Desafio Zup Orange Talents

Contexto:

1-Você está fazendo uma API REST que precisará controlar a aplicação de vacinas entre a população brasileira. O primeiro passo deve ser a construção de um cadastro de usuários, sendo obrigatórios dados como: nome, e-mail, CPF e data de nascimento, onde e-mail e CPF devem ser únicos.
2-O segundo passo é criar um cadastro de aplicação de vacinas, sendo obrigatórios dados como: nome da vacina, e-mail do usuário e a data que foi realizada a vacina.
3-Você deve construir apenas dois endpoints neste sistema, o cadastro do usuário e o cadastro da aplicação da vacina. Caso os cadastros estejam corretos, é necessário voltar o Status 201, caso haja erros de preenchimento de dados, o Status deve ser 400.

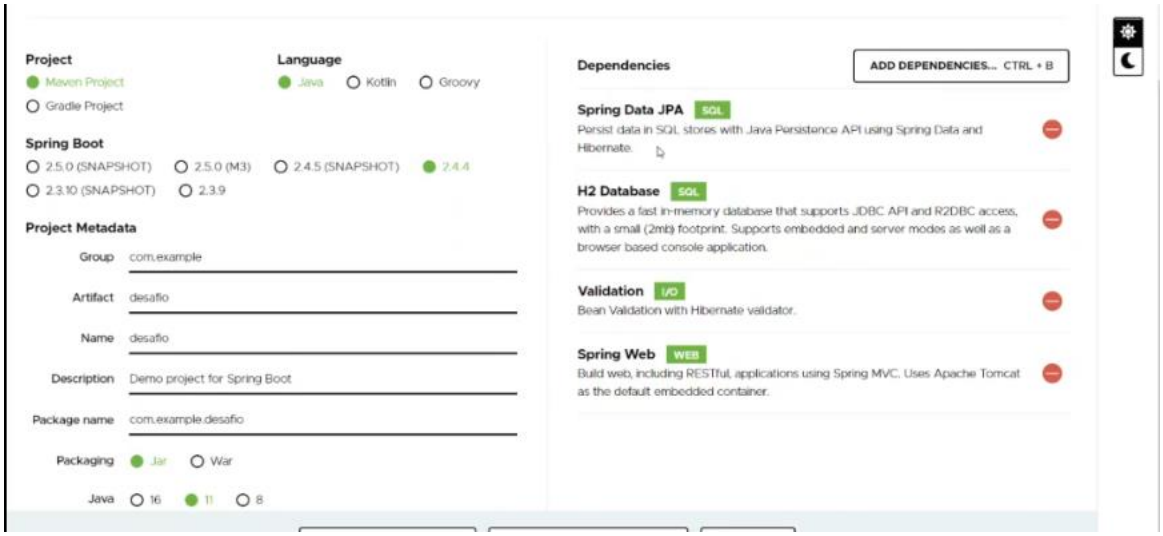
Se ficou fácil, te desafiamos a:

Construir a aplicação sem utilizar Lombok;

Substituir o e-mail do usuário na tabela de vacinação para uma chave estrangeira associada ao id do usuário

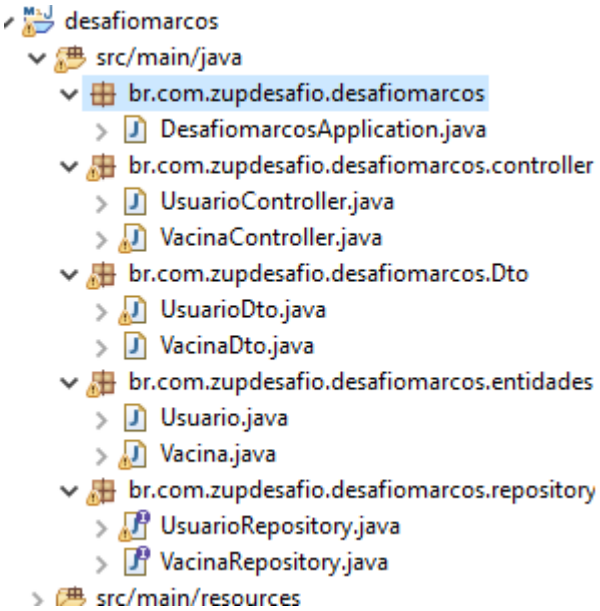
Quais tecnologias Spring usaremos?

Uma dúvida que pode surgir para qualquer profissional quando vai criar um novo projeto é quais tecnologias usar e quais dependências durante o início do projeto podem ser necessárias. Vamos **utilizar o Eclipse como nossa IDE** por ser prática e muito fácil de usar e o **Spring initializr** (https://start.spring.io/) para iniciarmos nosso projeto e escolhermos nossas dependências. Tendo em mente que nosso projeto tem uma boa complexidade, iniciaremos escolhendo um projeto **Maven** (uma ferramenta incrivel quando a questão é gerenciamento de projetos e dependências), com **Spring Data JPA** (por ser uma dependência que junto com o **Hibernate** nos ajuda a manter dados persistentes em um banco conforme nossa proposta, o JPA é essencial pois nos ajudará a mapear nossas entidades), o **H2 Database** (esse vai ser o nosso banco, devido a sua alta praticidade e facilidade de manuseio irá nos permitir testar nossa aplicação), **Bean Validation** (para que possamos validar objetos em diferentes camadas da aplicação, breve mostrarei como vou usar isso a nosso favor nas camadas) e **Spring Web** (para que possamos visualizar dados de um server remoto de forma fácil). Preencheremos os campos da aba Project Metadata e em seguida no botão Generate, está criada a base da nossa aplicação, basta importar agora para o Eclipse.



Quais pacotes, classes e camadas usaremos na implementação no nosso projeto?

Antes de explicar a fundo como funciona a aplicação precisamos deixar claro que o modelo de organização foi escolhido tenta visar boas práticas em programação separando devidamente ao máximo a responsabilidade atribuída a cada linha código.



Explicando de forma básica e rápida nossos pacotes e modelo escolhido para nosso sistema:

O pacote **br.com.zupdesafio.desafiomarcos** traz a aplicação principal junto com o método **main** é o nosso ponto inicial (brevemente vou mostrar com mais detalhes o código de todas os pacotes e classes).

O pacote **br.com.zupdesafio.desafiomarcos.controller** está relacionado com os controladores(regras) e também nossa validação e endpoint, além de se relacionar com o nosso pacote dto (Objeto de Transferência de Dados).

O pacote **br.com.zupdesafio.desafiomarcos.Dto** traz regras claras para a entrada de dados que aceitaremos e estaremos inserindo em nosso banco de forma mais segura e eficaz.

O pacote **br.com.zupdesafio.desafiomarcos.entidades** traz as classes Usuário e Vacina, nas quais seus atributos moldam todos os tipos de dados nas outras camadas e são a base do sistema após o método main.

O pacote **br.com.zupdesafio.desafiomarcos.repository** é responsável pelo repositório (a parte do banco de dados) e age junto ao **JPA** para trazer informações persistentes na aplicação.

A aplicação conta com 2 Classes básicas **Usuario e Vacina**, as demais são responsáveis por definir regras de **fluxo, tratamento e validação** e da entrada e registro das classes até o banco de dados **H2 Database.** Com conhecimento destas informações vamos pôr em prática!

Vamos começar!

Passo 1:

Após importar seu projeto e visualizar as classes e modelos adotados, vamos criar e detalhar a classe **Usuario**.

```
2  package br.com.zupdesafio.desafiomarcos.entidades;
3
4  import com.fasterxml.jackson.annotation.JsonFormat;
10
11 @Entity
12 public class Usuario {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16  private Long id;
17  private String nome;
18
19 @Column(unique = true)
20  private String email;
21
22 @Column(unique = true)
23  private String cpf;
24
25 @JsonFormat(pattern = "dd/MM/yyyy")
26  private LocalDate dataDeNascimento;
27
```

```
33     public Usuario(Long id) {
34         this.id = id;
35     }
36
37     public Usuario(String nome, String email, String cpf, LocalDate dataDeNascimento) {
38         this.nome = nome;
39         this.email = email.toLowerCase(Locale.ROOT);
40         this.cpf = cpf;
41         this.dataDeNascimento = dataDeNascimento;
42     }
43
44     public Long getId() {
45     return id;
46 }
47 public String getNome() {
48     return nome;
49 }
50 public String getEmail() {
51     return email;
52 }
53 public String getCpf() {
54     return cpf;
55 }
```

Vamos criar a classe e modelar seus atributos, id, nome, email, cpf e dataDeNascimento, note que, marcamos a classe com o @Entity (para dizermos que é uma entidade e seus objetos devem persistir no banco), usamos a anotação @Id e @GeneratedValue (strategy = GerationType.IDENTITY) com isso dizemos que o campo id é um identificador no banco de dados (**Primary Key**) e que esse campo deve ser auto incrementado na coluna. Note também o @Column que define que os dados dos campos email e cpf devem ser únicos ao entrar no banco de dados (como solicitado no desafio), @Jsonformat para passarmos o formato de data para o "dd/MM/yyyy".

Outro detalhe é a linha 39 na qual passamos o **toLowerCase** no email para assegurar que todo email será minúsculo e único no banco de dados e continuaremos nossa Classe criando construtor e getters para recebermos informações. Uma observação importante também, lembre-se de importar as dependências conforme o necessário enquanto você cria @anotações e usa dependências, pois sem o **import** não conseguirá executar os trechos de código que necessitam de dependências, um dos motivos pelo qual recomendo o **Eclipse** foi esse evitar dores de cabeça sempre que for possível, ele é uma IDE muito prática e auxiliará nessas tarefas.

Passo 2:

Agora que conseguimos modelar nossos dados do Usuario, vamos para a camada responsável por acessar esses dados no banco de dados! Vamos criar o pacote **br.com.zupdesafio.desafiomarcos.repository** e a Classe **UsuarioRepository** :

```
1  package br.com.zupdesafio.desafiomarcos.repository;
2
3  import org.springframework.data.jpa.repository.Query;
4  import org.springframework.data.repository.CrudRepository;
5
6  import br.com.zupdesafio.desafiomarcos.entidades.Usuario;
7  import org.springframework.stereotype.Repository;
8
9  @Repository
10 public interface UsuarioRepository extends CrudRepository<Usuario, Long> {
11
12     boolean existsByEmail(String email);
13     boolean existsByCpf(String cpf);
14
15 }
16
```

Por estarmos tratando agora da camada responsável pelo banco de dados, como se pode observar na linha 3 a 7 necessitaremos importar algumas dependências relacionadas ao **JPA** e **CrudRepository** e a própria classe Usuario, usaremos a anotação @Repository que define nossa classe UsuarioRepository como um repositório propriamente dito, afim de termos acesso ao banco de dados, outro fato importante é estender a mesma ao CrudRepository e também utilizei o recurso do spring que cria uma query JPQL por baixo dos panos quando utilizado um padrão de nome predefinido na documentação nos métodos, como exemplo o código da linha 12 e 13.

Passo 3:

O Projeto está tomando forma! Que tal tentarmos manter ao máximo a segurança? Vamos criar o pacote **br.com.zupdesafio.desafiomarcos.Dto**, a classe **UsuarioDto** e ditar algumas regras de entrada de dados em nosso banco.

```java
package br.com.zupdesafio.desafiomarcos.Dto;

import br.com.zupdesafio.desafiomarcos.entidades.Usuario;
import com.fasterxml.jackson.annotation.JsonFormat;
import org.hibernate.validator.constraints.br.CPF;

import javax.validation.constraints.Email;
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.PastOrPresent;
import java.time.LocalDate;

public class UsuarioDto {

    @NotBlank
    private String nome;

    @NotBlank
    @Email
    private String email;

    //@CPF
    @NotBlank
    private String cpf;

    @PastOrPresent
    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate dataDeNascimento;

    public String getNome() {
        return nome;
    }

    public String getEmail() {
        return email;
    }

    public String getCpf() {
        return cpf;
    }

    public LocalDate getDataDeNascimento() {
        return dataDeNascimento;
    }
```

Importe as dependências e a classe Usuario também, aqui usaremos bastante o **Bean validation**, primeiro vamos usar a anotação @NotBlank isso evitará que esse campo nome seja ignorado e tenhamos um cadastro de usuario sem nome em branco ou nulo, o mesmo ao campo email porém usando a anotação @Email também para que tenhamos um email com formatação válida e também temos a opção @CPF para que nossos cpfs sejam todos válidos e outra anotação interessante é a @PastOrPresent para definirmos que um Usuario nasceu ou está nascendo hoje, seria estranho que um usuário seja cadastrado em nosso banco e ainda nem tenha nascido, essa anotação define isso como regra. Não vamos esquecer de passar getters para todos os atributos da classe UsuarioDto para que o spring consiga utilizar os dados do request e os dados não fiquem como null e vamos usar esse método também, para que possamos instanciar um novo usuário.

```java
public Usuario transformarParaUsuario() {

    return new Usuario(nome, email, cpf, dataDeNascimento);
}
```

Passo 4:

Agora com as regras do dto definidas, vamos criar nosso pacote **br.com.zupdesafio.desafiomarcos.**

controller e também nossa classe **UsuarioController**, aqui definiremos nosso endpoint.

```
3  import br.com.zupdesafio.desafiomarcos.Dto.UsuarioDto;
4  import org.springframework.beans.factory.annotation.Autowired;
5  import org.springframework.http.HttpStatus;
6  import org.springframework.http.ResponseEntity;
7  import org.springframework.web.bind.annotation.*;
8
9  import br.com.zupdesafio.desafiomarcos.entidades.Usuario;
10 import br.com.zupdesafio.desafiomarcos.repository.UsuarioRepository;
11 import org.springframework.web.server.ResponseStatusException;
12 import org.springframework.web.util.UriComponentsBuilder;
13
14 import javax.validation.Valid;
15 import java.net.URI;
16 import java.util.Locale;
17
```

```
1  package br.com.zupdesafio.desafiomarcos.controller;
2
3  import br.com.zupdesafio.desafiomarcos.Dto.UsuarioDto;
17
18 @RestController
19
20 public class UsuarioController{
21
22     @Autowired
23     private UsuarioRepository usuarioRepository;
24
25 @PostMapping("/usuario")
26  public ResponseEntity<UsuarioDto> cadastarUsuario (@RequestBody @Valid UsuarioDto usuarioDto, UriComponentsBuilder
27
28     if(usuarioRepository.existsByEmail(usuarioDto.getEmail().toLowerCase(Locale.ROOT)) || usuarioRepository.existsE
29         throw new  ResponseStatusException(HttpStatus.BAD_REQUEST, "Dados invalidos");
30     }
31
32     Usuario usuario = usuarioDto.transformarParaUsuario();
33     usuarioRepository.save(usuario);
34     URI uri = uriBuilder.path("usuario/{id}").buildAndExpand(usuario.getId()).toUri();
35     return ResponseEntity.created(uri).body(usuarioDto);
36 }
```

Vamos começar importando do nosso pacote **br.com.zupdesafio.desafiomarcos.Dto** nossa classe **UsuarioDto** e mais algumas dependências relacionadas a **Spring Web** .

Note que temos uma nova anotação é a @RestController que define nossa classe como contralador que é responsável por controlar o fluxo da aplicação. Observe também a anotação @Autowired é responsável pela injeção de dependências/Classes estaremos usando ela na nosso **UsuarioRepository** e também fique de olho na anotação @PostMapping ela é de alta importância pois nos permite passar uma URI (nosso endpoint no caso) e como estamos terminando de relacionar a classe Usuario nada mais justo do que mapear de "/usuario" com esse uri poderemos abrir nosso link no **Postman** e junto passar a uri "/usuario" e observarmos se quando inserirmos valores eles realmente chegaram ao nosso banco(**Passo 9**).

Explicando um pouco sobre a linha 26 criamos uma função na qual retorna um ResponseEntity para que possamos retornar o código 201 e nosso header URI do nosso novo recurso criado. A anotação @RequestBody foi usada para que possamos atribuir os dados enviados pelo nosso cliente via JSON para nosso usuarioDto. O UriBuilder é utilizado para que possamos criar nossa URI e passar como parâmetro no retorno do nosso método no ResponseEntity.

Observação na linha 28 onde verificamos se existe um email já cadastrado no banco de dados igual ao informado no nosso cliente da api, se já tivermos um email existente então retornamos um ResponseStatusException que nos permite enviar um código HTTP, nesse caso 400 e uma mensagem de erro.

Na linha 32 convertemos nosso **usuarioDto** para um **usuario**, dessa forma temos nosso usuario será persistido no banco de dados.

Ufa que trabalho!!!

Com pacotes que moldaram as classes relacionadas ao **Usuario** esclarecidas, vamos falar da segunda parte, a outra entidade que demonstra muita importância em nosso sistema a Vacina.

Passo 5:

Vamos criar agora uma sequência de passos muito similar a tudo que fizemos e anotamos para criar a classe Usuario anteriormente, mas nos mínimos detalhes estaremos trazendo o foco.

```
1  package br.com.zupdesafio.desafiomarcos.entidades;
2
3  import com.fasterxml.jackson.annotation.JsonFormat;
10
11 @Entity
12 public class Vacina {
13
14     @Id
15     @GeneratedValue(strategy = GenerationType.IDENTITY)
16     private Long id;
17
18     private String nome;
19
20     @ManyToOne
21     private Usuario usuario;
22
23     @JsonFormat(pattern = "dd/MM/yyyy")
24     private LocalDate dataDaVacina;
25
26     public Vacina(String nome, Usuario usuario, LocalDate dataDaVacina) {
27         this.nome = nome;
28         this.usuario = usuario;
29         this.dataDaVacina = dataDaVacina;
```

No pacote **entidades**, criaremos a classe **Vacina**, ela possui a mesma base da classe **Usuario**, porém seus atributos são: id, nome, dataDaVacina e usuario, (usaremos esse atributo usuario para referenciar uma chave estrangeira em relação ao que foi proposto no desafio do blog).

A estrutura do código e das classes é bem similar mas temos uma nova anotação @ManyToOne (muitos para um, ela faz referência ao usuario para que ele possa ser uma chave estrangeira na tabela vacina e ligar as duas tabelas no banco de dados). Não se esqueça do Construtor e getters.

Passo 6:

Agora vamos criar a classe **VacinaRepository** no pacote **br.com.zupdesafio.desafiomarcos.repository**

```java
package br.com.zupdesafio.desafiomarcos.repository;

import br.com.zupdesafio.desafiomarcos.entidades.Usuario;

@Repository
public interface VacinaRepository extends CrudRepository<Vacina, Long> {

	@Query("Select u From Usuario u WHERE u.id = :id")
	Optional<Usuario>  buscarUsuario(Long id);
}
```

Bem similar ao que foi feito antes mas também com uma nova anotação temos a @Querry na qual passamos uma query JPQL (bem similar ao SQL mesmo) e nela solicitamos que ela selecione um usuario no banco de dados.

Passo 7:

Agora vamos fazer com a classe Vacina assim como fizemos com a classe Usuario e criar uma classe chamada **VacinaDto** no pacote **br.com.zupdesafio.desafiomarcos.dto** para mantermos a segurança da aplicação e as regras.

```java
    private String nome;

    @NotNull
    private Usuario usuario;

    @JsonFormat(pattern = "dd/MM/yyyy")
    @NotNull
    private LocalDate dataDaVacina;

    public String getNome() {
        return nome;
    }

    public Long getUsuario() {
        return usuario.getId();
    }

    public LocalDate getDataDaVacina() {
        return dataDaVacina;
    }

    public Vacina toModel() {
        return new Vacina(nome, usuario, dataDaVacina);
```

A base da estrutura do código é bem similar, criando e limitando cada campo e como bonus estamos usando o toModel() afim de converter nossa **vacinaDto** para um objeto do tipo **vacina.**

Passo 8:

Estamos quase acabando! Vamos criar agora nossa classe VacinaController no pacote **br.com.zupdesafio.desafiomarcos.controller.**

```
3  import br.com.zupdesafio.desafiomarcos.Dto.VacinaDto;
18
19 @RestController
20 public class VacinaController {
21
22     @Autowired
23     private VacinaRepository vacinaRepository;
24
25     @PostMapping("/vacina")
26     public ResponseEntity<VacinaDto> cadastrarVacina(@RequestBody @Valid VacinaDto vacinaDto, UriComponentsBuilder
27
28         Optional usuario = vacinaRepository.buscarUsuario(vacinaDto.getUsuario());
29
30         if(usuario.isEmpty()) {
31             throw new ResponseStatusException(HttpStatus.BAD_REQUEST, "Dados invalidos");
32         }
33
34         Vacina vacina = vacinaDto.toModel();
35         vacinaRepository.save(vacina);
36         URI uri = uriBuilder.path("/vacina/{id}").buildAndExpand(vacina.getId()).toUri();
37         return  ResponseEntity.created(uri).body(vacinaDto);
38     }
```

Basicamente é o mesmo conceito que já vimos na classe Usuario sendo reaplicados na classe Vacina e aqui também temos nosso outro **endpoint** (o "/vacina" lembre-se dele é muito importante cria-lo para que possamos fazer testes no nosso banco de dados, **Passo 9**), porém com um diferencial na linha 34 no qual transformamos nosso objeto **vacinaDto** em um objeto do tipo **vacina** para poder persistir no banco de dados.

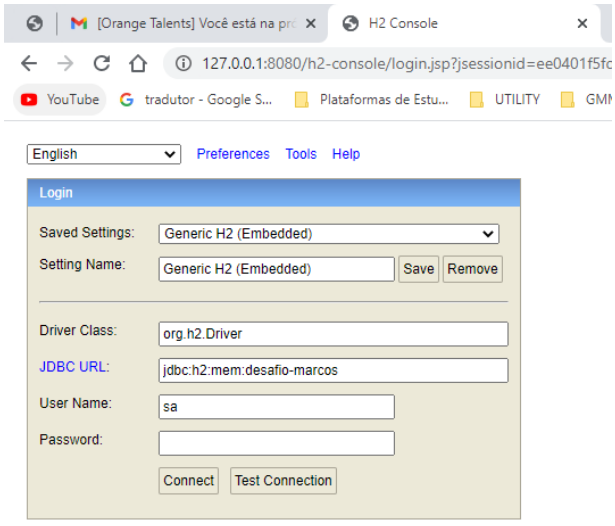
Na linha 28 utilizamos o Optional que pode receber ou não um usuario da nossa query. Depois utilizamos uma função presente nesse optional para verificar se o usuario está vazio, se estiver respondemos com a exceção novamente.

Passo 9:

Concluindo o processo de implementarmos duas classes precisamos testar e ver os resultados, então mãos-a-obra!

Vamos ver o que realmente resultou, rodando sua aplicação agora, você verá que o terminal está ok, para se conectar ao banco de dados utilize o endereço a seguir no seu browser:

127.0.0.1:8080/h2-console/



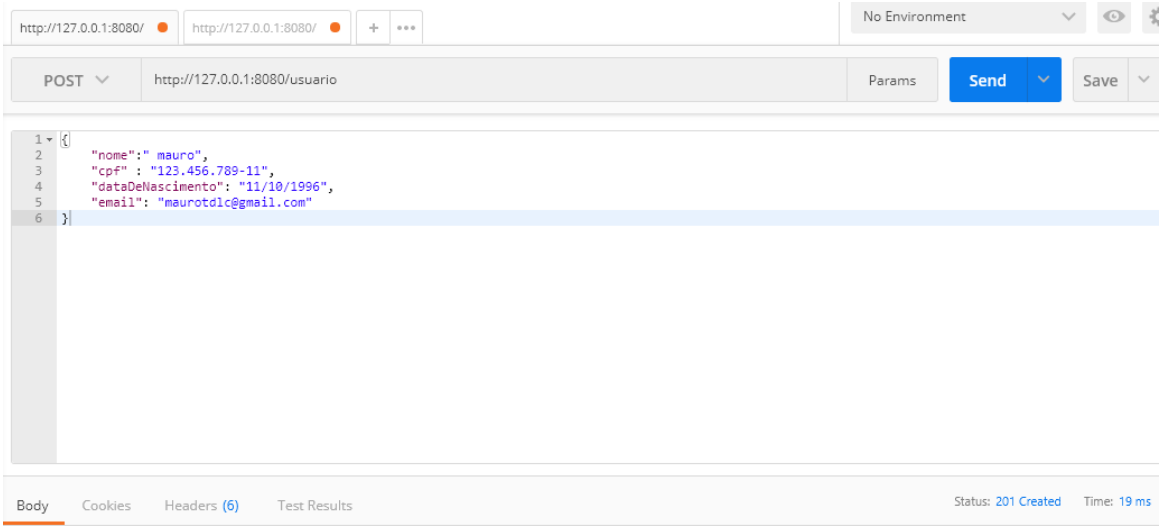
Vamos usar o **Postman** para inserir valores com método POST através de JSON em nosso banco de dados.

Abra o **Postman** tenha em vista que o método POST deve ser selecionado;

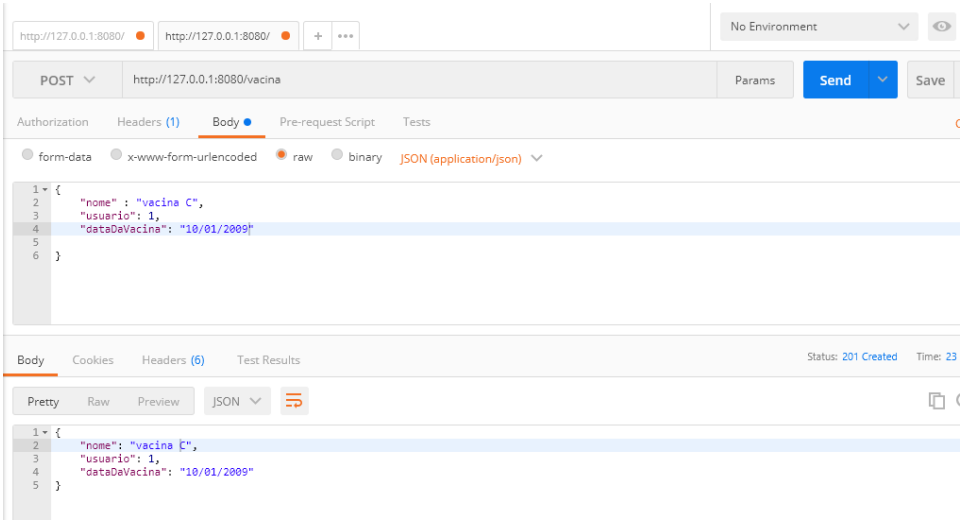
Logue no endereço de nosso banco "127.0.0.1:8080" e acrescente "/URI" no nosso caso "/usuario": "127.0.0.1:8080/usuario";

Vá na aba "Headers" e marque uma caixa, escolha a opção Key: Content-type, em Value complete com aplication/json (conforme a foto abaixo):

Agora escolha a opção ao lado: "Body" escolha a bolha radial com a opção "raw" e insira os valores do JSON no campo abaixo, uma vez inseridos use o botão Send e observe os status e o tempo, conforme o exemplo:

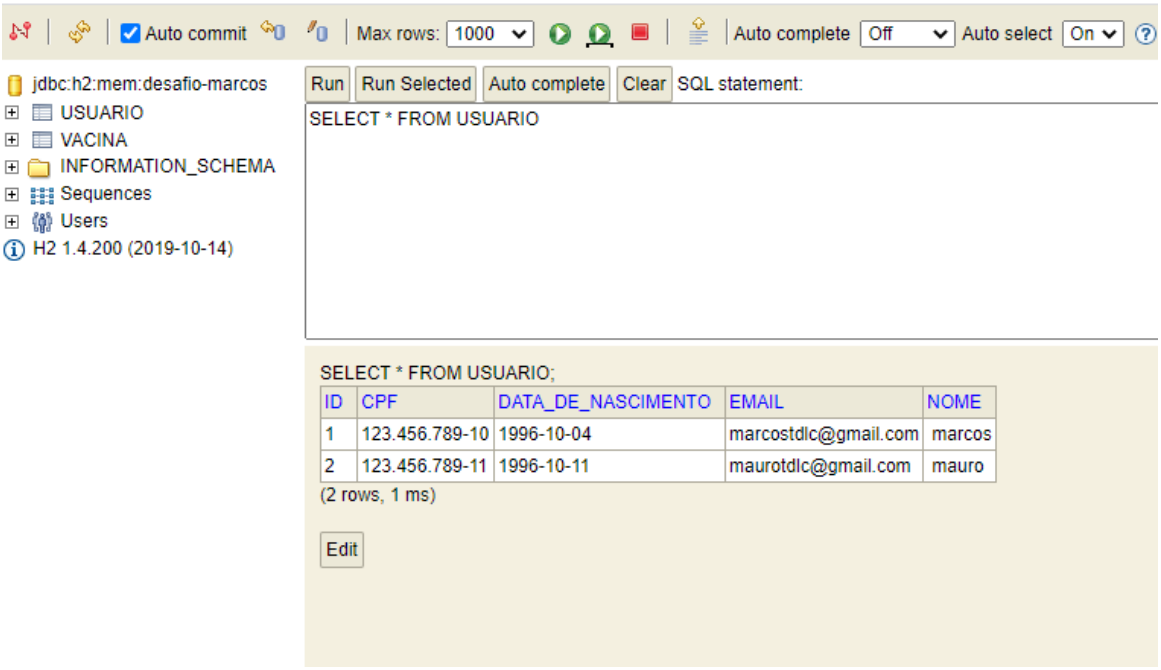


Como solicitado no desafio conseguimos o Status 201, está tudo correto com nossa aplicação! O mesmo aconteceu quando criamos uma nova aba no **Postman** e fizemos com o endpoint da classe Vacina:



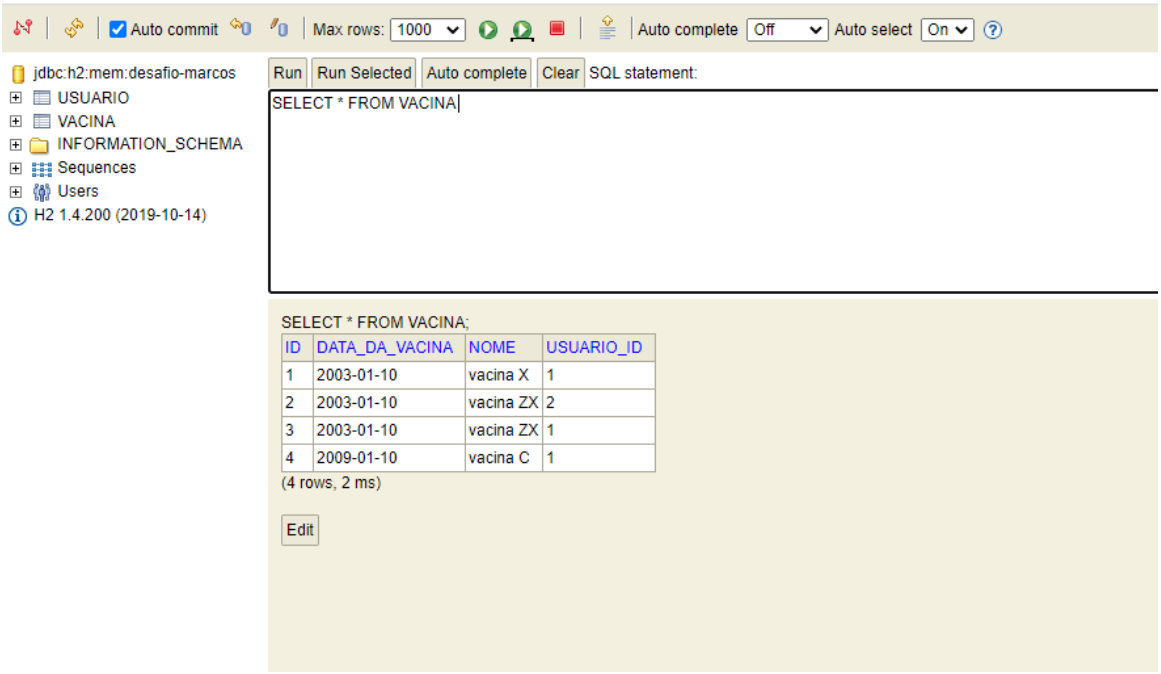
Uau que incrivel! Vamos conferir o nosso banco de dados:

Logando no banco e clicando na tabela Usuario e passando uma simples query e run podemos observar:



Inserindo 2 usuarios para teste observamos que, fomos capazes de criar o id que auto incrementa, o cpf está formatado também, conseguimos adicionar data de nascimento, email e o nome para nossos usuários.

Passando uma querry e observando a tabela Vacina:



Fomos capazes de criar um id para cada vacina que se auto incrementa, um nome da vacina, data da vacina e referenciar qual usuario tomou qual vacina de acordo com a chave estrangeira, no caso o usuario_id (1) tomou 3 vacinas enquanto o usuario_id (2) apenas 1 vacina.

Ótimo, funcionou perfeitamente, espero que tenha gostado e aprendido com esse nosso projetinho até uma próxima!!!