

API Rest Orange Talents

desafio:

implementar esse sistema utilizando Java como linguagem e Spring + Hibernate como stacks de tecnologia fundamentais da aplicação: Escreva um post de blog explicando de maneira detalhada tudo que você faria para implementar esse código.

Nome: Daniel Zeferino Ferreira

- Olá!! meu nome é Daniel,
vou estar criando uma
API Rest com a função de
controlar aplicação de
vacinas entre a
população brasileira

Mão na massa!!

Tecnologias e dependências que utilizarei no projeto

Tecnologías:

- Java
- Mysql

Frameworks:

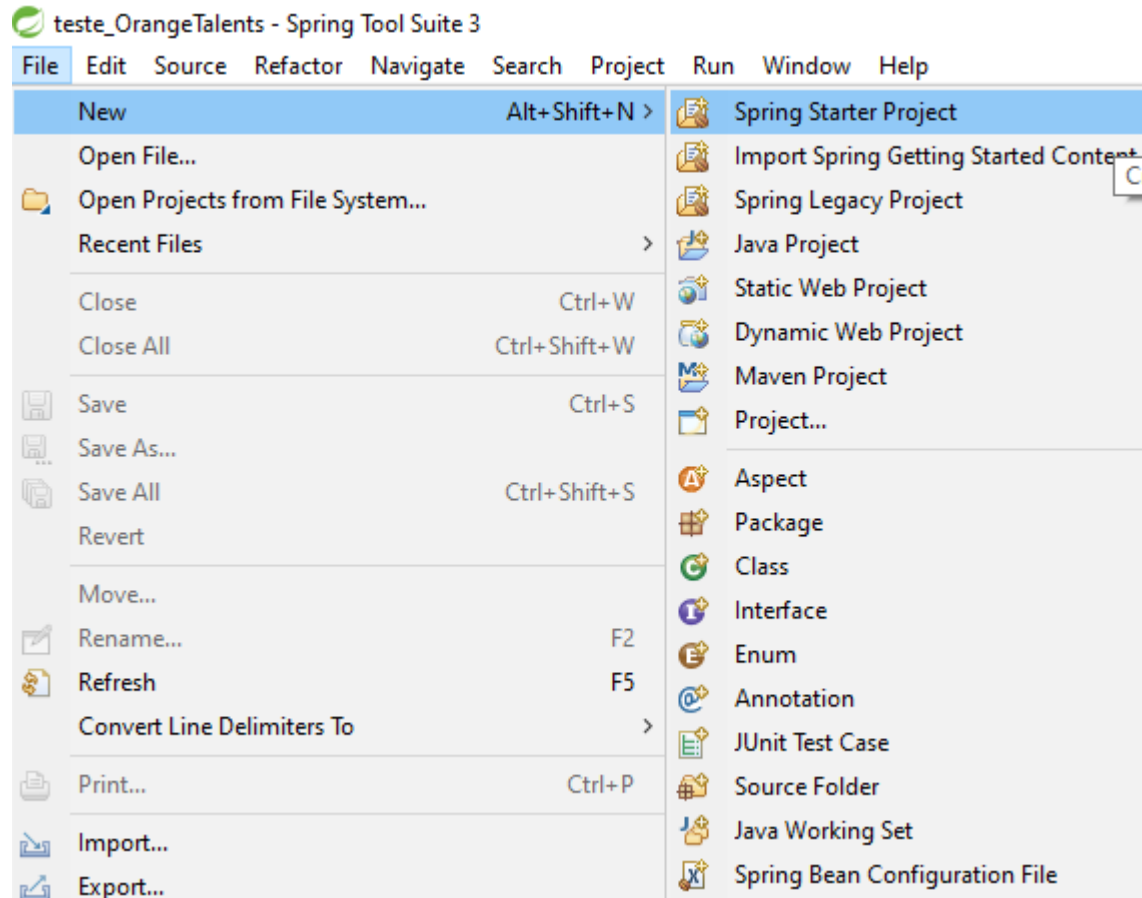
- Spring boot
- Hibernate (ORM)

Dependências:

- Spring boot DevTools - Será responsável por reload da api sempre que houver alguma atualização no código.
- Validation - será responsável por fazer as validações dos dados como, CPF, Email etc..
- Spring Web - Irá adicionar as dependências do spring MVC.
- MySQL Driver - Será o responsável por conectar a API ao banco de dados
- Spring data JPA - Adiciona as dependências do Spring Data, para que seja possível a persistência dos dados, adiciona também o Hibernate, que será o ORM, responsável por transformar os dados da estrutura lógica de um banco de dados em objetos relacionais.

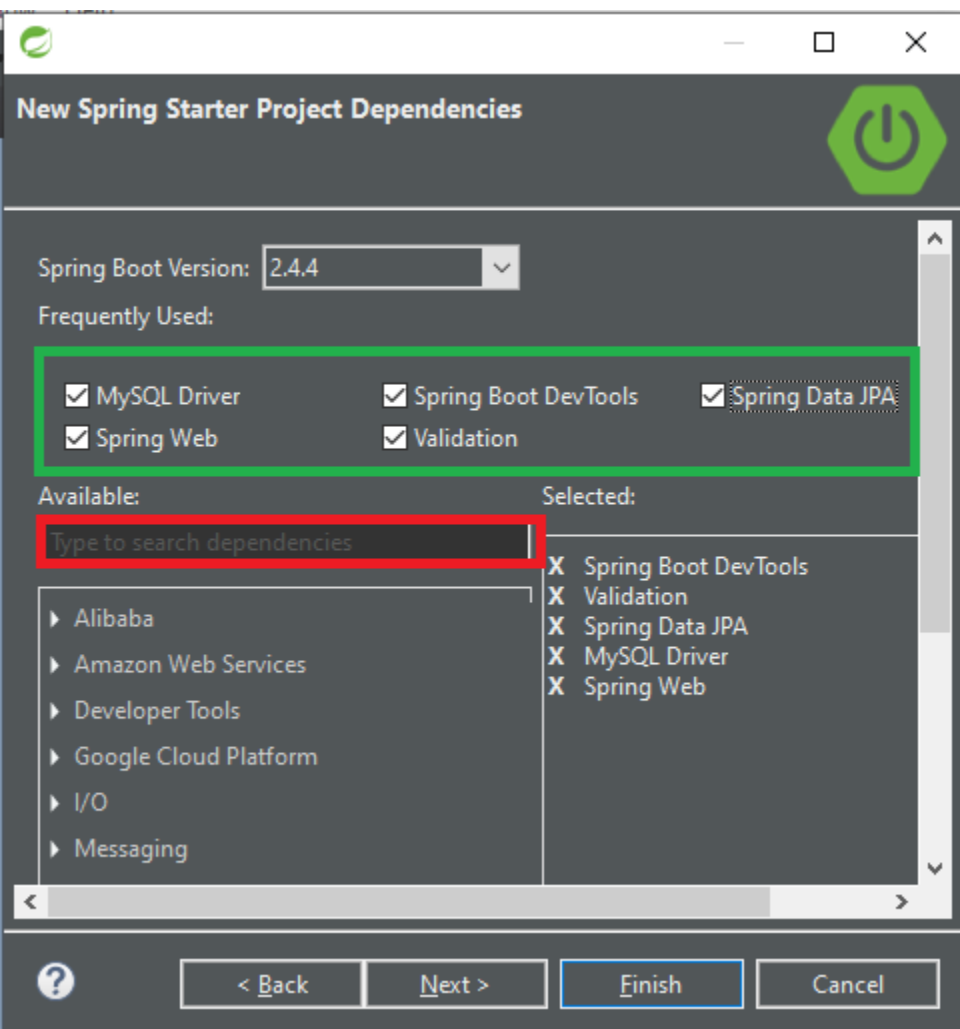
Vou estar utilizando a IDE STS (Spring Tool Suite), assim facilita a criação do projeto colocando as dependências.

Passo 1 - na IDE STS vai em File, New e clica em Spring Starter Project

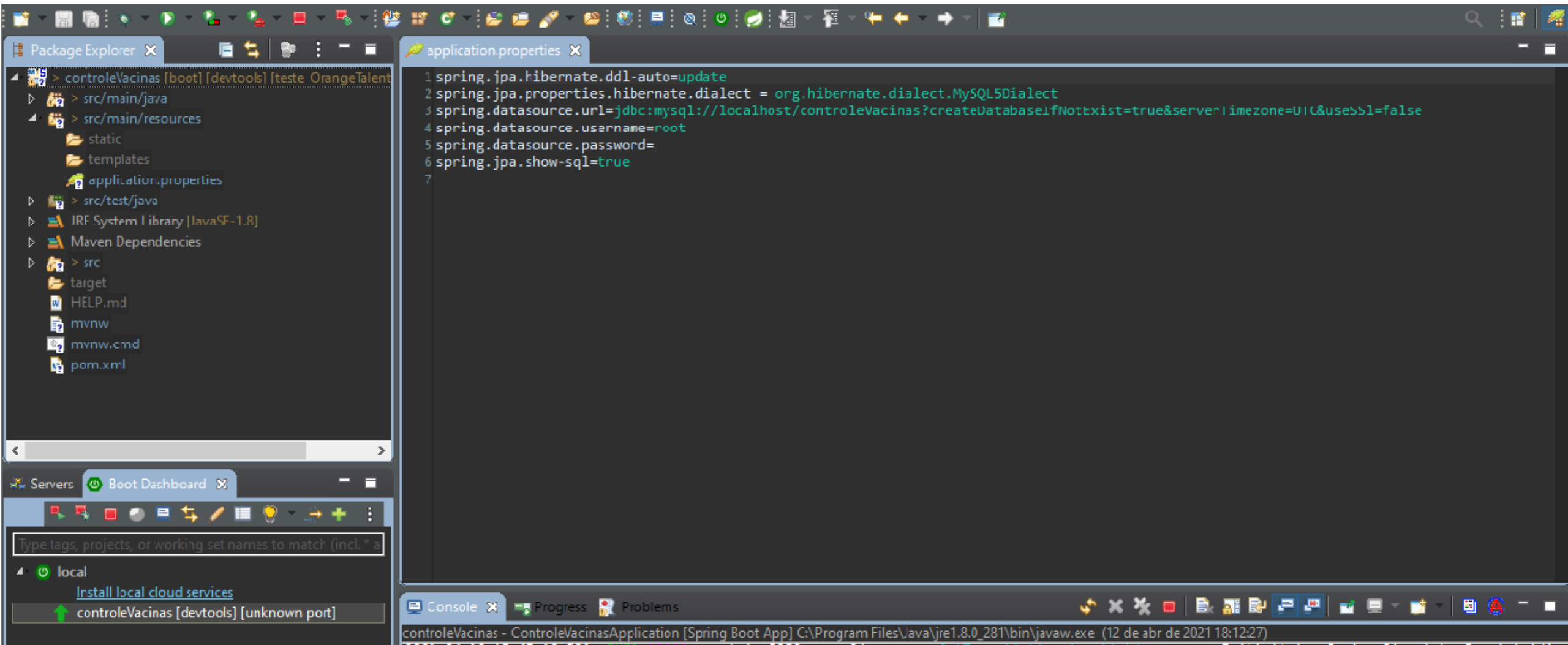


Passo 2 – Escolha as dependências que utilizara no projeto, no meu caso como eu já tinha utilizado elas em outros projetos já aparece, mais se caso não aparecer é só digitar o nome da dependência no campo vermelho como esta na imagem a baixo, e selecionar as desejadas.

Agora só Clica em Finish e pronto projeto criado com sucesso.



Passo 3 - Com o projeto criado corretamente, irei começar a configuração do **Application.properties**, este arquivo é encontrado **src/main/resources**, nele vamos configurar por exemplo: a conexão com o banco de dados, inserções de usuário e senha e até mesmo em alguns casos alterar a porta do local host quando necessário.



`spring.jpa.hibernate.ddl-auto=update` Na linha acima, nós estamos dando uma instrução para que sempre que a aplicação for iniciada, o JPA faça uma varredura no banco de dados, onde já temos uma relação estabelecida, e então, faça uma comparação verificando se o banco de dados está exatamente igual aos padrões da API, no momento em que ela está sendo executada, caso contrário ele fará uma atualização no banco para que ambos tenham as mesmas informações.

`spring.datasource.url=jdbc:mysql://localhost/controleVacinas?createDatabaseIfNotExist=true&serverTimezone=UTC&useSSL=false`

Esta linha indica qual o caminho do banco de dados e o seu nome, logo após a interrogação diz que caso não exista um banco com este nome, que um novo banco seja criado. Depois dizemos o padrão de Timezone que será UTC e que não estamos utilizando nenhuma certificação SSL.

`spring.datasource.username=root`

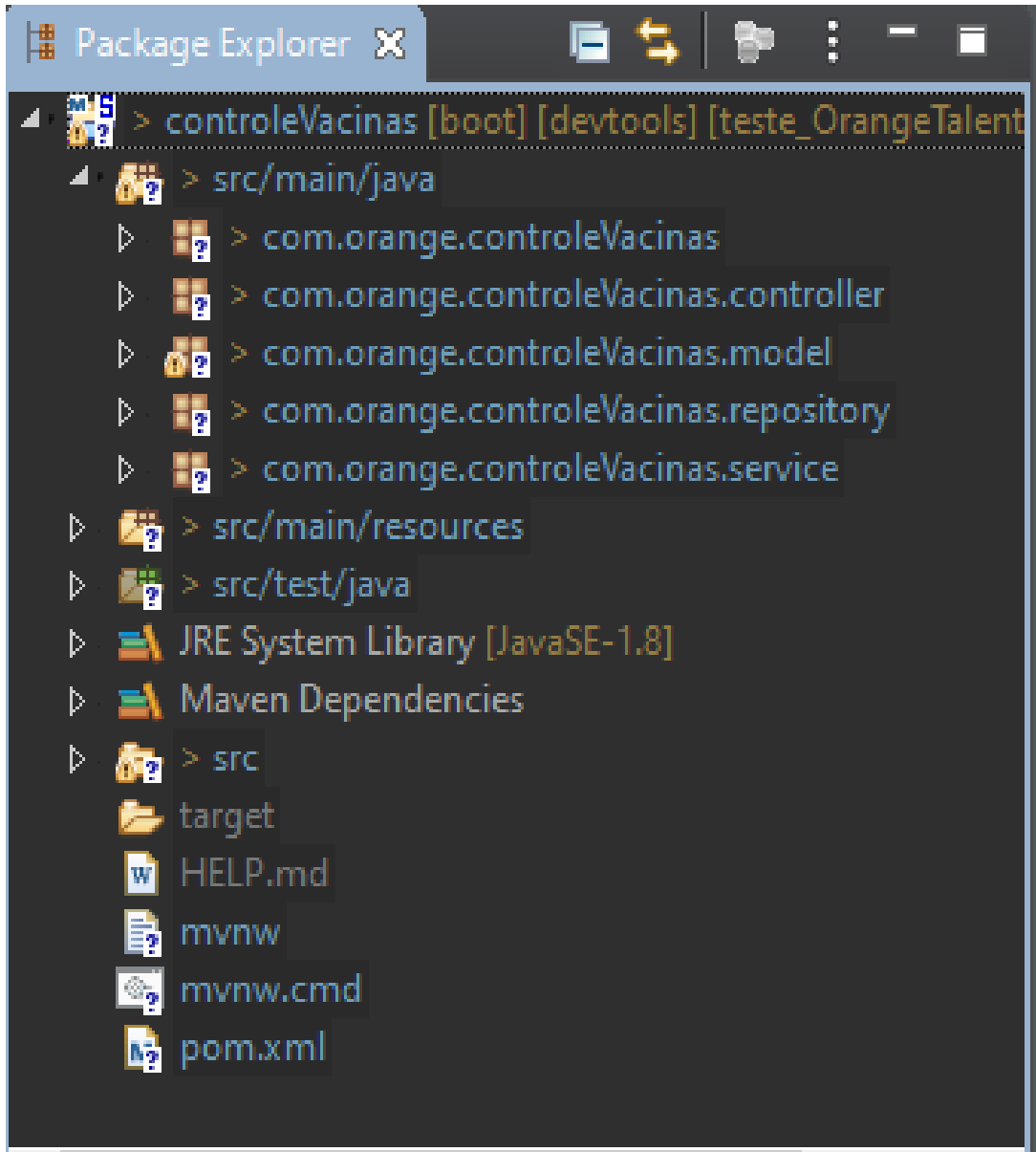
`spring.datasource.password=`

As linhas cima informam o usuário e senha do banco de dados.

`spring.jpa.show-sql=true`

E aqui dizemos que cada alteração realizada no banco de dados, deverá ser mostrada no console. Trabalhando com O projeto foi estruturado baseando-se na arquitetura MVC, onde cada camada possui sua responsabilidade, desta maneira tornando o código mais fácil para manutenções futuras.

O projeto foi separado nas seguintes camadas: Model, Repository, Service e Controller.



Criando a Model

A model é na onde geramos o objeto e criamos sua tabela ao executar o programa. Dentro do pacote da model, vamos criar duas classes, uma classe de usuário e outra classe de Vacina. A classe de usuário é responsável por armazenar todos os atributos de usuário como mostrado na imagem a seguir.

```
Usuario.java X
1 package com.orange.controleVacinas.model;
2
3 import java.time.LocalDate;
17
18
19 @Entity
20 @Table(name="tb_usuario")
21 public class Usuario {
22
23     @Id
24     @GeneratedValue(strategy = GenerationType.IDENTITY)
25     private long id;
26
27
28     @NotNull @NotEmpty
29     @Column(name = "nome")
30     private String nome;
31
32     @Email
33     @NotNull @NotEmpty
34     @Column(name = "email", unique = true)
35     private String email;
36
37     @CPF
38     @NotNull @NotEmpty
39     @Column(name = "cpf", unique = true)
40     private String cpf;
41
42     @DateTimeFormat(pattern = "dd/MM/yyyy")
43     private LocalDate dataNascimento;
44
45     public long getId() {
46         return id;
47     }
48 }
```

```
Usuario.java X
49
50     public void setId(long id) {
51         this.id = id;
52     }
53
54     public String getNome() {
55         return nome;
56     }
57
58     public void setNome(String nome) {
59         this.nome = nome;
60     }
61
62     public String getEmail() {
63         return email;
64     }
65
66     public void setEmail(String email) {
67         this.email = email;
68     }
69
70     public String getCpf() {
71         return cpf;
72     }
73
74     public void setCpf(String cpf) {
75         this.cpf = cpf;
76     }
77
78     public LocalDate getDataNascimento() {
79         return dataNascimento;
80     }
81
82     public void setDataNascimento(LocalDate dataNascimento) {
83         this.dataNascimento = dataNascimento;
84     }
85 }
```

A classe de vacina é responsável por armazenar todos os atributos de vacina e possui um relacionamento com a tabela de usuário, muitos para um, muitas vacinas para um usuário assim temos controle de qual vacina é de qual usuário e a data em que foi aplicada, como mostrado na imagem a seguir.

```
Usuario.java  Vacina.java X
1 package com.orange.controleVacinas.model;
2
3+ import java.time.LocalDate;
17
18 @Entity
19 @Table(name="tb_vacinas")
20 public class Vacina {
21
22     @Id
23     @GeneratedValue(strategy = GenerationType.IDENTITY)
24     private long id;
25
26     @NotNull @NotEmpty
27     @Column(name = "nomeVacina")
28     private String nomeVacina;
29
30     @ManyToOne
31     @JoinColumn(name = "id_usuario")
32     private Usuario usuario;
33
34
35
36     @DateTimeFormat(pattern = "dd/MM/yyyy")
37     private LocalDate dataRealizada;
38
39
40
```

```
Usuario.java  Vacina.java X
43
46     public void setId(long id) {
47         this.id = id;
48     }
49
50
51     public String getNomeVacina() {
52         return nomeVacina;
53     }
54
55
56     public void setNomeVacina(String nomeVacina) {
57         this.nomeVacina = nomeVacina;
58     }
59
60
61     public Usuario getUsuario() {
62         return usuario;
63     }
64
65
66     public void setUsuario(Usuario usuario) {
67         this.usuario = usuario;
68     }
69
70
71     public LocalDate getDataRealizada() {
72         return dataRealizada;
73     }
74
```

OBS: Lembrem-se de anotar as classes.

Por exemplo:

@Entity - dirá que a classe anotada se trata de uma entidade

@Table - informa que se trata de uma tabela no banco de dados

@Column – utilizamos com o (unique = true) nos atributos que queremos que seja únicos.

Temos também as marcações de validações que vem da dependência Validation que é:

@NotNull – Informa que o atributo não pode ser nulo

@NotEmpty – informa que o atributo não pode ser vazio

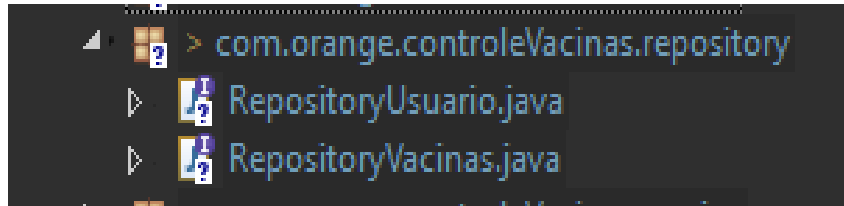
@CPF – aceita apenas CPF validos

@Email – verifica se tem todas as características de um email

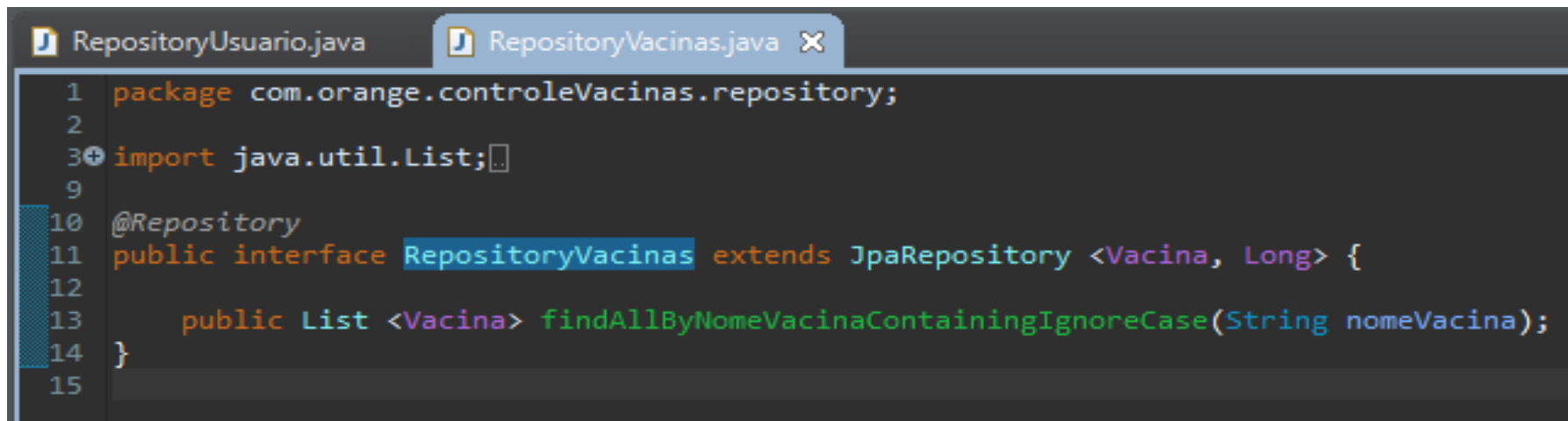
E como não utilizamos o Lombok, tivemos que gerar os Getter e Setter, para isso utilizamos um atalho com o botão direito do mouse e clicamos em source, Generate getters and setter.

Criando o Repository

Após a criação de nossas models, iremos criar o nosso repository, este pacote será responsável por nossa conexão com o banco de dados. Iremos criar um pacote e dentro dele duas interfaces, uma para nosso usuário e outra para nossa vacina.



Precisamos anotar a classe com `@Repository` para dizer que se trata de uma classe repositório, e também precisamos usar o `extends`, com ele dizemos que esta classe recebe as funcionalidades do JPA. Por exemplo: o método abaixo, busca por nome uma vacina no banco de dados. Temos vários outros métodos prontos, como `save` e `findAll` por exemplo.

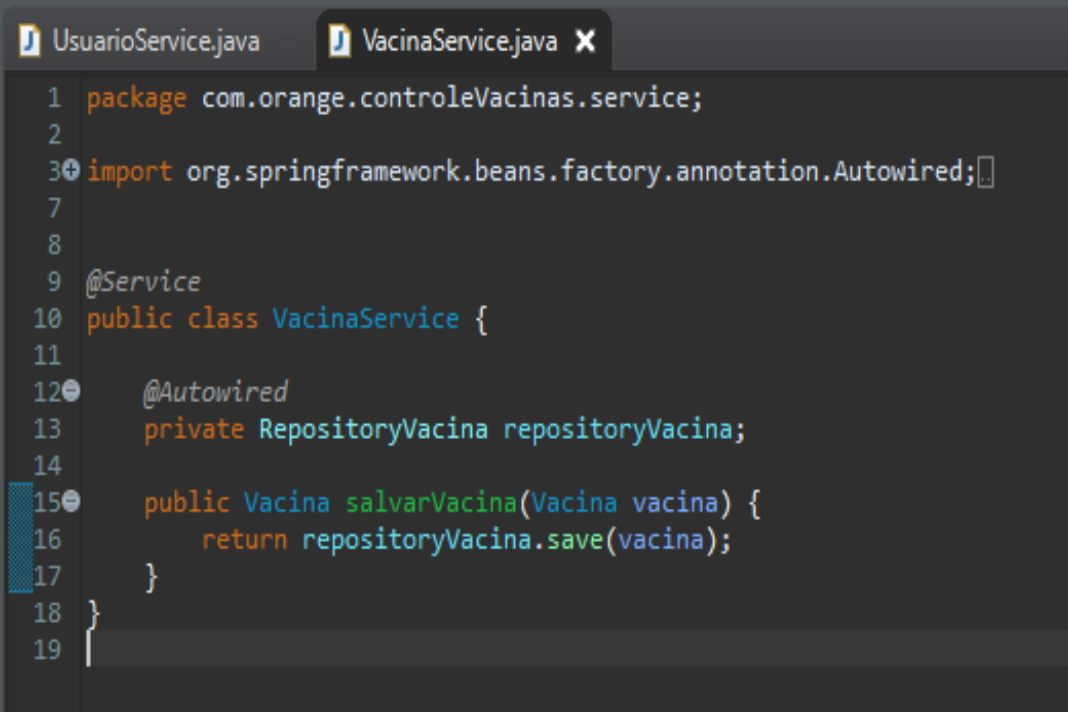
A screenshot of an IDE showing the code for `RepositoryVacinas.java`. The code defines a package, imports `java.util.List`, and creates a `@Repository` annotated interface `RepositoryVacinas` that extends `JpaRepository` for the `Vacina` entity. It includes a `findAllByNomeVacinaContainingIgnoreCase` method.

```
1 package com.orange.controleVacinas.repository;
2
3 import java.util.List;
4
5
6
7
8
9
10 @Repository
11 public interface RepositoryVacinas extends JpaRepository <Vacina, Long> {
12
13     public List <Vacina> findAllByNomeVacinaContainingIgnoreCase(String nomeVacina);
14 }
15
```

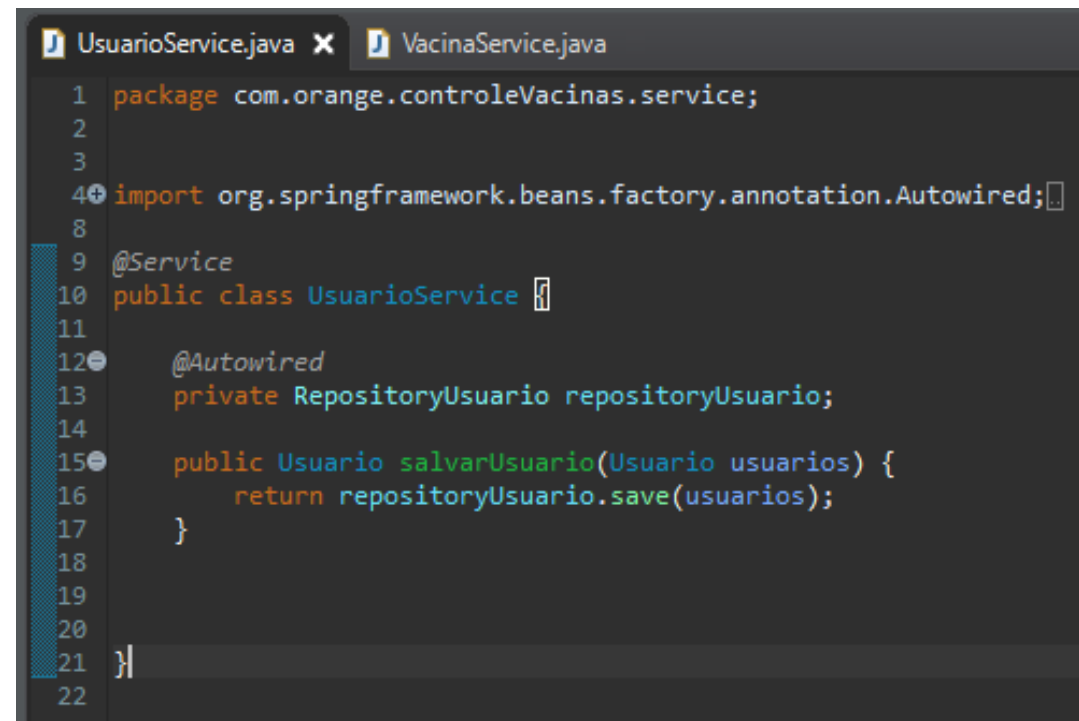
Camada Service

Chegamos à camada service, esta camada tem uma grande importância, pois ela será responsável por toda nossa regra de negócio, ou seja, toda a lógica de nossa API será feita nela. É necessário anotar a classe com `@Service`, é muito importante anotar todas as classes, para que o Spring saiba do que se trata cada classe.

Lembram-se que falamos sobre o repository e sua responsabilidade de conectar a API ao banco de dados? Pois bem, aqui iremos chamar nosso repository, para que possamos utilizar aqueles métodos do JPA que citamos anteriormente. Ao invés de instanciarmos o `RepositoryUsuario` sempre que fomos utilizar, vamos utilizar um “truque” do Spring. Ao anotarmos com `@Service`, “ganhamos” o `@Autowired`, esta anotação será responsável por instanciar o `RepositoryUsuario` sempre que formos utiliza-lo.



```
1 package com.orange.controleVacinas.service;
2
3 import org.springframework.beans.factory.annotation.Autowired;
4
5
6
7
8
9 @Service
10 public class VacinaService {
11
12     @Autowired
13     private RepositoryVacina repositoryVacina;
14
15     public Vacina salvarVacina(Vacina vacina) {
16         return repositoryVacina.save(vacina);
17     }
18 }
19
```



```
1 package com.orange.controleVacinas.service;
2
3
4 import org.springframework.beans.factory.annotation.Autowired;
5
6
7
8
9 @Service
10 public class UsuarioService {
11
12     @Autowired
13     private RepositoryUsuario repositoryUsuario;
14
15     public Usuario salvarUsuario(Usuario usuarios) {
16         return repositoryUsuario.save(usuarios);
17     }
18
19
20
21 }
22
```

Controller

Chegamos à reta final da nossa API, estamos na camada controller, esta camada é responsável por nossas requisições HTTP. É aqui que realizamos busca, criação, atualização e deletes. A primeira coisa que vamos fazer é criar dentro do pacote controller, uma classe que chamaremos de ControllerUsuario. Esta classe será anotada como @RestController. Vamos anotá-la também com @RequestMapping("/usuario"), esta anotação nos permite definir o caminho da URL. Por exemplo: Estamos utilizando nossa API localmente, então nossa URL será: <http://localhost:8080/usuario> O que passamos dentro de @RequestMapping será o final da url da nossa API para acessar aquele endpoint. Vamos começar instanciando o nosso usuarioService para que possamos utilizar os métodos que citamos anteriormente, lembrando sempre de anotar com @Autowired.

Fiz os seguintes métodos

POST - Agora vamos cadastrar um usuário, para isso precisamos criar um método Post, nosso método de criação. Vamos anotar nosso método com @PostMapping("/cadastrar"), a url será: <http://localhost:8080/usuario/cadastrar> assim conseguimos cadastrar um usuário, e fizemos o mesmo para a Vacina porém para cadastrar a vacina substituímos o usuário por /vacina, <http://localhost:8080/vacina/cadastrar>

Get - Temos 2 tipos de Get, GetAll buscamos todos os usuários, ou todas as vacinas e temos o GetById, buscamos passando um ID como parâmetro. Assim só busca aquele usuário ou vacina em que corresponde com o ID passado.

PUT - Este método atualiza o usuário ou vacina desejado, caso queira fazer alguma atualização de dados

Delete - Este método deleta o usuário ou vacina do nosso banco de dados, passando a não existir mais

```
ControllerUsuarios.java ControllerVacinas.java X
1 package com.orange.controleVacinas.controller;
2
3 import java.util.List;
4
22
23
24 @RestController
25 @RequestMapping("/vacina")
26 @CrossOrigin(origins = "*", allowedHeaders = "*")
27 public class ControllerVacinas {
28
29     @Autowired
30     private VacinaService vacinaService;
31     @Autowired
32     private RepositoryVacina repository;
33
34     @PostMapping("/cadastrar")
35     public ResponseEntity<Vacina>Post(@Valid @RequestBody Vacina vacina){
36         return ResponseEntity.status(HttpStatus.CREATED)
37             .body(vacinaService.salvarVacina(vacina));
38     }
39
40
41     @GetMapping
42     public ResponseEntity <List <Vacina>> GetAll(){
43         return ResponseEntity.ok (repository.findAll());
44     }
45
46
47     @GetMapping("/{id}")
48     public ResponseEntity <Vacina> GetById (@PathVariable Long id){
49         return repository.findById(id)
50             .map(resp -> ResponseEntity.ok(resp))
51             .orElse(ResponseEntity.notFound().build());
52
53     }
54 }
```

```
ControllerUsuarios.java ControllerVacinas.java X
30 private VacinaService vacinaService;
31 @Autowired
32 private RepositoryVacina repository;
33
34 @PostMapping("/cadastrar")
35 public ResponseEntity<Vacina>Post(@Valid @RequestBody Vacina vacina){
36     return ResponseEntity.status(HttpStatus.CREATED)
37         .body(vacinaService.salvarVacina(vacina));
38 }
39
40
41 @GetMapping
42 public ResponseEntity <List <Vacina>> GetAll(){
43     return ResponseEntity.ok (repository.findAll());
44 }
45
46
47 @GetMapping("/{id}")
48 public ResponseEntity <Vacina> GetById (@PathVariable Long id){
49     return repository.findById(id)
50         .map(resp -> ResponseEntity.ok(resp))
51         .orElse(ResponseEntity.notFound().build());
52 }
53
54
55 @PutMapping
56 public ResponseEntity<Vacina> put(@RequestBody Vacina vacinas){
57     return ResponseEntity.ok(repository.save(vacinas));
58 }
59
60 @DeleteMapping("/{id}")
61 public void delete (@PathVariable Long id) {
62     repository.deleteById(id);
63 }
64 }
65 }
```


Nossa API está pronta, agora vamos testa-la com o postman.

Criando um novo usuário:

The screenshot displays the Postman interface for a POST request to `http://localhost:8080/usuario/cadastrar`. The request body is a JSON object with the following fields: `nome`, `email`, `cpf`, and `dataNascimento`. The response status is `201 Created` with a time of `708 ms` and a size of `383 B`. The response body is a JSON object containing the created user's details, including an `id`.

Request:

- Method: POST
- URL: `http://localhost:8080/usuario/cadastrar`
- Body Type: raw (JSON)
- Body Content:

```
1 {  
2   "nome": "Daniel Ferreira2",  
3   "email": "daniel.ze.ferreira@hotmail.com",  
4   "cpf": "44715450870",  
5   "dataNascimento": "1997-07-17"  
6 }
```

Response:

- Status: 201 Created
- Time: 708 ms
- Size: 383 B
- Body Type: Pretty (JSON)
- Body Content:

```
1 {  
2   "id": 1,  
3   "nome": "Daniel Ferreira2",  
4   "email": "daniel.ze.ferreira@hotmail.com",  
5   "cpf": "44715450870",  
6   "dataNascimento": "1997-07-17"  
7 }
```


Criando uma nova vacina

Lembrando que a vacina esta relacionada com um usuário

POST

http://localhost:8080/vacina/cadastrar

Send

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

1

{

2

"nomeVacina": "covid19",

3

"dataRealizada": "2021-04-12",

4

"usuario":{

5

"id": 1

6

}

7

}

Body

Cookies

Headers (8)

Test Results

Status: 201 Created

Time: 97 ms

Size: 395 B

Save

Pretty

Raw

Preview

Visualize

JSON

1

{

2

"id": 1,

3

"nomeVacina": "covid19",

4

"usuario": {

5

"id": 1,

6

"nome": null,

7

"email": null,

8

"password": null,

Agora iremos fazer a busca dessa Vacina cadastrada já com os dados do usuário.

The screenshot shows a REST client interface with the following details:

- Method:** GET
- URL:** http://localhost:8080/vacina/1
- Buttons:** Params, Authorization, Headers (8), **Body**, Pre-request Script, Tests, Settings
- Body Type:** none, form-data, x-www-form-urlencoded, **raw**, binary, GraphQL, **JSON**
- Response Status:** 200 OK
- Response Time:** 71 ms
- Response Size:** 449 B
- Response Body (JSON):**

```
{
  "id": 1,
  "nomeVacina": "covid19",
  "usuario": {
    "id": 1,
    "nome": "Daniel Ferreira2",
    "email": "daniel.ze.ferreira@hotmail.com",
    "cpf": "44715450870",
    "dataNascimento": "1997-07-17"
  },
  "dataRealizada": "2021-04-12"
}
```

Aqui irei estar colocando um CPF não valido assim deve retornar erro 400.

POST

http://localhost:8080/usuario/cadastrar

Send

Save

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

Cookies

Co

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

Beautif

1

{

2

"nome": "Daniel Ferreira",

3

"email": "daniel.z.ferreira@hotmail.com",

4

"cpf": "44715450880",

5

"dataNascimento": "1997-07-17"

6

}

Body

Cookies

Headers (7)

Test Results

Status: 400 Bad Request

Time: 197 ms

Size: 6.13 KB

Save Response

Pretty

Raw

Preview

Visualize

JSON

1

{

2

"timestamp": "2021-04-13T17:17:52.237+00:00",

3

"status": 400,

4

"error": "Bad Request",

5

"trace": "org.springframework.web.bind.MethodArgumentNotValidException: Validation failed for argument [0] in public org.springframework.http.ResponseEntity<com.orange.controleVacinas.model.Usuario> com.orange.controleVacinas.controller.ControllerUsuarios.Post(com.orange.controleVacinas.model.Usuario): [Field error in object 'usuario' on field 'cpf': rejected value [44715450880]; codes [CPF.usuario.cpf,CPF.cpf,CPF.java.lang.String,CPF]; arguments

E assim finalizamos a API-Rest....

Link do projeto no GitHub:

https://github.com/danielferreira3/teste_OrangeTalents