## 01 Preparando o ambiente: MySQL

No curso será utilizado o **MySQL** como sistema gerenciador de banco de dados, sendo recomendado que você também o utilize.

Caso você não tenha o MySQL instalado em seu computador, pode seguir o tutorial de instalação disponível nesse artigo: MySQL: da instalação até a configuração

## 02 Adicionando dependências

### Transcrição

Agora que aprendemos a enviar dados para a nossa *API*, podemos continuar o desenvolvimento da funcionalidade do cadastro de médicos.

Podemos seguir com os próximos passos, que são fazer a validação das informações e a persistência no banco de dados.

Para fazer isso, precisaremos adicionar novos módulos do *Spring Boot*. Para fazer isso, vamos abrir o arquivo "target > pom.xml". Vamos rolar a página até a linha 28, onde encontramos as dependências.

Vamos adicionar o driver do banco de dados, o módulo do *Spring Data*, o módulo de validação e alguns outros módulos. Poderíamos fazer todo esse processo manualmente, mas isso não é recomendado.

Para fazer isso, inicialmente, abriremos o site start.spring.io em outra aba. Lá, clicaremos no botão "Add Dependencies", para adicionar dependências. Depois que a *pop-up* abrir, vamos adicionar as

dependências que precisaremos para essa parte de dependência e validação.

Vamos buscar por "Validation", segurar o botão "Ctrl" e clicar na alternativa que aparecerá. Na sequência, buscaremos por "MySQL Driver", que utilizaremos como banco de dados. Segurando "Ctrl" e clicando, também o adicionaremos.

Repetiremos o processo com "Spring Data JPA" e "Flyway Migrations". Agora podemos fechar a *pop-up*. Encontraremos as quatro dependências que selecionamos.

Vamos clicar no botão "Explore", que nos apresentará como o projeto funcionará com as especificações que estão apresentadas na tela. Seremos redirecionados para o arquivo "pom.xml" dessas novas configurações.

Basta procurar as dependências, dentro do código, selecioná-las, copiá-las e levá-la para nosso arquivo "pom.xml", na *IDE*, logo abaixo da última dependência:

```
    <dependencies>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-web</artifactId>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-devtools</artifactId>
            <scope>runtime</scope>
```

```
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <optional>true</optional>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-test</artifactId>
            <scope>test</scope>
        </dependency>

        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>org.springframework.boot</groupId>
            <artifactId>spring-boot-starter-validation</artifactId>
        </dependency>
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-core</artifactId>
        </dependency>
        <dependency>
            <groupId>org.flywaydb</groupId>
            <artifactId>flyway-mysql</artifactId>
        </dependency>
```

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

COPIAR CÓDIGO

Obs: Lembre de acessar a aba do *Maven*, à direita da *IDE*, e clicar em "Reload All Maven Projects", para recarregar as dependências do projeto.

Sempre que adicionarmos novas dependências, é interessante pausar o servidor, clicando no ícone do quadrado vermelho, que aparece no canto superior direito.

Depois que o *Maven* baixar todas as dependências, podemos seguir. Já temos as dependências necessárias para lidar com dependência e validação. Vamos voltar a executar o projeto, clicando sobre o ícone de *play* verde, também no canto superior direito.

Se executarmos a aba *Run*, perceberemos que um erro foi identificado. Isso faz com que a inicialização do projeto seja parada. No *log*, descobrimos que isso aconteceu porque algumas configurações do *DataSource* não foram feita.

Vamos fazer isso no arquivo "src > resources > application.properties". Precisamos adicionar três propriedades a ele, a *URL* de conexão com o banco de dados, o *login* e a senha do banco de dados.

Como esse arquivo funciona seguindo a lógica chave-valor, vamos passar algumas chaves para configurar *URL, username e password* no banco de dados.

Para passar a *URL* de conexão com o banco de dados, vamos inserir o código `spring.datasource.url=jdbc:mysql://localhost:3306/vollmed_api`. Passaremos também `spring.datasource.username=root`, para definir o *username*, e `spring.datasource.password=root`, para definir a senha:

```
spring.datasource.url=jdbc:mysql://localhost:3306/vollmed_api

spring.datasource.username=root

spring.datasource.password=root

COPIAR CÓDIGO
```

Agora o *Spring Data* conseguirá se conectar ao banco de dados da aplicação corretamente. No logo, encontraremos a informação de que o database "vollmed_api" não foi encontrado.

O que precisamos fazer é acessar o *MySQL* e criar o database, porque esse não é um processo automático. Vamos abrir o terminal e logar no *MySQL* com o comando `mysql -u root -p`. Ele solicitará a senha. Depois de inseri-la, estamos logados.

Agora criaremos o database, com o comando `create database vollmed_api;`. Vamos sair com *MySQL* com `mysql> exit`.

Depois disso, conseguiremos inicializar o projeto corretamente. Adicionamos as dependências e validação e persistência e aprendemos a fazer algumas configurações.

## 03 Mudança do MySQL no Maven

Houve uma mudança no Maven em relação à dependência do MySQL, na qual o **group-id** e o **artifact-id** foram alterados.
A partir de agora, você deve adicionar a dependência do driver MySQL no arquivo `pom.xml` da seguinte maneira:

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

Além disso, pode acontecer do Spring Boot não encontrar automaticamente o driver do MySQL no projeto, sendo recomendado que você adicione mais uma propriedade no arquivo **application.properties**:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver COPIAR CÓDIGO
```

## 04 Para saber mais: arquivo properties ou yaml?

As configurações de uma aplicação Spring Boot são feitas em arquivos externos, sendo que podemos usar arquivo de propriedades ou arquivo YAML. Neste “Para saber mais”, vamos abordar as principais diferenças entre eles.

## Arquivo de propriedades

Por padrão, o Spring Boot acessa as configurações definidas no arquivo `application.properties`, que usa um formato de `chave=valor`:

```
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver

spring.datasource.url=jdbc:mysql://localhost:3306/clinica

spring.datasource.username=root

spring.datasource.password=rootCOPIAR CÓDIGO
```

Cada linha é uma configuração única, então é preciso expressar dados hierárquicos usando os mesmos prefixos para nossas chaves, ou seja, precisamos repetir prefixos, neste caso, `spring` e `datasource`.

## YAML Configuration

YAML é um outro formato bastante utilizado para definir dados de configuração hierárquica, como é feito no Spring Boot.

Pegando o mesmo exemplo do nosso arquivo `application.properties`, podemos convertê-lo para YAML alterando seu nome para `application.yml` e modificando seu conteúdo para:

```
spring:

    datasource:

        driver-class-name: com.mysql.cj.jdbc.Driver

        url: jdbc:mysql://localhost:3306/clinica

        username: root

        password: root

COPIAR CÓDIGO
```

Com YAML, a configuração se tornou mais legível, pois não contém prefixos repetidos. Além de legibilidade e redução de repetição, o uso de YAML facilita o armazenamento de variáveis de configuração de ambiente, conforme recomenda o *12 Factor App*, uma metodologia bastante conhecida e utilizada que define 12 boas práticas para criar uma aplicação moderna, escalável e de manutenção simples.

## Mas afinal, qual formato usar?

Apesar dos benefícios que os arquivos YAML nos trazem em comparação ao arquivo properties, a decisão de escolher um ou outro é de gosto pessoal. Além disso, não é recomendável ter ao mesmo tempo os dois tipos de arquivo em um mesmo projeto, pois isso pode levar a problemas inesperados na aplicação.

Caso opte por utilizar YAML, fique atento, pois escrevê-lo no início pode ser um pouco trabalhoso devido às suas regras de indentação.

# 05 Entidades JPA

### Transcrição

Agora vamos continuar a implementar a funcionalidade de cadastros de médicos.
Vamos voltar para "MedicoController.java". Nele, estamos recebendo um *DTO*, representado pelo *record* `DadosCadastroMedico` e dando um `System.out`.

O que precisamos fazer, porém, é pegar o objeto e salvá-lo no banco de dados. O *Spring Data JPA* utiliza o *JPA* como ferramenta de mapeamento de objeto relacional. Por isso, criaremos uma entidade *JPA* para representar um tabela no banco de dados.
Como esse não é o foco do treinamento, vamos apenas utilizar a *JPA*.
Acessaremos "src > java > medico". Nessa pasta, criaremos a classe "Medico".

Faremos isso usando o atalho "Alt + Insert", selecionar a opção "Class" e digita seu nome, "Medico". Agora adicionaremos, à classe, os atributos que representam o médico.

Vamos declarar os atributos `private Long id;`, `private String nome;`, `private String email;`, `private String crm;`, `private Especialidade especialidade;` e `private Endereco endereco;`:

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


public class Medico {


        private Long id;

        private String nome;

        private String email;

        private String crm;

        private Especialidade especialidade;

        private Endereco endereco;


}COPIAR CÓDIGO
```

Precisaremos criar a classe `Endereco`, com a ajuda de um "Alt + Enter", substituindo o pacote por "med.voll.api.endereco", para criá-la.

Dentro de "Endereco.java", na classe `Endereco`, passaremos os campos `private String logradouro;`, `private String bairro;`, `private String cep;`, `private String numero;`, `private String complemento`, `private String cidade;` e `private String uf;`:

```
package med.voll.api.endereco;


public class Endereco {


        private String logradouro;

        private String bairro;

        private String cep;

        private String numero;

        private String complemento;

        private String cidade;

        private String uf;


}COPIAR CÓDIGO
```

Agora que tudo está mapeado corretamente, vamos voltar à entidade `Medico`, que até o momento é uma classe Java, com nada da *JPA*.

Vamos adicionar as anotações da JPA para transformar isso em uma entidade:

```
@Table(name = "medicos")

@Entity(name = "Medico")

public class Medico {


    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)

    private Long id;
```

```
    private String nome;

    private String email;

    private String crm;


    @Enumerated(EnumType.STRING)

    private Especialidade especialidade;


    @Embedded

    private Endereco endereco;COPIAR CÓDIGO
```

Vamos usar *Embeddable Attribute* da *JPA* para que `Endereco` fique em uma classe separada, mas faça parte da mesma tabela de `Medicos` junto ao banco de dados.

Para que isso funcione, vamos acessar a classe `Endereco` e adicionar, no topo do código, a anotação `@Embeddable` logo acima da classe.

Vamos importar a biblioteca *Lombok*, para gerar os códigos *Java* que faltam automaticamente.

Adicionaremos `@Getter`, `@NoArgsConstructor`, `@AllArgsConstructor`, `@EqualsAndHashCode(of = "id")`:

```
@Table(name = "medicos")

@Entity(name = "Medico")

@Getter

@NoArgsConstructor

@AllArgsConstructor

@EqualsAndHashCode(of = "id")

public class Medico {COPIAR CÓDIGO
```

Faremos a mesma coisa na classe `Endereco`. Junto de `@Embeddable`, adicionaremos as mesmas informações, exceto `@EqualsAndHashCode`:

```
@Embeddable

@Getter
```

```
@NoArgsConstructor

@AllArgsConstructor

public class Endereco { COPIAR CÓDIGO
```

Agora realizamos o mapeamento da *JPA*.

## 06 Interfaces Repository

### Transcrição

Para fazer a persistência, 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.

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:

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


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

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

{

} COPIAR CÓDIGO
```

Os métodos da interface `JpaRepository` agora estão presentes na interface que acabamos de criar.

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

Criaremos o atributo `private MedicoRepository repository;`. Precisamos avisar ao *Spring* que esse novo atributo pelo ser instanciado. Faremos a injeção de dependências inserindo a anotação `@Autowired` acima do atributo.

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

No *controller*, porém, não recebemos o objeto `Medico`. Precisaremos fazer a conversão para transformá-lo em uma entidade *JPA*. Para isso, usaremos o construtor.

No método `save`, vamos criar um construtor para receber `DadosCadastroMedico`. Primeiro, passaremos `repository.save(new Medico(dados));`. 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();` e `this.endereco = new Endereco(dados.endereco());`:

```
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());

    } COPIAR CÓDIGO
```

Criaremos o construtor que recebe o objetos `dados.endereco` na classe `Endereco`.

Para resolver isso, vamos criar o construtor usando "Alt + Enter" mais uma vez. No construtor, receberemos `dados` e vamos inserir os atributos `this.logradouro = dados.logradouro();`, `this.bairro = dados.bairro();`, `this.cep = dados.cep();`, `this.uf = dados.uf();`, `this.cidade = dados.cidade();`, `this.numero = dados.numero();` e `this.complementos = dados.complemento();`.

```
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();
    } COPIAR CÓDIGO
```

Vamos tentar salvar no banco de dados. No *insomnia*, tentaremos disparar 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`.

No próximo vídeo, resolveremos o problema da tabela e aprenderemos a alterá-la conforme precisarmos.

## 07 Para saber mais: e as classes DAO?

Em alguns projetos em Java, dependendo da tecnologia escolhida, é comum encontrarmos classes que seguem o padrão **DAO**, utilizado para isolar o acesso aos dados. Entretanto, neste curso utilizaremos um outro padrão, conhecido como **Repository**.

Mas aí podem surgir algumas dúvidas: qual a diferença entre as duas abordagens e o porquê dessa escolha?

### Padrão DAO

O padrão de projeto DAO, conhecido também por **Data Access Object**, é utilizado para persistência de dados, onde seu principal objetivo é separar regras de negócio de regras de acesso a banco de dados. Nas classes que seguem esse padrão, isolamos todos os códigos que lidam

com conexões, comandos SQLs e funções diretas ao banco de dados, para que assim tais códigos não se espalhem por outros pontos da aplicação, algo que dificultaria a manutenção do código e também a troca das tecnologias e do mecanismo de persistência.

### Implementação

Vamos supor que temos uma tabela de produtos em nosso banco de dados. A implementação do padrão DAO seria o seguinte:

Primeiro, seria necessário criar uma classe básica de domínio `Produto`:

```
public class Produto {

   private Long id;

   private String nome;

   private BigDecimal preco;

   private String descricao;


   // construtores, getters e setters

}
```

COPIAR CÓDIGO

Em seguida, precisaríamos criar a classe `ProdutoDao`, que fornece operações de persistência para a classe de domínio `Produto`:

```
public class ProdutoDao {


    private final EntityManager entityManager;


    public ProdutoDao(EntityManager entityManager) {
        this.entityManager = entityManager;
```

```
    }

    public void create(Produto produto) {
        entityManager.persist(produto);
    }

    public Produto read(Long id) {
        return entityManager.find(Produto.class, id);
    }

    public void update(Produto produto) {
        entityManger.merge(produto);
    }

    public void remove(Produto produto) {
        entityManger.remove(produto);
   }

}
```

COPIAR CÓDIGO

No exemplo anterior foi utilizado a JPA como tecnologia de persistência dos dados da aplicação.

## Padrão Repository

De acordo com o famoso livro *Domain-Driven Design*, de Eric Evans:

> O repositório é um mecanismo para encapsular armazenamento, recuperação e comportamento de pesquisa, que emula uma coleção de objetos.

Simplificando, um repositório também lida com dados e oculta consultas semelhantes ao DAO. No entanto, ele fica em um nível mais alto, mais próximo da lógica de negócios de uma aplicação. Um repositório está vinculado à regra de negócio da aplicação e está associado ao agregado dos seus objetos de negócio, retornando-os quando preciso.

Só que devemos ficar atentos, pois assim como no padrão DAO, regras de negócio que estão envolvidas com processamento de informações não devem estar presentes nos repositórios. Os repositórios não devem ter a responsabilidade de tomar decisões, aplicar algoritmos de transformação de dados ou prover serviços diretamente a outras camadas ou módulos da aplicação. Mapear entidades de domínio e prover as funcionalidades da aplicação são responsabilidades muito distintas.

Um repositório fica entre as regras de negócio e a camada de persistência:

1. Ele provê uma interface para as regras de negócio onde os objetos são acessados como em uma coleção;
2. Ele usa a camada de persistência para gravar e recuperar os dados necessários para persistir e recuperar os objetos de negócio.

## Por que o padrão repository ao invés do DAO utilizando Spring?

O padrão de repositório incentiva um design orientado a domínio, fornecendo uma compreensão mais fácil do domínio e da estrutura de

dados. Além disso, utilizando o repository do Spring não temos que nos preocupar em utilizar diretamente a API da JPA, bastando apenas criar os métodos que o Spring cria a implementação em tempo de execução, deixando o código muito mais simples, menor e legível.

## 08 Migrations com Flyway

### Transcrição

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. Já registramos o *Flyway*, uma dessas ferramentas suportadas pelo *Spring Boot*.

Vamos acessar o "pom.xml" do projeto. Lá veremos que criamos dependências de duas versões do *Flyway*, o `flyway-core` e o `flyway-mysql`. Vamos usar isso para fazer o controle e modificação de tabelas do banco de dados.

Para cada mudança que quisermos executar no banco de dados, precisamos criar um arquivo `.sql` no projeto e, nele, escrever o trecho do comando *SQL* que será executado no banco de dados.

Precisamos salvá-los em um diretório específico. Criaremos essa nova pasta em "main > resources". Com "Alt + Insert", vamos escolher a opção "Directory" e digita o nome da pasta: "db/migration".

Obs: 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*.

Faremos isso clicando no ícone do quadrado vermelho no canto superior direito.

Vamos clicar na pasta "db > migration" e, com o atalho "Alt + Insert", selecionaremos a opção "File". O nome da *migration* será "V1__create-table-medicos.sql".

Obs: Esse tipo de arquivo sempre começará com "V", seguido pelo número que repersenta a ordem de criação dos arquivos e, depois de dois *underlines*, um nome descritivo.

Vamos abrir o nome 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)


); COPIAR CÓDIGO
```

Agora vamos inicializar o projeto novamente e ver no *log* do *Spring* que a *migration* foi identificada e executada.

Se logarmos no *MySQL*, acessando o terminal e executando o comando `mysql -u -root -p`, e inserirmos a senha, poderemos ver a tabela após executar, em ordem, os comandos `use vollmed_api` e `show tables;`.

Com o comando `desc medicos`, veremos a descrição da tabela. Ela foi criada conforme descrevemos.

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`.

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*.

Vamos abrir o terminal para conferir se a nova informação foi salva no banco de dados. Para isso, executaremos `select * from medicos;`. Veremos que o registro foi salvo corretamente, conforme enviamos na requisição.

No próximo vídeo, faremos validações dos campos obrigatórios.

## 09 Validação com Bean Validation

### Transcrição

Como usamos as validações que já adicionamos ao nosso artigo "pom.xml"? No caso, `spring-boot-starter-validation`.

Para isso, vamos acessar nosso *controller* e observar o método `cadastrar`. Ele recebe o *DTO* `DadosCadastroMédico` como parâmetro. Se o acessarmos, clicando duas vezes sobre ele com o botão esquerdo, veremos os campos que estão chegando na requisição.

São eles: `String nome`, `String email`, `String crm`, `Especialidade especialidade`, e `DadosEndereco endereco`.

É nele que usaremos o *Bean Validation*, a partir de anotações. Ele vai verificar, no caso, se as informações que chegam estão de acordo com as anotações.
Vamos adicionar uma anotação a cada um dos atributos, começando pelo atributo `nome`, que é obrigatório e não pode ser nulo e que, também, não pode ser vazio: precisa de um texto. Para informar isso ao *Bean Validation*, passaremos a anotação `@NotBlank`.

Também passaremos a anotação `@NotBlank` acima de `String email`. Para dar a formatação de e-mail, passaremos também a anotação `@Email`.

Acima de `String crm`, vamos passar `@NotBlank` e `@Pattern`, porque ele é um número de 4 a 6 dígitos. Dentro da segunda anotação, para esclarecer a quantidade de dígitos passaremos a expressão regular `(regexp = "\\d{4,6}")`.

Como `Especialidade especialidade` é um campo obrigatório, vamos adicionar a anotação `@NotNull`. Por fim, acima de `DadosEndereco endereco`, passaremos a anotação `@NotNull`, porque ele também não pode ser nulo, e `@Valid`, para validar o *DTO*:

```
public record DadosCadastroMedico(
        @NotBlank
        String nome,
        @NotBlank
        @Email
        String email,
```

```
        @NotBlank
        @Pattern(regexp = "\\d{4,6}")
        String crm,
        @NotNull
        Especialidade especialidade,

        @NotNull @Valid DadosEndereco endereco) {
} COPIAR CÓDIGO
```

Agora, vamos dar dois clique com o botão esquerdo sobre `DadosEndereco` e repetir o processo, adicionando anotações do *Bean Validation*.
`logradouro` não pode ser branco ou vazio, então adicionaremos `@NotBlank`. Faremos o mesmo até `String uf`. `String complemento` e `String numero`, por suas vezes, são opcionais. Por isso, não adicionaremos anotações do *Bean Validation*.

Vamos passar `@Pattern` e a expressão regular `(regexp = "\\d{8}")` acima de `String cep`:

```
public record DadosEndereco(
        @NotBlank
        String logradouro,
        @NotBlank
        String bairro,
        @NotBlank
        @Pattern(regexp = "\\d{8}")
        String cep,
        @NotBlank
```

```
        String cidade,
        @NotBlank
        String uf,
        String complemento,
        String numero) {
} COPIAR CÓDIGO
```

Agora vamos acessar "MedicoController.java". Lá, adicionaremos `@Valid`, para solicitar queo Spring se integre ao *Bean Validation* e execute as validações. Agora só precisamos salvar. Vamos abrir o *Insomnia* para executar um teste. Vamos disparar uma requisição, mas deixando o campo `"nome"` em branco. Como resultado, receberemos o erro "400 Bad Request". Isso acontece porque informamos, via *Bean Validation*, que esse campo não pode ser nulo.

Em outras palavras, a validação foi pega corretamente. Se adicionarmos um nome a esse campo, mas removermos a informação do campo `"crm"`, teremos o erro novamente, pelo mesmo motivo.

Tudo está funcionando. No próximo vídeo, ajustaremos uma *migration* do nosso projeto.

## 10 Para saber mais: anotações do Bean Validation

Como explicado no vídeo anterior, o Bean Validation é composto por diversas anotações que devem ser adicionadas nos atributos em que desejamos realizar as validações. Vimos algumas dessas anotações, como a `@NotBlank`, que indica que um atributo do tipo `String` não pode ser nulo e nem vazio.

Entretanto, existem dezenas de outras anotações que podemos utilizar em nosso projeto, para os mais diversos tipos de atributos. Você pode conferir uma lista com as principais anotações do Bean Validation na documentação oficial da especificação.

## 11 Validando dados

Você está trabalhando em um projeto que utiliza o Bean Validation, entretanto, as validações não estão sendo realizadas e as informações estão chegando ao banco de dados de maneira inválida.

Analise os seguintes trechos de códigos desse projeto:

```
@PostMapping
public void cadastrar(@RequestBody DadosCadastroProduto dados) {
    repository.save(new Produto(dados));
}COPIAR CÓDIGO
public record DadosCadastroProduto(
        @NotBlank String nome,
        @NotBlank String descricao,
        @NotNull @DecimalMin("1.00") BigDecimal preco
) {
}
COPIAR CÓDIGO
```

Escolha a alternativa CORRETA que identifica o problema mencionado:

- Faltou anotar o parâmetro `dados`, recebido no método `cadastrar` do Controller, com **@Valid**.

  Sem essa anotação o Spring não vai disparar o processo de validação do Bean Validation.

- Alternativa correta

  O método `cadastrar` está declarado com retorno `void`.

- Alternativa correta

  Faltou anotar o método `cadastrar` com `@Transactional`.

Parabéns, você acertou!

# 12 Nova migration

## Transcrição

No último vídeo, esquecemos de adicionar o campo "Telefone" à entidade médico. Vamos aprender a adicionar uma coluna ao banco de dados quando a *migration* já existe.

Vamos acessar "MedicoController.java > DadosCadastroMedico". Precisaremos criar um atributo para receber esse novo campo. Vamos adicioná-lo logo após o campo `String email`. Como ele será obrigatório, passaremos a anotação `@NotBlank`:

```
public record DadosCadastroMedico(
        @NotBlank
        String nome,
        @NotBlank
        @Email
        String email,

        @NotBlank
        String telefone,
        @NotBlank
        @Pattern(regexp = "\\d{4,6}")
        String crm,
        @NotNull
        Especialidade especialidade,

        @NotNull @Valid DadosEndereco endereco) {
```

```
} COPIAR CÓDIGO
```

Abaixo disso, passaremos a `String telefone`. Vamos precisar fazer alterações também na entidade *JPA* "Medico.java". Lá, adicionaremos o atributo abaixo de `private String email;`. Será o atributo `private String telefone;`.

Nesse mesmo arquivo, vamos informar ao construtor `public Medico`, que recebe `DadosCadastroMedico`, desse novo atributo. Faremos isso passando `this.telefone = dados.telefone()`:

```
public class Medico {

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

    private String telefone;

    private String crm;

    @Enumerated(EnumType.STRING)
    private Especialidade especialidade;

    @Embedded
    private Endereco endereco;

    public Medico(DadosCadastroMedico dados) {
        this.nome = dados.nome();
```

```
        this.email = dados.email();
        this.telefone = dados.telefone();
        this.crm = dados.crm();
        this.especialidade = dados.especialidade();
        this.endereco = new Endereco(dados.endereco());
    } COPIAR CÓDIGO
```

Já temos uma *migration* criada, "V1_create-table-medicos.sql". Porém, não podemos alterá-la, porque *migrations* executadas no banco de dados não podem mais ser alteradas.

Portanto, para passar o atributo `telefone` para o banco de dados, precisaremos criar uma nova *migration*. Mas, antes disso, precisamos parar o projeto, clicando no ícone do quadrado vermelho, no canto superior direito.

Para isso, criaremos um novo arquivo, chamado "V2__alter-table-medicos-add-column-telefone.sql", dentro da pasta "db.migration".

Nele vamos digitar o código *SQL* abaixo, para alterar o banco de dados:

```
alter table medicos add telefone varchar(20) not null; COPIAR CÓDIGO
```

Vamos verificar no *log* se o projeto executará essa nova *migration*. Para isso, inicializaremos o projeto e maximizaremos o terminal. Veremos que a tabela foi atualizada corretamente.

Vamos abrir o *Insomnia* para cadastrar nosso novo registro no banco de dados. Se tentarmos cadastrar um novo médico sem informar o

telefone, teremos como resultado o erro *400 Bad Request*, o que significa que tivemos sucesso.

## 13 Para saber mais: Erro na migration

Conforme orientado ao longo dessa aula é importante sempre **parar** o projeto ao criar os arquivos de **migrations**, para evitar que o Flyway os execute antes da hora, com o código ainda incompleto, causando com isso problemas.

Entretanto, eventualmente pode acontecer de esquecermos de parar o projeto e algum erro acontecer ao tentar inicializar a aplicação. Nesse caso será exibido o seguinte erro ao tentar inicializar a aplicação:

```
Exception encountered during context initialization - cancelling
refresh attempt:
org.springframework.beans.factory.BeanCreationException: Error
creating bean with name 'flywayInitializer' defined in class path
resource
[org/springframework/boot/autoconfigure/flyway/FlywayAutoConfiguration
$FlywayConfiguration.class]: Validate failed: Migrations have failed
validation COPIAR CÓDIGO
```

Perceba na mensagem de erro que é indicado que alguma migration falhou, impedindo assim que o projeto seja inicializado corretamente. Esse erro também pode acontecer se o código da migration estiver inválido, contendo algum trecho de SQL digitado de maneira incorreta.

Para resolver esse problema será necessário acessar o banco de dados da aplicação e executar o seguinte comando sql:

```
delete from flyway_schema_history where success = 0; COPIAR CÓDIGO
```

O comando anterior serve para apagar na tabela do Flyway todas as migrations cuja execução falhou. Após isso, basta corrigir o código da migration e executar novamente o projeto.

Obs: Pode acontecer de alguma migration ter criado uma tabela e/ou colunas e com isso o problema vai persistir, pois o flyway não vai apagar as tabelas/colunas criadas em migrations que falharam. Nesse caso você pode apagar o banco de dados e criá-lo novamente:

```
drop database vollmed_api;

create database vollmed_api; COPIAR CÓDIGO
```

## 14 Faça como eu fiz: cadastro de pacientes

Agora é com você! Faça o mesmo procedimento que eu fiz na aula, porém, para a funcionalidade de **cadastro de pacientes**.

# Opinião do instrutor

- 
- 

Você precisará criar a entidade `Paciente`: ```java @Getter @EqualsAndHashCode(of = "id") @NoArgsConstructor @AllArgsConstructor @Entity(name = "Paciente") @Table(name = "pacientes") public class Paciente { @Id @GeneratedValue(strategy = GenerationType.IDENTITY) private Long id; private String nome; private String email; private String cpf; private String telefone;

@Embedded private Endereco endereco; public Paciente(DadosCadastroPaciente dados) { this.nome = dados.nome(); this.email = dados.email(); this.telefone = dados.telefone(); this.cpf = dados.cpf(); this.endereco = new Endereco(dados.endereco()); } } ``` Na sequência, precisará criar um repository: ```java public interface PacienteRepository extends JpaRepository { } ``` Depois, precisará alterar as classes Controller e DTO: ```java @RestController @RequestMapping("pacientes") public class PacienteController { @Autowired private PacienteRepository repository; @PostMapping @Transactional public void cadastrar(@RequestBody @Valid DadosCadastroPaciente dados) { repository.save(new Paciente(dados)); } } ``` ```java public record DadosCadastroPaciente( @NotBlank String nome, @NotBlank @Email String email, @NotBlank String telefone, @NotBlank @Pattern(regexp = "\\d{3}\\.?\\d{3}\\.?\\d{3}\\-?\\d{2}") String cpf, @NotNull @Valid DadosEndereco endereco ) { } ``` E, por fim, vai precisar criar uma *migration* (**Atenção!** Lembre-se de parar o projeto antes de criar a migration!): ``` create table pacientes( id bigint not null auto_increment, nome varchar(100) not null, email varchar(100) not null unique, cpf varchar(14) not null unique, telefone varchar(20) 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) ); ```

## 15 O que aprendemos?

Nessa aula, você aprendeu como:

- Adicionar novas dependências no projeto;
- Mapear uma entidade JPA e criar uma interface Repository para ela;
- Utilizar o Flyway como ferramenta de Migrations do projeto;
- Realizar validações com Bean Validation utilizando algumas de suas anotações, como a `@NotBlank`.