
            .fromCurrentRequest()
            .path("/{id}")
            .buildAndExpand(usuario.getId())
            .toUri();
        return ResponseEntity.created(uri).body(usuario);
    }
}
```

Testando no Postman



Testando no Postman

POST http://localhost:8080/usuarios Send

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

```
1 {
2   ...."nome": "bulinha",
3   ...."email": "bulinha@hotmail.com",
4   ...."senha": "123456"
5 }
```

Body Cookies (1) **Headers (6)** Test Results 201 Created 862 ms 285 B Save Response

Pretty Raw Preview Visualize JSON Copy Search

```
1 {
2   "id": 1,
3   "nome": "bulinha",
4   "email": "bulinha@hotmail.com",
5   "senha": "123456"
6 }
```

POST http://localhost:8080/usuarios Send

Params Authorization Headers (10) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON Beautify

```
1 {
2   ...."nome": "bulinha",
3   ...."email": "bulinha@hotmail.com",
4   ...."senha": "123456"
5 }
```

Body Cookies (1) **Headers (6)** Test Results 201 Created 862 ms 285 B Save Response

KEY	VALUE
Location	ⓘ http://localhost:8080/usuarios/1
Content-Type	ⓘ application/json
Transfer-Encoding	ⓘ chunked
Date	ⓘ Wed, 07 Sep 2022 20:19:17 GMT
Keep-Alive	ⓘ timeout=60
Connection	ⓘ keep-alive

Testando email duplicado



POST ▼ http://localhost:8080/usuarios Send ▼

Params Authorization Headers (10) **Body** ● Pre-request Script Tests Settings Cookies Beautify

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL **JSON** ▼

```
1 {  
2   "nome": "antonio da silva",  
3   "email": "bulinha@hotmail.com",  
4   "senha": "234567"  
5 }
```

Body Cookies (1) Headers (5) Test Results 🌐 **422 Unprocessable Entity (WebDAV) (RFC 4918)** 19 ms 221 B Save Response ▼

Pretty Raw Preview Visualize Text ▼ 🔍

```
1 Email já existente
```

Testando get



GET ▼ http://localhost:8080/usuarios Send ▼

Params Authorization Headers (8) Body Pre-request Script Tests Settings Cookies

☒ none ☐ form-data ☐ x-www-form-urlencoded ☐ raw ☐ binary ☐ GraphQL

This request does not have a body

Body Cookies (1) Headers (5) Test Results 200 OK 27 ms 238 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼ ≡

```
1 [
2   {
3     "id": 1,
4     "nome": "bulinha",
5     "email": "bulinha@hotmail.com",
6     "senha": "123456"
7   }
8 ]
```

DTO



Data Transfer Object é um padrão de projetos que serve para o transporte de dados entre diferentes componentes de um sistema, diferentes instâncias ou processos de um sistema distribuído ou diferentes sistemas via serialização. A ideia consiste basicamente em agrupar um conjunto de atributos em uma classe simples de forma a otimizar a comunicação. Um DTO não tem acesso a banco de dados e faz o isolamento dos dados. Podemos projetar os dados que queremos exibir na resposta de uma API.

```
public class UsuarioDTO {  
    private Long id;  
    private String nome;  
    private String email;  
    public UsuarioDTO() {  
  
    }  
    public UsuarioDTO(Long id, String nome, String email) {  
        this.id = id;  
        this.nome = nome;  
        this.email = email;  
    }  
    public UsuarioDTO(Usuario usuario) {  
        this.id = usuario.getId();  
        this.nome = usuario.getNome();  
        this.email = usuario.getEmail();  
    }  
    // gets sets  
}
```

No nosso exemplo abaixo vamos criar a classe **UsuarioDTO** para não exibir o campo senha e perfil na resposta da requisição

O Jackson a biblioteca usada para serialização e deserialização, compara os atributos do JSON com os atributos da classe do objeto como parâmetro do método que está recebendo a requisição, no caso, o objeto usuário, com isso, basicamente, todos os atributos do JSON que tiverem um correspondente do mesmo nome na classe **Usuario** serão preenchidos.

Criamos um construtor recebendo um **Usuario** para facilitar a cópia dos dados para classe **UsuarioDTO**.

Uma entidade pode ter mais de um DTO, para atender situações diferentes. Como por exemplo, a entidade Usuario pode ter um UsuarioInsertDTO que além da senha tenha um outro atributo confirmaSenha, e o service faria a verificação se as senhas são idênticas, se não forem ele poderia lançar a exceção SenhasNaoBatemException.

DTO



Vamos alterar o método **findAll** para retornar um **UsuarioDTO**

Alterando o método **findAll** do **UsuarioService**

```
@Service
public class UsuarioService {

    public List<UsuarioDTO> findAll() {
        List<Usuario> usuarios = usuarioRepository.findAll();
        List<UsuarioDTO> usuariosDTO = new ArrayList<UsuarioDTO>();
        for(Usuario usuario: usuarios) {
            usuariosDTO.add(new UsuarioDTO(usuario));
        }
        return usuariosDTO;
    }
}
```

Ou usando Stream

```
public List<UsuarioDTO> findAll() {
    return usuarioRepository
        .findAll()
        .stream()
        .map(UsuarioDTO::new)
        .collect(Collectors.toList());
}
```

Alterando no **UsuarioController**

```
@RestController
@RequestMapping("/usuarios")
public class UsuarioController {
    @Autowired
    UsuarioService usuarioService;

    @GetMapping
    public ResponseEntity<List<UsuarioDTO>> listar() {
        return ResponseEntity.ok(usuarioService.findAll());
    }
}
```

DTO



Testando no Postman o GET
O atributo senha não veio como resposta

The screenshot shows the Postman interface for a GET request to `http://localhost:8080/usuarios`. The request is configured with the method `GET` and the URL `http://localhost:8080/usuarios`. The `Body` tab is selected, showing the message "This request does not have a body". The response is displayed in the bottom panel, showing a `200 OK` status, `24 ms` response time, and `221 B` response size. The response body is a JSON object with the following structure:

```
1  [
2    {
3      "id": 1,
4      "nome": "bulinha",
5      "email": "bulinha@hotmail.com"
6    }
7  ]
```


DTO



Vamos criar a classe **UsuarioInserirDTO** para inserção de dados na requisição pois não precisamos inserir o campo **id**.

```
public class UsuarioInserirDTO {  
    private String nome;  
    private String email;  
    private String senha;  
    private String confirmaSenha;  
  
    public UsuarioInserirDTO() {  
    }  
  
    public String getNome() {  
        return nome;  
    }  
    public void setNome(String nome) {  
        this.nome = nome;  
    }  
    public String getEmail() {  
        return email;  
    }  
    }  
  
    // gets sets !!!!
```

A classe **UsuarioInserirDTO** é para entrada de dados no controller e **UsuarioDTO** para retorno dos dados no frontend.

Com ela é usada apenas para inserção, não é necessário o atributo **id**, já que esse vai ser gerado pelo banco.

- criar os gets e sets
- Não é necessário o construtor recebendo o objeto Usuario, pois este DTO vai ser utilizado na inserção, ou seja, o controller irá instanciar ele.

DTO



Vamos alterar o método inserir do **Service**

```
public UsuarioDTO inserir(UsuarioInserirDTO user) throws EmailException {
    if (!user.getSenha().equalsIgnoreCase(user.getConfirmaSenha())) {
        throw new SenhaException("Senha e Confirma Senha não são iguais");
    }
    if (usuarioRepository.findByEmail(user.getEmail()) != null) {
        throw new EmailException("Email já existente");
    }
    Usuario usuario = new Usuario();
    usuario.setNome(user.getNome());
    usuario.setEmail(user.getEmail());
    usuario.setSenha(passwordEncoder.encode(user.getSenha()));
    return new UsuarioDTO(usuarioRepository.save(usuario));
}
```

Criar a classe **SenhaException**

```
public class SenhaException extends RuntimeException {
    {
        public SenhaException(String message) {
            super(message);
        }
    }
}
```

Incluir o método para tratar a **SenhaException** no **ControllerExceptionHandler**

```
@ExceptionHandler(SenhaException.class)
protected ResponseEntity<Object> handleEmailExceptoin(SenhaException ex) {
    return ResponseEntity.unprocessableEntity().body(ex.getMessage());
}
```



Vamos alterar o método inserir no **Controller**

```
@PostMapping
public ResponseEntity<UsuarioDTO> inserir(@RequestBody UsuarioInserirDTO usuario) {
    UsuarioDTO usuarioDTO = usuarioService.inserir(usuario);
    URI uri = ServletUriComponentsBuilder
        .fromCurrentRequest()
        .path("/{id}")
        .buildAndExpand(usuarioDTO.getId())
        .toUri();
    return ResponseEntity.created(uri).body(usuarioDTO);
}
```

DTO



Testando no Postman

POST ▼ http://localhost:8080/usuarios Send ▼

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON ▼ Beautify

```
1 {
2   ...."nome": "claudio",
3   ...."email": "claudio@email.com",
4   ...."senha": "123456",
5   ...."confirmaSenha": "654321"
6 }
```

Body Cookies Headers (5) Test Results 422 Unprocessable Entity (WebDAV) (RFC 4918) 66 ms 241 B Save Response ▼

Pretty Raw Preview Visualize Text ▼ 📄 🔍

1 Senha e Confirma Senha não são iguais

POST ▼ http://localhost:8080/usuarios Send ▼

Params Authorization Headers (9) **Body** Pre-request Script Tests Settings Cookies

☐ none ☐ form-data ☐ x-www-form-urlencoded ☒ raw ☐ binary ☐ GraphQL JSON ▼ Beautify

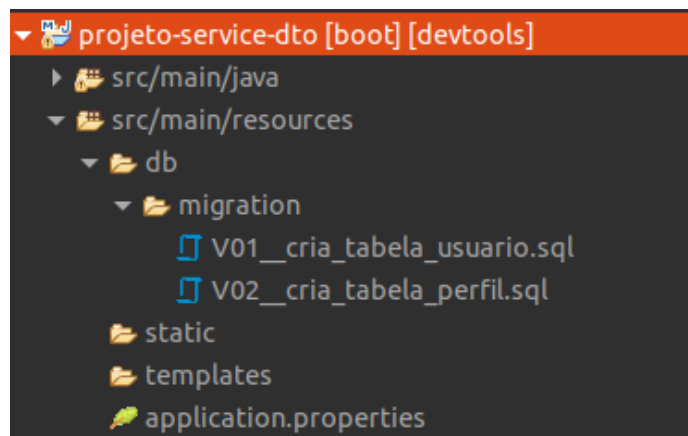
```
1 {
2   ...."nome": "claudio",
3   ...."email": "claudio@email.com",
4   ...."senha": "123456",
5   ...."confirmaSenha": "123456"
6 }
```

Body Cookies Headers (6) Test Results 201 Created 389 ms 266 B Save Response ▼

Pretty Raw Preview Visualize JSON ▼ 📄 🔍

```
1 {
2   "id": 2,
3   "nome": "claudio",
4   "email": "claudio@email.com"
5 }
```

Usuários e Perfis



Vamos criar as tabelas de **perfil** e **usuario_perfil** já com dois perfis. Basta criar o arquivo V02__cria_tabela_perfil.sql nas pasta migrations e executar o projeto novamente

```
CREATE TABLE perfil (  
    id_perfil serial primary key,  
    nome varchar(40)  
);  
  
CREATE TABLE usuario_perfil (  
    id_usuario int references usuario (id_usuario),  
    id_perfil int references perfil(id_perfil),  
    data_criacao date,  
    constraint pk_usuario_perfil primary key (id_usuario, id_perfil)  
);  
  
insert into perfil (id_perfil, nome) values  
(1, 'ROLE_ADMIN'),  
(2, 'ROLE_USER');
```

Atualizando o banco de dados

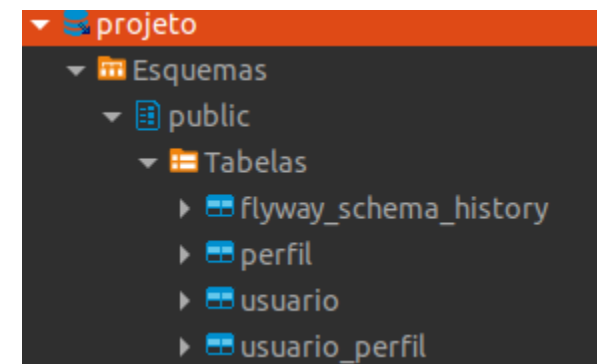


No log é possível verificar que o script de migração foi executado

```
.command.DbValidate      : Successfully validated 2 migrations (execution time 00:00.019s)
.command.DbMigrate       : Current version of schema "public": 01
.command.DbMigrate       : Migrating schema "public" to version "02 - cria tabela perfil"
.command.DbMigrate       : Successfully applied 1 migration to schema "public", now at version v02
```

As tabelas foram criadas corretamente e um novo registro foi inserido na tabela flyway_schema_history.

Através desta tabela que o flyway identifica quais scripts já foram executados e quais ainda não foram, para atualizar o banco de dados corretamente.



flyway_schema_history									
Enter a SQL expression to filter results (use Ctrl+Space)									
Grade	installed rank	version	description	type	script	checksum	installed by	installed on	
1	1	01	cria tabela usuario	SQL	V01_cria_tabela_usuario.sql	1.363.435.703	postgres	2022-09-07 16:54:40.177	
2	2	02	cria tabela perfil	SQL	V02_cria_tabela_perfil.sql	2.025.644.286	postgres	2022-09-07 17:36:38.768	

Classe Perfil



Criar a classe da entidade **Perfil**

```
@Entity
public class Perfil {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id_perfil")
    private Long id;

    private String nome;

    public Perfil() {
    }
    public Long getId() {
        return id;
    }
    public void setId(Long id) {
        this.id = id;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
}
```

Classe UsuarioPerfilPK



Criar a classe **UsuarioPerfilPK** que será anotada com a anotação **@Embeddable**. Ela vai conter os atributos **usuario** e **perfil** mapeados com os campos da tabela que serão a chave primária composta da tabela **usuario_perfil**.

```
@Embeddable
public class UsuarioPerfilPK implements Serializable{
    @ManyToOne
    @JoinColumn(name="id_usuario")
    private Usuario usuario;

    @ManyToOne
    @JoinColumn(name="id_perfil")
    private Perfil perfil;

    //... gets sets

    @Override
    public int hashCode() {
        return Objects.hash(perfil, usuario);
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        UsuarioPerfilPK other = (UsuarioPerfilPK) obj;
        return Objects.equals(perfil, other.perfil) && Objects.equals(usuario, other.usuario);
    }
}
```


Classe UsuarioPerfil



A anotação **EmbeddedId** é aplicada a um campo ou propriedade persistente de uma classe de entidade ou superclasse mapeada para indicar uma chave primária composta que é uma classe incorporável.

Na declaração do atributo id, instanciamos o objeto do tipo **UsuarioPerfilPK**

No construtor recebemos o **usuario** e o **perfil** e passamos as informações para o atributo id.

Criamos métodos **setUsuario** e **setPerfil** para atualizar o atributo id

```
@Entity
@Table(name="usuario_perfil")
public class UsuarioPerfil {
    @EmbeddedId
    private UsuarioPerfilPK id = new UsuarioPerfilPK();
    @Column(name= "data_criacao")
    private LocalDate dataCriacao;
    public UsuarioPerfil() {
    }
    public UsuarioPerfil (Usuario usuario, Perfil perfil, LocalDate dataCriacao) {
        this.id.setUsuario(usuario);
        this.id.setPerfil(perfil);
        this.dataCriacao=dataCriacao;
    }
    public UsuarioPerfilPK getId() {
        return id;
    }
    public void setId(UsuarioPerfilPK id) {
        this.id = id;
    }
    public LocalDate getDataCriacao() {
        return dataCriacao;
    }
    public void setDataCriacao(LocalDate dataCriacao) {
        this.dataCriacao = dataCriacao;
    }
    public void setUsuario(Usuario usuario) {
        this.id.setUsuario(usuario);
    }
    public void setPerfil(Perfil perfil) {
        this.id.setPerfil(perfil);
    }
}
```

Repository



Criar as interfaces de repositório **PerfilRepository**, e **UsuarioPerfilRepository**

```
@Repository
public interface PerfilRepository extends JpaRepository<Perfil, Long>{
}
```

```
@Repository
public interface UsuarioPerfilRepository extends JpaRepository<UsuarioPerfil, Long>{
}
```

Relacionamentos



Adicionar o relacionamento na classe **Usuario** não esquecendo de incluir os gets e sets

Usuario

```
@OneToMany(mappedBy = "id.usuario", fetch = FetchType.EAGER, cascade = CascadeType.ALL)
private Set<UsuarioPerfil> usuarioPerfis = new HashSet<>();

public Set<UsuarioPerfil> getUsuarioPerfis() {
    return usuarioPerfis;
}

public void setUsuarioPerfis(Set<UsuarioPerfil> usuarioPerfis) {
    this.usuarioPerfis = usuarioPerfis;
}
```

- FetchType.EAGER - Ao carregar um usuário, o spring data/hibernate irá carregar os perfis junto com ele
- CascadeType.ALL - ao salvar ou remover um usuário, os perfis serão salvos e removidos juntos com ele

DTO



Atualizar a classe **UsuarioDTO** e a classe **UsuarioInserirDTO** para incluir os Perfis (não esquecer o get e set)

```
public class UsuarioDTO {
    private Long id;
    private String nome;
    private String email;
    private Set<Perfil> perfis;
    public UsuarioDTO() {

    }
    public UsuarioDTO(Long id, String nome, String email) {
        this.id = id;
        this.nome = nome;
        this.email = email;
    }
    public UsuarioDTO(Usuario usuario) {
        this.id=usuario.getId();
        this.nome=usuario.getNome();
        this.email=usuario.getEmail();
        this.perfis = new HashSet<>();
        for(UsuarioPerfil usuarioPerfil: usuario.getUsuarioPerfis()){
            this.perfis.add(usuarioPerfil.getId().getPerfil());
        }
    }
    //gest e sets !!!
}
```

```
public class UsuarioInserirDTO {
    private String nome;
    private String email;
    private String senha;
    private String confirmaSenha;
    private Set<Perfil> perfis;

    public UsuarioInserirDTO() {
    }

    public String getNome() {
        return nome;
    }
    //gest e sets !!!
}
```

Service



Modificar o método da classe **UsuarioService**, e incluir os Autowired para **PerfilService** e **UsuarioPerfilRepository**

```
@Autowired
private PerfilService perfilService;

public UsuarioDTO inserir(UsuarioInserirDTO user) throws EmailException {
    if (!user.getSenha().equalsIgnoreCase(user.getConfirmaSenha())) {
        throw new SenhaException("Senha e Confirma Senha não são iguais");
    }
    if (usuarioRepository.findByEmail(user.getEmail()) != null) {
        throw new EmailException("Email já existente");
    }
    Usuario usuario = new Usuario();
    usuario.setNome(user.getNome());
    usuario.setEmail(user.getEmail());
    usuario.setSenha(user.getSenha());

    Set<UsuarioPerfil> perfis = new HashSet<>();
    for(Perfil perfil: user.getPerfis()) {
        perfil = perfilService.buscar(perfil.getId());
        UsuarioPerfil usuarioPerfil = new UsuarioPerfil(usuario, perfil, LocalDate.now());
        perfis.add(usuarioPerfil);
    }
    usuario.setUsuarioPerfis(perfis);
    usuario = usuarioRepository.save(usuario);

    return new UsuarioDTO(usuario);
}
```

Criar a classe **PerfilService**

```
@Service
public class PerfilService {

    @Autowired
    private PerfilRepository perfilRepository;

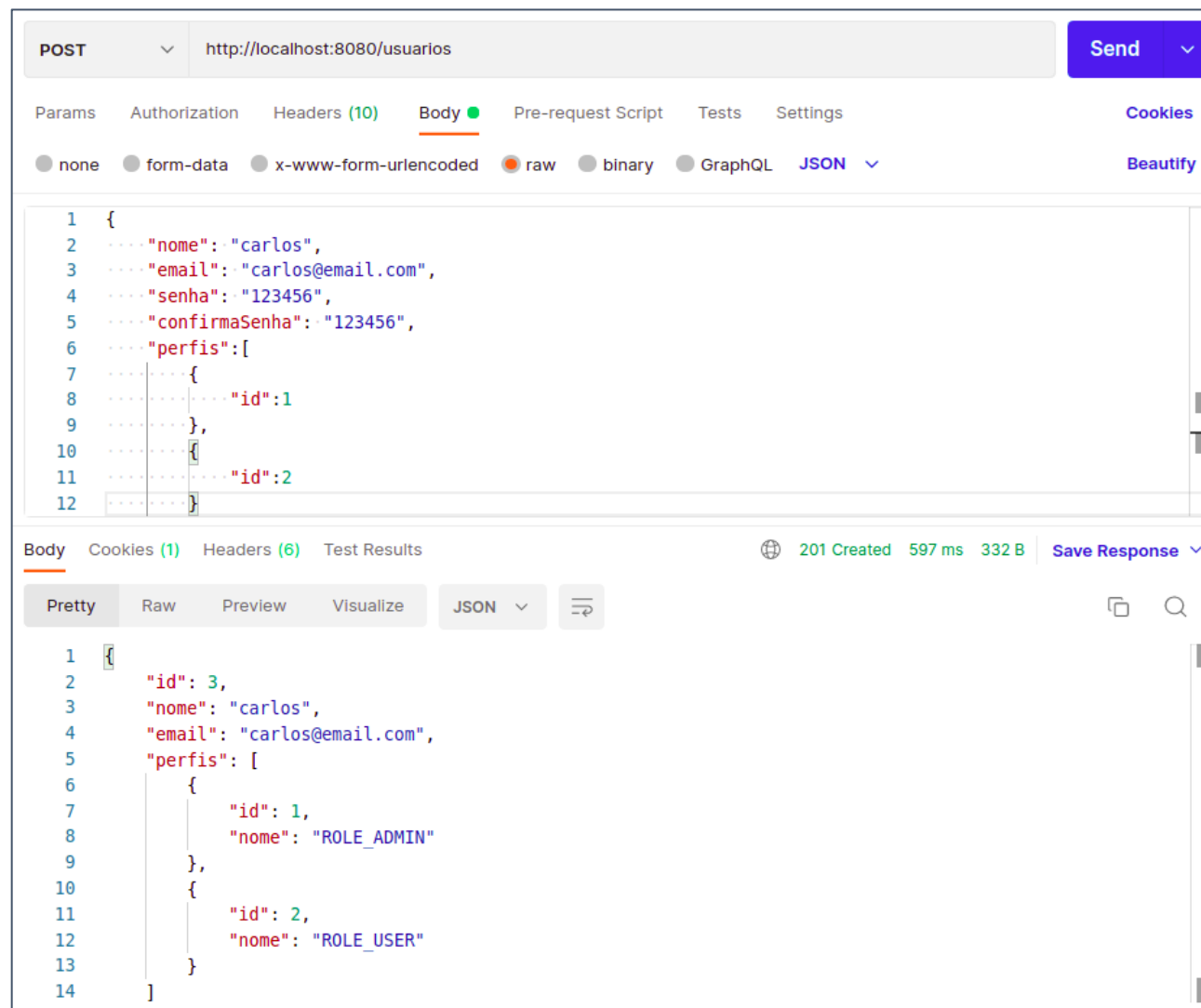
    public Perfil buscar(Long id) {
        Optional<Perfil> perfil =
            perfilRepository.findById(id);
        return perfil.get();
    }
}
```

Testando Postman



Inserindo o usuário Carlos com o perfil Admin e User como exemplo abaixo.

Não é necessário passar todo o objeto perfil, basta passar apenas o atributo id.

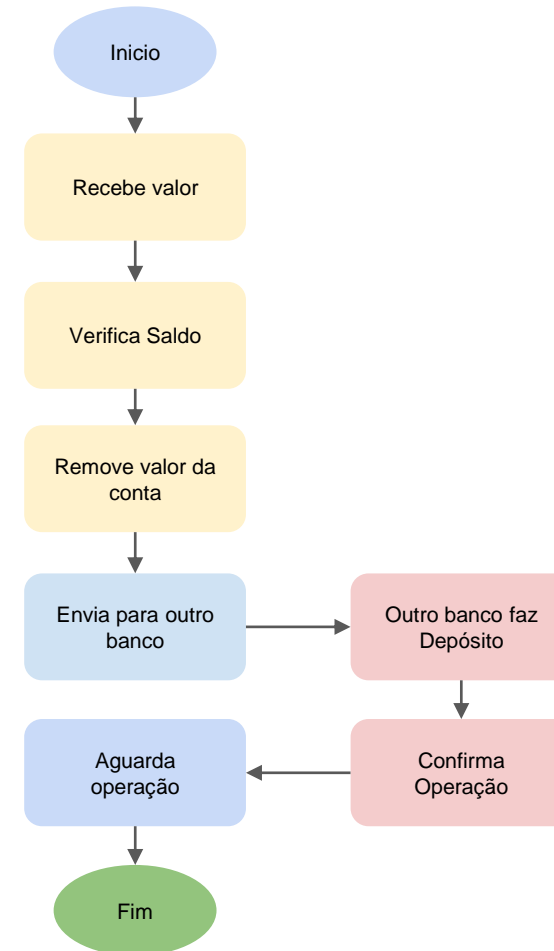


Transações



Realizar uma transferência bancária:

- Recebe o valor da transferência
- Verifica se há saldo na conta
- Remove o valor da conta
- Envia para o outro banco o valor
 - Faz comunicação com o banco
 - Aguarda a confirmação do outro banco
- Encerra a operação

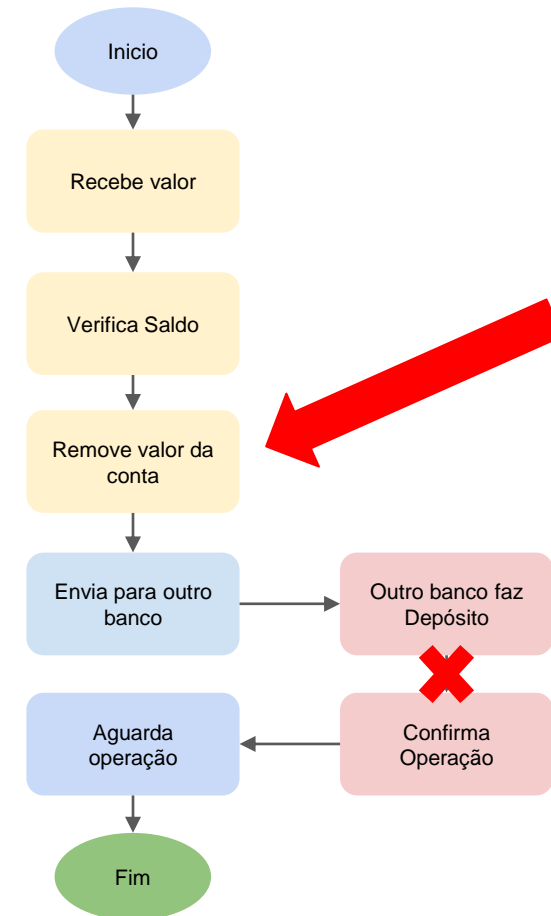


Transações



Realizar uma transferência bancária:

- Recebe o valor da transferência
- Verifica se há saldo na conta
- Remove o valor da conta
- Envia para o outro banco o valor
 - Faz comunicação com o banco
 - **Aguarda a confirmação do outro banco**
- **Encerra a operação**

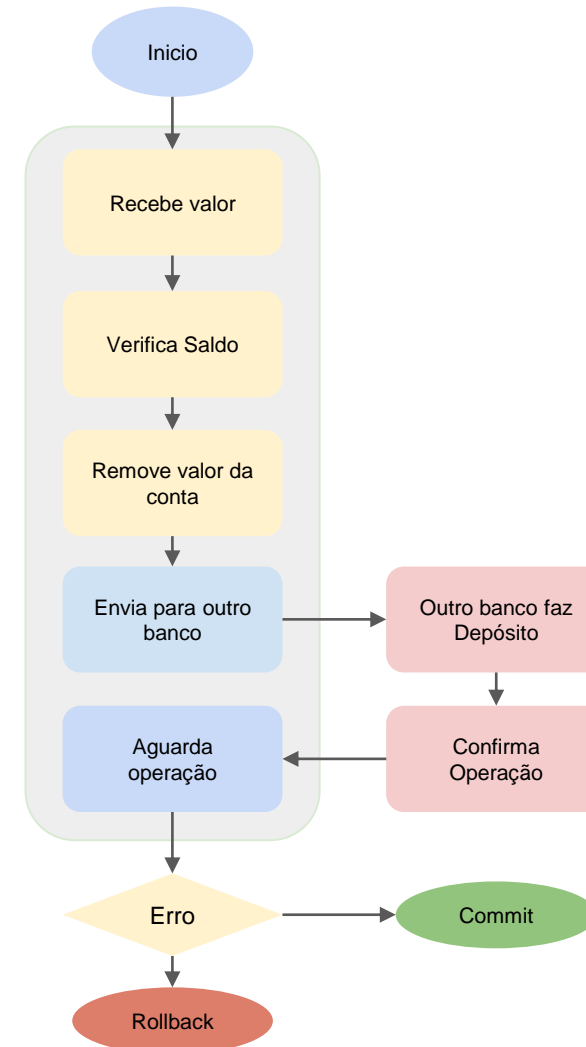


Transações



Realizar uma transferência bancária:

- Tentar (Begin Transaction - SQL)
 - Recebe o valor da transferência
 - Verifica se há saldo na conta
 - Remove o valor da conta
 - Envia para o outro banco o valor
 - Faz comunicação com o banco
 - Aguarda a confirmação do outro banco
 - Encerra a operação
- Se não houve erro
 - Confirma (Commit - SQL)
- Se houver erro
 - Reverte (Rollback - SQL)



Transações



A anotação `@Transactional` indica que todas as operações do método serão executadas dentro de uma transação do banco.

Por exemplo, se no método `inserir` de `UsuarioService`, ocorrer um erro no envio do e-mail, a anotação `@Transactional` fará com que a inserção do usuário seja desfeita, garantindo a integridade da operação.

```
@Transactional
public UsuarioDTO inserir(UsuarioInserirDTO user) throws EmailException {
```

IMPORTANTE: Existem duas anotações `@Transactional`, uma do Spring e outra do Java, utilizar a do Spring, cujo pacote e classe é:
`org.springframework.transaction.annotation.Transactional`

Mais informações <https://hellokoding.com/spring-boot/transactional/>