

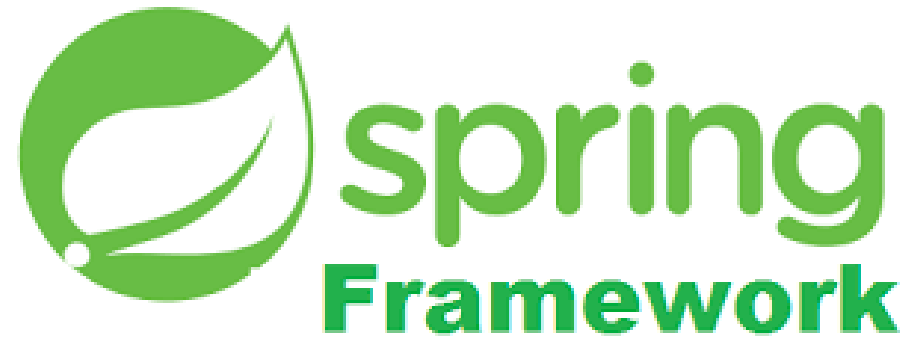
Tecnologias Para Back-End

Prof. JUNIO FIGUEIRÊDO

JUNIOINF@GMAIL.COM

AULA 04 – SPRING DATA JPA – Continuação....

Interfaces Repository



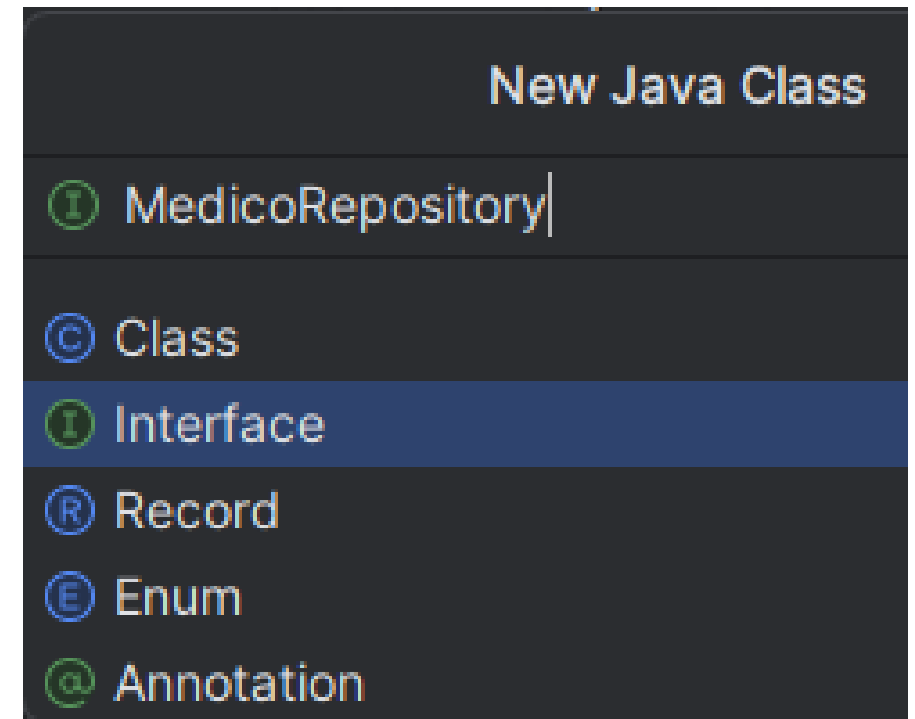
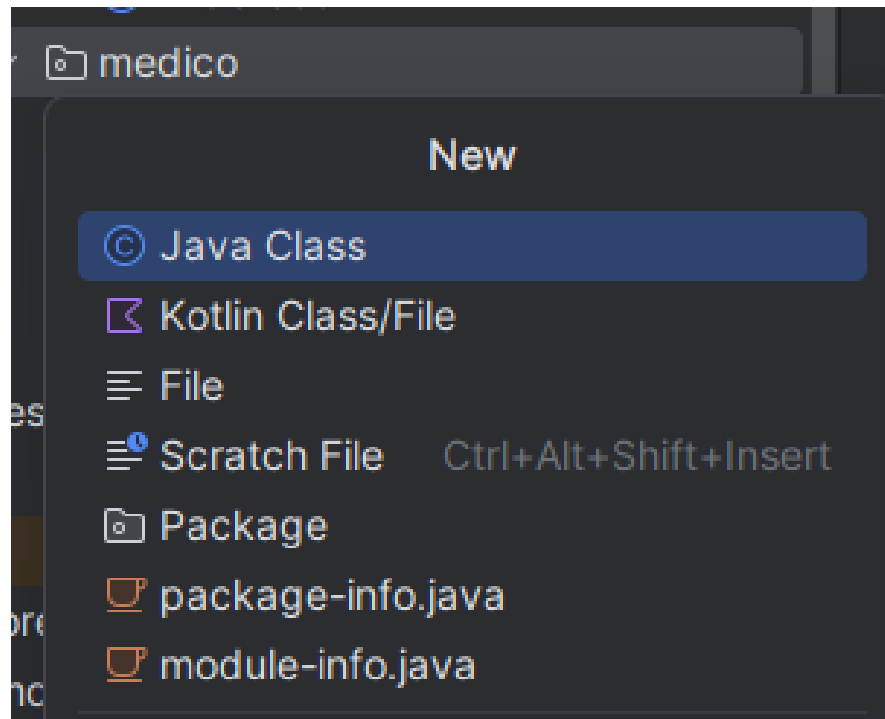
Spring

Persistência

- Para fazer a **persistência**, vamos pegar o **objeto Medico** e **salvar no banco de dados**,
- O Spring Data tem o **Repository**, que são **interfaces**.
- O Spring já **nos fornece a implementação**.
- **Interfaces** = nada mais que uma espécie de contrato de regras que uma classes deve seguir em um determinado contexto.

Spring

- Vamos criar uma nova interface em "java > med.voll.api > Medico". Como o atalho "**Alt** + **Insert**" e selecionar a opção "**Interface**". O nome será "**MedicoRepository**".



- Criaremos uma interface Java, sem elementos do Spring Data. Vamos herdar de uma interface chamada JpaRepository, usando um extends. Entre <>, passaremos dois tipos de objeto.
- O primeiro será o tipo da entidade trabalhada pelo repository, Medico, e o tipo do atributo da chave primária da entidade, Long.
- A interface está criada:

- A interface está criada:

```
package med.voll.api.medico;

import org.springframework.data.jpa.repository.JpaRepository;


public interface MedicoRepository extends JpaRepository<Medico, Long> {}
```

- Agora já podemos utilizar o repository no controller. Nele, apagaremos `System.out.println(dados);`. Vamos declarar o repository como um atributo da classe `MedicoController`.

- A Classe MedicoControle está assim:

```
@RestController
@RequestMapping("medicos")
public class MedicoController {

    no usages
    @PostMapping
    public void cadastrar(@RequestBody DadosCadastroMedico dados) {
        System.out.println(dados);
    }
}
```



- Apague `System.out.println(dados);`.

Spring

Persistência

- Criaremos o atributo `private MedicoRepository repository;`. Precisamos avisar ao Spring que esse novo atributo será instanciado por você(Spring).
- Como esse atributo é uma `Interface Repository`, e você(Spring) sabe fazer o carregamento d'ela. Então crie esse objeto e passe automaticamente para o controler.
- Faremos a `injeção de dependências` inserindo a `anotação @Autowired` acima do atributo.

- A **Injeção de Dependência** é uma técnica de desenvolvimento utilizada para evitar o **alto nível de acoplamento de código** (Quando os componentes de uma aplicação dependem muito uns dos outros) dentro de uma aplicação.
- A **anotação @Autowired** é uma das anotações mais usadas no Spring Framework **para injetar automaticamente dependências em classes gerenciadas pelo Spring**.

Spring

Persistência

- A anotação `@Autowired` é usada para injetar um objeto gerenciado pelo Spring em outra classe.
- O Spring Framework é capaz de detectar automaticamente as dependências que precisam ser injetadas em uma classe e fazer isso automaticamente.

- A Classe **MedicoControlle** está assim:

```
@RestController
@RequestMapping("medicos")
public class MedicoController {

    no usages
    @Autowired
    private MedicoRepository repository;

    no usages
    @PostMapping
    public void cadastrar(@RequestBody DadosCadastroMedico dados) {

    }
}
```

Persistência

- Vamos preparar a Persistencia:

```
public class MedicoController {  
    1 usage  
    @Autowired  
    private MedicoRepository repository;  
  
    no usages  
    @PostMapping  
    public void cadastrar(@RequestBody DadosCadastroMedico dados) {  
        repository.save(medico);  
    }  
}
```

- No método `cadastrar`, vamos inserir o método que fará o insert na tabela do banco de dados, `repository.save(medico);`.

Spring

Persistência

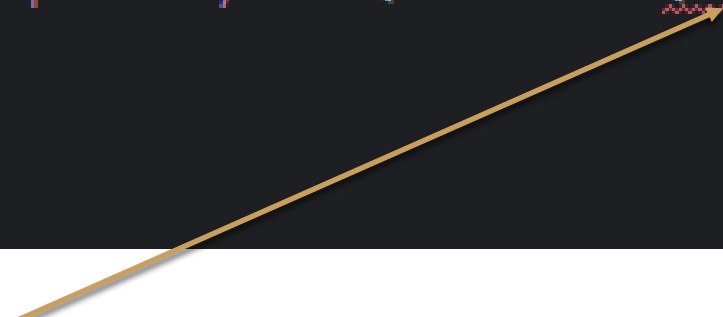
- No controller, não recebemos o objeto Medico, e sim um DTO (JSON) que é representado pelo java record DadosCadastroMedico.
- Precisaremos fazer a conversão!!!! Recebo como parâmetro um DTO e temos que transformá-lo em uma objeto do tipo medico, em um entidade JPA.
- Para isso, usaremos o construtor.
- No método save, vamos criar um construtor para receber DadosCadastroMedico.

Spring

Persistência

- Primeiro, passaremos `repository.save(new Medico(dados));`.

```
public class MedicoController {  
    1 usage  
    @Autowired  
    private MedicoRepository repository;  
    no usages  
    @PostMapping  
    public void cadastrar(@RequestBody DadosCadastroMedico dados) {  
        repository.save(new Medico(dados));  
    }  
}
```



- Quem é `dados`???

```
public void cadastrar(@RequestBody DadosCadastroMedico dados) {  
    repository.save(new Medico(null, dados.nome(), dados.crm(), dados.email(),  
        new Endereco(dados)));  
}
```



- As coisa estão começando a ficar complicado!!!!
- E ainda tem muita responsabilidade!!!!
- Vamos simplificar as coisas

Spring

Persistência

- Na sequência, daremos um "Alt + Enter" e selecionaremos **Create constructor**.

```
@PostMapping
public void cadastrar(@RequestBody DadosCadastroMedico dados) {
    repository.save(new Medico(dados));
}
```

 Create constructor
 Remove redundant arguments
Introduce local variable
Press Ctrl+Q to toggle preview

// Medico.java
public Medico(DadosCadastroMedico dados) {

}

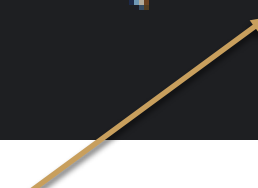
- O construtor ficará na classe Medico.

- O construtor da classe Medico.

```
1 usage  
public Medico(DadosCadastroMedico dados) {  
}
```

- Agora faremos a atribuição dos atributos, com:

```
public Medico(DadosCadastroMedico dados) {  
    this.nome = dados.nome();  
    this.email = dados.email();  
    this.crm = dados.crm();  
    this.especialidade = dados.especialidade();  
    this.endereco = new Endereco(dados.endereco());  
}
```



- Na sequência, daremos um "Alt + Enter" e selecionaremos **Create constructor**.

- Agora faremos a atribuição dos atributos, com
`this.nome = dados.nome();`
`this.email = dados.email();`
`this.crm = dados.crm();`
`this.especialidade = dados.especialidade();`
`this.endereco = new Endereco(dados.endereco());`

- O construtor da classe Endereco.

```
public Endereco(DadosEndereco dados) {  
    this.logradouro = dados.logradouro();  
    this.bairro = dados.bairro();  
    this.cep = dados.cep();  
    this.uf = dados.uf();  
    this.cidade = dados.cidade();  
    this.numero = dados.numero();  
    this.complemento = dados.complemento();  
}
```

- Agora faremos a atribuição dos atributos de Endereco.

```
this.logradouro = dados.logradouro();
```

```
this.bairro = dados.bairro();
```

```
this.cep = dados.cep();
```

```
this.uf = dados.uf();
```

```
this.cidade = dados.cidade();
```

```
this.numero = dados.numero();
```

```
this.complemento = dados.complemento();
```

- Vamos testar...
- Inicie o servidor.
- Veja o log, no próximo slide.
- Agora vá para o Insomnia
- Faça um requisição no Insomnia..
- O que aconteceu??? O porque do erro????
- Veja no log da aplicação que fica melhor de visualizar!!!!

- Veja o Log...

```
Database: jdbc:mysql://localhost:3306/vollmed_api (MySQL 8.0)
Schema history table `vollmed_api`.`flyway_schema_history` does not exist yet
Successfully validated 0 migrations (execution time 00:00.081s)
No migrations found. Are your locations set up correctly?
Creating Schema History table `vollmed_api`.`flyway_schema_history` ...
Current version of schema `vollmed_api`: << Empty Schema >>
Schema `vollmed_api` is up to date. No migration necessary.
HHH000204: Processing PersistenceUnitInfo [name: default]
HHH000412: Hibernate ORM core version 6.4.1.Final
```

Spring

Persistência

- No insomnia, foi disparado a requisição com as informações que temos. Como resposta, receberemos um erro 500.
- O problema foi causado porque no database vollmed_api, não existe a tabela .medicos.

Migrations com Flyway



Spring

- Vamos fazer com que a tabela seja encontrada pelo banco de dados.
- Usaremos **migrations**, ou ferramentas de migrações, para registrar as atualizações no banco de dados.

Spring

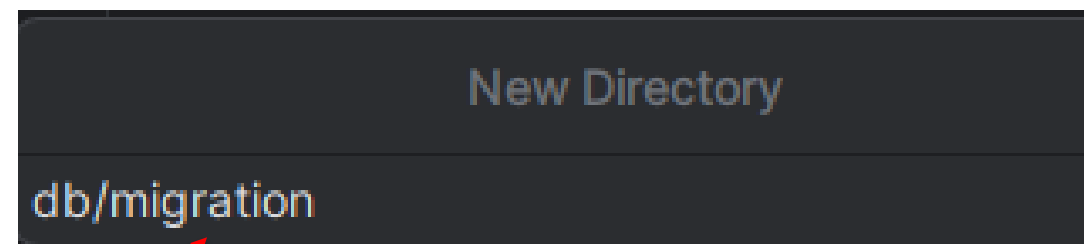
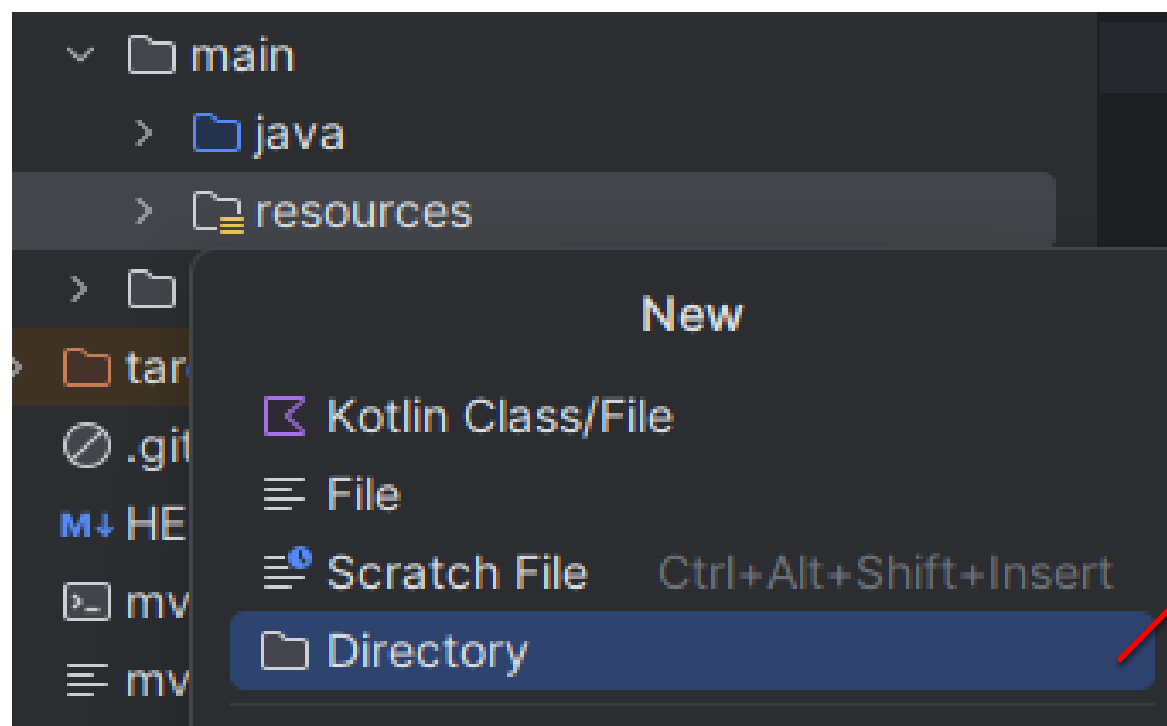
- Já registramos([pom.xml](#)) o **Flyway**, uma dessas ferramentas suportadas pelo Spring Boot.

```
<dependency>
  <groupId>org.flywaydb</groupId>
  <artifactId>flyway-mysql</artifactId>
</dependency>


<dependency>
  <groupId>com.mysql</groupId>
  <artifactId>mysql-connector-j</artifactId>
  <scope>runtime</scope>
</dependency>
```

Spring

- Para cada mudança (**criação de tabela, insert e etc**) que quisermos executar no bd, devemos criar um arquivo com extensão **.sql**.
- Então **na pasta** em "**resources**". "**Botão direito**", vamos **escolher a opção** "**Directory**" e digita o **nome da pasta**: "**db/migration**".



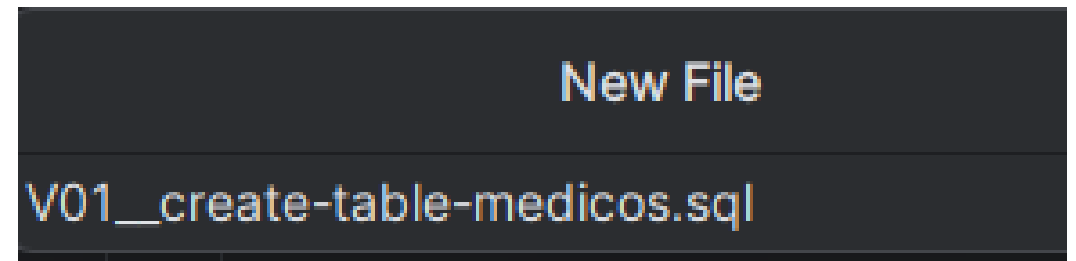
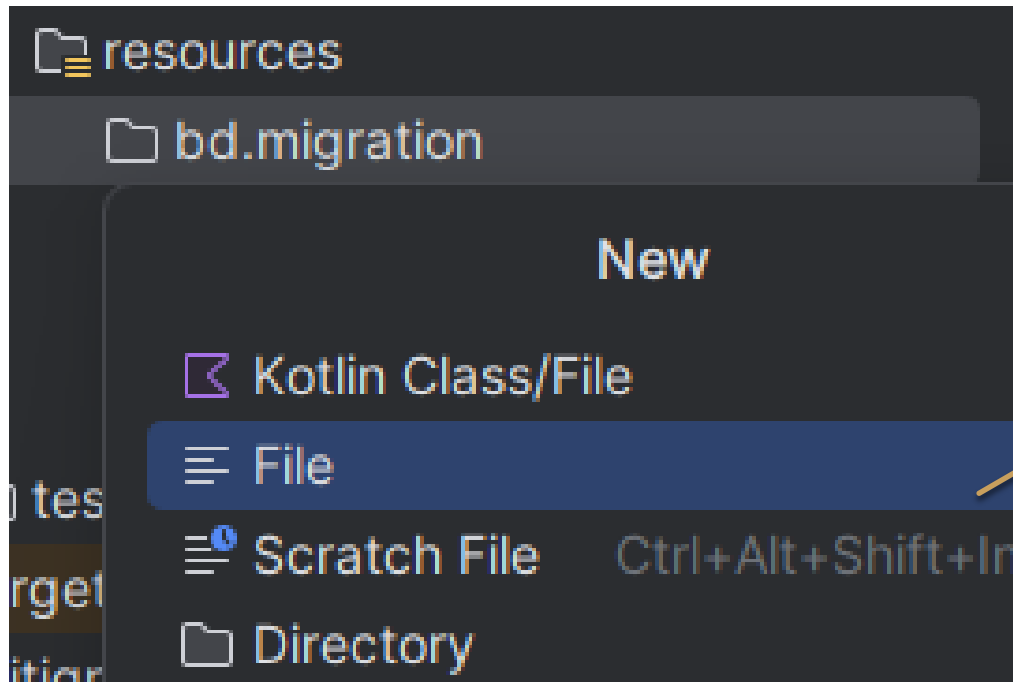
Spring

- No IntelliJ o nome fica : 
- A barra significa que será criada uma subpasta, "migration", dentro da pasta "db"
- Dentro da pasta, criaremos um arquivo SQL que servirá como nossa primeira migration, responsável por criar a tabela de médicos.
- Antes disso, é preciso interromper o projeto antes de usar migrations.

- Obs: Sempre interrompa o projeto ao usar migrations.

Spring

- Vamos clicar na pasta "db > migration" e, com o atalho "Alt + Insert ou botão direito", selecionaremos a opção "File".



- O nome da migration será "V01__create-table-medicos.sql".
- Obs: Esse tipo de arquivo sempre começará com "V", seguido pelo número que representa a ordem de criação dos arquivos e, depois de dois underlines, um nome descritivo.
- Vamos abrir o arquivo e digitar o comando SQL para criar a tabela:


```
create table medicos(  
id bigint not null auto_increment,  
nome varchar(100) not null,  
email varchar(100) not null unique,  
crm varchar(6) not null unique,  
especialidade varchar(100) not null,  
logradouro varchar(100) not null,  
bairro varchar(100) not null,
```

```
cep varchar(9) not null,  
complemento varchar(100),  
numero varchar(20),  
uf char(2) not null,  
cidade varchar(100) not null,  
primary key(id)  
);
```

Spring

```
create table medicos(  
    id bigint not null auto_increment,  
    nome varchar(100) not null,  
    email varchar(100) not null unique,  
    crm varchar(6) not null unique,  
    especialidade varchar(100) not null,  
    logradouro varchar(100) not null,  
    bairro varchar(100) not null,  
    cep varchar(9) not null,  
    complemento varchar(100),  
    numero varchar(20),  
    uf char(2) not null,  
    cidade varchar(100) not null,  
    primary key(id)  
);
```

Spring

- Vamos iniciar o projeto e ver o log do Spring que a migration foi identificada e executada.

```
Database: jdbc:mysql://localhost:3306/vollmed_api (MySQL 8.0)
```

```
Schema history table `vollmed_api`.`flyway_schema_history` does not exist yet
```

```
Successfully validated 1 migration (execution time 00:00.078s)
```

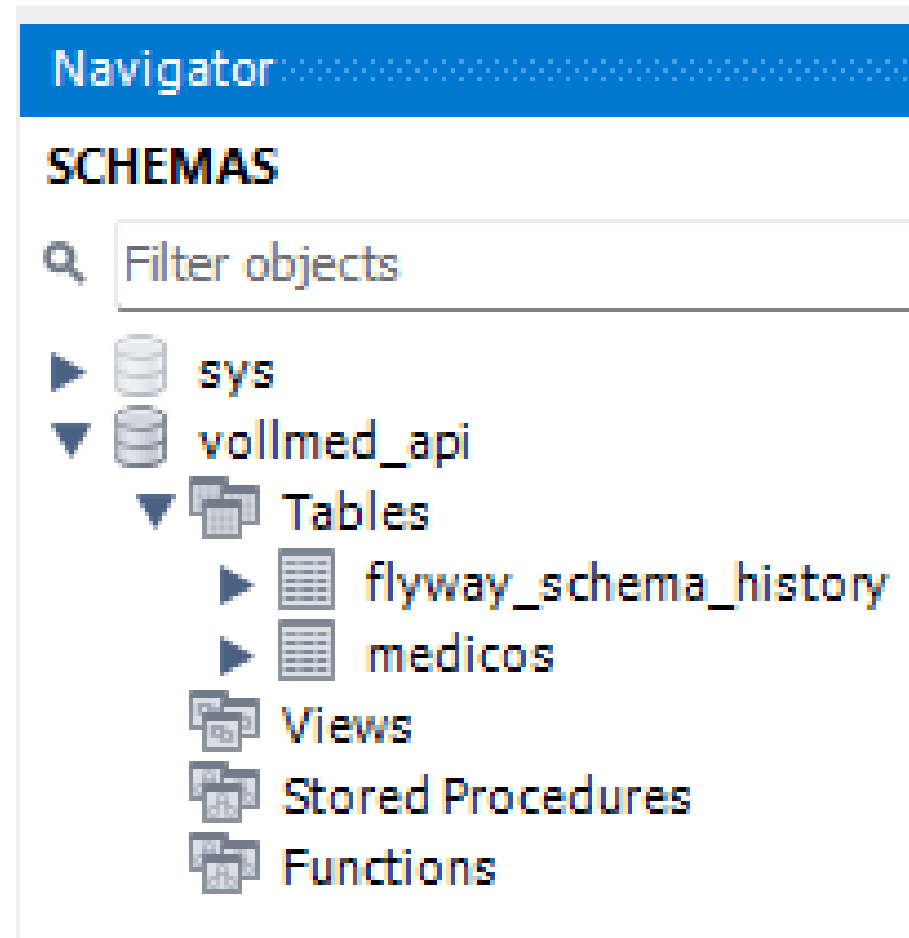
```
Creating Schema History table `vollmed_api`.`flyway_schema_history` ...
```

```
Current version of schema `vollmed_api`: << Empty Schema >>
```

```
Migrating schema `vollmed_api` to version "01 - create-table-medicos"
```

```
Successfully applied 1 migration to schema `vollmed_api`, now at version v01 (execution
```

- Veja como está no MySql.



- Agora podemos testar se conseguimos salvar um objeto Medico no banco de dados.
- Vamos abrir o arquivo MedicoController.java. Acima do método, abaixo da anotação `@PostMapping`, vamos inserir a anotação `@Transactional`.

Spring

```
public class MedicoController {  
    1 usage  
    @Autowired  
    private MedicoRepository repository;  
    no usages  
    @PostMapping  
    @Transactional  
    public void cadastrar(@RequestBody DadosCadastroMedico dados) {  
        repository.save(new Medico(dados));  
    }  
}
```

Spring

- Como esse é um **método de escrita**, que **consiste em um insert no banco de dados**, **precisamos ter uma transação ativa com ele**.
- De volta ao Insomnia, vamos tentar rodar a aplicação.
- Disparando a requisição, receberemos o código 200, que significa OK, e não veremos nenhuma mensagem de erro na IDE.
- Faça uma consulta na tabela medico do Banco vollmed_api

Transação com JPA:

- `begin()`: Inicia uma transação;
- `commit()`: Finaliza uma transação;
- `rollback()`: Cancela uma transação.

- Ao chamar o `begin()`, uma transação é iniciada e tudo que for feito daqui em diante será considerado transacional.
- Quando o `commit()` é chamada então as informações são persistidas.
- Se algum erro ocorrer dentro do `businesslogic()`, o nosso fluxo vai direto ao bloco `catch()` onde um `rollback()` é chamado garantindo que nada será persistido.

Transação com JPA:

```
1  UserTransaction utx = entityManager.getTransaction();
2
3  try {
4      utx.begin();
5
6      businessLogic();
7
8      utx.commit();
9  } catch (Exception ex) {
10     utx.rollback();
11     throw ex;
12 }
```

Listagem 1

Spring Framework: @Transactional

- Ao chamar o `begin()`, uma transação é iniciada e tudo que for feito daqui em diante será considerado transacional.
- Quando o `commit()` é chamada então as informações são persistidas.
- Se algum erro ocorrer dentro do `businesslogic()`, o nosso fluxo vai direto ao bloco `catch()` onde um `rollback()` é chamado garantindo que nada será persistido.

Spring Framework: @Transactional

```
1 | @Transactional
2 |     public void businessLogic() {
3 |         //lógica necessária aqui
4 |     }
```

Listagem 2.

Na **Listagem 1** precisamos capturar o entityManager, iniciar uma transação, finalizar a transação, tratar os erros dentro do bloco try-catch e chamar o rollback() caso necessário. Na **Listagem 2** só anotamos o método com a anotação @Transactional e pronto, tudo que fizemos na **Listagem 1** já está pronto na **Listagem 2**.

Spring



DTO - Data Transfer Object

