

API REST - Conta Bancaria

Post por: Gildenor Junior da Silva Costa
junior.silva_costa@live.com

No post de hoje temos uma missão a cumprir, tentar criar e explicar o funcionamento de uma API REST Java, que tem por objetivo realizar operações básicas de um cadastro de pessoa tal como uma abertura de conta de banco.

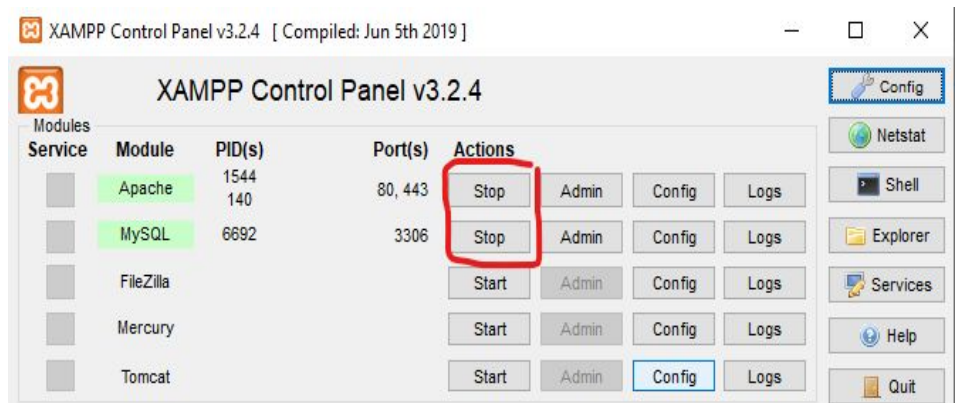
Apenas para recapitular e dar embasamento ao nosso post de hoje, vou citar alguns dos pontos obrigatórios que deveremos ter em nossa aplicação como os seguintes atributos: Nome, E-mail, CPF e Data de Nascimento. E também as tecnologias como linguagem de programação Java e Spring + Hibernate, tanto as tecnologias como os atributos serão complementados e explicados de maneira mais clara ao decorrer da execução desta nossa missão.

Visto isso, mãos à massa temos um objetivo a cumprir!

Primeiramente, um passo básico, mas não menos importante é verificar se temos alguns recursos essenciais instalados e funcionando em nossa máquina. São eles: XAMPP, Eclipse e MySQL Workbench.

XAMPP: É um software gratuito OpenSource que simula servidores em nossa máquina local.

Será necessário ter os serviços Apache e MySQL rodando durante todo o trabalho com a API.



Eclipse: É um Ambiente de Desenvolvimento Integrado, nele iremos escrever e executar nossos códigos Java.

MySQL: Será utilizado como serviço de Banco de dados, ele é um sistema gerenciador de banco de dados relacional onde poderemos acessar e fazer consultas ao nosso banco de dados.

Mas estas são ferramentas utilizadas para o desenvolvimento de qualquer aplicação Java, não necessariamente uma API REST com Spring, pois então, vamos ao lado obscuro da força!

Calma, Calma, Calma! Mas não antes de dar um overview sobre estas “siglazinhas” que estamos falando desde o começo como API, REST, SPRING, JAVA que nem todo mundo pode conhecer.

API: Em uma explicação simplificada, uma API é um “pedaço” de código que pode ser implementado em qualquer sistema sem necessariamente mexer nesse local. É uma forma de integrar diferentes sistemas.

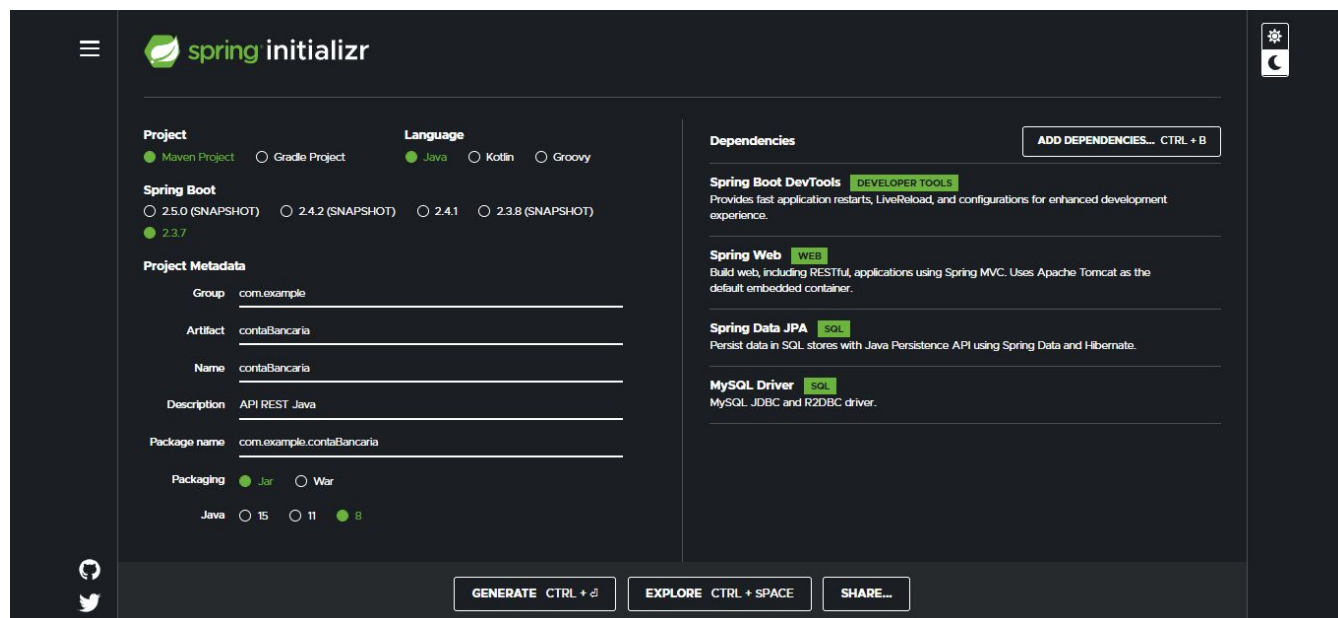
REST: Representational State Transfer, é um estilo de arquitetura de software que define um conjunto de restrições a serem usadas para a criação de web services.

JAVA: É uma linguagem de programação orientada a objetos, utilizada para desenvolver aplicações computacionais. Ou seja, um dos idiomas falados pelos computadores e programadores.

SPRING: É um framework Java criado com o objetivo de facilitar o desenvolvimento de aplicações, fazendo por exemplo o as tarefas de Inversão de Controle e Injeção de Dependências.

Ao decorrer do post iremos explicando novas siglas e termos que poderão e irão surgir ao longo do desenvolvimento e estudo. Agora sim, com todo mundo situado nesse mundo binário podemos dar continuidade a nossa missão.

Primeiro passo de fato para dar início a uma API REST com Spring é criar um projeto, para isso utilizaremos o *Spring Initializr* que facilita alguns pontos dessa tarefa.



The screenshot shows the Spring Initializr web application interface. The top navigation bar includes the Spring logo and the text 'spring initializr'. On the right, there are icons for a menu, a sun/moon theme toggle, and a user profile. The main content area is divided into several sections:

- Project:** Includes radio buttons for 'Maven Project' (selected) and 'Gradle Project'. Below this, there are radio buttons for 'Spring Boot' versions: '2.5.0 (SNAPSHOT)', '2.4.2 (SNAPSHOT)', '2.4.1', and '2.3.8 (SNAPSHOT)'. The version '2.3.7' is also listed.
- Language:** Includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.
- Project Metadata:** Includes input fields for 'Group' (com.example), 'Artifact' (contaBancaria), 'Name' (contaBancaria), 'Description' (API REST Java), and 'Package name' (com.example.contaBancaria). There are also radio buttons for 'Packaging' (Jar selected, War) and 'Java' versions (15, 11, 8 selected).
- Dependencies:** A section on the right with a button 'ADD DEPENDENCIES... CTRL + B'. It lists several dependencies: 'Spring Boot DevTools' (DEVELOPER TOOLS), 'Spring Web' (WEB), 'Spring Data JPA' (SQL), and 'MySQL Driver' (SQL). Each dependency has a brief description.

At the bottom, there are three buttons: 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

Como podemos visualizar na imagem acima, criaremos um projeto *Maven* que é uma ferramenta que serve para gerenciar dependências e automatizar builds, assim facilitando e não sendo necessário que criemos manualmente essa etapa. Daremos o nome do projeto de *contaBancaria* e utilizaremos a versão 8 do Java. As dependências que utilizaremos serão as seguintes:

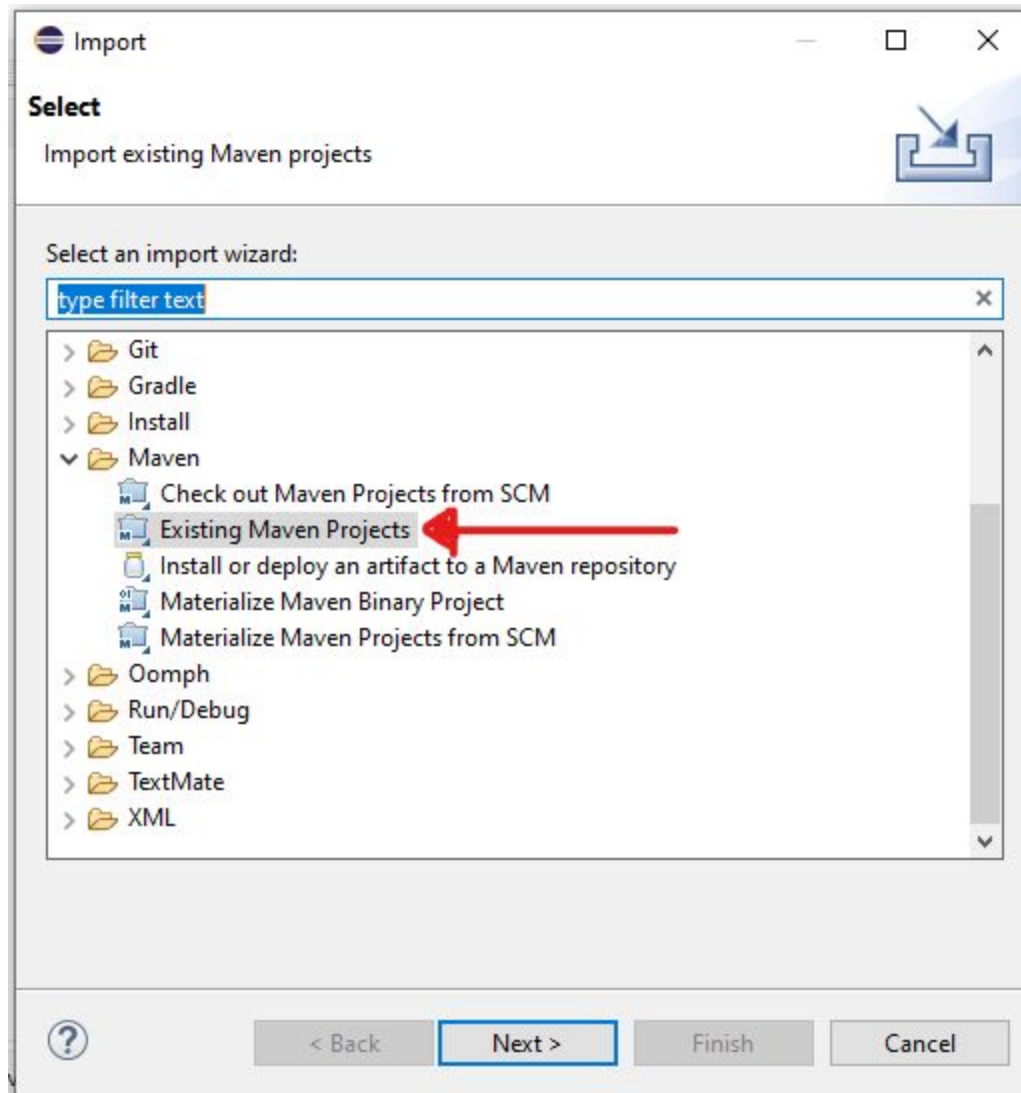
Spring Boot DevTools: Facilita na hora do desenvolvimento de modo que atualiza a cada vez que salvamos o código e assim não precisamos ficar reiniciando o servidor.

MySQL Driver: Dependência responsável por realizar a conexão da nossa API com o banco de dados MySQL.

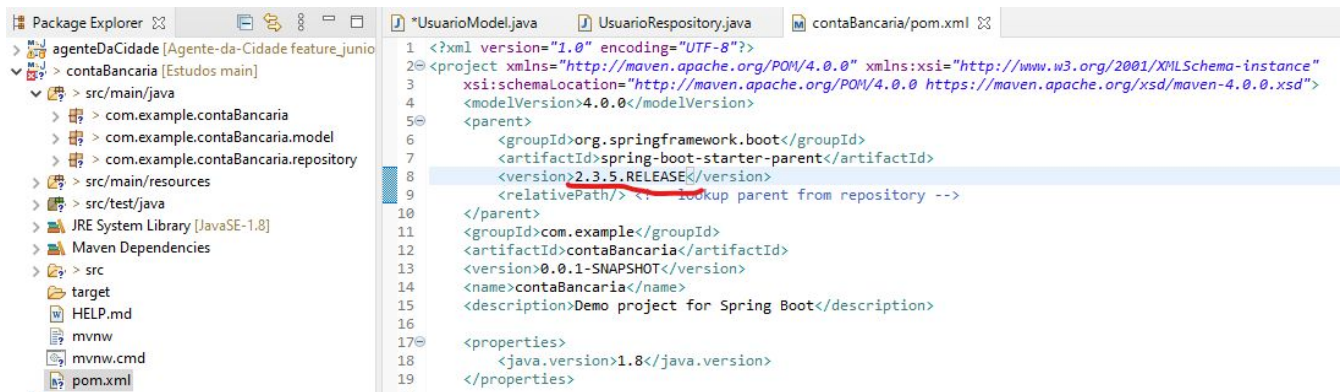
Spring Data JPA: Faz a persistência dos dados SQL usando Spring Data e Hibernate.

Spring Web: É um framework que auxilia no desenvolvimento de aplicações web no padrão MVC.

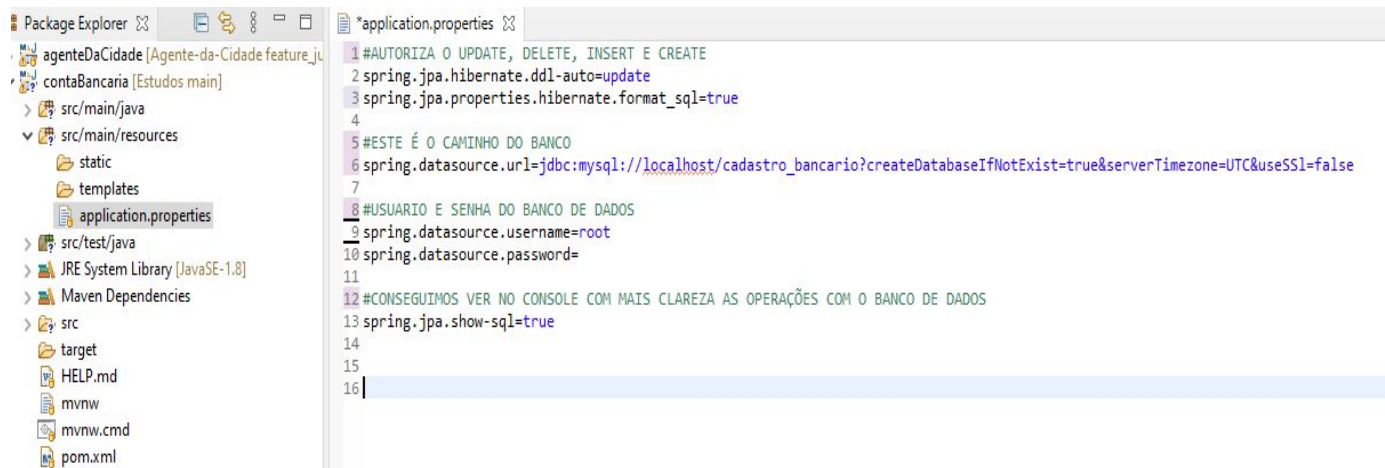
Feito isso, podemos gerar nosso projeto fazendo o download que virá em um arquivo compactado. Descompactando este arquivo, será necessário apenas fazer o import do mesmo para o eclipse.



Um detalhe nessa etapa é que precisaremos mudar a versão do nosso artefato *spring-boot-starter-parent* apenas por questões de bugs que estão sendo gerados nas versões mais atuais. Portanto, precisaremos ir no arquivo `pom.xml` e na tag `version` do artefato *spring-boot-starter-parent* mudaremos para a versão 2.3.5.RELEASE. Ficará parecido com a imagem abaixo.



Com o projeto importado no nosso eclipse agora vamos definir algumas configurações do nosso banco de dados, no arquivo *application.properties* é onde definimos todas essas configurações como o caminho específico para a criação do banco, usuário e senha desse banco e também alguns outros comandos do *jpa hibernate*, assim como podemos ver na imagem abaixo.



Após a configuração do nosso *application.properties*, agora criaremos um pacote, dentro desse pacote criaremos nossa primeira classe que será uma classe Model, ou seja, a partir dela será criada a tabela no nosso banco de dados, a chamaremos de **UsuarioModel** e nela teremos os seguintes atributos: *idUsuario* do tipo primitivo Long, *nome* do tipo String, *senha* do tipo String, *email* do tipo String, *cpf* do tipo String e *dataNascimento* do tipo LocalDate, todos terão os modificadores de acesso no modo private para a segurança dos nossos atributos e seguindo o conceito de encapsulamento criaremos também os métodos Getters e Setters de todos.

O atributo `idUsuario` será utilizado como uma espécie de índice de um array, para identificarmos os objetos e também para podermos fazer algumas operações buscando diretamente por ele.

Nessa classe também daremos início ao uso de Annotations do Java Bean Validation, que servem para dar instruções complementares ao programa e também validar e impor algumas regras aos campos que colocamos.

@Entity: Servirá para informar que a classe também é uma entidade do banco de dados.

@Table: com o parâmetro (`name = "nome_da_tabela"`), definimos o nome específico que queremos para aquela tabela do banco de dados.

@Id: Que representa o campo onde queremos impor a chave primária na tabela e que este campo será único.

@GeneratedValue: Serve para definir como será a geração desse campo, nele temos alguns parâmetros que podemos utilizar, nesse caso utilizamos (`strategy = GenerationType.IDENTITY`), assim informamos que um valor deverá ser gerado a cada registro inserido no banco.

@Column: Serve para categorizar aquele atributo como uma coluna no banco de dados.

@NotNull: É uma anotação do Bean Validation onde não permite que o campo seja nulo.

@NotEmpty: Valida se a propriedade não é nula ou vazia.

@Email: Valida se a propriedade anotada é um endereço de email válido.

@JsonFormat: Com essa anotação definimos o formato da data de nascimento que queremos.

@UniqueConstraint: Com ela conseguimos validar os campos de email e cpf para serem campos únicos no banco de dados.


Lembrando que para as anotações do Bean Validation funcionar precisamos adicionar as seguintes dependências no arquivo pom.xml.

```
<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator</artifactId>
    <version>6.0.2.Final</version>
</dependency>

<dependency>
    <groupId>org.hibernate.validator</groupId>
    <artifactId>hibernate-validator-annotation-processor</artifactId>
    <version>6.0.2.Final</version>
</dependency>

<dependency>
    <groupId>javax.validation</groupId>
    <artifactId>validation-api</artifactId>
    <version>2.0.0.Final</version>
</dependency>
```

Finalizando esta classe podemos ver na imagem abaixo como ficará seu resultado final após nossas implementações.

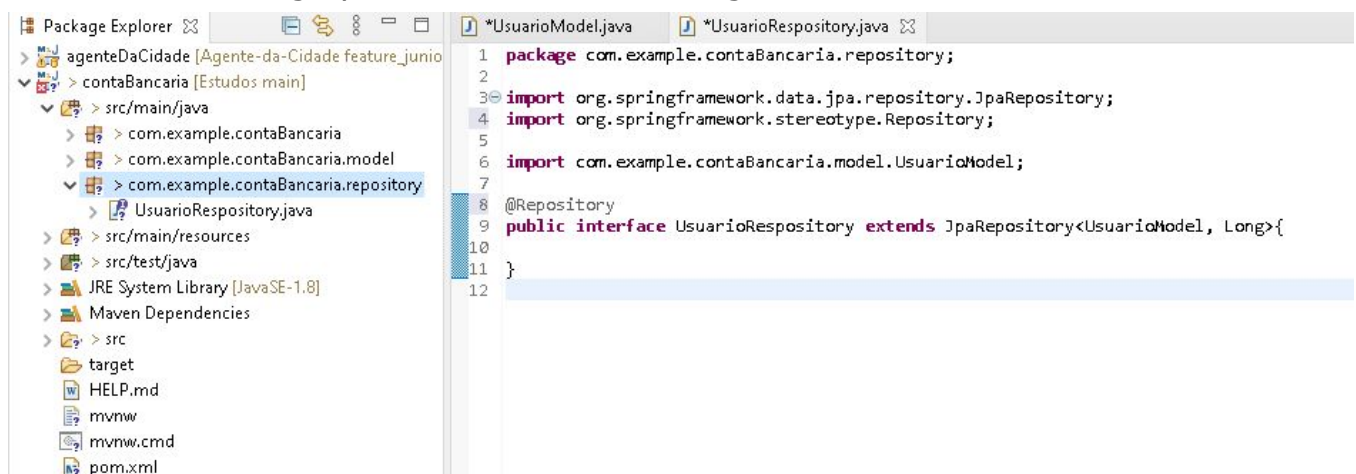


```
18
19 @Entity
20 @Table (name = "tb_usuarios", uniqueConstraints={@UniqueConstraint(columnNames={"cpf"})})
21 public class UsuarioModel {
22
23     //ATRIBUTOS
24     @Id
25     @GeneratedValue (strategy = GenerationType.IDENTITY)
26     private Long idUsuario;
27
28     @Column
29     @NotNull @NotEmpty(message= "Por favor, verifique o campo nome o mesmo não pode ser vazio ou nulo!")
30     private String nome;
31
32     @Column
33     @NotNull @NotEmpty(message= "Por favor, verifique o campo senha o mesmo não pode ser vazio ou nulo!")
34     private String senha;
35
36     @Column(unique = true)
37     @NotNull @NotEmpty(message= "Por favor, verifique o campo email o mesmo não pode ser vazio ou nulo!")
38     @Email
39     private String email;
40
41     @Column(name = "cpf")
42     @NotNull @NotEmpty(message= "Por favor, verifique o campo cpf o mesmo não pode ser vazio ou nulo!")
43     private String cpf;
44
45     @Column
46     @NotNull
47     @JsonFormat(pattern = "yyyy-MM-dd")
48     private LocalDate dataNascimento;
49 }
```


Após finalizada essa primeira etapa daremos início agora a nossa camada de Repository que tem por função persistir/recuperar os dados do banco de dados. Para isso, criaremos um pacote chamado repository e dentro dele criaremos uma interface com o nome de **UsuarioRepository**. Essa interface estenderá outra interface JpaRepository, essa interface faz parte do Spring Data que funciona como um intermédio de operações comuns com banco de dados como CRUD's e busca por campos das entidades do sistema, implementando ela precisaremos passar dois parametros que serão a classe de domínio e o tipo do ID que utilizamos na nossa classe model. Muito importante também darmos a instrução para identificar essa interface como um repository e para isso utilizaremos a seguinte anotação:

@Repository: Serve para identificar a classe como um repositório.

Ao fim teremos algo parecido como o da imagem abaixo:



No próximo passo iremos construir a nossa camada de controller que lida com as requisições dos usuários, ele fica responsável de retornar uma resposta com a ajuda das demais camadas da nossa API, nessa resposta que o usuário fará por meio de um EndPoint que criaremos na URL, passaremos uma resposta do objeto no formato de *JSON*.

Já podemos criar um novo pacote chamado controller e dentro dele criaremos essa nossa classe chamada UsuarioController. Nessa classe criaremos cinco EndPoints dos verbos vinculados ao protocolo HTTP são eles: GET, POST, PUT e DELETE e teremos um GET adicional onde poderemos

pesquisar um usuário a partir do id dele. Veremos mais alguns detalhes dos termos utilizados abaixo:

HTTP: É o protocolo de comunicação da Web, ele funciona como um protocolo de requisição e resposta nos modelos de cliente e servidor.

JSON: É um modelo de armazenamento e transmissão de informações no formato de texto. As informações que o usuário requisitar na nossa API serão retornadas como resposta neste formato por ser bem simples e também um dos mais usados atualmente nas aplicação Web.

GET: Verbo utilizado para recuperar e ler os dados, com ele podemos ter uma visualização do que solicitamos. Teremos dois desse método na nossa API, um para recuperar o usuário específico através de seu ID e também outro método para recuperar uma lista com todos os usuários cadastrados em nosso sistema.

POST: Utilizamos para criar algo novo, nesse caso podemos criar um novo usuário no nosso sistema de cadastro. Com ele precisamos especificar um body com todos os recursos que desejamos incluir.

PUT: Utilizado para atualizar informações. Podemos atualizar qualquer informação dos recursos já existentes na nossa aplicação apenas chamando pela URL com o Id do determinado usuário e no body todo o conjunto de especificações que o objeto já tem e o que será modificado.

DELETE: Utilizado para excluir algum recurso do nosso sistema. Através da URL passamos o ID específico do objeto e com isso esse recurso será excluído, sendo assim não necessário passar nada no body.

@GetMapping, @PostMapping, @PutMapping, @DeleteMapping: Através dessas anotações conseguimos fazer o mapeamento de retorno de acordo com cada método HTTP que queremos.

@RestController: Usado para sinalizar que essa é uma classe que terá a função de controller e também utilizado para redirecionar view, além de retornar tudo em formato JSON.

@RequestMapping: Serve para mapearmos a url inicial que queremos para os EndPoints.

@CrossOrigin: É utilizado para a troca de recursos entre os navegadores e o servidor em domínios diferentes.

@Autowired: Utilizamos para injetar a dependência do repositório que criamos anteriormente.

@PathVariable: Utilizamos para quando o determinado valor será passado diretamente pela URL, como no caso do Get quando recuperamos um recurso puxando pelo seu ID, esse ID vai diretamente na URL.

@RequestBody: Utilizado para os métodos que precisaremos passar o corpo do objeto na requisição.

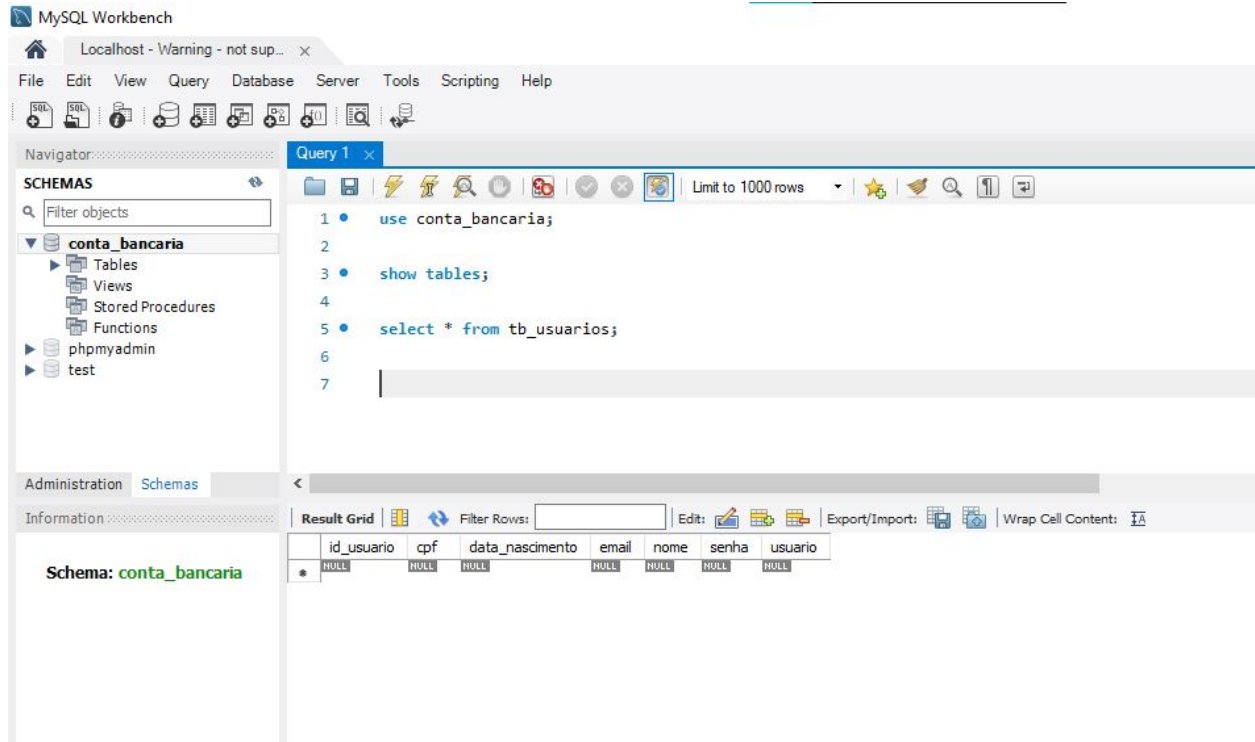
Agora que já sabemos de cada um dos itens que utilizamos na nossa classe controller, teremos um resultado parecido com o da imagem abaixo (imagem cortada devido proporções da screenshot):

```

25
26 @Autowired
27 private UsuarioRepository repository;
28
29 //MÉTODO GET QUE BUSCA TODOS USUARIOS
30 @GetMapping
31 public ResponseEntity<List<UsuarioModel>> getAll(){
32     return ResponseEntity.ok(repository.findAll());
33 }
34
35 //MÉTODO GET QUE BUSCA O USUARIO PELO ID
36 @GetMapping("/{id}")
37 public ResponseEntity<UsuarioModel> getById(@PathVariable Long id){
38     return repository.findById(id)
39         .map(resp-> ResponseEntity.ok(resp))
40         .orElse(ResponseEntity.notFound().build());
41 }
42
43 //MÉTODO POST QUE CRIA UM NOVO USUARIO
44 @PostMapping
45 public ResponseEntity<UsuarioModel> post(@RequestBody UsuarioModel usuario){
46     return ResponseEntity.status(HttpStatus.CREATED).body(repository.save(usuario));
47 }
48
49 //MÉTODO PUT QUE ATUALIZA AS INFORMAÇÕES DE UM USUARIO
50 @PutMapping
51 public ResponseEntity<UsuarioModel> put(@RequestBody UsuarioModel usuario){
52     return ResponseEntity.ok(repository.save(usuario));
53 }
54
55 //MÉTODO DELETE QUE APAGA UM USUARIO
56 @DeleteMapping("/{id}")
57 public void delete(@PathVariable Long id) {
58     repository.deleteById(id);

```

Chegamos na hora da verdade... Hora de executarmos nosso código para ver o seu funcionamento. Portanto, na classe principal **ContaBancariaApplication** que contém o método main, iremos executá-la como uma Java Application. Se a aplicação for executada com sucesso e sem nenhum erro poderemos verificar o banco de dados criado no MySQL Workbench.



Muito legal, não é mesmo?!

Agora vamos brincar um pouco com o nosso CRUD. Utilizaremos agora um programa chamado PostMan, ele serve para nos auxiliar nas funções de testes da API, simulando requisições HTTP, com ele poderemos fazer todas as operações que criamos no nosso controller e ver nosso sistema de fato funcionando.

No nosso primeiro manejo com as requisições faremos uma busca GET por algum usuário já cadastrado, porém como iniciamos agora e ainda não temos nenhum usuário cadastrado no nosso banco de dados o retorno correto da aplicação deve uma lista vazia. E percebe também que agora veremos que a cada requisição feita receberemos um status HTTP de acordo com o resultado da nossa consulta.

Temos cinco grupos de respostas, cada status de resposta HTTP indica se aquela requisição foi efetuada e se não a possível causa.

1XX - Informações Gerais

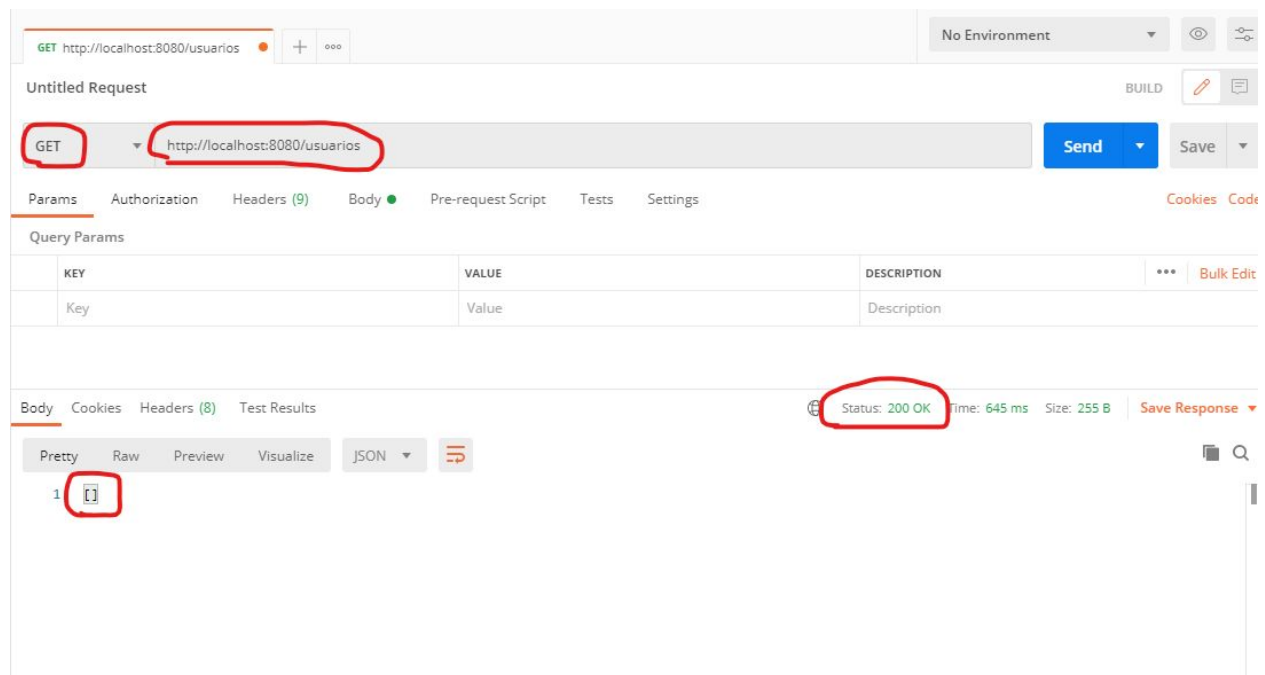
2XX - Sucesso

3XX - Redirecionamento

4XX - Erro no cliente

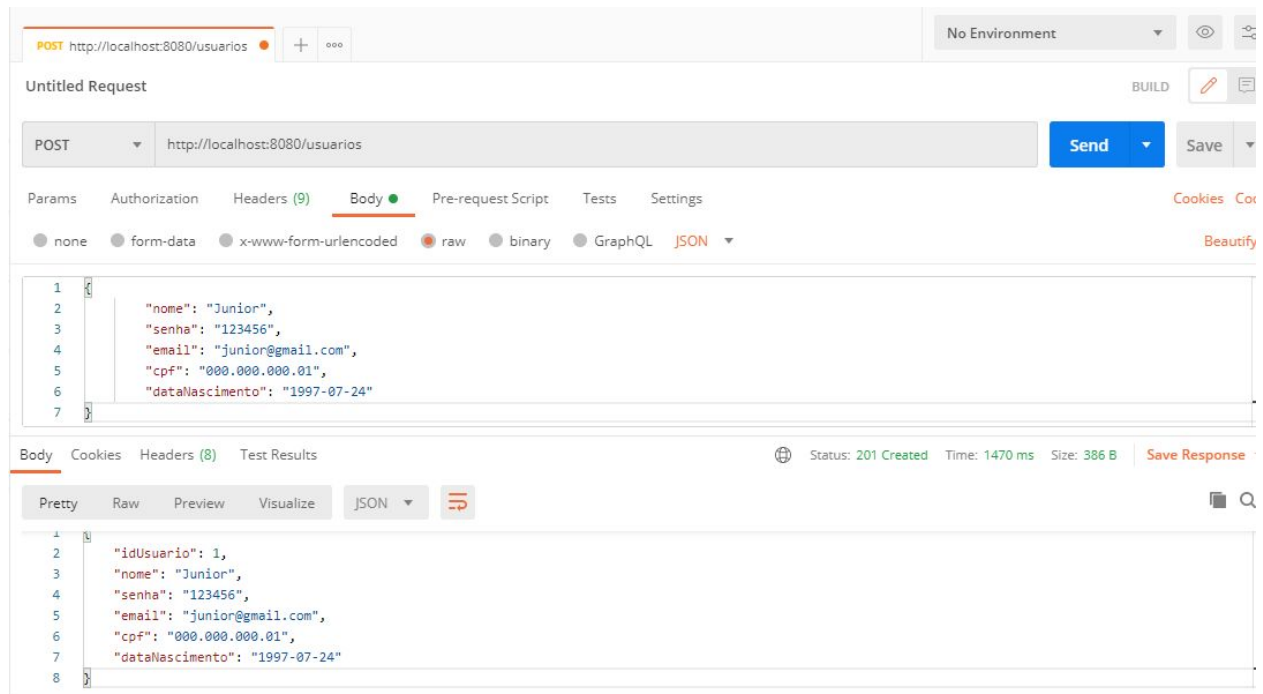
5XX - Erro no servidor

Como podemos ver no nosso primeiro exemplo de requisição, foi retornado um código de status 200 onde indica que tudo ocorreu sem nenhum erro.



Está na hora de chamarmos clientes para se filiar a esse nosso banco não é mesmo?! Pois então, vamos lá cadastrar nosso primeiro usuário.

Para o cadastro precisaremos alterar o método de GET para POST no PostMan e também em body precisaremos passar todos os parâmetros que nosso cliente precisa para ser criado como, nome, e-mail, senha, data de nascimento e cpf. Lembrando que não precisamos passar o ID por causa das nossas validações na Model onde cada ID será gerado automaticamente e que tudo isso precisaremos mandar no formato de texto JSON. Com esses passos corretamente feitos, teremos o próprio objeto usuário como resposta e também teremos o status HTTP 201 que vem significando que deu tudo certo e o objeto foi criado corretamente.



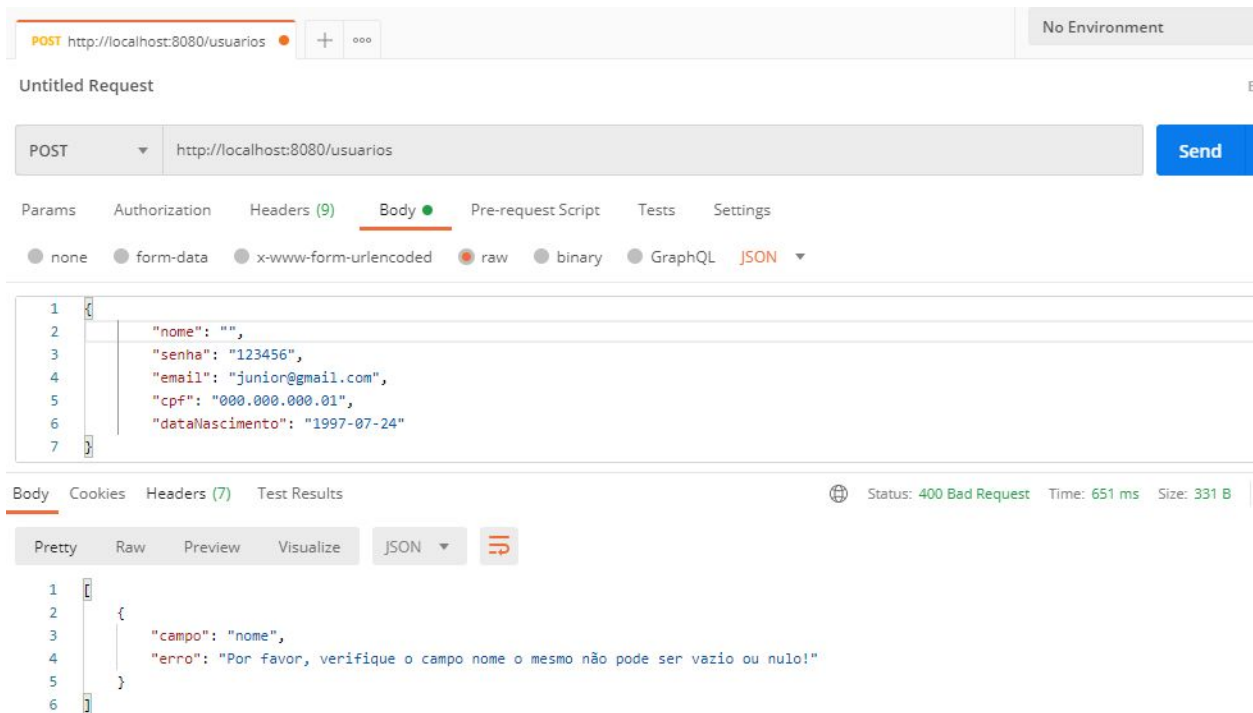
Mas agora você deve estar se perguntando, e se eu colocar algum atributo errado ou até mesmo não colocar... Boa questão! Lembra daquelas validações que fizemos lá na classe Model onde falamos que determinado campo não poderia ser vazio ou nulo, é agora que eles estão em ação, se deixarmos de colocar o nome por exemplo, ao invés de retornar o próprio objeto com o status de 201, ele nos retornará um código 400 indicando uma BadRequest no Cliente da nossa aplicação além de um código gigante de erros que chegam a dor os olhos. Ao trabalharmos com uma API que será consumida por outros desenvolvedores no front end não é muito legal deixarmos esses erros gigantes e sem tratamento passar, de modo que chegando no front ficará mais difícil para o desenvolvedor saber o que de fato aconteceu ali. Para isso, criaremos uma espécie de filtro interceptador do spring chamado ControllerAdvice, ele intercepta todas as exceções que forem geradas a partir dos endpoints da nossa aplicação e com isso poderemos configurar nele uma mensagem de erro mais amigável e de acordo com a exceção que está ocorrendo ali no momento da execução. Nosso primeiro passo para essa resolução será criar um novo pacote chamado config.validacao, nele criaremos também duas novas classes ErroDeFormularioDto e ErroDeValidacaoHandler. Na classe ErroDeFormularioDto teremos apenas dois atributos e seus métodos Getters e também um construtor da classe, basicamente nessa classe criamos os

recursos que serão mostrados a cada mensagem de erro. Já na próxima classe é onde vamos fazer as instruções e o tratamento de fato desses erros. Podemos ver na próxima imagem como ficará o resultado dessa classe com os métodos implementados e também utilizaremos a seguinte anotação:

@RestControllerAdvice: Vai indicar para o Spring que essa é uma classe interceptadora de exceções.

```
1 package com.example.contaBancaria.config.validacao;
2
3+ import java.util.ArrayList;
15
16
17 @RestControllerAdvice
18 public class ErroDeValidacaoHandler{
19
20-     @Autowired
21     private MessageSource messageSource;
22
23-     @ResponseStatus(code = HttpStatus.BAD_REQUEST)
24     @ExceptionHandler(MethodArgumentNotValidException.class)
25     public List<ErroDeFormularioDto> handle(MethodArgumentNotValidException exception) {
26         List<ErroDeFormularioDto> dto = new ArrayList<>();
27         List<FieldError> fieldErrors = exception.getBindingResult().getFieldErrors();
28
29         fieldErrors.forEach(e -> {
30             String mensagem = messageSource.getMessage(e, LocaleContextHolder.getLocale());
31             ErroDeFormularioDto erro = new ErroDeFormularioDto(e.getField(), mensagem);
32             dto.add(erro);
33         });
34
35         return dto;
36     }
37
38 }
39
```

Após ter esses recursos corretamente configurados, ao tentar cadastrar um usuário que não tenha o nome teremos aquela validação da classe Model ativada nos retornando a mensagem que configuramos e também o status code HTTP 400 indicando uma BadRequest. Além é claro, desse usuário não ser cadastrado no nosso banco de dados por conta dessa segurança, sendo necessário a correção desses dados.



E da mesmo forma também temos as restrições que colocamos lá na Model dos campos Email e Cpf com o recurso unique que faz com que se tentarmos adicionar um outro usuário mas com o mesmo email ou cpf dará erro e não deixará adicionar pois já existe uma chave dessa no nosso banco de dados, tendo que alterar esses campos para um valor diferente para que se possa cadastrar um novo usuário.

Agora depois do cadastro de mais alguns usuários no sistema do nosso banco se formos no MySQL Workbench verificar como está populado nosso banco de dados poderemos os dados de todos esses usuários.

Query 1 x

Limit to 1000 rows

```

1 • use conta_bancaria;
2
3 • show tables;
4
5 • select * from tb_usuarios;
6

```

Result Grid

Filter Rows:

Edit:

Export/Import:

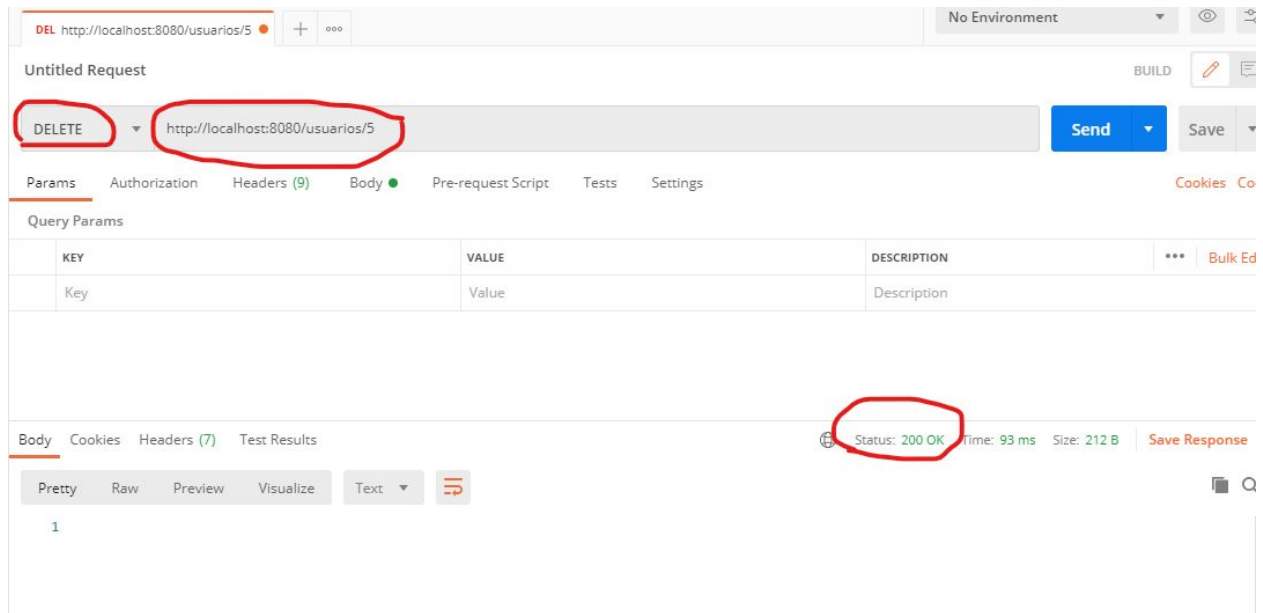
Wrap Cell Content:

	id_usuario	cpf	data_nascimento	email	nome	senha
▶	1	000.000.000.01	1997-07-24	junior@gmail.com	Junior	123456
	2	000.000.000.02	1997-04-24	maria@gmail.com	Maria	654321
	3	000.000.000.03	1998-06-10	antonia@gmail.com	Antonia	112233
	4	000.000.000.04	1996-01-18	jose@gmail.com	Jose	258741
	5	000.000.000.05	1995-03-20	pedro@gmail.com	Pedro	001144
*	NULL	NULL	NULL	NULL	NULL	NULL

E se quisermos atualizar o cadastro de algum usuário do nosso sistema? Temos um método no nosso controller para isso! Com o método put podemos atualizar qualquer dado dos usuários do nosso sistema, da mesma forma que cadastramos os usuários no postman podemos fazer para atualizar, precisamos passar todo o corpo do objeto junto com o atributo que queremos atualizar e na resposta teremos o objeto atualizado com o seu status code HTTP.

Temos também um método adicional GET que podemos recuperar qualquer usuário buscando pelo seu id.

E por fim temos o nosso método para deletar um usuário do nosso cadastro, apenas passando na URL o id do usuário que desejamos deletar, no corpo da resposta do PostMan não será retornado nada pois o método Delete tem o retorno definido como void mas o status code HTTP será retornado normalmente se a função for concluída corretamente. Na imagem abaixo podemos ver a deleção do usuário Pedro que tem o id 5.



E se fossemos consultar o banco de dados no MySQL WorkBench poderíamos confirmar essa função vendo que o usuário Pedro foi devidamente deletado.

Com isso, exploramos todos os pontos e aprendemos um pouquinho sobre o funcionamento da nossa API REST Java para o cadastro de pessoas em um sistema simulando uma conta de banco.

Por fim, finalizamos o nosso post tutorial de hoje com a nossa missão primária concluída, até o próximo post!