Início do projeto dia 23.12.20

Considerações iniciais::

Escreverei todo o artigo, em tempo real de desenvolvimento, ou seja, vou codificando, copiando as telas e escrevendo sobre o que acabei de fazer, inclusive na medida que for desenvolvendo e acontecer algum erro, deixarei escrito no artigo incluindo a solução que eu encontrar.

API Rest para abertura de uma nova conta em um banco

Nesse artigo iremos abordar além da criação do projeto, diversos conceitos importantes.

Tecnologia que iremos utilizar neste projeto

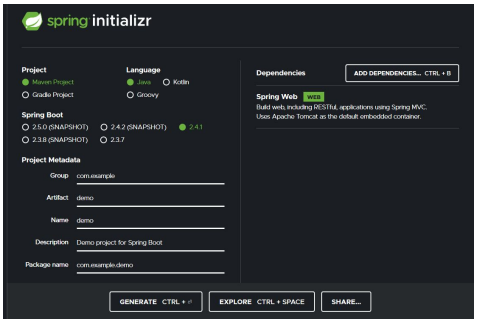
- Linguagem Java;
- Framework Spring : projeto spring boot e mais algumas dependências;
- Gerenciador de dependências Maven;
- Biblioteca Lombok para auxiliar nos códigos boilerplates
- Hibernate
- Para desenvolver iremos utilizar o IntelliJ, porém outra IDE muito boa é a STS (Um eclipse com um plus específico para spring boot).
- Postman - para testar nossa API

Um pouco de conceitos para entendermos o funcionamento do projeto antes de **#codar**.

- **Spring boot** - *infraestrutura do nosso projeto*
  - que facilita o processo de configuração inicial e publicação do nosso projeto, veremos duas formas de iniciar a nossa aplicação utilizando o spring boot.
  - A grande sacada do spring boot é reduzir e facilitar a inicialização de um novo projeto, fazendo com que a equipe consiga focar mais na regra de negócio da própria aplicação.
  - Importante ressaltar que mesmo o spring boot deixando tudo praticamente configurado para você, existe uma grande flexibilidade para que você possa assumir o controle na parte que achar necessária.

Como inicializar um projeto com Spring boot

Você pode fazer através da própria IDE IntelliJ, STS ou através do site Spring Initializr



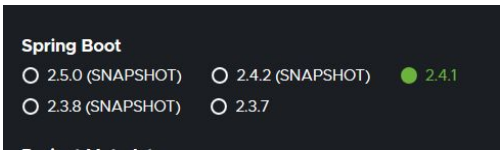
Acessando o site, poderemos configurar toda a parte inicial do nosso projeto.

Abaixo irei detalhar cada etapa da criação

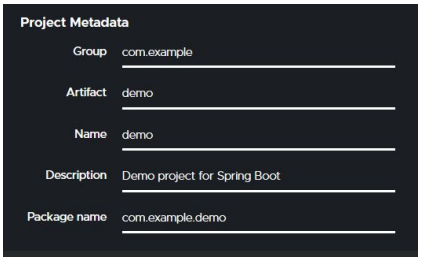
obs: iremos criar direto a partir da IDE Intellij.

Você inicia escolher o gerenciador de sua preferência, Maven ou Gradle.
No nosso projeto iremos utilizar Maven, por ser mais antigo e me sentir mais confortável, porém existem diversos artigos falando sobre o desempenho do Gradle ser melhor.

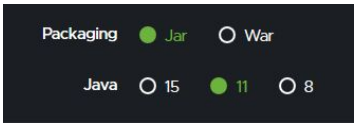
Nessa etapa você escolhe a linguagem que irá trabalhar, no nosso caso será Java.



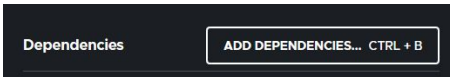
Aqui você pode escolher a versão do spring boot que deseja trabalhar, aconselho escolher a última versão sem o (SNAPSHORT) pois ela ainda está em desenvolvimento.



Group: grupo do projeto referenciado no pom.xml como groupId, geralmente usamos o domínio da empresa ao contrário ex: br.com.nomedaempresa;
Artifact: referenciado no pom.xml como artifactId, também infere o nome do projeto;
Name: Nome de exibição do projeto que também determina o nome do aplicativo;
Description: podemos fazer uma breve descrição sobre o projeto que está sendo construído;
Package name: nome do pacote raiz, se nao for especificado ele assume o nome do ***group***.



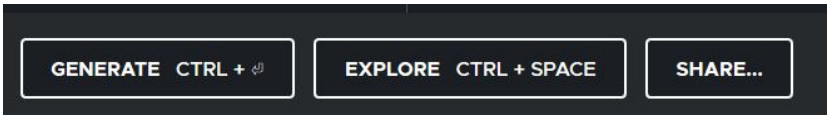
Nesta etapa você pode escolher entre o empacotamento Jar ou War, podemos diferenciar de forma mais superficial que o Jar aceita bibliotecas, classes e sozinho já roda toda sua aplicação, enquanto que o War leva apenas os seus arquivos WEB e você precisaria de um servidor instalado em seu computador para poder executar.
E na opção Java você pode escolher a versão que irá trabalhar, também aconselho utilizar, neste caso, a versão 11, pois é a última LTS - Long Term Support.



Aqui, escolhemos as dependências que iremos precisar utilizar no nosso projeto

obs: na medida que formos utilizar as dependências eu irei explicar cada uma delas.

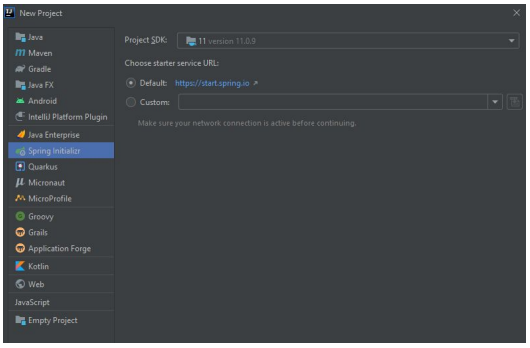
Para finalizar, podemos escolher gerar um pacote com o ***GENERATE.*** Explorar essas dependências vendo como ficará o projeto ***EXPLORE*** ou ainda você pode fazer toda a sua configuração e gerar um link para enviar para outro desenvolvedor ou equipe utilizando o botão ***SHARE***



Agora que você já sabe como iniciar uma aplicação vamos dar continuidade iniciando o nosso desenvolvimento.

para o projeto, irei a cada passo fazer um commit para o seguinte repositório
https://github.com/eduardoalbuquerque/cadastrobancoapi.git

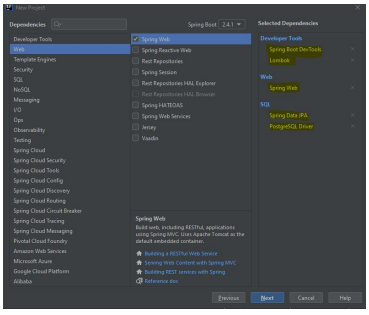
A segunda maneira é criar a estrutura inicial com suas dependências utilizando o spring boot na própria IDE
//file - new - project



Não irei me aprofundar muito nas telas da IDE porque segue o mesmo raciocínio do site explicado anteriormente, coloquei a imagem apenas para que se possa ter uma ideia da tela da IDE.

Configurações do projeto

- ***Group :*** *com.orangetalents*
- ***Artifact :*** *orangetalents*
- ***Name :*** *orangetalents*
- ***Description :*** *Cadastro de usuários de um banco*
- ***Package:*** *com.orangetalents.apicadastro*
- ***Type:*** *Maven* | ***Language:*** *Java* | ***Packaging:*** *Jar* |***Java version:*** *11*



Dependências do Projeto

Spring Boot DevTool : Ele cria cache e melhora a performance quando precisamos reiniciar o servidor para que as alterações surtam efeitos, porém noto que na IDE IntelliJ não é tão constante quanto quando usado no STS (posso está equivocado);

Lombok : irá nos auxiliar nos boilerplates (com configurações simples os getters e setters, assim como o Equals, hashcode, tostring, serão gerenciados por ele;

Spring Web : essencial para aplicação web, traz toda a estrutura embutida inclusive o padrão MVC;

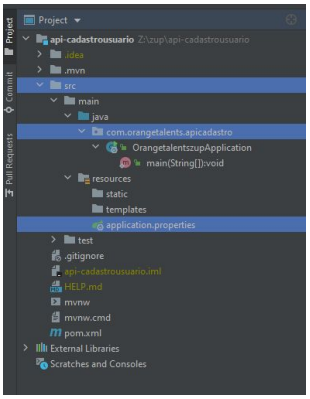
Spring Data JPA : dependência para trabalharmos com a persistência de dados, é uma abstração da JDBC;

PostgreSQL Driver : Iremos preparar uma conexão com o banco de dados PostgreSql, e este é o drive para permitir essa comunicação.

Para finalizar essa primeira parte usaremos o github para subir nosso projeto inicial

git init : cria repositório local
git add . : adiciona as alterações
git commit -m "started our project with basic structure" : comitou localmente
git remote add origin https://github.com/eduardoalbuquerque/cadastrobancoapi.git : cadastrando repositório do github (remoto)
git push -u origin main: mandando as informações locais para o remoto (cópia)

Estrutura inicial de pastas



Após criar o projeto pela IDE ou pelo site, e nesse último caso, importar para dentro da IDE, temos a seguinte estrutura de pastas, como mostra na figura ao lado.
Para facilitar nosso entendimento, deixei destacado em azul as principais que iremos trabalhar.

src : pasta raiz e dentro dela vai está a estrutura do nosso código fonte;

com.orangetalents.apicadastro : nome do primeiro package, definimos isso no Package quando estávamos criando a aplicação;

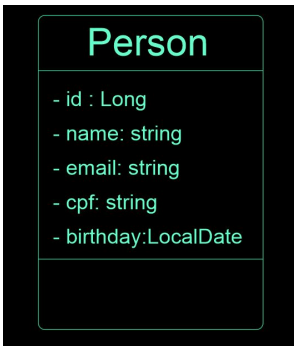
application.properties : um arquivo de configurações, iremos usar ele nesse tutorial para colocar as configurações de acesso ao banco de dados;

pom.xml : arquivo que não marquei mas também é muito importante, ele é o local onde fica referenciada todas as nossas dependências.

Criando a nossa classe

conceito de classe : A classe em POO é um modelo que representa a abstração que fazemos de um assunto, esse assunto pode ser um objeto da vida real ou uma ação como por exemplo pagarconta. No nosso caso para representar uma pessoa irei criar a classe Person escrita em inglês (padrão de desenvolvimento)
obs: As classes no Java devem iniciar sempre com primeira letra maiúsculas e em caso de palavras composta usar o padrão CamelCase.

Atributos*: os atributos de uma classe representa as suas características criadas a partir da abstração que fazemos do assunto dentro de um contexto, por exemplo, no nosso caso usaremos nome, email, cpf e data de aniversário, porém se essa mesma classe Person fosse para representar um aluno, poderíamos ter outros atributos com curso, turma e etc.*
obs: para os atributos devemos iniciar com a primeira letra em minúsculo e em caso de palavras compostas a partida da segunda palavra utilizar a primeira letra em maiúsculo. Ex: numeroFichaCadastral.



class : Person
attributes :
id - (long) que será a chave primária e receberá um valor sequencia gerenciado pelo próprio banco de dados;

name - (string) usaremos um tamanho máximo para 100 caracteres e um mínimo de 10 já que para cadastro bancário deveremos ter um nome completo;
obs: sempre procuro identificar os requisitos de forma única para evitar consumo de espaço desnecessário. em uma busca rápida pela internet encontrei pessoas com mais de 800 letras assim com um registro de uma pessoa com apenas uma letra (um pai minimalista hehehe), mas são exceções.

email - (string) tamanho máximo de 60 e mínimo 10;

cpf - (string), por que neste caso poderemos guardar inclusive a máscara do CPF e outra justificativa é que já que não se fará operações aritméticas com o CPF então fica como string.

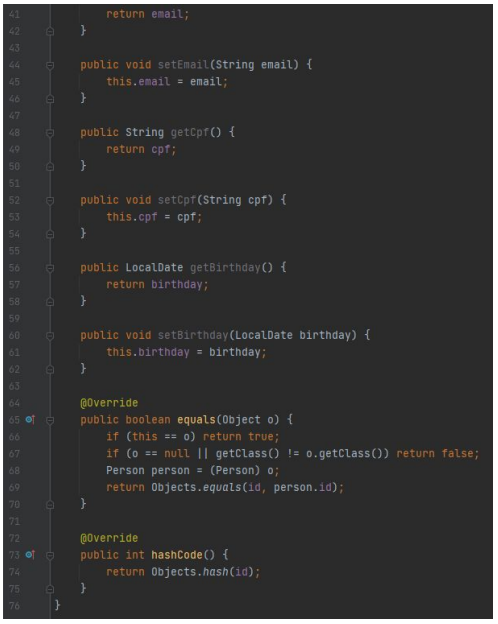
birthday - (LocalDate) que veio na versão do Java 8, se não me engano, e é uma API bem completa para lidar com datas

Criando a classe Person na IDE

Iremos criar a class ***Person*** dentro de um package domain

O código abaixo sem o lombok. Notem que chegamos fácil em 76 linhas, então além de ter que digitar muito mais código, ainda corremos o risco de errar em algum momento, além de perder produtividade.

```java
package com.orangetalents.apicadastro.domain;

import java.time.LocalDate;
import java.util.Objects;

public class Person {
    private Long id;
    private String name;
    private String email;
    private String cpf;
    private LocalDate birthday;

    public Person() {
    }

    public Person(Long id, String name, String email, String cpf, LocalDate birthday) {
        this.id = id;
        this.name = name;
        this.email = email;
        this.cpf = cpf;
        this.birthday = birthday;
    }

    public Long getId() {
        return id;
    }

    public void setId(Long id) {
        this.id = id;
    }

    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getEmail() {
```



```java
        return email;
    }

    public void setEmail(String email) {
        this.email = email;
    }

    public String getCpf() {
        return cpf;
    }

    public void setCpf(String cpf) {
        this.cpf = cpf;
    }

    public LocalDate getBirthday() {
        return birthday;
    }

    public void setBirthday(LocalDate birthday) {
        this.birthday = birthday;
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        Person person = (Person) o;
        return Objects.equals(id, person.id);
    }

    @Override
    public int hashCode() {
        return Objects.hash(id);
    }
}
```

Agora veja o mesmo código depois de aplicar o lombok

```java
package com.orangetalents.apicadastro.domain;

import lombok.Data;

import java.time.LocalDate;

@Data
public class Person {
    private Long id;
    private String name;
    private String email;
    private String cpf;
    private LocalDate birthday;
}
```

menos de 15 linhas, claro que iremos ainda acrescentar algumas linhas antes de finalizar nosso projeto, o intuito até agora foi apenas mostrar quanto podemos economizar em linhas de códigos “padrões” e com isso ganhar um pouco mais de produtividades, mesma a IDE nos ajudando com os códigos boilerplates, mas daí temos também outra vantagem que vejo é que fica um código mais limpo.
Obs: você pode também alterar o comportamento do Lombok como por exemplo escolher qual propriedade será utilizada no hashcode, já que por padrão ele seleciona todas, e não precisamos de todas as propriedades sejam comparadas já temos um ID único na entidade.

Para isso usaremos a seguinte annotation do lombok:

```java
@Data
@EqualsAndHashCode(onlyExplicitlyIncluded = true)
@Entity
public class Person {

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

A anotação ***@Data*** já deixa pronto para gente os getters,setters,equals, Hashcode, tostring e construtor com argumento.
@EqualsAndHashCode(onlyExplicitlyIncluded = true) Iremos usar para informar que é para incluir o Equals somente os atributos que estiverem anotados com o Include.
Acima do atributo que que queremos incluir deveremos colocar a notação ***@EqualAndHashcode.Include***

observação importante*: escolhi usar o lombok para mostrar como ganhar produtividade, e vamos concordar que no projeto tão pequeno como esse nem precisaria e ainda existem diversos artigos na internet que não aconselha o uso do mesmo em projetos maiores, as principais justificativas são que o java evolui muito rápido e o lombok nem tanto, o que pode gerar algumas inconsistências e quando ocorre um erro, como o lombok não explicita o código que ele gerencia, as vezes fica difícil encontrar onde está o problema. o que posso falar é que usem com responsabilidade e em projetos maiores seria mais interessante codar tudo que precisa realmente. Ex: get quando precisar de get e set quando precisar de set.*

Hora de atualizar o nosso repositório

~~*git init : cria repositório local*~~ *uma vez criado não preciso mais usar esse comando.*
git add .
git commit -m “created a Person class using lombok”
~~*git remote add origin https://github.com/eduardoalbuquerque/cadastrobancoapi.git : cadastrando repositório do github (remoto)*~~
~~*git push -u origin main: mandando as informações locais para o remoto (cópia)*~~
git push

Continuando na nossa Person class

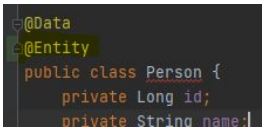
Agora iremos configurar mais algumas coisas para que tudo funcione

- para que essa classe se torne uma entidade no meu banco de dados (que ainda não foi criado) preciso colocar uma anotação antes do nome da classe:

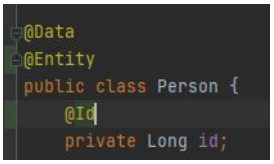
  Contextualizando: para conseguirmos fazer uma relação entre objetos da linguagem com o paradigma OO para um banco de dados com outro paradigma, no caso o relacional, ou seja, existir uma conversar entre dois paradigmas diferentes o ORM nos ajudar de forma muito eficiente. com poucas anotações nós conseguimos informar que uma classe java irá ser persistida no banco em forma de tabela (entidade) e cada atributo será uma campo e cada objeto será uma tupla (linha).

  Então iremos utilizar as seguinte anotações:

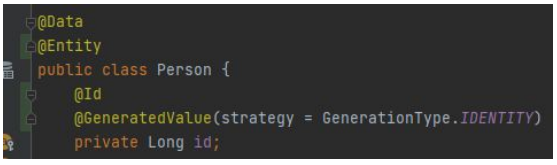
  @Entity, para informar que uma classe será uma entidade no banco de dados



  Colocamos a anotação antes da classe e notamos que a própria IDE nos mostra um erro, sublinhado de vermelho o nome da classe. Isso acontece porque quando informamos que uma classe será uma entidade no banco, precisamos informar também qual atributo da classe será a chave primária (pk), por tanto iremos colocar mais uma anotação em cima do atributo que será a (pk) no banco.



  de forma bem sugestiva, basta colocar a anotação ***@Id*** antes do atributo, no nosso caso, id.
  porém ainda podemos configurar mais uma coisa bem interessante, vamos informar que além de ser uma chave primária, queremos que o campo id no banco seja auto incremento, ou seja, que o banco atribua valores sequenciais a cada tentativa de inclusão de um novo registro, e quando falo tentativa, é porque mesmo que ocorra um erro, dependendo do momento, e o registro não seja gravado com sucesso, essa sequência pode ter sido incrementada.



  @generatedValue informa ao banco que será um campo auto incremento. obs: já li em outros artigos que para o postgres é necessário fazer outro tipo de configuração, porém usando essa estratégia conseguir gerar o campo auto incremento sem maiores problemas.

  para finalizar essas configurações iniciais do JPA iremos apenas informar mais algumas características para criação da entidade no banco

  As anotações @Column utilizada antes dos atributos permite ao desenvolvedor, definir algumas configurações das entidades criadas.

```java
@Data
@Entity
public class Person {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    @Column(nullable = false, length = 100)
    private String name;
    @Column(nullable = false, length = 60)
    private String email;
    @Column(nullable = false, length = 14)
    private String cpf;
    private LocalDate birthday;
}
```

No projeto iremos utilizar ***nullable = false*** que obrigar o campo no banco ser preenchido e a propriedade ***length*** informa o tamanho do campo no banco. obs: apesar dessa configuração nos trazer um pouco de restrição, é muito limitada quando se trata de entrada de dados, por exemplo, não tenho como verificar se um email tem um formato válido ou se o cpf é válido, para isso utilizaremos outros recurso que falaremos mais na frente.

Hora de atualizar o nosso repositório com implementação da ORM

git add .
git commit -m “using Hibernate for persistence Person class in to the Postgres ”
git push

Será que roda?

Se tentarmos rodar o programa nesse momento irá gerar um erro, por enquanto fácil de entender ;-)))

```
***************************
APPLICATION FAILED TO START
***************************

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class
```

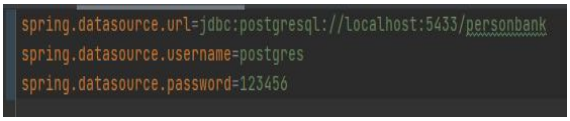
Como nós colocamos a dependência JPA no nosso projeto, precisamos configurar o acesso ao banco, *mas que banco?* , bom, nós ainda não criamos, então iremos no próximo passo criar o banco e as configurações de acesso para que possamos rodar nossa aplicação e ver o que irá acontecer

Criando nosso banco no postgresql de maneira default

No cenário onde o banco ainda não existe, iremos criar um para podermos testar nossa API. No meu caso, como tenho o postgresql instalado em minha máquina, usarei o pgAdmin para criar de forma default o banco,

```
CREATE DATABASE personbank
    WITH
    OWNER = postgres
    ENCODING = 'UTF8'
    CONNECTION LIMIT = -1;
```

Uma vez criado o banco e sem criar nenhuma tabela ainda. Pelo nosso projeto iremos configurar para que a própria aplicação crie quando necessário nossas entidades no banco. para isso iremos utilizar um arquivo chamado ***aplication.properties*** que se encontra dentro da pasta resources e dentro dele colocaremos as seguintes configurações:



essas três linha já irá garantir nossa aplicação rodar tranquilamente

url : informe qual o banco, porta de acesso e nome do banco que será acessado;

username : nome do usuário do banco (no nosso caso é o usuário padrão)

password : senha, que nesse caso é a 123456 (bem forte por sinal) ;-)))

Porém, apesar do sistema rodar tranquilamente, você irá notar que nenhuma tabela foi criada no banco, isso porque não informamos dentro do arquivo que precisamos que seja criada uma tabela caso ela não exista. Vamos lá.

colocando o comando ***spring.jap.hibernante.dll-auto***, será ofertada algumas opções:

create : irá criar o banco sempre que a aplicação for inicializada, nesse caso apaga sempre todas as informações sempre que você inicializar o projeto, porém, quando você finaliza sua aplicação, o que foi criado permanece persistido no banco até o momento de reinicialização da aplicação novamente, mais utilizando quando você está iniciando um projeto e está modificando muito suas classes e você precisa sempre de um banco novo e limpo para testar;

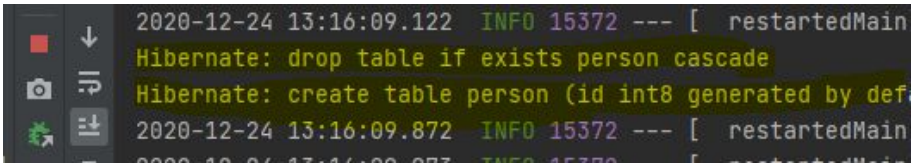
update, é irá apenas atualizar seu banco quando for encontrada alguma mudança em suas classes;

create-drop : parecido com o create, o create-drop ele apagar o banco sempre que a aplicação finalizar;

none: bem sugestivo, não irá alterar seu banco, mesmo que exista mudanças em suas classes;

validation : com o banco já criado, se houver alguma alteração entre banco - classes é gerado um erro na hora de rodar a aplicação

durante o desenvolvimento eu gosto de deixar essa opção como true para poder acompanhar os comando SQL executados pelo Hibernate como mostrado no exemplo abaixo

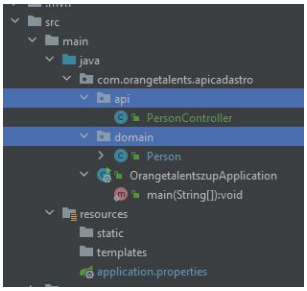


O próximo passo

Agora iremos criar o nosso controller, que será exatamente a camada que terá comunicação com o nosso cliente, seja um site, um aplicativo mobile ou até mesmo outra API
Iremos criar um package ***api*** e dentro desse pacote criaremos nossa classe ***PersonController.***

obs: o motivo de criar um novo pacote é para que os arquivos possam ficar organizados de acordo com suas respectivas camadas.

Temos a seguinte estrutura até o momento.



Na classe PersonController dentro do pacote api nós iremos utilizar as seguintes annotation

@RestController - essa anotação mostra o retorno de um método, nesse caso evita eu ter que colocar em cada método a annotation ***@ResponseBody***;

@RequestMapping("/people") - essa anotação é um endpoint da minha API é um endereço que será respondido de alguma forma, neste nosso caso usaremos "/people" para representar Person.
Obs: o endereço colocamos people por que devemos utilizar o plural para nossos end points.

Nosso primeiro método será para salvar no banco um Person, entretanto no ponto que nós estamos no projeto, isso ainda não será possível, o que farei agora é criar apenas um teste para saber se minha aplicação está rodando sem erros.

Iremos criar agora um método que retorne uma string, meramente um teste que costumo fazer.

```
@RestController
@RequestMapping("/people")
public class PersonController {


    public String hello(){
        return "Hello!";
    }

}
```

Nossa classe ***PersonControler*** está pronta, porém ainda não nos dará nenhum retorno, para que realmente funcione eu preciso anotar esse método com algum verbo http.

Nós temos 8 verbos , sendo 5 os mais utilizados.

get - buscar recursos na api;

post - gravar recursos na api;

put - atualizar de forma completa um recurso na api;

patch - atualizar de forma parcial um recurso na api;

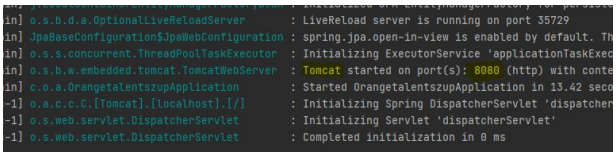
delete - deletar um recurso na api;

logo, para retornar um simples texto usaremos o verbo http GET, para tanto, basta colocar a seguinte annotation ***@Getmapping*** antes do método que retornará algum recurso.
obs: caso não seja especificado nenhum endereço adicional, esse método retornará no endereço /people

Obs: como estou desenvolvendo e preparando esse material, poderá ocorrer algum erro e eu tenho que refatorar algum trecho do código e para evitar de ficar modificando o material irei mostrar também o erro e a refatoração.

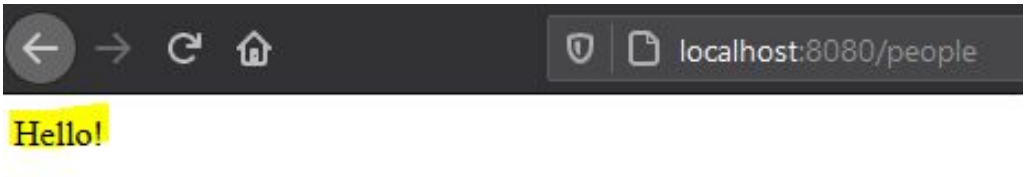
Um ponto que eu gostaria de alterar é a questão do endereço, que eu coloquei /people, porém por questão de versionamento iremos adicionar o "***/api/v1/people***" no ***@RequesttMapping*** da classe. ok? ;-)))

##boratestar!



*Ok, o serviço subiu perfeitamente e uma coisa que eu ainda não tinha falado é que por padrão o tomcat inicia na porta **8080**.*

para testar vou utilizar um browser, no meu caso o firefox, para ver o resultado



Notem o endereço que coloquei na url ***localhost:8080/people*** devolve uma resposta que é exatamente o retorno do método da ***PersonController***

Aparentemente tudo funcionando, vou atualizar o repositório agora e partiremos para a próxima etapa

Hora de atualizar o nosso repositório com implementação da ORM

git add .
git commit -m "created layer controller with the PersonController included"
git push

Construindo o end point para salvar a pessoa

http://localhost:8080/people

Criamos então um método no controle que sua ação é salvar a **person** e retornar a própria **person** salva.
Para isso precisamos colocar a anotação ***@PostMapping***, para informar que iremos utilizar o verbo POST, explicado anteriormente, e como não passamos nenhum parâmetro será entendido que no end point "/people" passando um corpo cairá nesse método.
Outra anotação que foi colocada à ***@RequestBody*** que faz de forma automática a deserialização do que está chegando do cliente em JSon para o tipo em Java.
Obs: o Json enviado pelo cliente deverá ter o mesmo formato que estamos esperando.

Implementar a camada de repositório aproveitando todos os benefícios da Jpa.

Criando nosso repository

A primeira coisa que iremos fazer é criar um pacote dentro da pasta domain chamado repository, em seguida criar uma interface com o nome ***PersonRepository***. O sufixo repository é opcional, porém ajuda a gente saber de cara do que se trata.

Essa parte do código é bastante simples, porém bem poderosa. o que nós fizemos foi estender nossa interface da JpaRepository que abstrai a complexidade do CRUD principalmente nos fornecendo um acesso facilitado, e é claro que ainda podemos modificar ou customizar outras consultas ao banco.
obs: JpaRepository<Person,Long> onde Person é a entidade na qual o JPA vai atuar e Long o tipo do campo que representa o ID na tabela de referência ao objeto;

Uma vez implementando essa interface e a estendendo-a, vamos voltar ao nosso controller e injetá-la para podermos utilizar.

Voltando ao nosso controller, criamos um ***atributo do tipo_da_interface*** criada anteriormente e colocar com o nome de ***repository***, pode ser qualquer nome, geralmente é colocado, o mesmo nome da interface (personRepository) mas prefiro o nome reduzido, já que nesse contexto só terei esse repository.

Criaremos um construtor para instanciar esse atributo, daí neste ponto poderíamos fazer de algumas maneiras:

- 1 - colocar acima do próprio atributo a anotação ***@Autowired***;
- 2 - colocar acima do método **setNomeDoAtributo** a anotação ***@Autowired***
- 3 - criar um construtor para instanciar esse atributo e colocar ou não a anotação ***@Autowired*** acima do construtor, A anotação não é necessário se tivermos apenas um construtor na classe.

Usaremos no nosso projeto a injeção no construtor pelos seguintes motivos:

- posso deixar meu atributo imutável
- eu garanto que sempre que a classe for instanciada obrigatoriamente será instanciada a minha dependência, ao contrário de se colocar no set que deixa de forma opcional
- e se deixar no próprio atributo se essa classe for instanciada em outro local poderá gerar um nullpointerexception

obs: poderia utilizar o lombok também nesse contexto, bastava colocar antes da classe a seguinte annotation ***@AllArgsConstruct*** *, que daí seria gerado automaticamente um construtor com atributos*

Bom, depois de criado o ***repository*** e colocando no nosso controller como um atributo chegou a hora de utilizá-lo.

```java
@RestController
@RequestMapping("/api/v1/people")
public class PersonController {

    private final PersonRepository repository;

    public PersonController(PersonRepository repository){
        this.repository=repository;
    }

    @PostMapping
    public Person save(@RequestBody Person person){
        return repository.save(person);
    }

}
```

o que foi feito na classe ***Personcontroller***:

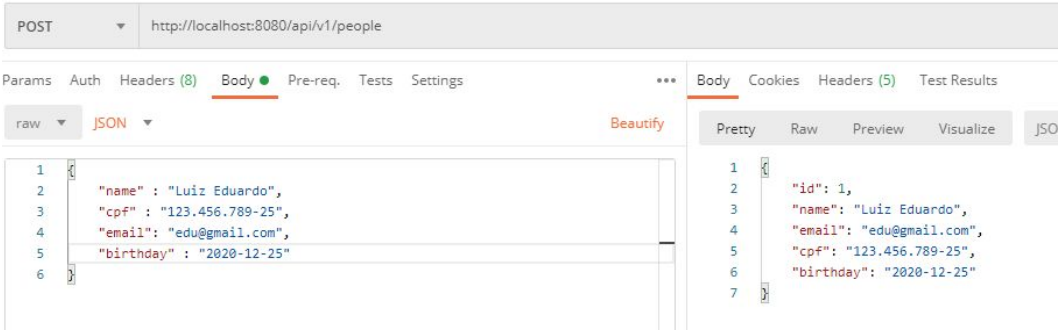
Alterei o atributo para final;

Criei um construtor para instanciar o meu atributo repository

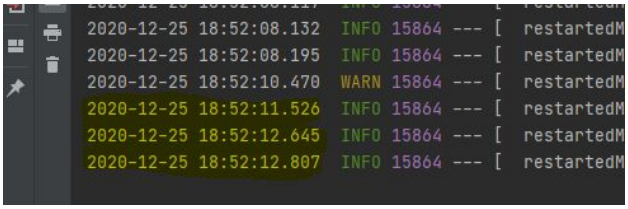
Utilizei um método que já está pronto na interface do JpaRepository que é o ***save***.

obs: Iremos utilizar agora o aplicativo postman para testar nossa API
link: https://www.postman.com/

No corpo iremos passar o seguinte json para testar a inclusão do recurso



Aplicação rodando de boa :-)))



Notem na imagem acima que foi passado o json com os dados da minha classe de domínio Person e o método da ***PersonController*** me retornou os dados persistidos no banco. Ao lado a imagem do aplicativo rodando.

Tudo funcionando às mil maravilhas.

Hora de atualizar o nosso repositório com a implementação da camada repository e o save já funcionando.

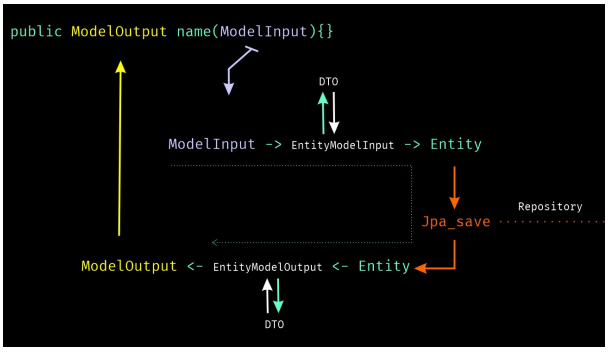
git add .
git commit -m "created layer repository with a interface personController and a method save"
git push

Implementado o DTO

Agora iremos implementar o DTO - Data Transfer Object, que é um padrão de projeto onde, explicando de maneira simples, é onde iremos transportar dados entre os objetos. No nosso caso iremos criar uma classe para representar a entrada de dados e uma classe para retornar a saída dos dados.

E porque utilizar DTO?, com o padrão DTO a gente tem um controle refinado do que pode entrar e o que pode ou deve ser mostrado como retorno da nossa API.
Quando você retorna a própria classe de domínio, você poderá está expondo informações sensíveis da sua entidade ou informações desnecessárias para quem consome a API.
Ex: Imagine você retornar uma entidade que tenha CPF ou senha de usuário, isso não é interessante, um outro motivo bem legal é que você pode tratar os dados nessa classe para só depois persistir ou exibir.

Fluxo dos dados quando salvamos Person



Os dados que você irá receber será do tipo ***ModelInput***, e dentro desta classe iremos validar algumas informações e só depois passar para a classe de domínio e salvar. Como o método save retorna a própria entidade de domínio, nesse momento iremos transformá-la para ***ModelOutput*** e devolver como resposta para o cliente.

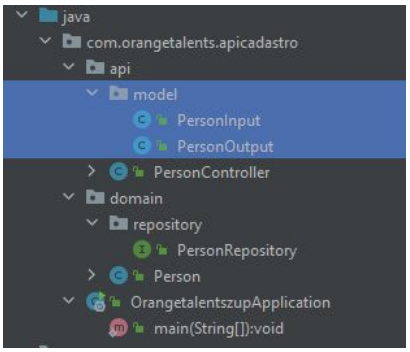
É comum utilizar uma camada service com o seguinte fluxo ***controller -> service -> repository -> service -> controller***, Os services ou camada de serviço, tem algumas funções como por exemplo enviar email, sms, entre outras coisas, ou ainda se eu precisar de um desacoplamento maior do ***repository*** e camada ***controller***. No nosso projeto não iremos utilizar por não achar necessário nesse cenário.

agora @boracodar!!!

Iremos criar um package model dentro da pasta API, acredito que muita gente use de forma diferente, mas fico confortável em colocar os modelos de entrada e saída juntos dos controlles.

Em seguida iremos criar as classes ***PersonInput*** e ***PersonOutput***, que representarão as entradas e saídas dos dados na nossa API, criaremos também uma classe ***PersonDTO*** para realizar as tranferência de dados entre os objetos **PersonInput | Person | PersonOutput.**

Nossa estrutura agora está dessa forma:



Neste nosso contexto a classe ***PersonInput*** será praticamente igual a classe de domínio com exceção do ID.

obs: Irei utilizar o lombok novamente nesta classe apenas para mostrar que podemos utilizar apenas para os Getters e/ou Setters.

NOTA: geralmente uso o lombok em TUDO ou em NADA, estou alternando nesse projeto apenas para mostrar que podemos utilizar parte dele ou até mesmo não usar, dentro de um mesmo projeto.

A classe ficou como está na figura ao lado, o próximo passo agora é fazer o controller receber uma ***PersonInput*** e não a ***Person*** propriamente dita. para isso trocamos o tipo de dados na entrada do método, como mostro na imagem abaixo:

```
@Getter
@Setter
public class PersonInput {

    private String name;
    private String email;
    private String cpf;
    private LocalDate birthday;

}
```

```
@PostMapping
public Person save(@RequestBody PersonInput personInput){
    return repository.save(personInput);
}
```

Podemos observar um erro no ***repository***, afinal ele espera receber um ***Person*** e estamos passando agora um ***PersonInput***.

Vamos agora resolver esse conflito de interesses :-))))

Existe diversas maneiras de resolver isso, porém, vou mostrar uma maneira mais “raiz” como a turma gosta de chamar, poderia também utilizar uma biblioteca que nos ajudaria nessa tarefa repetitiva como o ***ModelMapper*** ou ***MapStruct.***

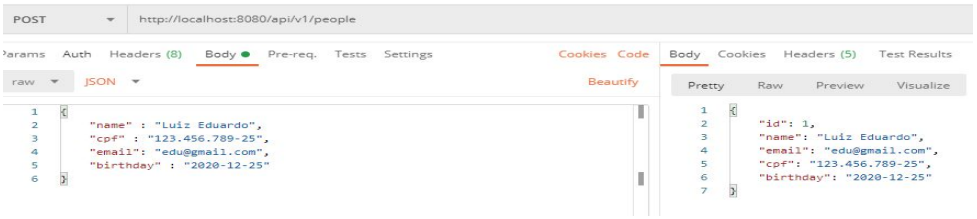
obs: não irei utilizar neste projeto por se tratar de um projeto pequeno.

```
@PostMapping
public Person save(@RequestBody PersonInput personInput){
    Person person = new Person();
    person.setName(personInput.getName());
    person.setCpf(personInput.getCpf());
    person.setEmail(personInput.getEmail());
    person.setBirthday(personInput.getBirthday());
    return repository.save(person);
}
```

Instanciei uma nova ***Person*** de forma manual e fui setando os atributos um a um, em seguida mando salvar, neste caso é bem trabalhoso e até então não estou verificado nenhuma informação que está vindo do cliente.

Notem que a aplicação roda tranquilamente e eu consigo cadastrar uma nova ***Person*** no banco, utilizando o PostMan

```
2020-12-26 10:02:34.504  INFO 13880 --- [nio-8080-exec-1] o.a.c.c.C.[
2020-12-26 10:02:34.504  INFO 13880 --- [nio-8080-exec-1] o.s.web.serv
2020-12-26 10:02:34.505  INFO 13880 --- [nio-8080-exec-1] o.s.web.serv
Hibernate: insert into person (birthday, cpf, email, name) values (?,
```



*Nossa classe **PersonController** ficou um pouco suja, vocês não acham? então vamos dar uma refatorada básica, para deixar o nosso código mais elegante:*

```
@PostMapping
public Person save(@RequestBody PersonInput personInput){
    Person person = new Person();
    person.setName(personInput.getName());
    person.setCpf(personInput.getCpf());
    person.setEmail(personInput.getEmail());
    person.setBirthday(personInput.getBirthday());
    return repository.save(person);
}
```

Antes estávamos com a classe dessa forma
e agora deixamos igual a imagem abaixo.
Não se preocupem que irei explicar o que foi feito ;-)))

```
@PostMapping
public PersonOutput save(@RequestBody PersonInput personInput){
    return personDTO.toPersonOutput(repository.save(personDTO.toPerson(personInput)));
}
```

Passo a passo

- para resolver ***PersonInput*** para ***Person*** que é a forma que poderemos salvar no banco utilizei o seguinte método da classe ***PersonDTO***

```
public Person toPerson(PersonInput personInput){
    Person person = new Person();
    person.setName( personInput.getName());
    person.setCpf(personInput.getCpf());
    person.setEmail(personInput.getEmail());
    person.setBirthday(personInput.getBirthday());
    return person;
}
```

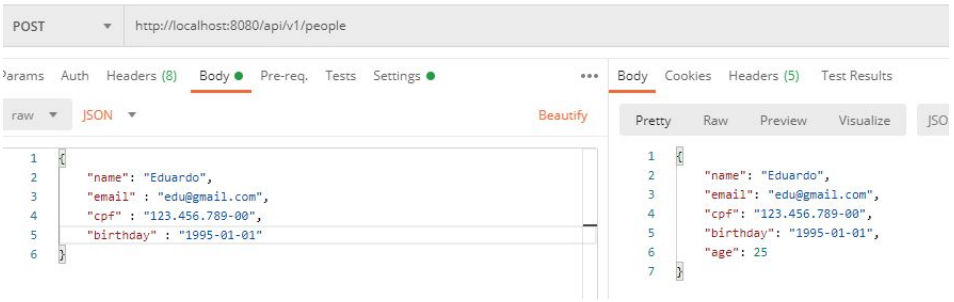
O método recebe um ***PersonInput*** e nos devolve um ***Person***
notem que quando chamo o ***repository.save*** passo como parâmetro o método ***toPerson*** que já me retorna uma ***Person***, dessa forma consigo salvar sem problemas.

- Agora precisamos resolver o retorno da função do ***PersonController*** que espera retornar um ***PersonOutput***, para isso criei outro método dentro da mesma classe ***PersonDTO*** para fazer essa transformação

```
public PersonOutput toPersonOutput (Person person){
    PersonOutput personOutput = new PersonOutput();

    personOutput.setName(person.getName());
    personOutput.setEmail(person.getEmail());
    personOutput.setCpf(person.getCpf());
    personOutput.setBirthday(person.getBirthday());
    personOutput.setAge(Period.between(person.getBirthday(),LocalDate.now()).getYears());


    return personOutput;
}
```

Aproveitei para incrementar com mais um atributo que mostra a idade do cliente, para mostrar que a saída dos dados pode ser formatada do jeitinho que a gente precisar.

Dessa forma, a entrada continua sendo padrão e a saída da seguinte forma:



Código completo da classe ***PersonDTO***

```java
@Component
public class PersonDTO {

    public Person toPerson(PersonInput personInput){
        Person person = new Person();
        person.setName( personInput.getName());
        person.setCpf(personInput.getCpf());
        person.setEmail(personInput.getEmail());
        person.setBirthday(personInput.getBirthday());
        return person;
    }

    public PersonOutput toPersonOutput (Person person){
        PersonOutput personOutput = new PersonOutput();

        personOutput.setName(person.getName());
        personOutput.setEmail(person.getEmail());
        personOutput.setCpf(person.getCpf());
        personOutput.setBirthday(person.getBirthday());
        personOutput.setAge(Period.between(person.getBirthday(),LocalDate.now()).getYears());

        return personOutput;
    }
}
```

Uma observação importante é que coloquei ***@Component*** para informar ao spring que essa será uma classe gerenciada pelo próprio spring. dessa forma poderemos usar a DI em outros pontos do nosso sistema

As classes ***PersonInput*** e ***PersonOutput***

```java
@Getter
@Setter
public class PersonInput {

    private String name;
    private String email;
    private String cpf;
    private LocalDate birthday;

}
```

```java
@Setter
@Getter
public class PersonOutput {

    private String name;
    private String email;
    private String cpf;
    private LocalDate birthday;
    private Integer age;

}
```

Utilizando o beanvalidation

vamos pensar um pouco na regra de negócio, temos os seguintes campos com as seguintes regras:
Obs: regras foram criadas por mim de forma arbitrária, apenas para fazer mais sentido o que iremos fazer.

name : nome do usuário que não pode ser nula e deve ter no mínimo 10 caracteres e no máximo 100. *obs: o máximo de 100 caracteres porque criei minha entidade no banco com espaço máximo para 100 caracteres(página 4)*
cpf : campo não pode ser nulo ou vazio e deve ser verificado se um cpf válido.
email : campo não pode ser nulo ou vazio e deve ser verificado se é válido.
birthday : campo não pode ser nulo ou vazio e deve ser verificado se é válido e se a data não é no futuro.

Usaremos a anotação ***@Valid*** (especificação bean validation) no parâmetro do método controller, automaticamente ele irá pegar os parâmetros que estamos recebendo e irá aplicar as validações da bean validation que foram setadas na classe em questão.

importante*: Na versão mais nova do spring o beanvalidation não vem mais dentro do startweb, então é necessário colocar uma dependência no pom.xml, e agora para validation funcionar foi necessário também colocar a anotação* ***@Validated*** *na classe* ***PersonController*** *para indicar que a classe em questão terá validações (pelo menos foi assim que entendi) ;-)))*

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-validation</artifactId>
    <version>2.4.1</version>
</dependency>
```

Outra coisa importante que fez eu passar a manhã toda pesquisando foi que mesmo com a dependência, só funcionou quando inserir a linha de ***<version>2.4.1</version>*** *Sem a versão não funcionou.*

Hora de atualizar o nosso repositório com a implementação da camada repository e o save já funcionando.

git add .
git commit -m “created layer DTO with a In and Out to entity convert”
git push

Validação dos campos de entrada da API

O primeiro campo será o ***name*** para que ele não aceita null nem vazio iremos utilizar o ***@NotNull*** e o ***@Size(min=10,Max=100)*** como mostra na imagem abaixo:

```
@NotNull
@Size(min = 10, max = 100)
private String name;
```

Além de colocar essas notações nos atributos da classe ***PersonInput*** não podemos esquecer de colocar no controle a anotação ***@Validated*** antes da classe e ***@Valid*** dentro do parâmetro do método que irá ser validado como falei anteriormente.

vamos testar antes de continuar:

```
{
    "name": "Eduardo",
    "email" : "edu@gmail.com",
    "cpf" : "123.456.789-00",
    "birthday" : "1995-01-01"
}
```

Passa esse Json como entrada teremos o seguinte erro

```
'trace": "org.springframework.web.bind.MethodArgumentNotValidException: Validation failed for argument
    [0] in public com.orangetalents.apicadastro.api.model.PersonOutput com.orangetalents.apicadastro.api.
    PersonController.save(com.orangetalents.apicadastro.api.model.PersonInput): [Field error in object
    'personInput' on field 'name': rejected value [Eduardo]; codes [Size.personInput.name,Size.name,Size.
    java.lang.String,Size]; arguments [org.springframework.context.support.
    DefaultMessageSourceResolvable: codes [personInput.name,name]; arguments []; default message [name],
    100,10]; default message [O nome tem que ter no mínimo 10 e no máximo 100 caracteres]] \r\n\tat org.
    springframework.web.servlet.mvc.method.annotation.RequestResponseBodyMethodProcessor.resolveArgument
    (RequestResponseBodyMethodProcessor.java:139)\r\n\tat org.springframework.web.method.support.
```

podemos observar pelo destaque que foi referente ao tamanho do campo nome que foi passado apenas 7 letras e no mínimo precisamos de 10 letras

raw JSON

```
{
    "name": "Luiz Eduardo",
    "email" : "edu@gmail.com",
    "cpf" : "123.456.789-00",
    "birthday" : "1995-01-01"
}
```

Alterando para ***Luiz Eduardo*** passa tranquilamente, e para esse artigo não virá um livro, já ficou claro que validação é uma forma de proteger a borda do seu sistema, nesse caso a borda mais externa (controller),

Setando as outras validações:

```
@Getter
@Setter
public class PersonInput {

    @NotNull
    @Size(min = 10, max = 100)
    private String name;
    @NotBlank
    @Email
    private String email;
    @NotBlank
    @CPF
    private String cpf;
    @Past
    private LocalDate birthday;

}
```

@NotNull : Não aceita campos nulos;
@Sice(min=10,max=100) : Tamanho min/max caracteres aceito;
@NotBlank : Não aceita campos nulos nem em branco;
@Email : Verificar se segue o padrão dos email, porém não temos como garantir que o mesmo exista;
@CPF : Verifica se o CPF é válido
@Past : Apenas permite datas até o dia anterior ao corrente

obs: dessa forma conseguimos validar a entradas de dados de acordo com a regra

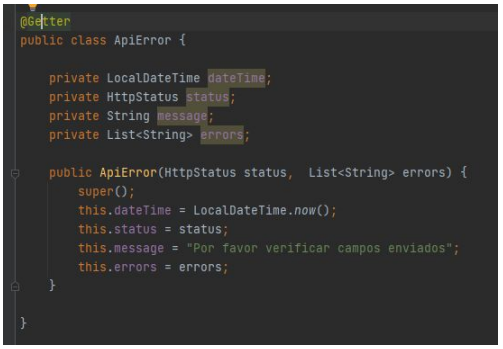
Como mostrar uma mensagem amigável para o consumidor

Quando tudo dá certo na inclusão de um recurso a mensagem que devemos mostrar é o ***201 - Created***, isso já está acontecendo, agora vamos verificar as possíveis mensagens de retorno que teremos em nossa API

Para tratar o retorno mais adequado para as validações, iremos criar uma classe chamada ***ApiExceptionHandler*** (poderia ser qualquer nome) e anotá-la com ***@ControllerAdvice***

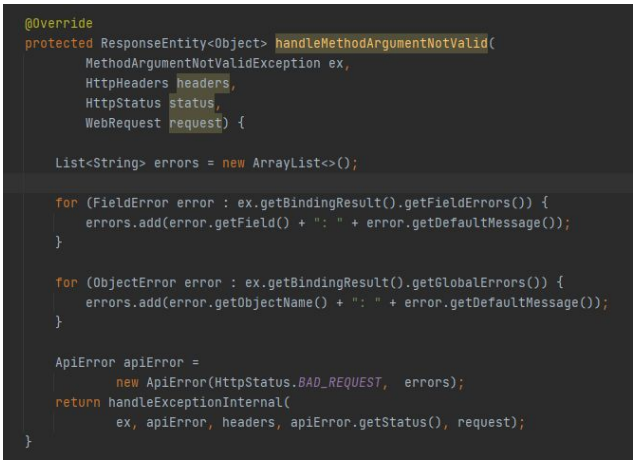
O ***@ControllerAdvice*** informa ao Spring que os erros dos controllers serão tratados pela classe que tiver esta annotation

Outra coisa que devemos fazer é estender a classe ***ApiExceptionHandler*** da ***ResponseEntityExceptionHandler***, para fornecer tratamento de forma centralizadas das exceptions



```
@Getter
public class ApiError {

    private LocalDateTime dateTime;
    private HttpStatus status;
    private String message;
    private List<String> errors;

    public ApiError(HttpStatus status,  List<String> errors) {
        super();
        this.dateTime = LocalDateTime.now();
        this.status = status;
        this.message = "Por favor verificar campos enviados";
        this.errors = errors;
    }

}
```

Criamos uma outra classe com alguns atributos apenas para receber nossas configurações personalizadas e retornar para o cliente a partir da classe ***ApiExceptionHandler***
acrescentamos dentro dos atributos um campo datetime para mostrar junto com a mensagem o momento da exceção.



```
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(
        MethodArgumentNotValidException ex,
        HttpHeaders headers,
        HttpStatus status,
        WebRequest request) {

    List<String> errors = new ArrayList<>();

    for (FieldError error : ex.getBindingResult().getFieldErrors()) {
        errors.add(error.getField() + ": " + error.getDefaultMessage());
    }

    for (ObjectError error : ex.getBindingResult().getGlobalErrors()) {
        errors.add(error.getObjectName() + ": " + error.getDefaultMessage());
    }

    ApiError apiError =
            new ApiError(HttpStatus.BAD_REQUEST,  errors);
    return handleExceptionInternal(
            ex, apiError, headers, apiError.getStatus(), request);
}
```

agora voltando para a classe ***ApiExceptionHandler:***

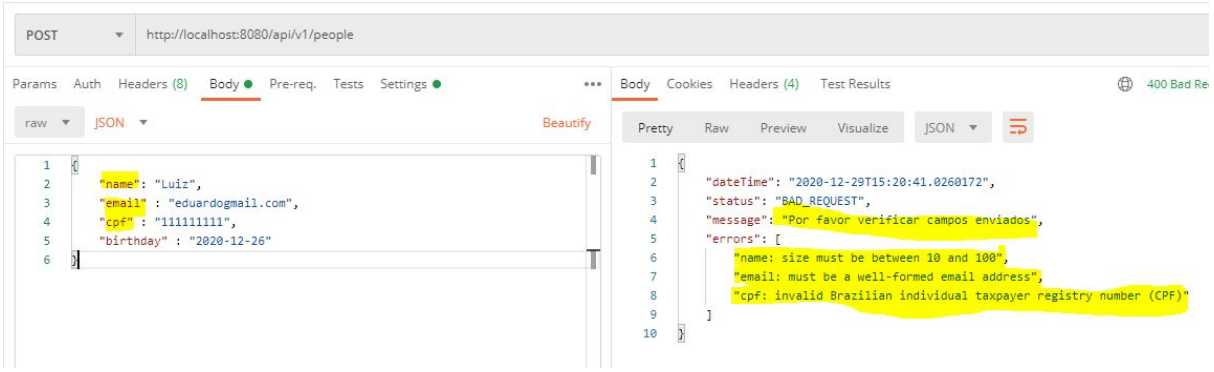
Sobrescrevemos o método que trata das validações ***handleMethodArgumentNotValid***

Criamos uma lista de String chamada ***erros*** para colocar os possíveis erros de validação.

usamos o ***BindingResult()*** que é um controlador de erros da exception ***MethodArgumentNotValidException*** e com isso percorremos o array e colocamos na lista ***errors*** as possíveis falhas de validação

instanciamos a classe ***ApiError*** criada anteriormente, passando o status e os erros encontrados

E finalmente retornamos um ***handleExceptionInternal***, que recebe as especificações alteradas.
como resultado teremos o seguinte cenário:



Como podemos notar na resposta para o cliente, é mostrado a data, o status, a mensagem e uma lista de erros que estão impedindo a execução correta do end point.

obs: A verificação é feita antes de salvar no banco, logo a verificação de duplicidade só ocorre se passar em todos os testes de validação;

Verificando CPF e Emails duplicados

Essa verificação foi feita na própria classe ***PersonController.***

```java
@Repository
public interface PersonRepository extends JpaRepository<Person, Long> {

    Optional<Person> findByCpf(String cpf);
    Optional<Person> findByEmail(String email);
}
```

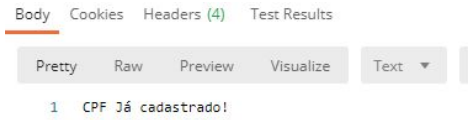
criei dois métodos, um para buscar o CPF no banco e outro para buscar o email.

```java
@PostMapping
public ResponseEntity<?> save(@Valid @RequestBody PersonInput personInput){

    if(!repository.findByCpf(personInput.getCpf()).isEmpty()){
        throw new CampoDuplicadoException("CPF Já cadastrado!");
    }

    if(!repository.findByEmail(personInput.getEmail()).isEmpty()){
        throw new CampoDuplicadoException("Email Já cadastrado!");
    }

    PersonOutput output= personDTO.toPersonOutput(repository.save(personDTO.toPerson(personInput)));
    return new ResponseEntity<>(output,HttpStatus.CREATED);

}
```

Verifico a existência do CPF e/ou email na base de dados com um If simples, disparando um erro quando encontrado qualquer duplicidade, evitando dessa forma salvar CPF ou emails repetidos.

Para o tratamento do erro disparado quando encontramos um CPF ou email duplicado, criaremos uma classe específica e a chamaremos de ***CampoDuplicadoException***. estendendo a mesma de ***RuntimeException*** utilizando apenas um construtor para enviar a mensagem para a super.

```java
public class CampoDuplicadoException extends RuntimeException{

    public CampoDuplicadoException(String message) {
        super(message);
    }

}
```



porém para que essa classe seja chamada iremos colocar dentro da classe ***ApiExceptionHandler*** o seguinte método:

```java
@ExceptionHandler(CampoDuplicadoException.class)
public ResponseEntity<?> handlerCPFDuplicado(CampoDuplicadoException ex){

    return ResponseEntity.status(HttpStatus.BAD_REQUEST)
            .body(ex.getMessage());
}
```

Anotado com ***@ExceptionHandler(nome_da_classe_que_ele_irá_tratar)*** e retornaremos um ResposeEntity com o status e a mensagem

Hora de atualizar o nosso repositório com tudo funcionando.

git add .
git commit -m “created exceptions and validations”
git push

Considerações finais::

Quero agradecer a oportunidade de participar desse projeto, e deixar meu depoimento de que foi um momento de aprendizado sem igual, tive que me esforçar muito para fazer o que sabia e estudar o que ainda não dominava e com isso o benefício que me trouxe foi único. Parabéns a equipe que criou esse processo seletivo, estou aprendendo mesmo antes de ser selecionado.

“in memoriam ao meu pai, que perdeu a vida para o COVID (23-12-20) no meio desse projeto, mas deixou uma frase que sempre usei, “filho, primeiro a obrigação antes de qualquer outra coisa” isso me motivou ainda mais a finalizar esse desafio da melhor maneira possível”