Implementação da API para controle da vacinação

Nesse artigo sera examinado objetivamente a utilização de tecnologias e ferramentas no ecossistema Spring e o seus recursos para o desenvolvimento do sistema solicitado no desafio.

Mapeamento objeto-relacional: De classe a tabela

Tendo em vista a necessidade de armazenamento de informações no banco de dados, existe uma técnica que estabelece uma interface que promove uma compatiblidade entre o uma linguagem OO (o Java) e o um banco de dados relacional (nesse caso o MySQL); esse paradigma se chama o ORM, e o foco dessa seção será justamente contemplar o framework **Hibernate**.

Para a comunicação direta com o banco de dados, foi empregado o auxílio do JPA (acrônimo de Java Persistence API) - é uma especificação que oferece um conjunto de regras e comportamentos esperados a serem seguidos, é um template que sugere formas do framework ser integrado. O Hibernate por sua vez, é reponsável por mapear as tabelas de um banco de dados para uma classe da linguagem, definindo um denominador comun nas entidades. O conceito que promove esse tipo de framework é a capacidade de persistir, consultar, atualizar, apagar, realizar todo o CRUD sem a necessidade de lidar diretamente com SQL.

Persistência: Característica de determinados dados permanecer na memória mesmo após o estado que o criou se ausentar. Sem essa capacidade o estado só existiria na RAM e seria perdido quando a RAM parasse.

Para melhorar a didática da interação entre JPA e Hibernate, veja o JPA como uma interface e o Hibernate como a classe dessa implementação.

Inicializando o projeto

Dependências selecionadas para o desenvolvimento da proposta

1. Spring Boot DevTools
   Esse módulo tem o objetivo de otimizar o tempo do desenvolvimento; O Spring BootDevTools aplica as mudanças do código reiniciando a aplicação a cada save

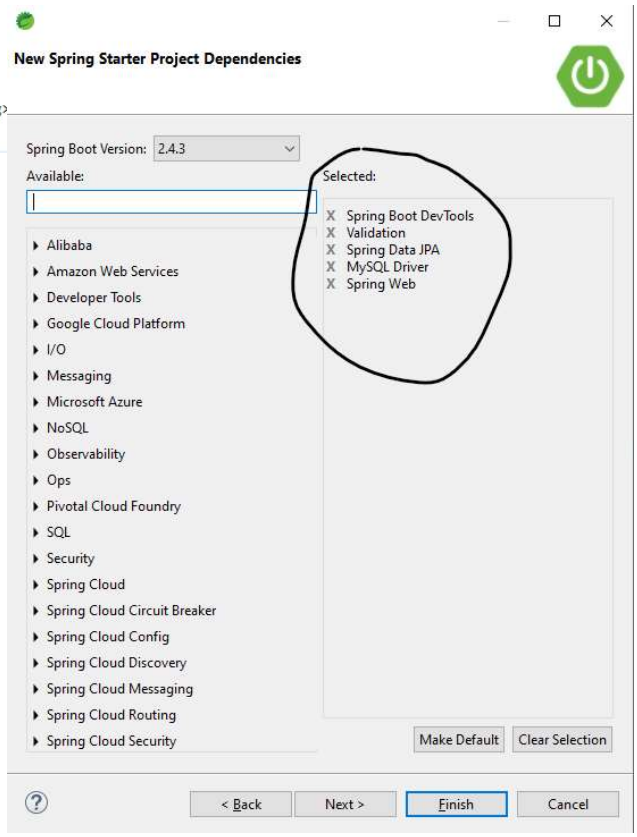
2. Validation
   É uma especificação que disponibliza um conjunto de restrições; A anotação @NotNull, por exemplo proibe campos nulos atendendo à obrigaridade exigida. (Veja a linha 26 do código abaixo)

```
18 @Entity
19 @Table(name = "vacinas")
20 public class VacModel {
21
22     @Id
23     @GeneratedValue(strategy = GenerationType.IDENTITY)
24     private Long id;
25
26     @NotNull
27     @Column
28     private String name;
```

3. Spring Data JPA
   Esse framework parte da grande família Spring, torna mais facil a implementação dos repository em JPA, além de que auxilia na comunicação com o banco de dados, traduzindo os comandos do Java para os comandos do mySQL.

/

```java
package com.zup.orangeTalent.repository;

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

public interface VacRepository extends JpaRepository<VacModel, Long>

}
```

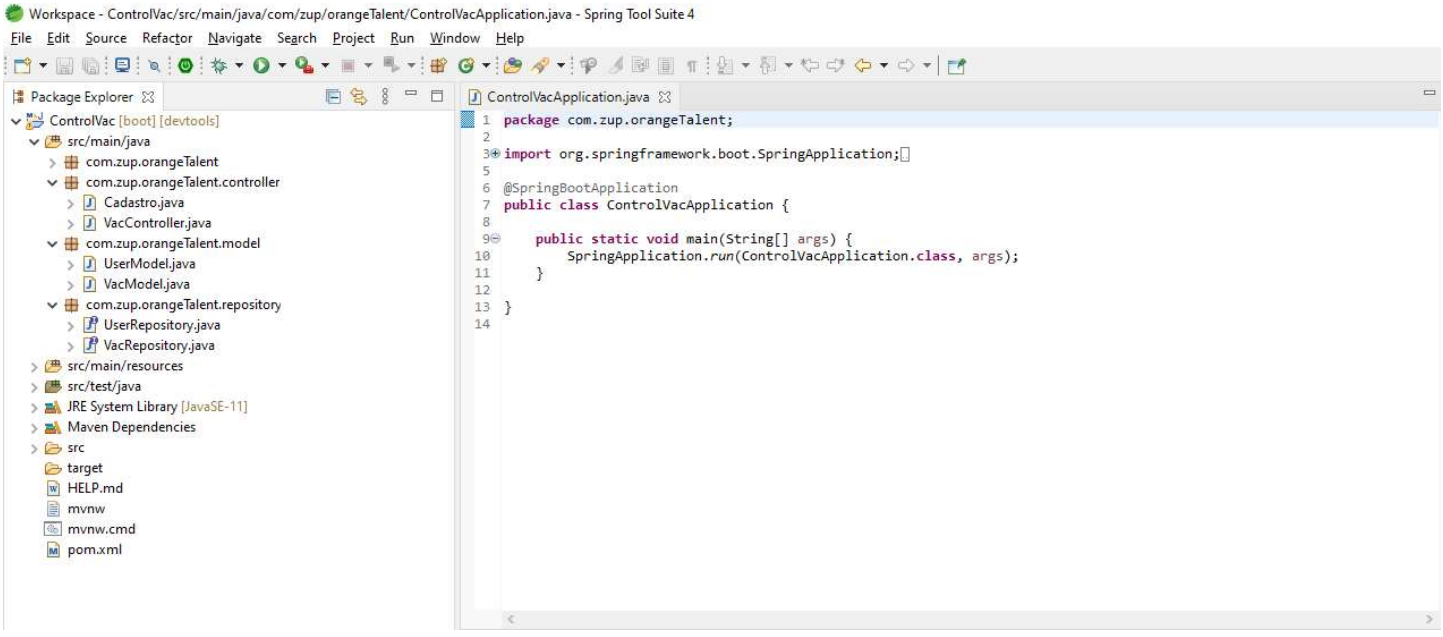


4. MySQL Driver
   Promove a conectividade o MySQL; utilizado para verificar a entrada dos dados.

5. Spring Web
   Essa dependência foi incluído para deixar o sistema "na agulha" para futursas implementação na web

- ## Padrão arquitetural das classes (em camadas)

Árvore dos pacotes com suas respectivas classes/interfaces e a classe primordial que executa a aplicação



Camada dos Repositórios

1. *Armazenamento para a tabela de vacinação*

```java
public interface VacRepository extends JpaRepository <VacModel, Long> { }
```

O repositório foi definido dessa forma como uma interface, que herda JpaRepository, e é parametrizada como argumento a classe da entidade(VacModel) e o tipo de dado Long; uma classe fornecida pela dependência Spring Data JPA (explicada na seção acima) que proporciona uma interação direta com o banco de dados

2. *Armazenamento para a tabela dos usuários*

```
@Repository
public interface UserRepository extends JpaRepository<UserModel, Long> { }
```

Mesma lógica do repositório de vacinação, alterando somente a classe da entidade para UserModel

Camada das Model

Camada que repercute por todo o sistema, aqui é modelado o objeto e seu respectivo relacionamento que permeia as outras entidades do banco de dados

1. *Classe de usuários*

No código abaixo começamos ver a importância do JPA + a bean Validation em uma abordagem mais prática: fornece propriedades desejaveis aos atributos, como @NotNull indicando obrigatoriedade e nas @Columns é possível exigir somente valores únicos

Da linha 39 à linha 40 é estabelecida uma Foreign Key

```
1  package com.zup.orangeTalent.model;
2
3  import java.util.List;
14
15 @Entity
16 @Table(name = "usuarios")
17 public class UserModel {
18
19     @Id
20     @GeneratedValue(strategy = GenerationType.IDENTITY)
21     public Long id;
22
23     @NotNull
24     private String nome;
25
26     @Column(unique = true)
27     @NotNull
28     private String email;
29
30     @Column(unique = true)
31     @NotNull
32     private String cpf;
33
34     @NotNull
35     private String birthday;
36
37
38     //Estabelecendo a chave estrangeira para cada aplicação da vacina:
39     @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL)
40     private List<VacModel> vacinacao;
41
42
43     //Getters and Setters
44
45     public String getNome() {
46         return nome;
47     }
48
49     public Long getId() {
50         return id;
51     }
52
```

2. *Classe de vacinação*

Nessa outra classe abaixo, é utilizado algumas novas anotações... @Temporal também é um propriedade do JPA, e nessa aplicação ele registra o exato momento em que o registro da entrada no banco de dados. (Obs: é utilizado dois atributos para marcar a data, da linha 37 é ajustavel.)

Da linha 41 à 43, o código atende o desafio bônus e passa a relacionar o id do usuário substituindo o email.

Da linha 39 à linha 40 é estabelecida uma Foreign Key

```java
package com.zup.orangeTalent.model;

import java.util.Date;

@Entity
@Table(name = "vacinas")
public class VacModel {

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

    @NotNull
    @Column
    private String name;

                .

    @Column
    @Temporal(TemporalType.TIMESTAMP)
    private Date dataInput = new java.sql.Date(System.currentTimeMillis());

    @NotNull
    private String data;

                                        .
    //Aderindo ao id do usuário em detrimento do email
    @ManyToOne
    @JsonIgnoreProperties("vacinacao")
    private UserModel usuario;


    //Getters and Setters


    public String getName() {
        return name;
    }

    public Long getId() {
        return id;
    }
```

Camada dos Controller

É nesse momento que os endpoints serão construídos

Cadastros

Na imagem abaixo que exibe as linha do código, podemos ver a anotação @RestContoller logo no ínicio, a anotação que índica que a classe se tratar de um controlador dos endpoints.
@RequestMaping define a rota de acesso (na linha 18) e na próxima linha vemos a anotação @CrossOrigin que designa que a API pode ser acessada em qualquer plataforma

Há duas anotações do @Autowired (injeção de dependência) para cada repositório de registro. Esse recurso permite trabalhar com os respectivos repositório com um baixo nível de acomplamento, o que resulta em uma certa independência das classes, e previne o projeto de possíveis problemas.

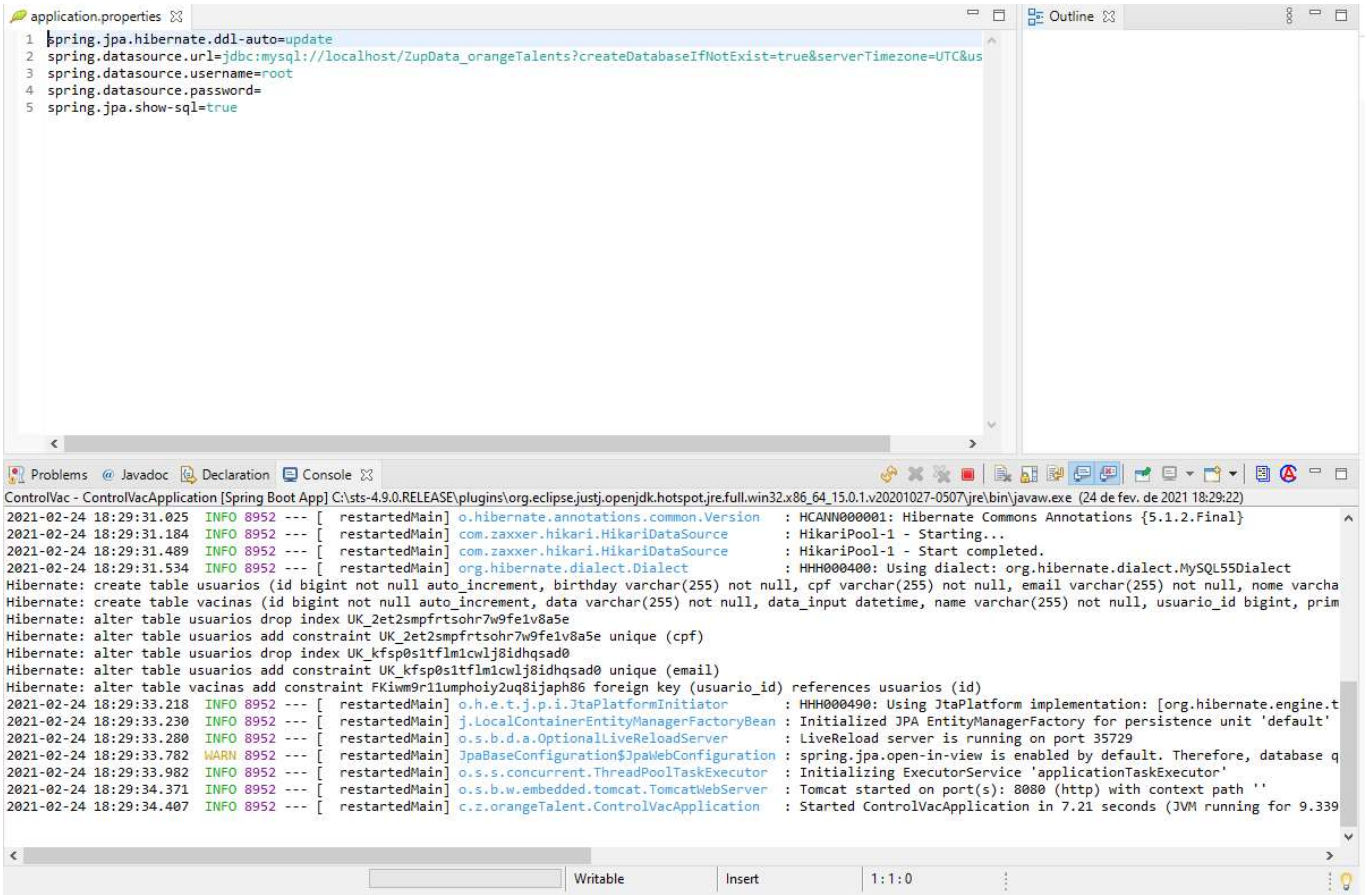
A anotação @PostMapping vai tentar fazer a persistência dos dados, e vai conseguir se todas as regras for atendida, criando um novo registro no banco de dados.
A anotação @RequestBody espera os dados no corpo da requisição para que seja então desserializado na construção do objeto e então sendo persistido nos registros através da comunicação com o banco de dados.

O try/catch faz uma especie de tratamento para que haja somente os status esperado (201 e 400)

```java
package com.zup.orangeTalent.controller;

import org.springframework.beans.factory.annotation.Autowired;

@RestController
@RequestMapping("/cadastro")
@CrossOrigin("*")
public class Cadastro {

    @Autowired
    private VacRepository aplyRepository;

    @Autowired
    private UserRepository userRepository;

    @PostMapping("/user")
    public ResponseEntity<UserModel> registro(@RequestBody UserModel usuario) {
        try {
            return ResponseEntity.status(HttpStatus.CREATED).body(userRepository.save(usuario));
        }
        catch (Exception e) {
            return ResponseEntity.badRequest().build();

        }

    }

    @PostMapping("/apply-vac")
    public ResponseEntity<VacModel> cadastro(@RequestBody VacModel vacina) {
        try {
            return ResponseEntity.status(HttpStatus.CREATED).body(aplyRepository.save(vacina));
        }
        catch (Exception er) {
            return ResponseEntity.badRequest().build();

        }
    }
}
```

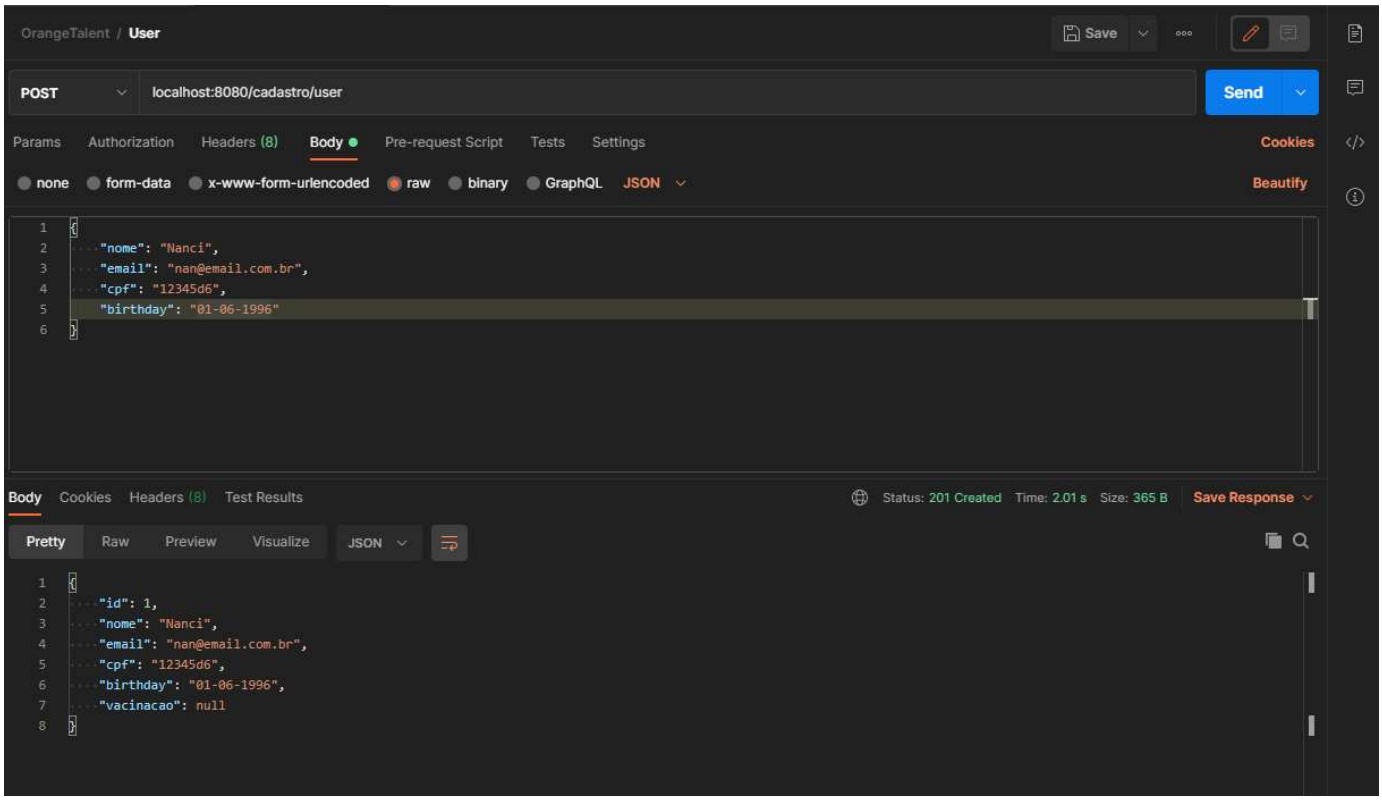
- # Comprovando em tempo de execução

Então configurando o **application.properties** conforme a ilustração, já é garantido o banco de dados.

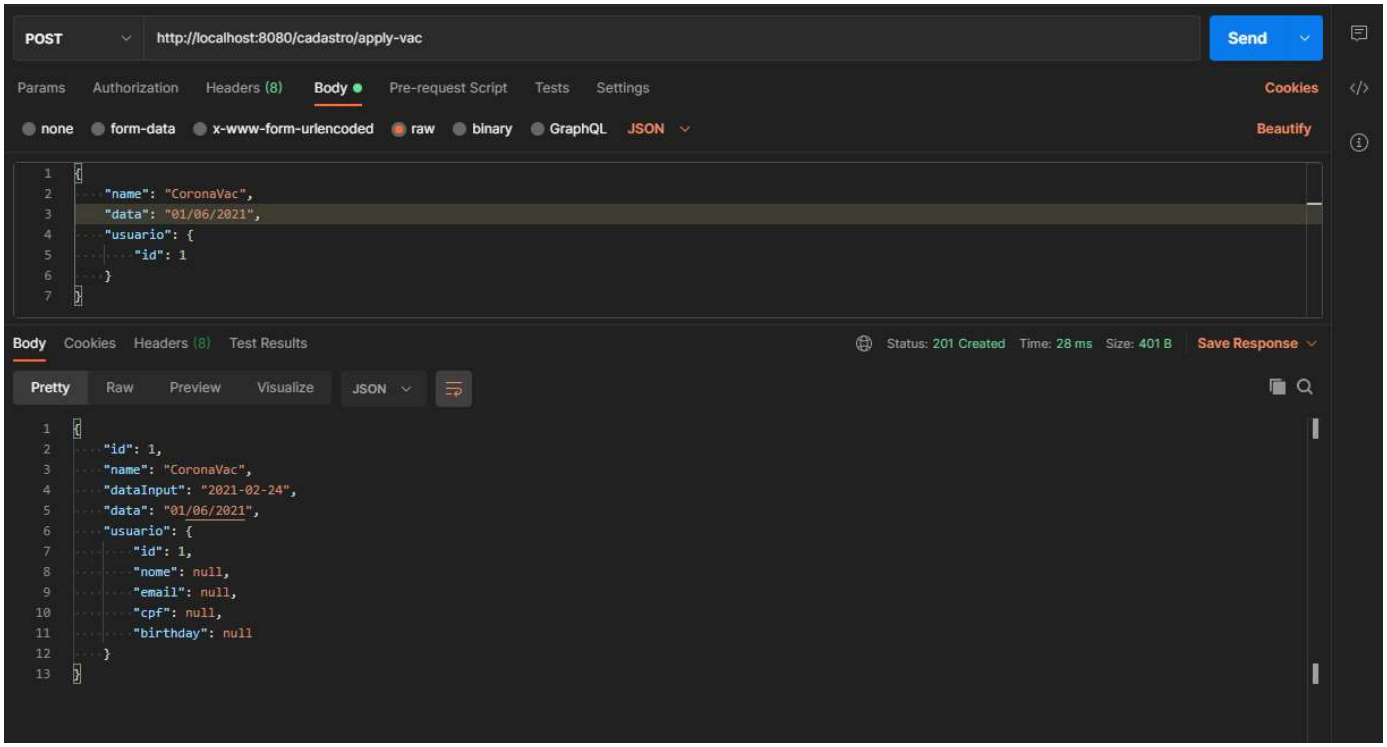


Com a aplicação rodando é efetuado o cadastro do primeiro usuário

/



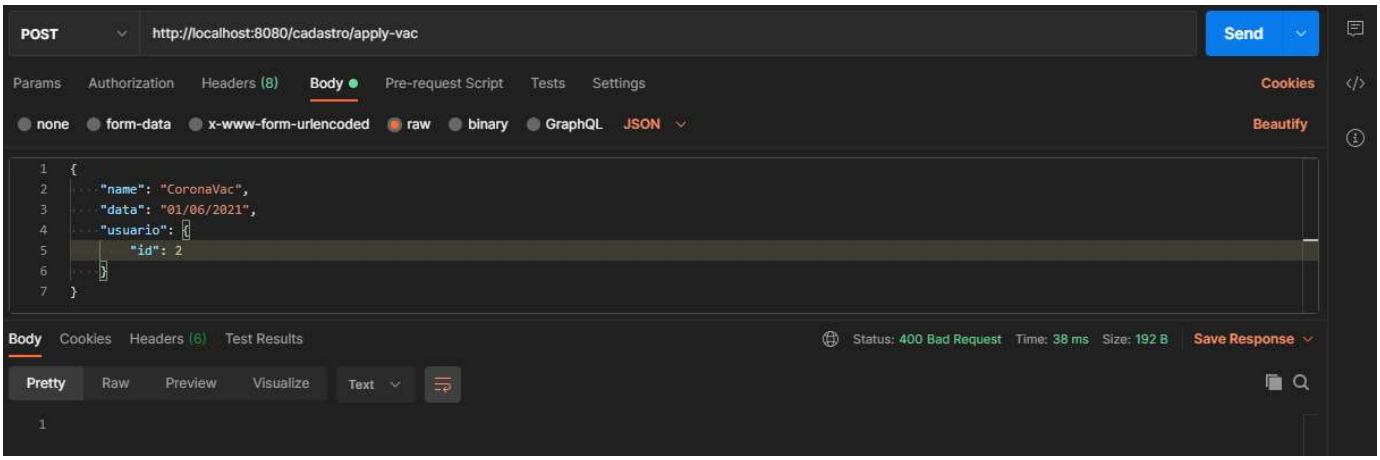
Passando para o registro da aplicação da vacína nessa usuário é efetuado o registro no banco de dados



br>

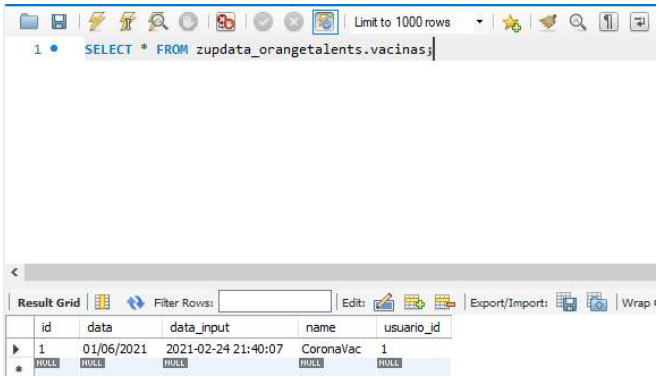
Porém se for indicado um id de usuário inexistente (o que pode ser facilmente impedido no front end) ele retorna o erro 400!

/



br>

E a imagem do banco de dados abaixo exibe o registro da aplicação da vacina.



- ## Como faria a implementação do sistema na Web

  Provavelmente seria embasado no wireframe abaixo (desenvolvido no figma) com o emprego do framework Angular, criando uma single-page; os caminho no service dos component seria indicado para o devido endpoint, e então com o projeto finalizado executaria uma build e teria uma relação de arquivos para submeter no public do projeto do Backend para que possa hospedar a aplicação no Heroku, por exemplo, e ver o seu comportamento em tempo real.