

Criando meu Projeto com Spring Framework

Atualmente a produtividade é uma das palavras-chave quando falamos em desenvolvimento de software. Criar e configurar projetos do zero é um processo bem demorado de ser realizado, não sendo à toa a infinidade de ferramentas que surgem todos os dias visando justamente acelerar esse processo de criação e implantação.

Se você gostaria de simplificar o seu desenvolvimento de aplicações Java sem ter que fazer inúmeras configurações e otimizações e ainda elevar seu nível de produtividade – Tenho algo a lhe apresentar: o **Spring Boot**.

Primeiramente quero lhe apresentar as tecnologias do mundo Spring que vamos utilizar neste projeto:

- **Spring Boot:** uma ferramenta que visa facilitar o processo de configuração e publicação de aplicações que utilizem o ecossistema Spring. Ele fornece a maioria dos componentes baseados no Spring necessários em aplicações em geral de maneira pré-configurada. O Spring Boot revolucionou a forma de desenvolver, abstraindo as tarefas mais onerosas, possibilitando um rápido desenvolvimento.
- **Spring MVC:** é uma estrutura MVC completa orientada a HTTP, gerenciada pela Spring Framework e baseada em Servlets. Seria equivalente ao JSF na pilha Java EE. Os elementos mais populares nele são as classes anotadas **@Controller**, nas quais você implementa métodos que podem ser acessados usando diferentes solicitações HTTP. Tem um equivalente **@RestController** para implementar APIs baseadas em REST.
- **Spring Data JPA:** é um framework que nasceu para facilitar a criação dos nossos repositórios (camada de persistência de dados) oferecendo funcionalidades sofisticadas e comuns à maioria dos métodos de acesso a banco de dados. Ele (o Spring Data JPA) é, na verdade, um projeto dentro de um outro maior que é o **Spring Data**. O Spring Data tem por objetivo facilitar nosso trabalho com persistência de dados de uma forma geral.
- **Spring Boot Devtools:** uma ferramenta que pode melhorar bastante o tempo de desenvolvimento dos seus projetos, não precisando fazer tarefas repetitivas como reiniciar a aplicação a todo instante, Algumas mensagens de erro ficarão mais claras e o aumento da velocidade da reinicialização.

Para facilitar ainda mais, o Spring disponibiliza a página **Spring Initializr**. Com apenas alguns cliques, você pode habilitar os módulos desejados em seu projeto.

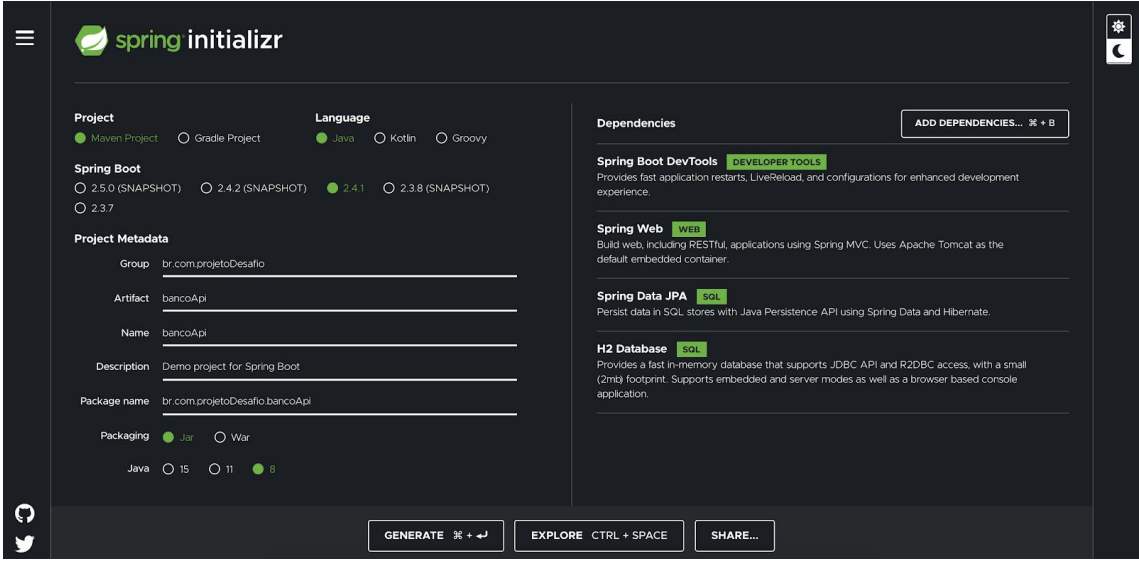
Começando o Projeto

Vamos acessar o Spring Initializr e começar o nosso projeto.

Configurações do projeto:

1. Gerenciador de projeto: Maven;
2. Linguagem de programação: Java;
3. Versão do spring boot: 2.4.1;
4. Identificador do Grupo ou Group Id: br.com.projetoDesafio;
5. Identificador do projeto ou Artefact Id: bancoApi;
6. O nome do projeto deixei o padrão que foi gerado;
7. A descrição também deixei a padrão;
8. O pacote é gerado automaticamente de acordo com Group ID e o Artefact Id;
9. O tipo de arquivo gerado após a compilação do projeto escolhi o JAR;
10. E a versão do Java JDK é a 1.8;
11. Dependências do projeto que iremos usar: Spring Boot Dev Tools, Spring Web e Spring Data JPA, h2 Database.

Seu projeto deve ficar assim:

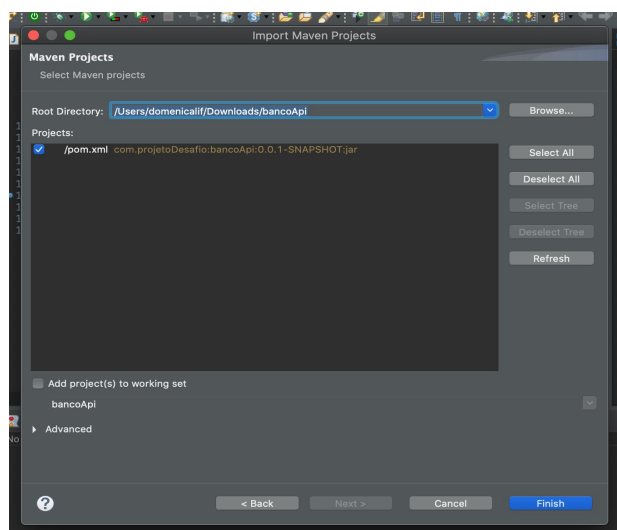
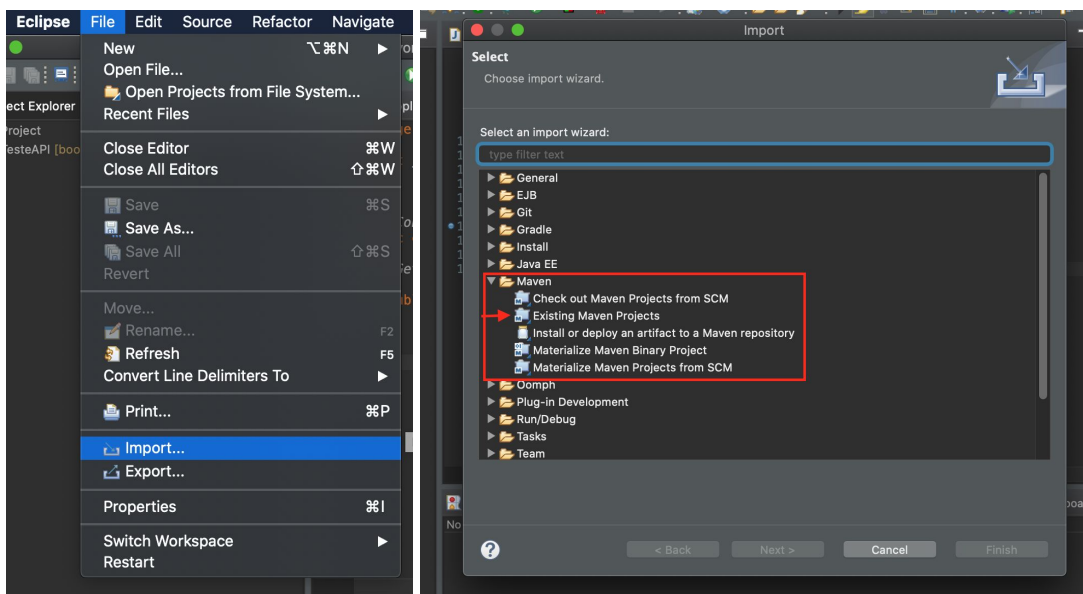


The screenshot shows the Spring Initializr web application interface. The left sidebar contains the Spring logo and social media icons. The main content area is divided into several sections:
 - **Project**: Includes radio buttons for 'Maven Project' (selected) and 'Gradle Project'.
 - **Language**: Includes radio buttons for 'Java' (selected), 'Kotlin', and 'Groovy'.
 - **Spring Boot**: Includes radio buttons for versions 2.5.0 (SNAPSHOT), 2.4.2 (SNAPSHOT), 2.4.1 (selected), and 2.3.8 (SNAPSHOT).
 - **Project Metadata**: Includes input fields for 'Group' (br.com.projetoDesafio), 'Artifact' (bancoApi), 'Name' (bancoApi), 'Description' (Demo project for Spring Boot), and 'Package name' (br.com.projetoDesafio.bancoApi).
 - **Packaging**: Includes radio buttons for 'Jar' (selected) and 'War'.
 - **Java**: Includes radio buttons for versions 15, 11, and 8 (selected).
 - **Dependencies**: A section on the right with a button 'ADD DEPENDENCIES... 3 + 8'. It lists selected dependencies: 'Spring Boot DevTools' (DEVELOPER TOOLS), 'Spring Web' (WEB), 'Spring Data JPA' (SQL), and 'H2 Database' (SQL).
 - At the bottom, there are three buttons: 'GENERATE' (with a download icon), 'EXPLORE' (with 'CTRL + SPACE'), and 'SHARE...'.

Página do Spring Initializr

Após finalizar, vamos clicar no botão "Generate" para baixar o arquivo zipado, logo após vamos importá-lo pela nossa IDE Eclipse seguindo os seguintes passos:

1. Com o seu Eclipse aberto, clique no primeiro botão no menu superior File;
2. E clique em Import... para abrir uma caixa de diálogo;
3. Encontre a pasta com o nome Maven, e clique na seta (>) para exibir as opções;
4. Escolha a opção Existing Maven Projects e clique em Next para abrir uma nova janela;
5. Na janela que abriu clique em Browse... e navegue até a pasta onde o seu projeto descompactado está, e depois selecione a pasta;
6. Agora se tudo ocorreu bem, na seção Projects, um campo de checkbox deve aparecer selecionado com as informações do seu **pom.xml**;
7. Agora basta clicar no botão Finish para importar seu projeto para o Eclipse.



Importando o Projeto na IDE

O Pom.xml é um dos arquivos mais importantes em um projeto Maven, ele descreve uma série de configurações que o projeto terá e quais repositórios e dependências seu projeto irá precisar. Então, todas as dependências que listamos em nosso projeto pelo Spring Initializr, o POM gerência.

```
21 <dependencies>
22   <dependency>
23     <groupId>org.springframework.boot</groupId>
24     <artifactId>spring-boot-starter-web</artifactId>
25   </dependency>
26
27   <dependency>
28     <groupId>org.springframework.boot</groupId>
29     <artifactId>spring-boot-devtools</artifactId>
30     <scope>runtime</scope>
31     <optional>true</optional>
32   </dependency>
33   <dependency>
```

Arquivo Pom.xml

Criando Classes para o nosso Projeto

Vamos começar criando a classe modelo, essa classe representa o objeto do mundo real, e no nosso caso, eu quero tratar informações de nome, e-mail, CPF e Data de nascimento.

Vamos criar um novo pacote dentro do pacote principal chamado model, logo em seguida vamos adicionar uma nova classe dentro deste pacote chamado Cliente e colocar as nossas variáveis e gerar os Getters e Setters.

```
21 public class Cliente {
22
23     private Long id;
24     private String nome;
25     private String email;
26     private Long cpf;
27     private Date dataNasc;
28
29     public Long getId() {
30         return id;
31     }
32
33     public String getNome() {
34         return nome;
35     }
36
37     public String getEmail() {
38         return email;
39     }
40
41     public Long getCpf() {
42         return cpf;
43     }
44
45     public Date getDataNasc() {
46         return dataNasc;
47     }
48
49     public void setId(Long id) {
50         this.id = id;
51     }
52
53     public void setNome(String nome) {
54         this.nome = nome;
55     }
56
57     public void setEmail(String email) {
58         this.email = email;
59     }
60
61     public void setCpf(Long cpf) {
62         this.cpf = cpf;
63     }
64 }
```

Classe Cliente do Projeto

Agora vamos criar o nosso primeiro endpoint da API REST, criando um novo pacote como fizemos anteriormente, e vamos adicionar uma nova classe para controlar as rotas na nossa API REST Spring Boot.

O nome do pacote vai ser controller e a classe ApiController. O próximo passo é informar para o Spring Boot que esta classe é um controller, para isso vamos adicionar a anotação **@RestController** na classe.

```
package com.projetoDesafio.bancoApi.controller;

import org.springframework.web.bind.annotation.RestController;

@RestController
public class ApiController {

}
```

Classe ApiController do pacote Controller

@RestController: Informa que a classe é um manipulador de solicitações. É usada para criar serviços da Web RESTful. Cuida do mapeamento dos dados da solicitação para o método definido do manipulador de solicitações. Depois que o corpo da resposta é gerado a partir do método manipulador, ela o converte em resposta JSON ou XML.

Vamos agora criar o nosso Pacote services com uma nova classe chamada ClienteServices. Vamos utilizar essa classe para regras de negócio e também incluindo a notação **@Service**, pois ela informa que a classe é um serviço.

```
package com.projetoDesafio.bancoApi.services;

import org.springframework.stereotype.Service;

@Service
public class ClienteServices {}

}
```

Classe ClienteServices do Pacote Services

Se comunicando com o Banco de Dados

Vamos aproveitar e criar o nosso Pacote Repository com a interface CadastroRepository, pois essa Classe é um objeto que isola os objetos ou entidades do domínio do código que acessa o banco de dados.

Temos um repositório que implementa parte das regras de negócio no que se refere à composição das entidades. Ele é fortemente vinculado ao domínio da aplicação e este é um reflexo direto das regras de negócio, pois ele abstrai armazenamento e consulta de um ou mais entidades de domínio.

```
package com.projetoDesafio.bancoApi.repository;

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

import com.projetoDesafio.bancoApi.model.Cliente;

public interface CadastroRepository extends JpaRepository<Cliente, Long> {

}
```

Classe CadastroRepository do Pacote Repository

Configuração

Precisamos fazer a configuração de banco para que possamos persistir os objetos. Neste post usarei o banco de dados H2. Este BD é um banco em memória bem simples.

Dentro do pacote **src/main/resources** existe o Application.properties, lá vamos setar as configurações do nosso Banco de dados em memória, ficará dessa maneira:

```
1 spring.datasource.driver-class-name=org.h2.Driver
2 spring.datasource.url = jdbc:h2:mem:cliente
3 spring.datasource.username=sa
4 spring.datasource.password=
5
6 spring.jpa.database-platform=org.hibernate.dialect.H2Dialect
7 spring.jpa.hibernate.ddl-auto=update
8
9 spring.h2.console.enabled=true
10 spring.h2.console.path=/h2-console
```

Configuração do applicationProperties

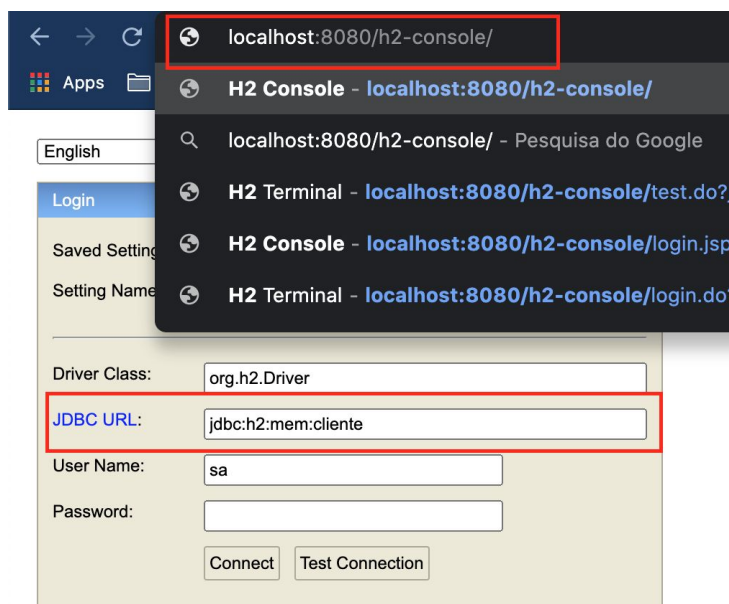
Agora vamos falar para a nossa classe Cliente que ela é uma Entidade e os atributos são as colunas da nossa Entidade Cliente. Para isso vamos usar a notação @Entity. A anotação @id é uma anotação obrigatória e determina qual campo da entidade representa a chave primária.

```
@Entity
public class Cliente {

    @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private String nome;
    private String email;
    private Long cpf;
    private Date dataNasc;
```

Classe Cliente anotada como Entity

Agora vamos acessar o navegador para entrar no nosso banco em memória. Para isso vamos acessar a URL <http://localhost:8080/h2-console> que foi setado lá no nosso applicationProperties e logar no campo JDBC URL: com a URL que também colocamos no nosso applicationProperties.



Conectando com o Banco de Dados H2

Assim que acessarmos, vemos que a nossa tabela e campos listados estão refletindo no Banco de dados

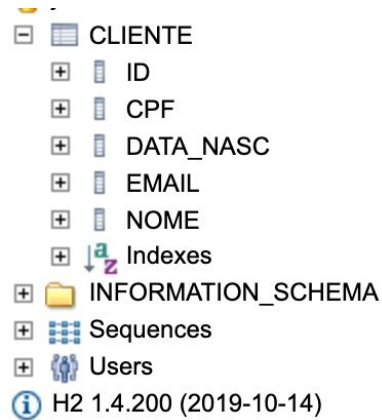


Tabela do Banco de Dados H2

Vamos agora fazer um Insert no Banco apenas para validarmos se está tudo certo. Vamos então criar um arquivo chamado data.sql e lá fazer o Insert.



Fazendo um Insert no Banco de Dados

Verique, já está refletindo no Banco de Dados:



Banco de Dados H2

Criando nossa API REST

Vamos criar a nossa Classe `ClienteDto` dentro do Pacote `Dto`, Data Transfer Object (DTO) ou simplesmente *Transfer Object* é um padrão de projetos bastante usado em Java para o transporte de dados entre diferentes componentes de um sistema, diferentes instâncias ou processos de um sistema distribuído ou diferentes sistemas via serialização.

Além disso, muitas vezes os dados usados na comunicação não refletem exatamente os atributos do seu modelo. Então, um DTO seria uma classe que provê exatamente aquilo que é necessário para um determinado processo.

Dentro da nossa classe iremos gerar os Getters e Setters e acessar as informações `Cliente` e também criar o método `obterDataFormatada` para termos o formato "dd/mm/yyyy".

```
public class ClienteDto {  
    private Long id;  
    private String nome;  
    private String email;  
    private Long cpf;  
    private String dataNasc;  
  
    public ClienteDto() {  
        // TODO Auto-generated constructor stub  
    }  
  
    public ClienteDto(Cliente cadastro) {  
        this.id = cadastro.getId();  
        this.nome = cadastro.getNome();  
        this.email = cadastro.getEmail();  
        this.cpf = cadastro.getCpf();  
        this.dataNasc = obterDataFormatada(cadastro.getDataNasc());  
    }  
  
    public static String obterDataFormatada(Date dataNasc){  
        SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");  
        return sdf.format(dataNasc);  
    }  
  
    public Long getId() {  
        return id;  
    }  
    public String getNome() {  
        return nome;  
    }  
}
```

Classe `ClienteDto` do Pacote `Dto`

Dentro da nossa Classe Services vamos criar um Método Array para listar os clientes cadastrados. Vamos também adicionar a anotação `@Autowired`, pois ela fornece controle sobre onde e como a ligação entre os beans deve ser realizada.

```
@Service
public class ClienteServices {

    @Autowired
    private CadastroRepository cadastroRepository;

    public List<ClienteDto> listarClientes() {
        List<Cliente> cadastros = cadastroRepository.findAll();

        return ClienteDto.converterList(cadastros);
    }
}
```

Adicionando uma ArrayList

O Converter é um método que vamos utilizar para converter para DTO, tanto os dados recebidos como os dados já existentes. Ele ficará na classe ClienteDto

```
public static List<ClienteDto> converterList(List<Cliente> cliente){
    return cliente .stream().map(ClienteDto::new).collect(Collectors.toList());
}

public static ClienteDto converter(Cliente cliente) {
    ClienteDto clienteDto = new ClienteDto();

    clienteDto.setId(cliente.getId());
    clienteDto.setNome(cliente.getNome());
    clienteDto.setCpf(cliente.getCpf());
    clienteDto.setEmail(cliente.getEmail());
    clienteDto.setDataNasc(obterDataFormatada(cliente.getDataNasc()));

    return clienteDto;
}
```

Métodos da Classe Cliente Dto

Vamos para a nossa classe ApiController e usar o @GetMapping, anotação para mapear solicitações HTTP GET em métodos manipuladores específicos e passar o valor do identificador da rota ou URI, que vai ser "clientes".

Agora precisamos fazer a injeção de dependência do ClienteServices, podemos fazer esta injeção pelo construtor, mas vamos optar por fazer com uma anotação chamada @Autowired e utilizar os métodos do ClienteServices.

```
@RestController
public class ApiController {

    @Autowired
    private ClienteServices clienteServices;

    @GetMapping("clientes")
    public List<ClienteDto> listarClientes() {
        List<ClienteDto> clientes = clienteServices.listarClientes();

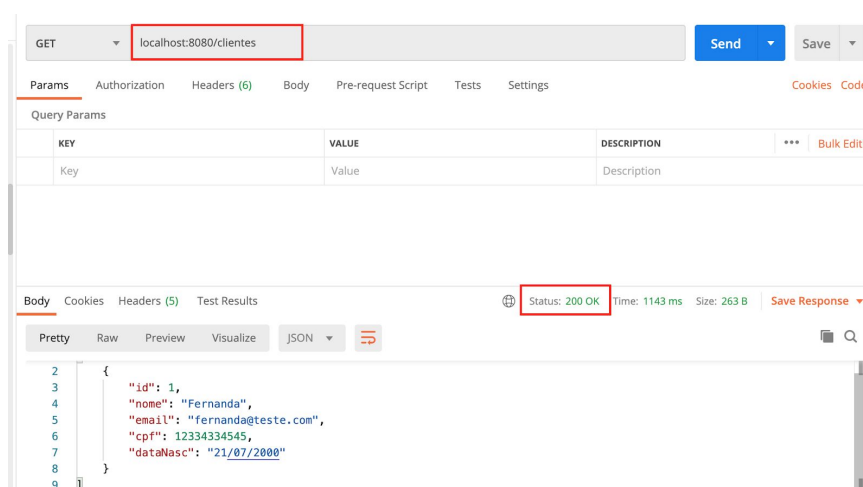
        return clientes;
    }
}
```

Classe ApiController para requisições http

Indo para o Aplicativo Postman, uma ferramenta que tem como objetivo testar serviços RESTful (Web APIs) por meio do envio de requisições HTTP e da análise do seu retorno.

Vamos selecionar a opção GET e passar "localhost:8080/clientes" que foi onde mapeamos na Classe ApiController e o resultado será 200 OK.

Nota-se que estão retornando os nossos dados que inserimos no banco de dados e com a formatação que fizemos para retornar tipo String.



Aplicação Postman GET

Precisamos agora criar uma classe que também vai ser um Dto mas para não ficarmos com classes e pacotes duplicados, vamos chamar de CadastroForm.

Essa Classe será para o Cliente efetuar o Cadastro de uma nova conta Bancária, terá todos os campos da classe Cliente exceto Id, pois o Banco de Dados já incrementa o Id.

```
package com.projetoDesafio.bancoApi.controller.form;

public class CadastroForm {

    private String nome;
    private String email;
    private Long cpf;
    private String dataNasc;

    public String getNome() {
        return nome;
    }
    public String getEmail() {
        return email;
    }
    public Long getCpf() {
        return cpf;
    }
    public String getDataNasc() {
        return dataNasc;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public void setCpf(Long cpf) {
        this.cpf = cpf;
    }
}
```

Classe CadastroForm do Pacote Form

Dentro da Classe Cliente vamos adicionar um método gerarClienteEntity, pois precisamos receber as informações e mandar para o Banco de Dados.

```
public static Cliente gerarClienteEntity(CadastroForm cadastroForm) {
    Cliente cliente = new Cliente();

    cliente.setCpf(cadastroForm.getCpf());
    cliente.setEmail(cadastroForm.getEmail());
    cliente.setNome(cadastroForm.getNome());
    try {
        cliente.setDataNasc(obterDataFormatada(cadastroForm.getDataNasc()));
    } catch (ParseException e) {
        e.printStackTrace();
    }

    return cliente;
}

private static Date obterDataFormatada(String data) throws ParseException {
    SimpleDateFormat sdf = new SimpleDateFormat("dd/MM/yyyy");
    return sdf.parse(data);
}
}
```

Classe Cliente com novo método

Vamos aproveitar e já criar a validação caso o cliente tente cadastrar um E-mail ou um CPF duplicado.

Vamos primeiro criar uma Classe Dto chamada CadastroDto dentro da classe Dto. Vamos adicionar os atributos para trazer o Status, mensagem e o corpo. Criando também os Getters e o Setters, ficará dessa forma

```
package com.projetoDesafio.bancoApi.controller.dto;

public class CadastroDto {

    private int statusCode;
    private String mensagem;
    private Object body;

    public int getStatusCode() {
        return statusCode;
    }
    public String getMensagem() {
        return mensagem;
    }
    public Object getBody() {
        return body;
    }
    public void setStatusCode(int statusCode) {
        this.statusCode = statusCode;
    }
    public void setMensagem(String mensagem) {
        this.mensagem = mensagem;
    }
    public void setBody(Object body) {
        this.body = body;
    }
}
```

Nova classe CadastroDto

Precisamos acessar o Repository e colocar a anotação @Query fornecida pelo JPQL. Criando o método buscarCpfEmailIguais

```
package com.projetoDesafio.bancoApi.repository;

import java.util.List;

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

import com.projetoDesafio.bancoApi.model.Cliente;

public interface CadastroRepository extends JpaRepository<Cliente, Long> {

    @Query(value = "SELECT c FROM Cliente c WHERE CPF = :cpf OR EMAIL = :email")
    public List<Cliente> buscarCpfEmailIguais(Long cpf, String email);

}
```

Classe CadastroRepository com uma nova notação

Agora vamos acessar o ClienteServices e jogar um throws Exception, para que possamos declarar que esse método é uma exceção de um determinado tipo. Vamos adicionar também a mensagem que queremos dar quando receber esse tipo de dado duplicado que será "CPF ou Email ja cadastrado".

```
10 import com.projetoDesafio.bancoApi.model.Cliente;
11 import com.projetoDesafio.bancoApi.repository.CadastroRepository;
12
13 @Service
14 public class ClienteServices {
15
16     @Autowired
17     private CadastroRepository cadastroRepository;
18
19     public List<ClienteDto> listarClientes() {
20         List<Cliente> cadastros = cadastroRepository.findAll();
21
22         return ClienteDto.converterList(cadastros);
23     }
24
25     public ClienteDto cadastrar(CadastroForm cadastroForm) throws Exception {
26         List<Cliente> cliente = cadastroRepository.buscarCpfEmailIguais(cadastroForm.getCpf(), cadastroForm.getEmail());
27
28         if (cliente.size() > 0) {
29             throw new Exception("CPF ou Email ja cadastrado.");
30         }
31
32         Cliente clienteEntity = cadastroRepository.save(Cliente.gerarClienteEntity(cadastroForm));
33
34         return ClienteDto.converter(clienteEntity);
35     }
36 }
37
38 }
```

Classe ClienteServices com throws Exception

Na Classe ApiController vamos adicionar a nossa requisição POST e também usar um Try Catch para o tratamento de exceções que ficará dessa forma

```
@PostMapping("cadastrar")
public ResponseEntity<CadastroDto> cadastrar (@RequestBody CadastroForm form){
    ClienteDto cliente = new ClienteDto();
    CadastroDto cadastroDto = new CadastroDto();

    try {
        cliente = clienteServices.cadastrar(form);

        cadastroDto.setStatusCode(HttpStatus.CREATED.value());
        cadastroDto.setMensagem(HttpStatus.CREATED.name());
        cadastroDto.setBody(cliente);
    } catch (Exception e) {
        cadastroDto.setStatusCode(HttpStatus.BAD_REQUEST.value());
        cadastroDto.setMensagem(e.getMessage());
    }

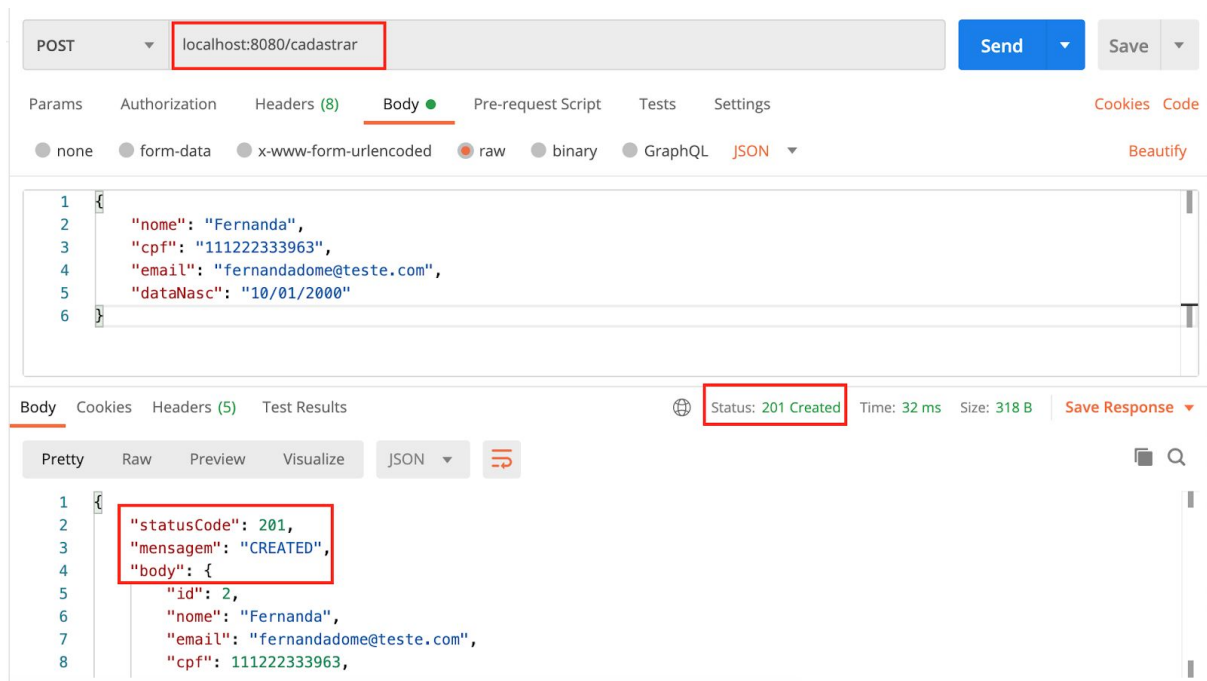
    return ResponseEntity.status(cadastroDto.getStatusCode()).body(cadastroDto);
}
```

Classe ApiController requisição POST

Nota-se que estamos utilizando o Bad_request, que é o código de status de resposta HTTP 400, indica que o servidor não pode ou não irá processar a requisição devido a alguma coisa que foi entendida como um erro do cliente, então pegamos o getMessage e mostramos o porque que a requisicao nao foi concluida com sucesso.

Vamos testar :)

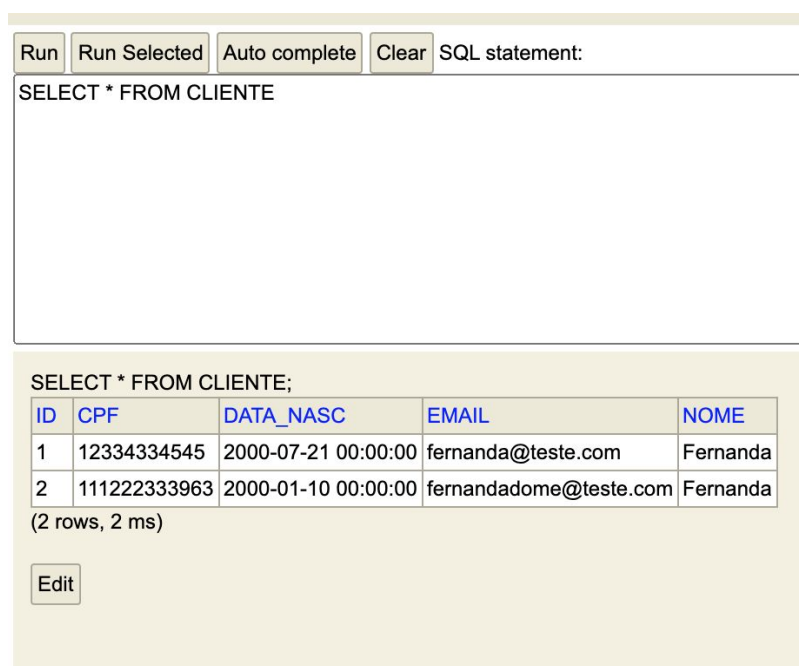
Entrando novamente no Postman, vamos para a Aba de POST e indicamos a Uri "Cadastro"



Postman requisição POST

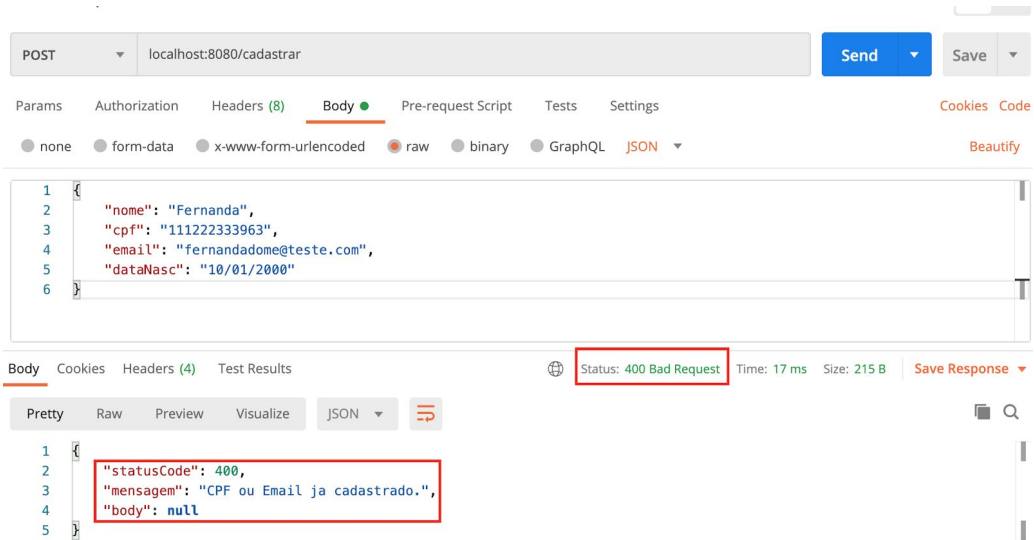
Verifique que trazemos o status **201 CREATED** juntamente com a mensagem para uma melhor visualização.

Visualizando também no Banco de Dados, vemos que foi cadastrado com sucesso



Banco de Dados após a requisição POST

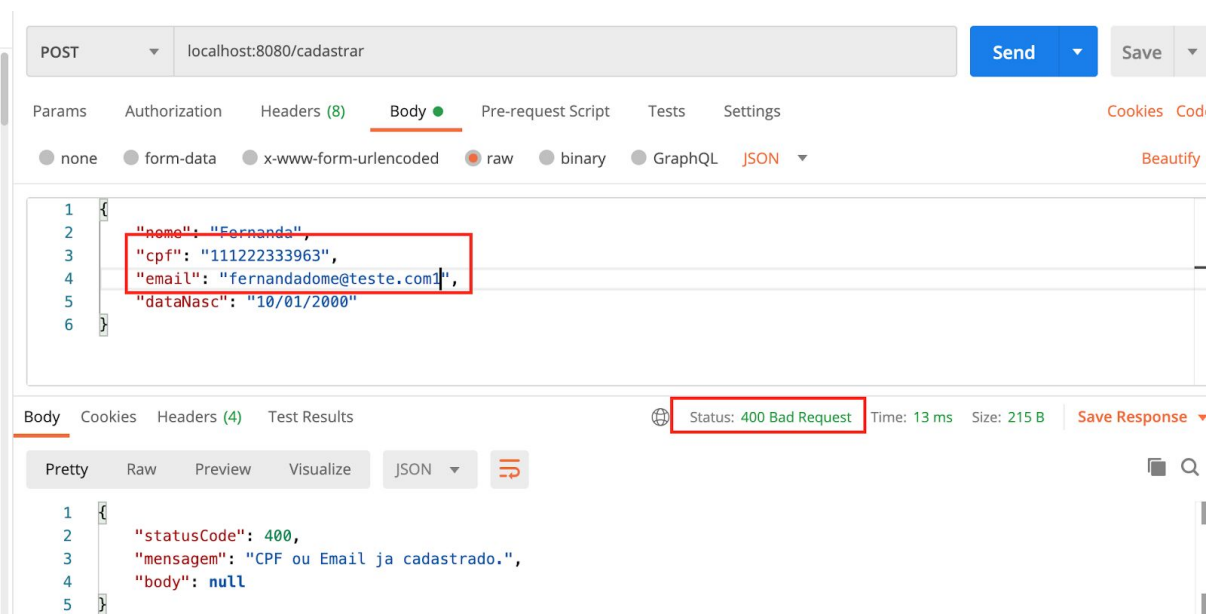
Testando agora a duplicidade dos campos CPF e Email, vamos dar novamente um "Send" para mandar a mesma Requisição e o resultado será:



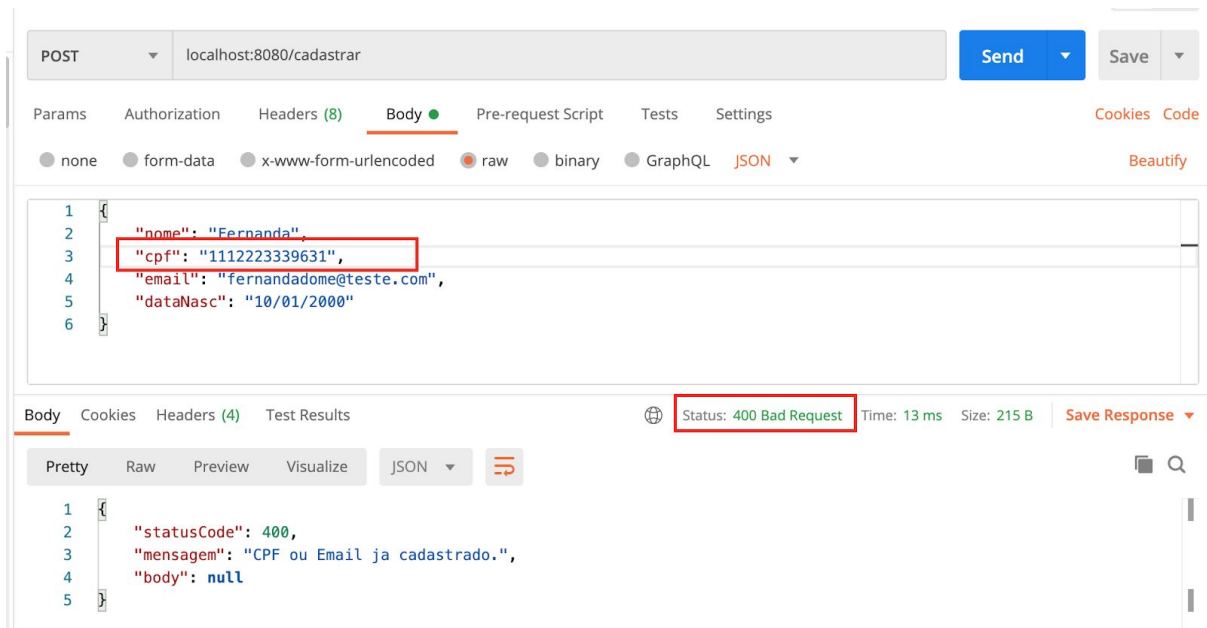
Postman Requisição POST com resultado 400

Isso mesmo! O retorno **400 BAD REQUEST**.

Vamos testar também o retorno trazendo só o CPF duplicado e logo em seguida só o Email duplicado.



CPF duplicado e Email não



Email duplicado e CPF não

Deu tudo certo!

Podemos também realizar outros tipos de validações como, por exemplo verificar se o CPF é válido ou Email é válido, e trazer mais segurança para a nossa aplicação.

Ferramentas Utilizadas nesse Projeto

- **Git:** sistema de controle de versão distribuído.
- **Maven:** Gerenciador de dependências
- **Tomcat:** Servidor de Aplicação web
- **Spring Initializr:** fornece uma interface web bem simples para o usuário, podendo gerar seu projeto a partir de uma estrutura de configurações pré-moldadas.
- **h2 Database:** Banco de Dados em memória
- **Postman:** Realizar Requisições http
- **Eclipse:** IDE

Conclusão

Vimos aqui como criar uma API REST com Spring Boot, entendemos um pouco melhor sobre as dependências que utilizamos neste projeto, como utilizar o banco de dados em memória e também como validar registros duplicados.

Minha programação ficou consideravelmente mais ágil depois do Spring Boot, criamos um serviço bem simples, mas que nos ajuda a compreender de forma rápida como podemos criar Web Services.

Espero que tenham gostado!

Obs: você pode baixar o código-fonte desse post no GitHub:

<https://github.com/nandadomenicali/ProjetoDesafio-API>