

DESAFIO

Criar uma API REST para controle de vacinas na população brasileira por meio do cadastro de usuários e da aplicação das vacinas, utilizando Java e Spring como tecnologias fundamentais.

Nome: Breno Nogueira Botelho Noccioli

E-mail: brenonoccioli@gmail.com

 www.linkedin.com/in/brenonoccioli/

 github.com/BrenoNoccioli

Olá, tudo bem com vocês?

Meu nome é **Breno Nocchioli** e sou desenvolvedor Full Stack Jr.

O desafio de hoje foi proposto pela **ZUP**, através do programa **Orange Talents**, e consiste em criar uma **API REST** para controle da aplicação de vacinas na população brasileira (achei bem propício! E você?! 😊).

E aí, topa vir nessa comigo?

Então, vamos começar! 😎

API REST

Se você nunca construiu uma **API REST**, deve estar se perguntando agora: "**O que é isso?!**"

API é um acrônimo em inglês para "*Interface de Programação de Aplicações*". Então, de modo geral, uma **API** é uma estrutura para fornecer dados a uma aplicação.

"Mas e o REST?!"

REST é uma arquitetura de envio de dados muito utilizada pelo protocolo HTTP, que é o protocolo que utilizamos na internet. Construir uma **API REST** será fundamental para fazermos os cadastros propostos em nosso desafio!

Nas próximas páginas, optei por me referir apenas como **API, mas tenha em mente que estamos falando de uma **API REST**, ok? 😊*

LINGUAGENS E TECNOLOGIAS

Agora que já sabemos um pouco mais sobre **API REST**, vamos às linguagens e tecnologias que utilizaremos em nossa aplicação:

- Linguagem de programação **Java**;
- Banco de dados **MySQL**;
- Framework **Spring**.

O **MySQL** é o gerenciador de banco de dados que utilizaremos para persistir ("salvar") os dados que cadastrarmos. O ecossistema **Spring** também possui uma série de tecnologias específicas que facilitarão nosso trabalho:

- **Spring web**: permite-nos trabalhar com web services, que são métodos usados para trafegar os dados da nossa **API** através de requisições HTTP;
- **Spring Boot DevTools**: auxilia-nos na produtividade, reiniciando a aplicação após alterações no código;
- **MySQL Driver**: para conexão com nosso banco de dados;
- **Validation**: para validação dos dados da nossa aplicação.

O INÍCIO

Nosso próximo passo será a criação do nosso projeto (uhuuu! 🎉). Podemos criá-lo de diversas formas, mas o **Spring** nos dá uma maneira muito rápida e prática através do site start.spring.io, onde podemos configurar os dados iniciais, importar dependências e gerar nosso projeto com alguns poucos cliques. Dá uma olhada:

The screenshot shows the Spring Initializr web interface. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected. Under 'Spring Boot', '2.4.3' is selected. The 'Project Metadata' section includes fields for Group (com.orangeTalents), Artifact (cadastroVacao), Name (cadastroVacao), Description (Controle de vacinação por pessoas), and Package name (com.orangeTalents.cadastroVacao). The 'Packaging' is set to 'Jar' and the 'Java' version is '8'. On the right, the 'Dependencies' section lists 'Spring Web' (WEB), 'Spring Boot DevTools' (DEVELOPER TOOLS), 'Spring Data JPA' (SQL), 'MySQL Driver' (SQL), and 'Validation' (I/O). At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'. A Windows activation notice is visible in the bottom right corner.

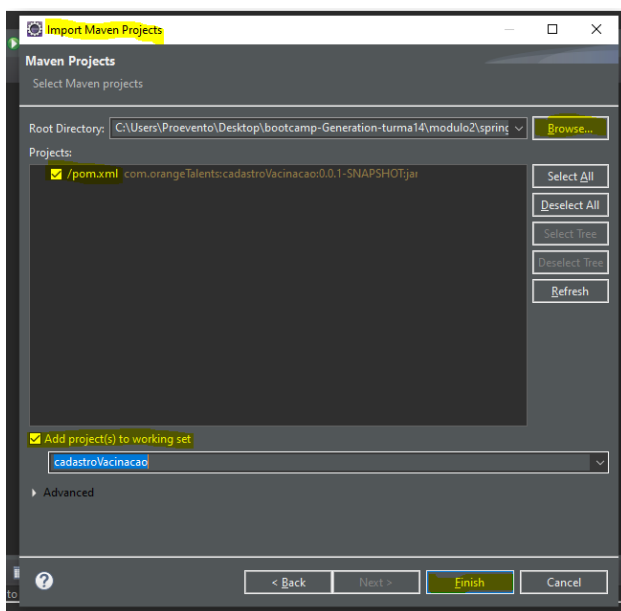
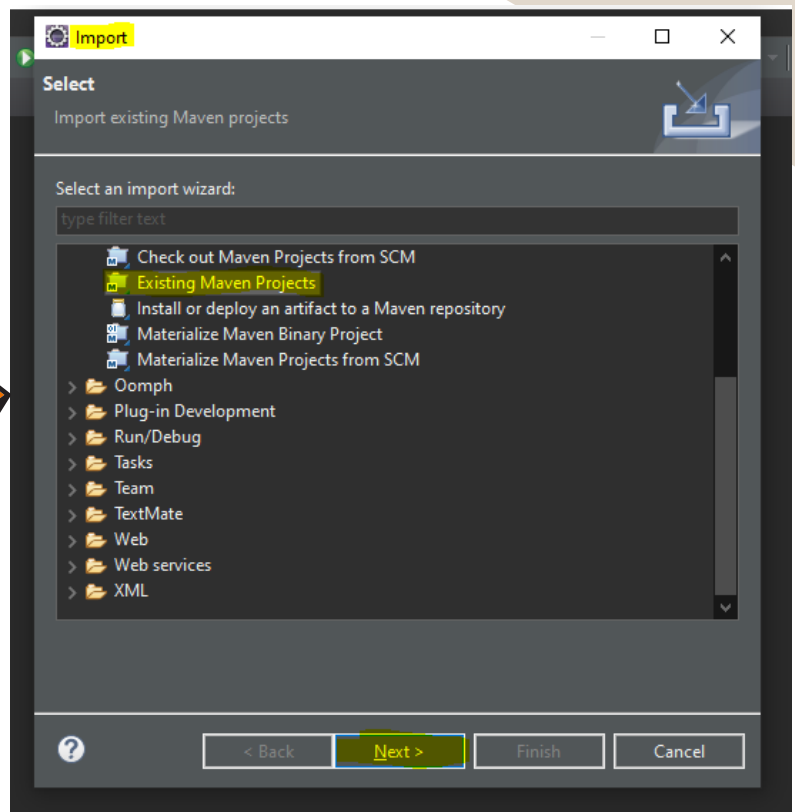
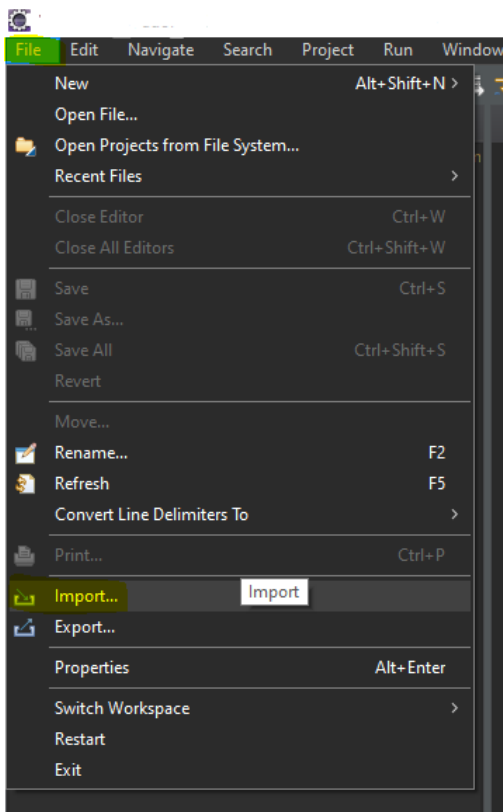
Conforme a imagem acima, definimos alguns pontos importantes para nosso projeto:

- nosso gerenciador de pacotes será o *Maven*;
- Definimos o nome do nosso projeto, dos seus pacotes e demos uma breve descrição nos campos "Project Metadata" (fique à vontade para criar os seus 😊)
- Também definimos como dependências as tecnologias que abordamos anteriormente.

Orange TALENTS

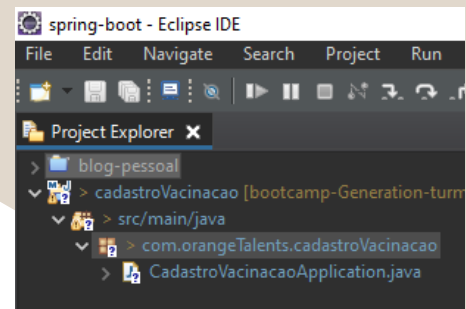
Após isso, ao clicar no botão **Generate**, o **Spring Initializr** gera um arquivo **ZIP** com nosso projeto, que pode ser importado para qualquer **IDE**.

Aqui utilizaremos o **Eclipse**; podemos importar o projeto pelo caminho *File -> Import -> Existing Maven Projects* e selecionar a pasta onde extraímos o projeto.



Não esqueça de marcar a opção **pom.xml, que é o arquivo que contém todas as nossas dependências.*

Feito isso, nosso projeto está importado e já temos uma **Classe** padrão criada automaticamente!



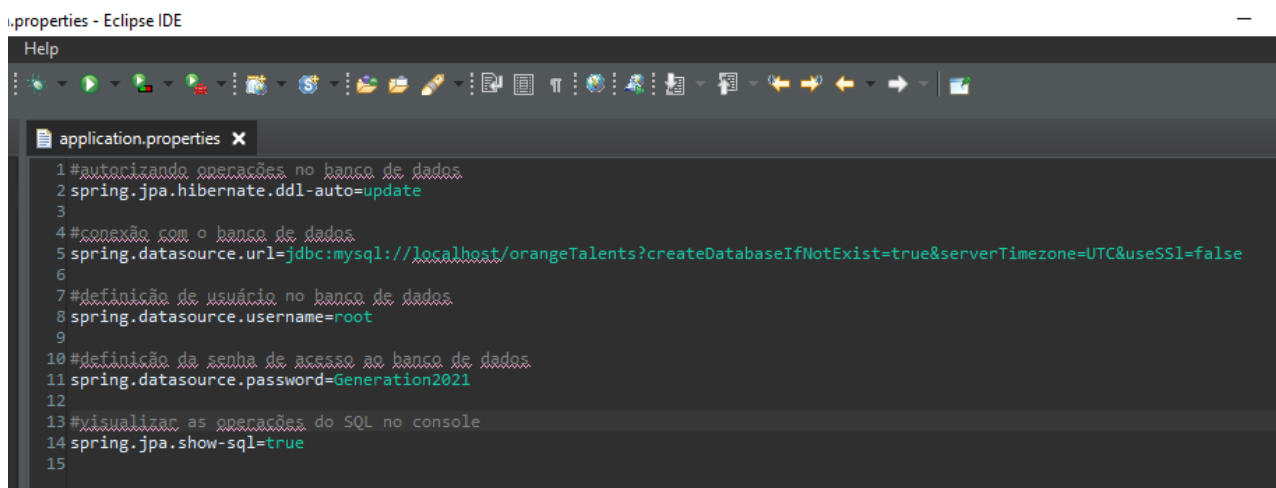
O BANCO DE DADOS

Antes de prosseguirmos com nossa **API**, precisamos instalar e configurar nosso gerenciador de banco de dados, o **MySQL Workbench 8.0**. Você pode baixá-lo neste link: <https://dev.mysql.com/downloads/workbench/>

CONFIGURANDO

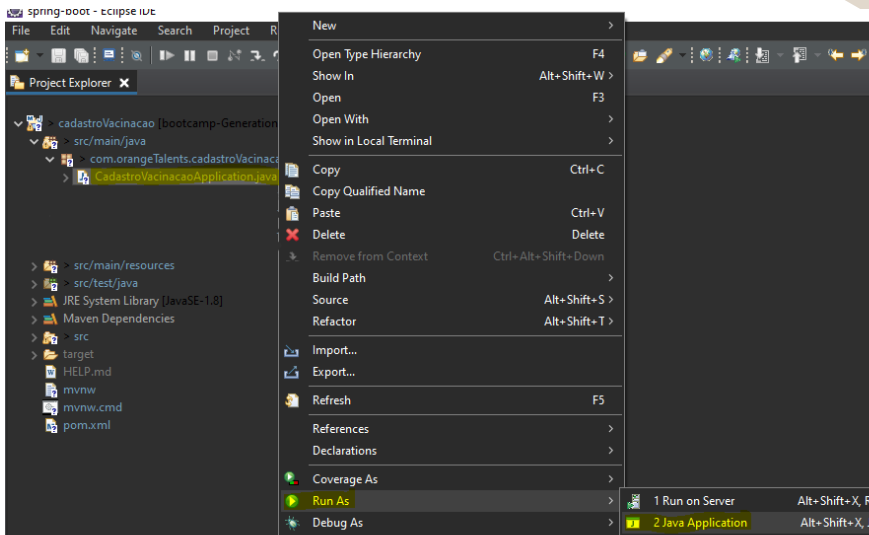
Agora que já criamos nosso projeto e instalamos o **MySQL**, vamos começar a construir nossa **API**!

A primeira coisa que precisamos fazer é configurar o acesso da nossa aplicação ao banco de dados, para que possamos enviar, salvar e manipular nossos dados. Vamos fazer isso pelo arquivo **application.properties** que está no caminho `src -> main -> resources`:

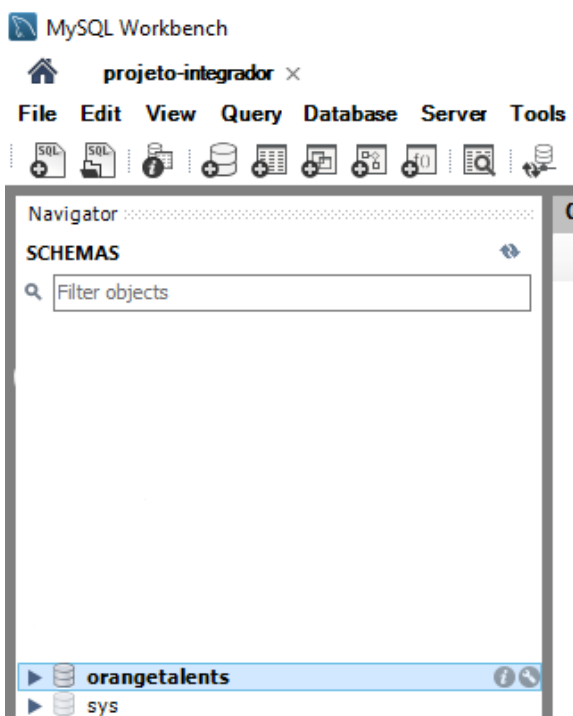
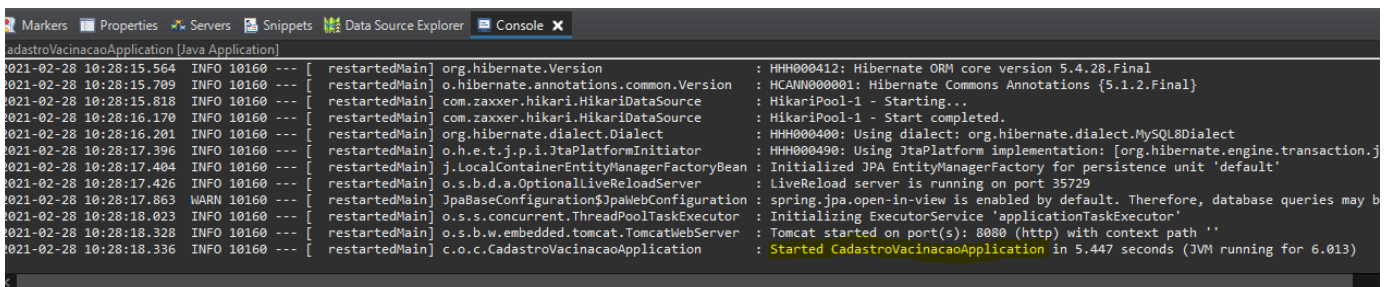


Orange TALENTS

Configurada nossa conexão com o banco de dados, vamos rodar nossa aplicação através do **CadastroVacinacaoApplication.java**, que está em *src -> main -> java -> com.orangeTalents.cadastroVacinacao*, clicando com o botão direito e *Run As -> Java Application*:



Se tudo der certo, a aplicação será inicializada no console e nosso banco de dados **orangeTalents** será criado no **MySQL**:



Criado nosso banco de dados, vamos falar um pouco sobre a *arquitetura* da nossa aplicação, quais classes e interfaces precisaremos criar e como vamos configurá-las para nossa **API** "rodar" lindamente! 🐱

MAS COMO FUNCIONA?

Após todas essas configurações, você deve estar se perguntando: "**Mas como vamos receber e enviar dados?**" 🤔

Para isso, precisamos criar alguns pacotes (iguais ao pacote criado automaticamente pelo **Spring Initializr**, lembra dele?). Esses pacotes são as camadas da nossa **API**: *Model*, *Repository*, *Service* e *Controller*.

Vamos dar uma olhada em cada uma delas:

- O pacote **Model** é onde ficam os modelos da nossa aplicação; aqui criaremos as classes que servirão de modelo para a criação das nossas tabelas no banco de dados;
- O pacote **Repository** é o responsável pelas transações diretas com o banco de dados, sem ele não conseguiremos persistir nossos cadastros;
- O **Service** é responsável por nossas regras de negócio e por fazer a ponte entre o *Controller* e o *Repository*;
- Finalmente, a responsabilidade do **Controller** é fazer a ponte entre o usuário e nosso sistema: ele recebe as requisições enviadas, executa as respectivas ações por meio do acesso à camada *Model* e envia os resultados obtidos de volta ao usuário.

MODEL

Para os fins dessa **API**, aqui criaremos duas classes: *Usuarios* e *VacinasAplicadas*, que serão, respectivamente, os modelos para cadastrarmos usuários e aplicações de vacina no banco de dados. Veja:

MODEL USUARIOS

```
1 package com.orangeTalents.cadastroVacinacao.model;
2
3 import java.util.Date;
4
5 @Entity //Definimos nossa classe como uma entidade que servirá de modelo ao banco de dados;
6 @Table(name="tb_usuarios") //Aqui definimos o nome da tabela a ser criada no banco de dados.
7 public class Usuarios {
8     //Definindo atributos e constraints ("restrições") que desejamos adicionar aos campos;
9     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
10     private long id;
11
12     @NotNull @Size(min=5, max=100)
13     private String nome;
14
15     @NotNull
16     @Email
17     @Column(unique=true)
18     private String email;
19
20     @NotNull
21     @CPF
22     @Column(unique=true)
23     private String cpf;
24
25     @NotNull
26     @JsonFormat(pattern="dd/MM/yyyy")
27     private Date nascimento;
28
29     @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL) //Definindo tipo de relacionamento
30     @JsonIgnoreProperties("usuario")
31     private List<VacinasAplicadas> vacinas;
32
33     //Métodos Getters e Setters
34     public long getId() {
35         return id;
36     }
37
38     public void setId(long id) {
39         this.id = id;
40     }
41
42     public String getNome() {
43         return nome;
44     }
45
46     public void setNome(String nome) {
47         this.nome = nome;
48     }
49
50     public String getEmail() {
51         return email;
52     }
53
54     public void setEmail(String email) {
55         this.email = email;
56     }
57
58     public String getCpf() {
59         return cpf;
60     }
61
62     public void setCpf(String cpf) {
63         this.cpf = cpf;
64     }
65
66     public Date getNascimento() {
67         return nascimento;
68     }
69
70     public void setNascimento(Date nascimento) {
71         this.nascimento = nascimento;
72     }
73 }
```

```
48
49 //Métodos Getters e Setters
50 public long getId() {
51     return id;
52 }
53
54 public void setId(long id) {
55     this.id = id;
56 }
57
58 public String getNome() {
59     return nome;
60 }
61
62 public void setNome(String nome) {
63     this.nome = nome;
64 }
65
66 public String getEmail() {
67     return email;
68 }
69
70 public void setEmail(String email) {
71     this.email = email;
72 }
73
74 public String getCpf() {
75     return cpf;
76 }
77
78 public void setCpf(String cpf) {
79     this.cpf = cpf;
80 }
81
82 public Date getNascimento() {
83     return nascimento;
84 }
85
86 public void setNascimento(Date nascimento) {
87     this.nascimento = nascimento;
88 }
89 }
```

MODEL VACINAS

```
1 package com.orangeTalents.cadastroVacinacao.model;
2
3 import java.util.Date;
4
5 @Entity
6 @Table(name="tb_vacina")
7 public class VacinasAplicadas {
8
9     @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
10     private long id;
11
12     @NotNull
13     private String nomeVacina;
14
15     @NotNull
16     @Email
17     private String emailUsuario;
18
19     private Date dataAplicacao = new Date(); //Atribuindo valor de data atual como padrão;
20
21     @ManyToOne
22     @JsonIgnoreProperties("vacinas")
23     private Usuarios usuario;
24
25     public long getId() {
26         return id;
27     }
28 }
```

```
37
38 public long getId() {
39     return id;
40 }
41
42 public void setId(long id) {
43     this.id = id;
44 }
45
46 public String getNomeVacina() {
47     return nomeVacina;
48 }
49
50 public void setNomeVacina(String nomeVacina) {
51     this.nomeVacina = nomeVacina;
52 }
53
54 public String getEmailUsuario() {
55     return emailUsuario;
56 }
57
58 public void setEmailUsuario(String emailUsuario) {
59     this.emailUsuario = emailUsuario;
60 }
61
62 public Date getDataAplicacao() {
63     return dataAplicacao;
64 }
65
66 public void setDataAplicacao(Date dataAplicacao) {
67     this.dataAplicacao = dataAplicacao;
68 }
69
70 }
71 }
```

Nessas duas classes da camada *Model* temos os atributos, que serão os campos das tabelas a serem geradas no banco de dados, e os *getters* e *setters*. Além disso, você deve ter percebido alguns "@s" no nosso código, eles são chamados de "anotações" (annotations) e servem para definirmos comportamentos e restrições nas classes e atributos, como a validação de atributos com **@NotNull**, **@Email** e **@CPF**. Você pode conhecer um pouco mais sobre elas neste link:

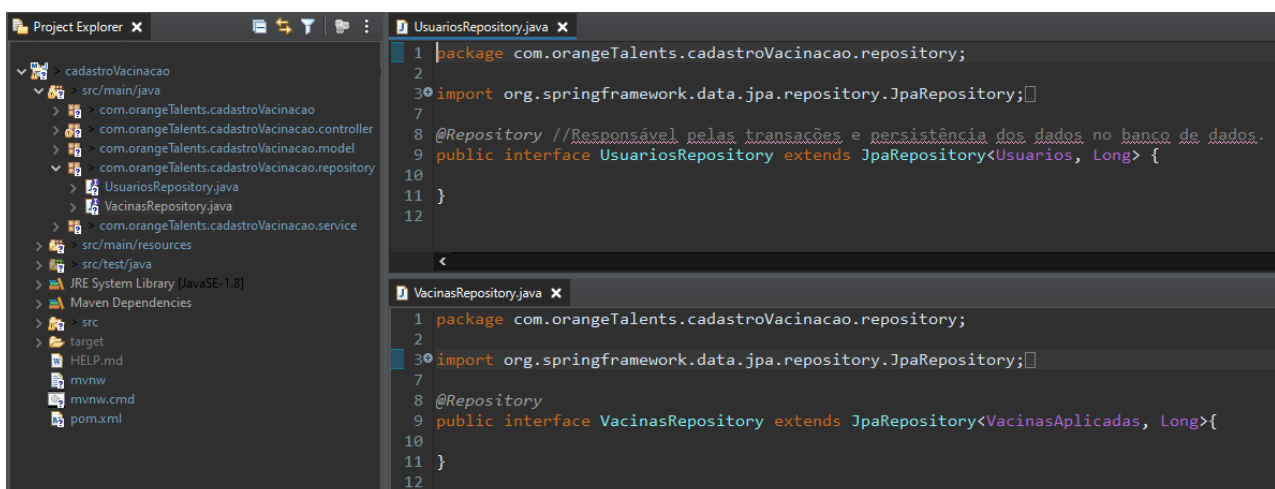
domineiospring.wordpress.com/2016/07/13/guia-das-annotations-do-spring/

**Aqui vale ressaltar que as anotações @ManyToOne e @OneToMany servem para definirmos um relacionamento entre tabelas, isso foi feito visando um desenvolvimento posterior dos métodos GET dessa API, mas não terá aplicação para os fins deste desafio pois apenas cadastraremos os dados.*

REPOSITORY

Feita a *Model*, vamos construir nossas interfaces *Repository*, para nos comunicarmos efetivamente com o banco de dados que criamos. Elas ficarão assim:

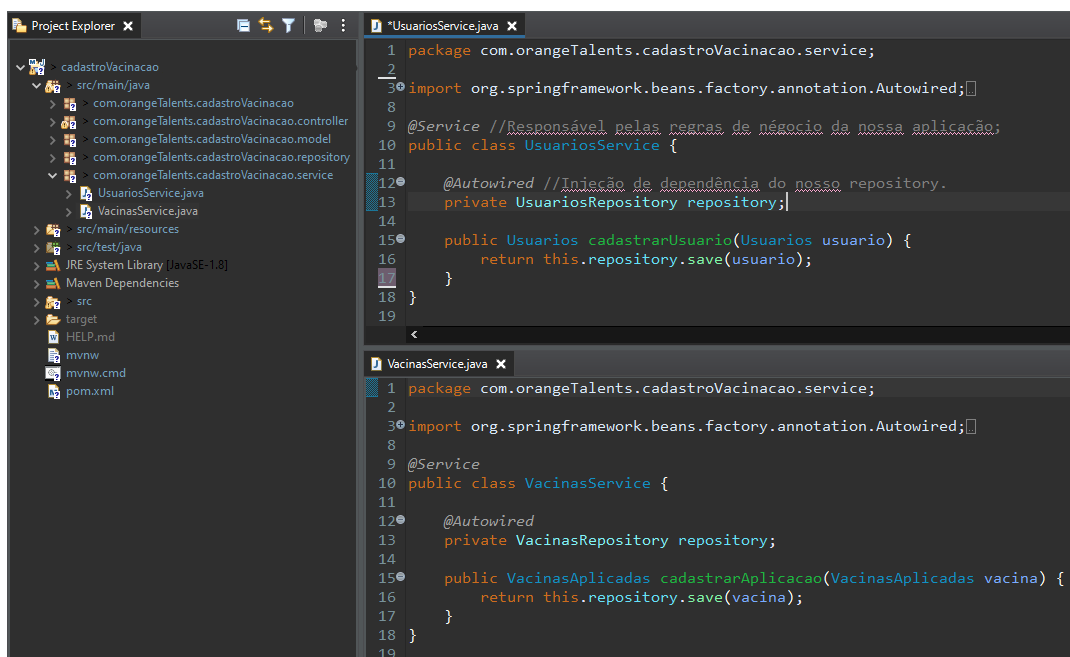
REPOSITORY USUARIOS E VACINAS



SERVICE

Agora é a vez do *Service*, onde vamos definir um método de cadastro para nossos dados e que será a ligação entre o *Controller* e o *Repository*; note que estamos injetando nosso *Repository* e utilizando-o em nosso método de cadastro. Veja:

SERVICE USUARIOS E VACINAS



CONTROLLER

Chegamos ao *Controller*, nossa camada de comunicação entre as requisições realizadas pelo usuário e a nossa aplicação. Como vamos realizar apenas cadastros por enquanto, criamos somente um método **Post**, que é usado para cadastro de dados. Veja como ficou:

CONTROLLER USUARIOS E VACINAS

```
15 @RestController //Definimos que nosso controller fornecerá serviços REST, retornando os dados em formato JSON;
16 @RequestMapping("/usuarios") //Definindo endpoint das requisições desta controller, ou seja, os métodos definidos aqui serão chamados pela url http://localhost:8080/usuarios
17 public class UsuariosController {
18
19     @Autowired
20     private UsuariosRepository repository;
21
22     @Autowired
23     private UsuariosService service;
24
25     //Método post utilizado para inserirmos dados;
26     @PostMapping("/cadastrar") //definindo caminho ao que nosso método será chamado.
27     public ResponseEntity<Usuarios> postUsuario(@RequestBody Usuarios usuario){
28         try {
29             //Utilizamos um objeto ResponseEntity para manipular o tipo de resposta HTTP que queremos
30             //HttpStatus para definirmos o tipo de resposta
31             return ResponseEntity.status(HttpStatus.CREATED).body(service.cadastrarUsuario(usuario));
32         } catch (Exception e) {
33             return new ResponseEntity<Usuarios>(HttpStatus.BAD_REQUEST);
34         }
35     }
36 }
```

```
15 @RestController
16 @RequestMapping("/vacinas")
17 public class VacinasController {
18
19     private VacinasRepository repository;
20
21     @Autowired
22     private VacinasService service;
23
24
25     @PostMapping("/cadastrar")
26     public ResponseEntity<VacinasAplicadas> postVacinas(@RequestBody VacinasAplicadas aplicacao){
27         try {
28             return ResponseEntity.status(HttpStatus.CREATED).body(service.cadastrarAplicacao(aplicacao));
29         } catch (Exception e) {
30             return new ResponseEntity<VacinasAplicadas>(HttpStatus.BAD_REQUEST);
31         }
32     }
33 }
34 }
```

HORA DE TESTAR!

Depois de construirmos todas as nossas camadas, é hora de testar se nossa **API** está, de fato, "rodando" lindamente! (Cruzou os dedos aí? 🙌)

Para testar nossa aplicação utilizaremos o **Postman**, que é um ambiente super fácil e eficiente para testes em **APIs** e requisições em geral. Você pode baixá-lo neste link: www.postman.com/downloads/ .

Agora vamos executar novamente nossa classe padrão **CadastroVacinacaoApplication** para rodar nossa **API** no endereço <http://localhost:8080> .

POSTMAN

Com nossa **API** rodando, vamos acessar no **Postman** as URLs, que definimos no nosso *Controller* para os cadastros de *usuários* e de *aplicação de vacinas*. Também vamos definir o método de envio da requisição como **Post** e passar no corpo da requisição, em formato JSON, as informações que desejamos enviar.

TESTANDO CADASTRO DE USUÁRIOS

The screenshot shows the Postman interface. The request is a POST to `http://localhost:8080/usuarios/cadastrar`. The body is in JSON format:

```
{
  "nome": "Breno Noccioli",
  "email": "brenonoccioli@gmail.com",
  "cpf": "99230640344",
  "nascimento": "05/09/1990"
}
```

The response is also shown in JSON format:

```
{
  "id": 1,
  "nome": "Breno Noccioli",
  "email": "brenonoccioli@gmail.com",
  "cpf": "99230640344",
  "nascimento": "05/09/1990"
}
```

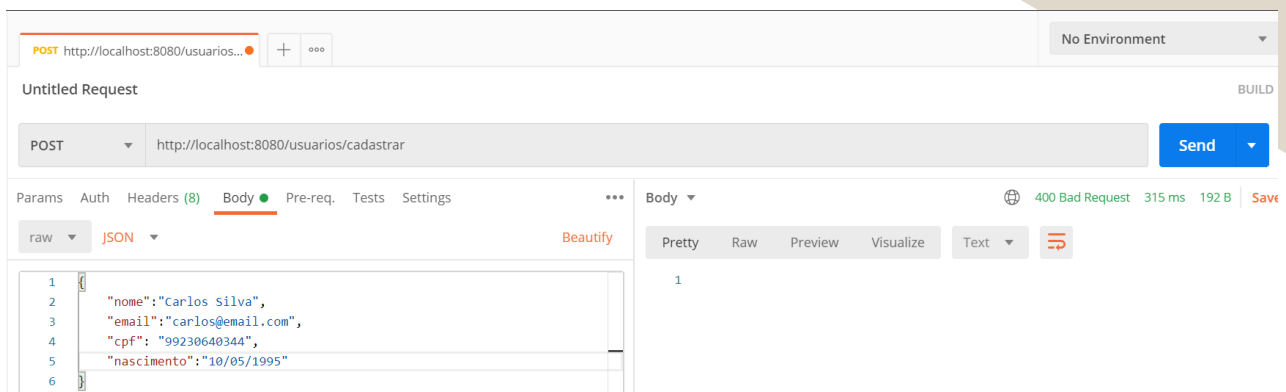


Se confirmarmos no banco de dados, veremos que o usuário foi cadastrado! 🎉

id	cpf	email	nascimento	nome
1	99230640344	brenonoccioli@gmail.com	1990-09-05 00:00:00.000000	Breno Noccioli
NULL	NULL	NULL	NULL	NULL

Mas e se passarmos um *e-mail* ou *CPF* já existentes em nosso banco de dados?

A resposta será um status de **erro 400 Bad Request**, conforme configuramos em nosso Controller! 🚩



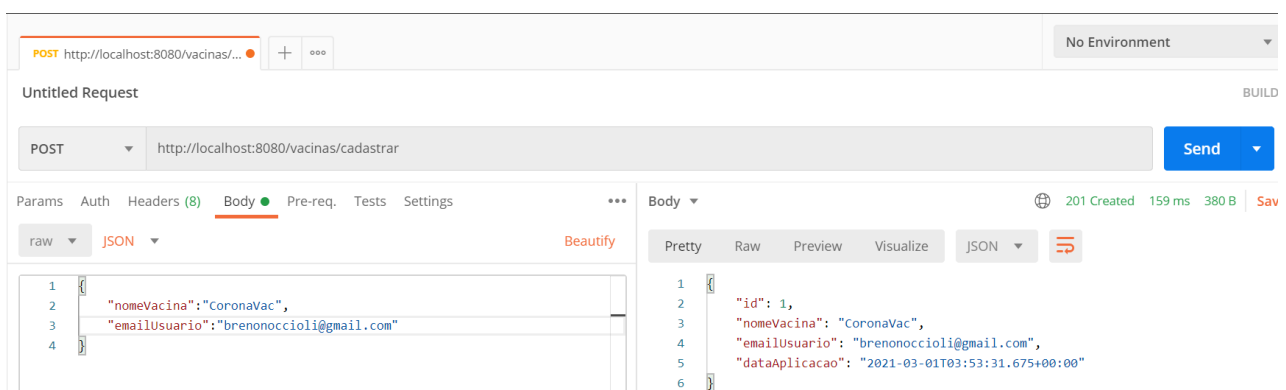
POST http://localhost:8080/usuarios/cadastrar

Body (JSON):

```
1 {
2   "nome": "Carlos Silva",
3   "email": "carlos@email.com",
4   "cpf": "99230640344",
5   "nascimento": "10/05/1995"
6 }
```

Response: 400 Bad Request 315 ms 192 B

TESTANDO CADASTRO DE APLICAÇÃO DE VACINA



POST http://localhost:8080/vacinas/cadastrar

Body (JSON):

```
1 {
2   "nomeVacina": "CoronaVac",
3   "emailUsuario": "brenonoccioli@gmail.com"
4 }
```

Response: 201 Created 159 ms 380 B

Body (JSON):

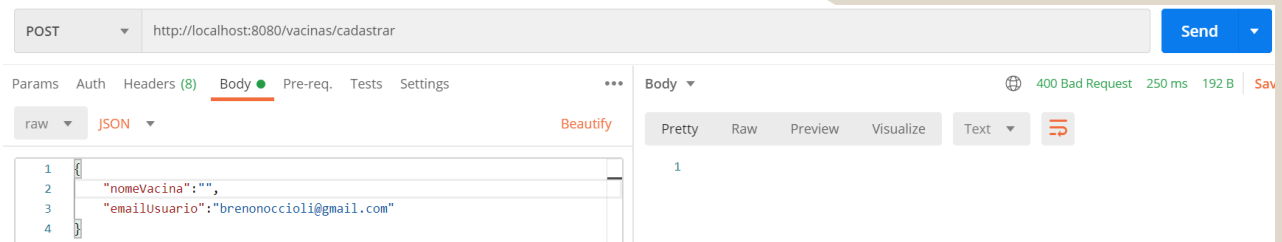
```
1 {
2   "id": 1,
3   "nomeVacina": "CoronaVac",
4   "emailUsuario": "brenonoccioli@gmail.com",
5   "dataAplicacao": "2021-03-01T03:53:31.675+00:00"
6 }
```



Status **201 Created** na resposta da requisição e dados cadastrados com sucesso no **MySQL**! 🚩

Result Grid					
Filter Rows:		Edit:		Export/Import:	
	id	data_aplicacao	email_usuario	nome_vacina	usuario_id
▶	1	2021-03-01 03:53:31.675000	brenonoccioli@gmail.com	CoronaVac	NULL
✱	NULL	NULL	NULL	NULL	NULL

Da mesma forma, se a requisição não atender as regras que definimos na **API**, a resposta será um **erro 400 Bad Request**.



IMPLEMENTANDO UM FORMULÁRIO

Após nossos testes no **Postman**, criei um formulário para implementarmos nossa aplicação em uma interface mais gráfica utilizando **HTML, CSS, Bootstrap, Typescript** e **Angular** para integrar nossa **API**:

A screenshot of a web browser displaying a user registration form titled 'Cadastro de usuários'. The form is set against a dark blue background with an orange header. It contains four input fields: 'Nome completo' (placeholder: 'Informe seu nome completo'), 'E-mail' (placeholder: 'exemplo@exemplo.com'), 'CPF' (placeholder: 'Apenas números - sem pontos e traços'), and 'Data de nascimento' (placeholder: 'dd/mm/aaaa'). A green 'Cadastrar' button is at the bottom. Below the button is a link that says 'Ir para Cadastro de Vacinas'. The browser's address bar shows 'localhost:4200/#/usuarios'.

*Antes do teste com o Angular, apaguei o banco de dados e reiniciei a **API** para criarmos um novo e não termos problemas com a inserção de **e-mail** e **CPF** duplicados.

CADASTRANDO UM USUÁRIO

localhost:4200/#/usuarios

Orange
TALENTS

localhost:4200 diz
Usuário cadastrado com sucesso!

OK

Cadastro de usuários

Nome completo

Breno Nocchioli

E-mail

brenonocchioli@gmail.com

CPF

99230640344

Data de nascimento

05/09/1990

Cadastrar

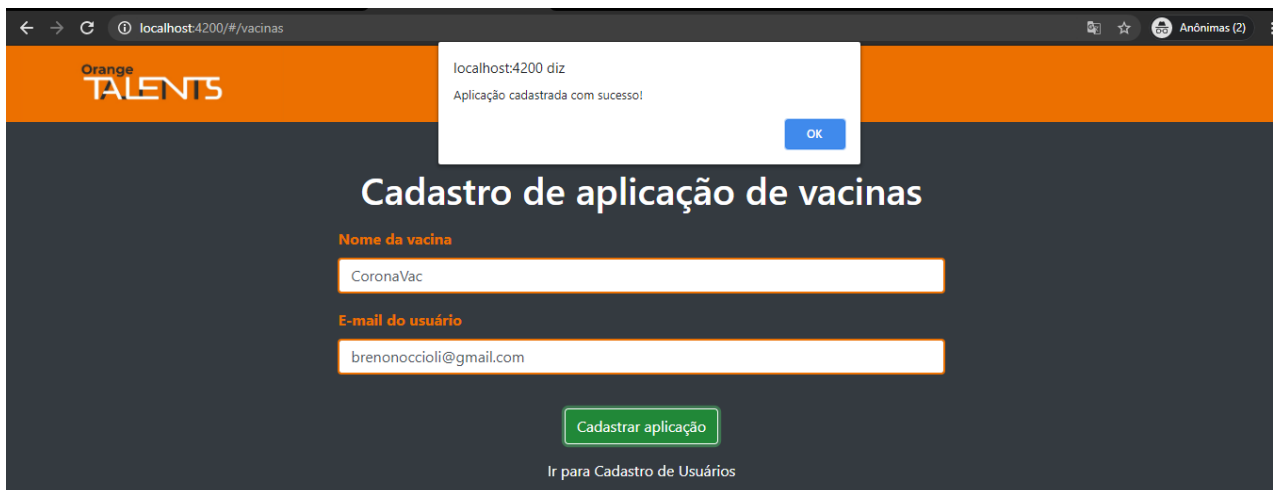
Ir para Cadastro de Vacinas



Mensagem de sucesso e
usuário cadastrado no
banco de dados!

Result Grid					
Filter Rows:		Edit: Export/Import:			
	id	cpf	email	nascimento	nome
▶	1	99230640344	brenonocchioli@gmail.com	1990-09-05 00:00:00.000000	Breno Nocchioli
*	NULL	NULL	NULL	NULL	NULL

CADASTRANDO A APLICAÇÃO DE UMA VACINA



localhost:4200/#/vacinas

Orange
TALENTS

localhost:4200 diz
Aplicação cadastrada com sucesso!

OK

Cadastro de aplicação de vacinas

Nome da vacina

CoronaVac

E-mail do usuário

brenonoccioli@gmail.com

Cadastrar aplicação

[Ir para Cadastro de Usuários](#)



E mais um cadastro
realizado com sucesso!

	id	data_aplicacao	email_usuario	nome_vacina	usuario_id
▶	1	2021-03-01 04:32:04.680000	brenonoccioli@gmail.com	CoronaVac	NULL
✱	NULL	NULL	NULL	NULL	NULL

DEU CERTO!

Se chegamos até aqui é porque a nossa **API** está funcionando! 🎉

Ficou alguma dúvida? Entre em contato comigo!
Esse projeto também estará no meu repositório no Github.

OBRIGADO!

por ter me acompanhado até aqui.

Breno Nocchioli
brenonocchioli@gmail.com

