API REST com Spring Boot

Implementando uma API com Java e Spring Boot.

09/05/2021 por Edilson Alves

O Java é uma linguagem de programação muito poderosa se utilizada corretamente. Trabalhando com ela orientado a objeto e utilizando as várias tecnologias disponíveis, pode se criar inúmeras coisas. E fica ainda mais poderosa tendo o Spring Boot como ferramenta.

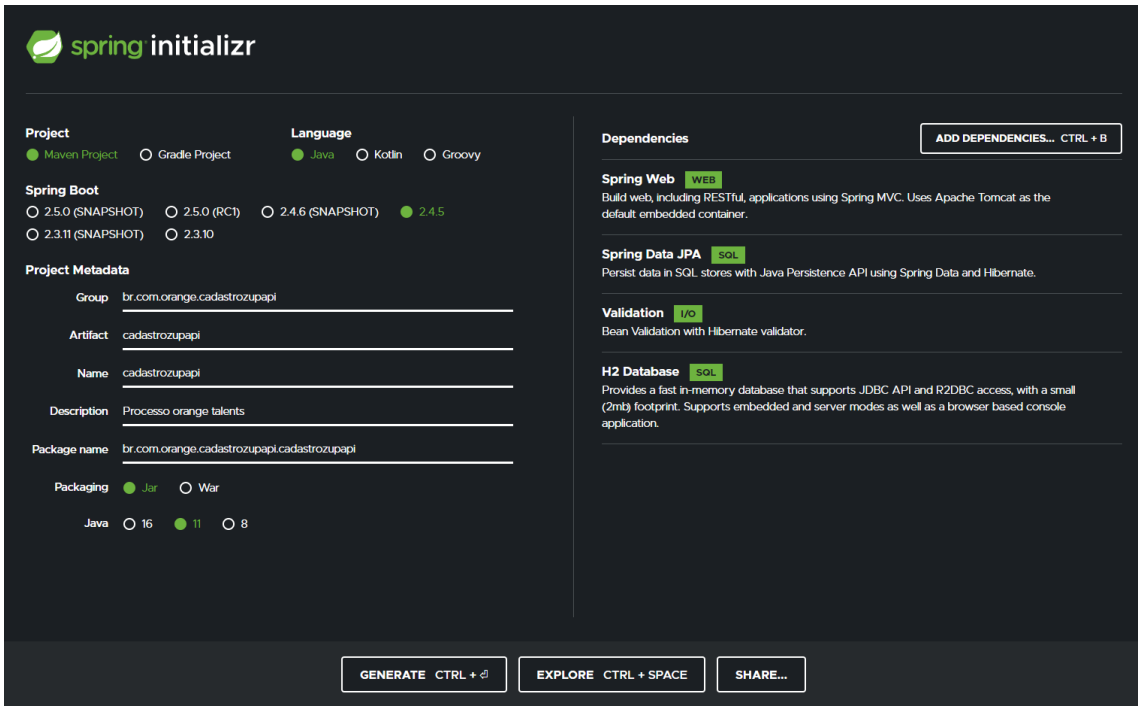
O Spring Boot é uma ferramenta ou de certa forma uma parte do Framework Spring. O Spring tem como um dos seus princípios a injeção de dependências. E para marcar os pontos de injeção da sua classe, basta utilizar a anotação @Autowired.

O terror de muitos programadores ao criar uma API com Java, era configurar tudo do zero. Para facilitar essa parte, surgiu o Spring Boot, facilitando o processo de iniciar um projeto, rodando rápido e sem complicações.

Com o site Spring Initializr você escolhe seu gerenciador de dependências (eu geralmente opto pelo Maven), assim como todas as dependências que deseja utilizar no projeto e a partir daí já estamos prontos para começar. Caso queira adicionar mais módulos, basta adicioná-los no arquivo pom.xml que vem no projeto.

Agora vou explicar com mais detalhes como eu criaria uma API REST utilizando o Spring. Essa API precisa suporta um cadastro de clientes, sendo necessário possuir os seguintes dados: Nome, E-mail, CPF e data de nascimento. Também iremos implementar o cadastro de endereços por cliente.

Tendo o ambiente preparado para trabalhar, basta acessar o site start.spring.io escolher suas configurações básicas e começar. Para esse projeto eu escolhi o Maven, linguagem Java11, Spring Boot versão 2.4.5. no Metadata escolhi utilizar uma convenção e nomeei o meu grupo br.com.orange.cadastrozupapi, pois o nome do meu projeto é Cadastro Zup APi



Já na parte de dependências adicionei: Spring web, Spring Data JPA,Validation e H2 Database.

Com o **Spring web** conseguimos criar uma aplicação web ou API REST no modelo MVC. Utiliza ainda o Apach Tomcat como o webservice.

Persistiremos os dados no bando de dados com a facilidade do **Hibernate** que vem dentro do **Spring Data JPA**.

O Bean Validation é um auxiliador de validações dos nossos dados.

E o H2Databse que nos fornece um banco de dados na memória rápido que suporta modos integrados e de servidor.

Extraí o arquivo baixado do Spring Initializr em uma pasta e abri com o Eclipse. Já de inicio, dentro da pasta src/main/resource abri o arquivo application.properties e adicionei algumas linhas para a configuração da

nossa aplicação e banco de dados:

```
spring.datasource.url=jdbc:h2:mem:projetodb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=
spring.jpa.database-platform=org.hibernate.dialect.H2Dialect

spring.h2.console.enabled=true
spring.h2.console.path=/h2-console

spring.jpa.show-sql=true
spring.jpa.properties.hibernate.format_sql=true
spring.jpa.hibernate.ddl-auto=update


server.error.include-message=always
```

Com isso já conseguimos testar se nossa aplicação está funcionando, abrindo o terminal da ide e digitando mvnm install, o Maven instala e atualiza as mudanças do nosso código. Após isso com

Java -jar target/file.jar tendo assim a nossa aplicação acessível pelo navegador ou insomnia.

Bom agora vamos ao que interessa: construir nosso código.

Criei cinco pacotes dentro de src/main/java, sendo elas clientes, controllers, dtos, models e repositories.

Criei a classe Usuario dentro do pacote models para formatar nossa tabela que vai ser adicionada ao bando de dados. A classe contendo as informações da tabela e suas validações ficou assim.

```java
package br.com.orange.cadastroenderecozupapi.models;

import javax.persistence.*;
import java.time.LocalDate;

@Entity
public class Usuario {

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

    @Column(unique = true)
    private String email;

    @Column(unique = true, length = 11)
    private String cpf;
    private LocalDate dataNascimento;

    public Usuario(String nome, String email, String cpf, LocalDate dataNascimento) {
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }

    @Deprecated
    public Usuario() {
    }


    public String getNome() {
        return this.nome;
    }
}

```

Percebemos que por meio de anotações conseguimos eliminar códigos clichês do nosso projeto e adicionamos funcionalidades e validações de maneira fácil e assertiva. Nosso ID com a anotação @Id e @GerenatedValue passou a ser considerado um id e tendo suas configurações já ajustadas.

Adicionei mais validações em todas os outros atributos, como mostrado acima, tendo assim uma aplicação mais segura e com relação de dados inconsistente.

Com a anotação @Column(unique = true) adicionada em e-mail e CPF, garantimos que não existirá dados duplicados nesses campos e sendo obrigatório 11 caracteres no CPF.

Na pasta controllers criei a classe UsuarioController, responsável pelas rotas dos endpoints e a chamada das suas respectivas funções.

Adicionei a anotação @RestController no topo da classe para indicar que o mapa model dessa classe vai ser voltada para uma API REST e assim retornar ao cliente os dados como bodyparser (JSON no caso).

Deixei a classe bem enxuta adicionei apenas dos IFs para manda mensagem para o usuário caso um E-mail ou CPF já tenha sido cadastrado.

```java
package br.com.orange.cadastroenderecozupapi.controllers;

import br.com.orange.cadastroenderecozupapi.dtos.NovoUsuarioRequest;
import br.com.orange.cadastroenderecozupapi.models.Usuario;
import br.com.orange.cadastroenderecozupapi.repositories.UsuarioRepository;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import javax.validation.Valid;

@RestController
@RequestMapping("api/usuarios")
public class UsuarioController {

    @Autowired
    private UsuarioRepository usuarioRepository;

    @PostMapping
    public ResponseEntity<?> cadastro(@RequestBody @Valid NovoUsuarioRequest request) {
        //if para verificar se o email já está cadastrado
        if (usuarioRepository.existsByEmail(request.getEmail())){
            return ResponseEntity.status(400).body("Email já cadastrado");
        }
        //if para verificar se o email já está cadastrado
        if(usuarioRepository.existsByCpf(request.getCpf())){
            return ResponseEntity.status(400).body("CPF já cadastrado");
        }

        Usuario novoUsuario = request.toUsuario();

        usuarioRepository.save(novoUsuario);

        return ResponseEntity.status(201).body("Usuário cadastrado");
    }
}

```

configurando-o assim para o método HTTP POST, pois iremos inserir/criar informações na aplicação. Esse método recebe ainda uma variável do tipo NovoUsuarioRequest, um DTO. Com a anotação @Valid para informar que essas informações devem ser validadas e @RequestBody para deixar explícito que queremos os dados do corpo da requisição e via JSON.

Nosso DTO NovoUsuarioRequest Ficou assim:

```java
package br.com.orange.cadastroenderecozupapi.dtos;

import org.hibernate.validator.constraints.br.CPF;

public class NovoUsuarioRequest {

    @NotBlank
    private String nome;
    @NotBlank
    @Email
    private String email;
    @NotBlank
    @CPF
    private String cpf;
    @NotNull
    private LocalDate dataNascimento;

    public String getNome() {
        return nome;
    }

    public String getEmail() {
        return email;
    }

    public String getCpf() {
        return cpf;
    }

    public LocalDate getDataNascimento() {
        return dataNascimento;
    }


    public Usuario toUsuario() {
        return new Usuario(nome, email, cpf, dataNascimento);
    }
}

```

Declarei todos meus atributos private e fiz algumas validações como @NotBlank que não deixa o campo ser vazio @Email e @CPF que valida como tal. Depois criei os Getters de cada atributo publico para ter acesso na minha classe UsuarioController.

Para o extra iremos cadastrar endereços por clientes, sendo assim irei utilizar uma API externa. E iremos utilizar o Spring cloud feign , então

adicionei essa dependência no arquivo pom.xml

```
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-dependencies</artifactId>
                <version>2020.0.1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
        </dependencies>
    </dependencyManagement>

    <build>
        <plugins>
            <plugin>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-maven-plugin</artifactId>
            </plugin>
        </plugins>
    </build>
```

O acesso é feito por uma interface. Então criei uma interface no pacote clientes chamada ViaCepClient e utilizei a anotação @FeignClient para informar que essa interface é dessa nova dependência que adicionei.

Vou também dar um nome e informar uma URL, chamei ela de ViaCep mesmo e a URL ficou assim http://viacep.com.br/ws/{CEP}/json/

Onde o cep entre chaves será uma variável que será passada pra esse API pra ela fazer a busca do endereço pra min.

Vamos utilizar o método Get e criar um response que será a resposta que essa API vai nos dar.

E nossa interface ficará assim:

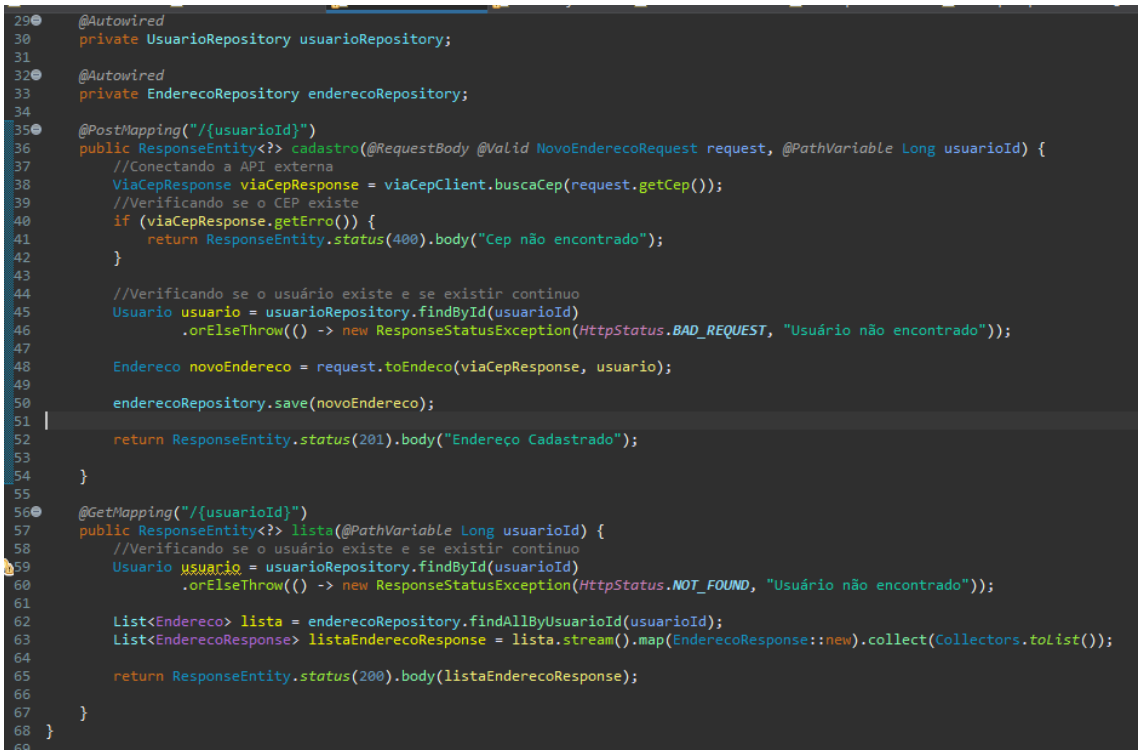
```
package br.com.orange.cadastroenderecozupapi.clients;

import org.springframework.cloud.openfeign.FeignClient;

@FeignClient(name = "ViaCep", url = "https://viacep.com.br/ws")
public interface ViaCepClient {

    @GetMapping("/{cep}/json/")
    ViaCepResponse buscaCep(@PathVariable String cep);
}

```

Vamos agora criar uma classe chamada ViaCepResponse que vai representar os dados que estamos recebendo. Da mesma forma que criamos o UsuarioRequest , terá as variáveis e os getters e ficará assim:

```java
package br.com.orange.cadastroenderecozupapi.clients;

public class ViaCepResponse {

    private String cep;
    private String logradouro;
    private String complemento;
    private String bairro;
    private String localidade;
    private String uf;
    private Boolean erro = false;

    public Boolean getErro() {
        return erro;
    }

    public String getCep() {
        return cep;
    }

    public String getLogradouro() {
        return logradouro;
    }

    public String getComplemento() {
        return complemento;
    }

    public String getBairro() {
        return bairro;
    }

    public String getLocalidade() {
        return localidade;
    }

    public String getUf() {
        return uf;
    }
}
```

Agora teremos uma nova classe que chamei de EnderecoController que vai trabalhar da mesma forma que o UsarioController porém com uma diferença que vou especificar ele pelo um id de um usuário já cadastrado e passaremos também algumas mensagens amigáveis caso haja erro ao cadastrar o novo endereço.

E nossa classe ficara assim:



```java
    @Autowired
    private UsuarioRepository usuarioRepository;

    @Autowired
    private EnderecoRepository enderecoRepository;

    @PostMapping("/{usuarioId}")
    public ResponseEntity<?> cadastro(@RequestBody @Valid NovoEnderecoRequest request, @PathVariable Long usuarioId) {
        //Conectando a API externa
        ViaCepResponse viaCepResponse = viaCepClient.buscaCep(request.getCep());
        //Verificando se o CEP existe
        if (viaCepResponse.getErro()) {
            return ResponseEntity.status(400).body("Cep não encontrado");
        }

        //Verificando se o usuário existe e se existir continuo
        Usuario usuario = usuarioRepository.findById(usuarioId)
                .orElseThrow(() -> new ResponseStatusException(HttpStatus.BAD_REQUEST, "Usuário não encontrado"));

        Endereco novoEndereco = request.toEndeco(viaCepResponse, usuario);

        enderecoRepository.save(novoEndereco);

        return ResponseEntity.status(201).body("Endereço Cadastrado");

    }

    @GetMapping("/{usuarioId}")
    public ResponseEntity<?> lista(@PathVariable Long usuarioId) {
        //Verificando se o usuário existe e se existir continuo
        Usuario usuario = usuarioRepository.findById(usuarioId)
                .orElseThrow(() -> new ResponseStatusException(HttpStatus.NOT_FOUND, "Usuário não encontrado"));

        List<Endereco> lista = enderecoRepository.findAllByUsuarioId(usuarioId);
        List<EnderecoResponse> listaEnderecoResponse = lista.stream().map(EnderecoResponse::new).collect(Collectors.toList());

        return ResponseEntity.status(200).body(listaEnderecoResponse);

    }
}
```

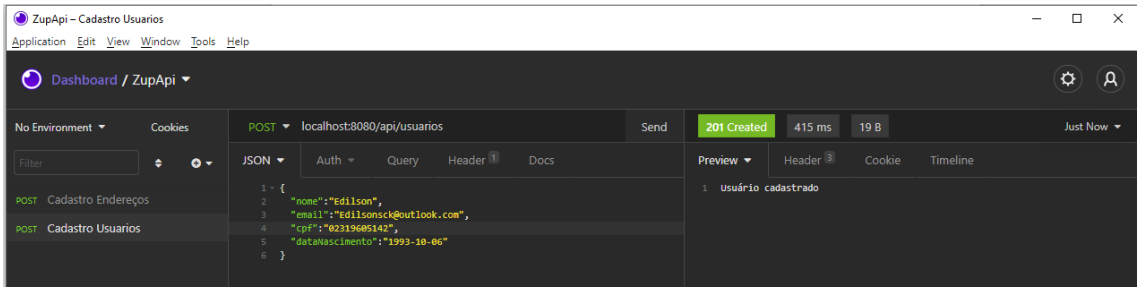
O DTO que passei como NovoEnderecoRequest que são os dados que vamos pegar do usuário e queremos apenas cep, número e complemento

Essa nova classe criada ficará assim:

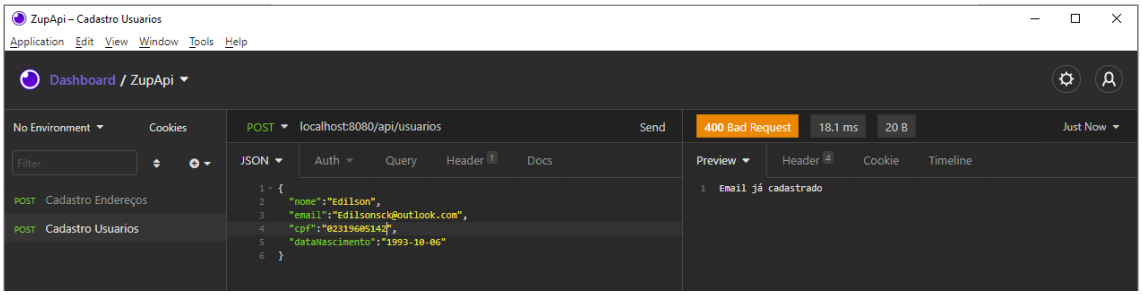
```java
package br.com.orange.cadastroenderecozupapi.dtos;

import br.com.orange.cadastroenderecozupapi.clients.ViaCepResponse;

public class NovoEnderecoRequest {

    @NotBlank
    @Pattern(regexp = "[0-9]{8}")
    private String cep;
    @NotBlank
    private String numero;
    private String complemento;

    public String getCep() {
        return cep;
    }

    public String getNumero() {
        return numero;
    }

    public String getComplemento() {
        return complemento;
    }

    public Endereco toEndeco(ViaCepResponse cepResponse, Usuario usuario) {
        return new Endereco(cepResponse.getCep(), cepResponse.getLogradouro(), numero, cepResponse.getComplemento(),
                complemento,
                cepResponse.getBairro(), cepResponse.getLocalidade(), cepResponse.getUf(), usuario);
    }
}
```

Executamos agora nosso projeto e faremos os testes.

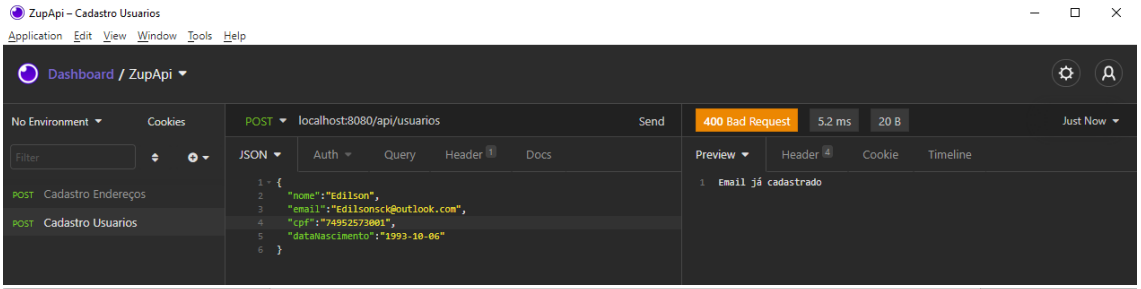
Pelo programa insomnia eu vou criar dois método POST chamados Cadastro Usuário e Cadastro Endereço vou passar texto do tipo JSON pra ele e clicar em send o teste ficara assim:

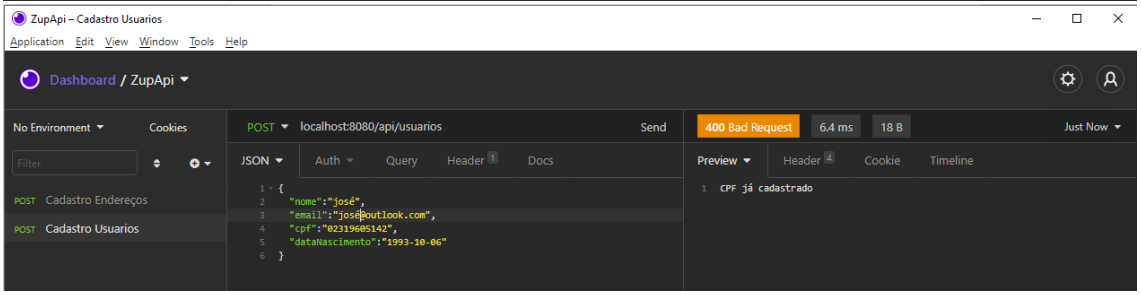


Caso eu tente cadastrar o mesmo o usuário 2x ele nos retornará a mensagem que o usuário já foi cadastrado e com erro 400



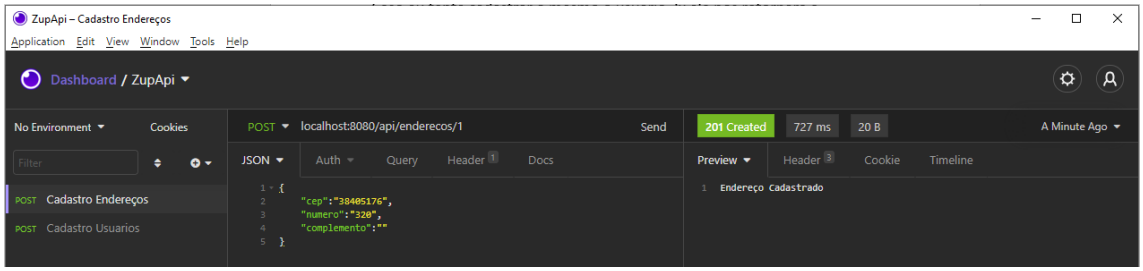
Da mesma forma para e-mail e cpf.



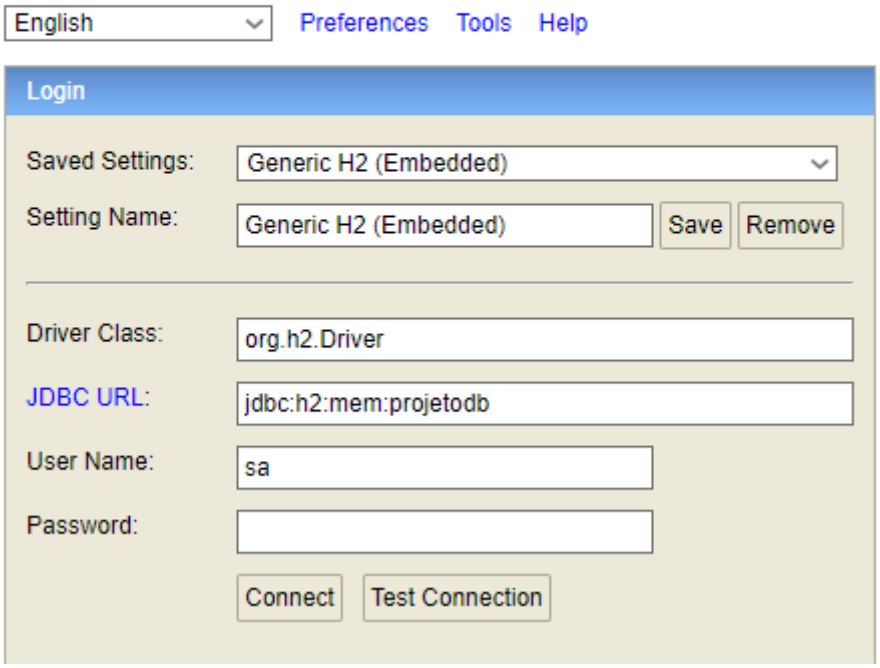


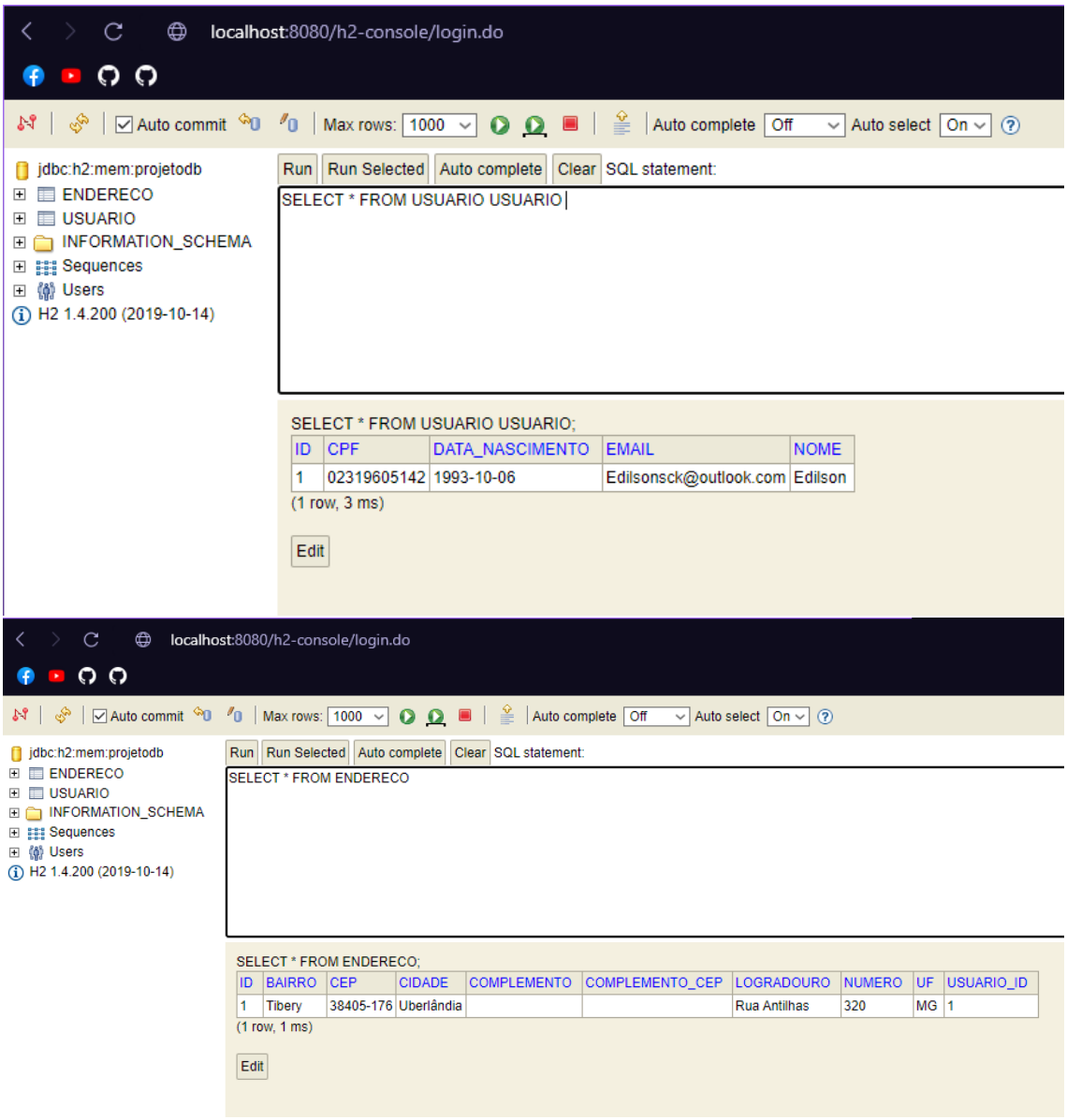
E a mesma coisa para os endereços alterando apenas a URL que a gente colocou api/endereço/1 e o id do usuário no final .

Ficando assim:



E como estamos utilizando a dependência do H2 temos acesso ao banco de dados pelo navegador utilizando a URL http://localhost:8080/h2-console/login onde usuário e senha está no nosso arquivo application.properties podemos chegar a tabela de usuários e endereços.





Sendo assim podemos ver o que o Java pode ser muito mais prático e produtivo se soubermos utilizar as ferramentas certas, entre elas o Spring Boot e o Spring Initializr.

Obs: Para a implementação do ViaCep foi consultado alguns links externos.

https://viacep.com.br

https://nstecnologia.com.br/blog/viacep-como-consultar-cep/

Gostaria de agradecer à Zup através do programa Orange Talents pela oportunidade de demonstrar um pouco do meu aprendizado na prática e também de evoluir muito durante o desafio desta API. Aprendi coisas novas como no caso da comunicação com a API externa do ViaCep, por exemplo, entre outros. Enfim, muito obrigado