

Autor: Jean Pandolfi

Antes de iniciarmos esse primeiro blog post do ano, queria desejar-lhes um 2021 com muita saúde, paz e sucesso.

Hoje vamos criar uma API REST em Spring para cadastrar os dados pessoais de uma pessoa. Para isso precisamos de apenas algumas informações obrigatórias:

- Nome
- E-mail
- CPF
- Data de nascimento

Como dependências do projeto vamos utilizar:

- **Spring Web** para criar os controllers REST para acesso a nossa API e que inclui o Apache Tomcat como servidor embutido.
- **MySQL Driver** para conectarmos ao banco de dados.
- **Spring Data JPA** para acesso ao banco de dados e que possui interfaces que facilitam muito a manipulação dos dados.
- **Validation** para podermos realizar as validações necessárias dos dados de entrada.

Eu gerei o projeto pelo <https://start.spring.io/> e assim ficou as propriedades:

The screenshot shows the Spring Initializr configuration page. On the left, under 'Project', 'Maven Project' is selected. Under 'Language', 'Java' is selected with version '8'. Under 'Spring Boot', version '2.3.7' is selected. The 'Project Metadata' section includes fields for Group (com.jeanpandolfi), Artifact (cadastros), Name (cadastros), Description (API para cadastros de pessoas), and Package name (com.jeanpandolfi.cadastros). Packaging is set to 'Jar'. On the right, the 'Dependencies' section lists 'Spring Web' (WEB), 'MySQL Driver' (SQL), 'Spring Data JPA' (SQL), and 'Validation' (I/O). At the bottom, there are buttons for 'GENERATE CTRL + G', 'EXPLORE CTRL + SPACE', and 'SHARE...'.

Apenas troquei a versão do Spring Boot para 2.3.5.RELEASE, por ser uma versão release.

Como opção vou deixar um docker-compose.yml para levantarmos um banco de dados MySQL com Docker, mas você pode criar um banco no servidor instalado na sua máquina.

Após abrir o projeto em sua IDE favorita vamos criar um pacote de *pessoa*, já que queremos realizar um cadastro de Pessoas.

Precisamos definir um controller para expor a API e recebermos as requisições dos nossos clientes, para isso vamos criar uma classe *CadastroPessoaController* e será mapeado como */api/cadastrar-pessoa*. Assim ficou nossa classe.

```
@RestController
@RequestMapping(value = "/api/cadastrar-pessoa")
public class CadastroPessoaController {

}
```

Agora, para recebermos os dados de cadastro da pessoa, vamos criar um método POST que recebe esses dados através de um objeto *NovaPessoaRequest*. Como esses dados vão vir pelo body da requisição precisamos anotar o parâmetro como *@RequestBody*.

Objeto de entrada:

Vamos criar a classe *NovaPessoaRequest* que será responsável por representar o objeto de entrada da requisição na nossa API. Nela vamos colocar os atributos que iremos receber, citado anteriormente: Nome, e-mail, CPF e data de nascimento.

```
import java.time.LocalDate;

public class NovaPessoaRequest {

    private String nome;

    private String email;

    private String cpf;

    private LocalDate dataNascimento;

}
```

Como esses dados são obrigatórios, precisamos validar esses dados de entrada, para isso vamos utilizar as anotações do *javax.validation*.

```

@NotBlank
private String nome;

@NotBlank
@email
private String email;

@NotBlank
@CPF
private String cpf;

@NotNull
@Past
private LocalDate dataNascimento;

```

As anotações `@NotBlank`, `@NotNull`, `@Past` e `@Email` são fornecidas pela API do Jakarta Bean Validation que faz as validações.

- `@NotNull` verifica se o campo anotado é nulo.
- `@NotBlank` verifica se o campo anotado está em branco.
- `@Past` verifica se o campo anotado que tem que ser uma data, está no passado, isso é válido por ser uma data de nascimento em que ela tem que está no passado.
- `@Email` verifica se o campo anotado tem o formato de email.

A anotação `@CPF` é fornecida pelo `HibernateValidator` e verifica se o campo anotado é um CPF válido. Como CPF é uma identificação específica do Brasil você pode observar que ela vem do pacote `br` como mostrado.

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

Para conseguir validar esses atributos na entrada da requisição, ainda precisamos anotar o parâmetro com `@Valid` ficando nessa forma.

```

@PostMapping
ResponseBody<Void> cadastrarDadosPessoais(@RequestBody @Valid NovaPessoaRequest novaPessoaRequest){
    return null;
}

```

Em nossa classe vamos adicionar os Getters para todos os atributos para o Jackson conseguir serializar o Json vindo da requisição para um objeto Java.

Estamos prontos para executar nossa aplicação, mas antes vamos colocar a configuração do banco de dados senão o Spring lançará um erro. Fica dessa forma.

```
#INFORMAÇÕES DE CONEXÃO COM BANCO
spring.datasource.url=jdbc:mysql://localhost/cadastros?useTimezone=true&serverTimezone=UTC
spring.datasource.username=root
spring.datasource.password=root
spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver
spring.jpa.properties.hibernate.dialect = org.hibernate.dialect.MySQL5Dialect

# INDICA PARA O HIBERNATE GERAR AS TABELAS E
# MUDA AS ESTRUTURAS DE ACORDO COM AS ENTIDADES FORNECIDAS
spring.jpa.hibernate.ddl-auto=update

#INDICA PARA O HIBERNATE FORMATAR O SQL
spring.jpa.properties.hibernate.format_sql=true

#INDICA PARA JPA GERAR OS SQL DE DEFINIÇÃO DE DADOS
spring.jpa.generate-ddl=true

#INDICA PARA JPA MOSTAR O SQL
spring.jpa.show-sql=true
```

Agora podemos executar a aplicação e para testarmos podemos utilizar o Insomnia. Montei a requisição com o seguinte body para saber se as validações irão ocorrer.

```
{
  "nome": "",
  "email": "",
  "cpf": "",
  "dataNascimento": ""
}
```

Ao enviar a requisição a aplicação irá retornar um erro com status HTTP 400, mas com poucas informações. Isso ocorre pois o Spring não mostra por padrão os erros a menos que seja exposto de forma explícita no código (que é o que vamos fazer posteriormente) isso reduz o risco de vaziar [informações indesejadas](#) para um cliente. Para mostrar essas informações de erro é necessário adicionar as propriedades no application.properties:

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

```
server.error.include-binding-errors=always
```

Indicam inclusão da mensagem e erros de ligação, respectivamente.

Ao reiniciar a aplicação podemos ver os erros lançados. Como a mensagem é grande coloquei em um arquivo separado que pode ser visto clicando [aqui](#).

Podemos observar que todos os as validações funcionaram e que atributos com duas anotações como CPF é gerado os dois erros, de campo vazio e de campo inválido.

Tratando erros

Agora que temos todos os dados dos erros, precisamos tratá-los para apresentar melhor para nosso cliente. Para isso o Spring possui a anotação [@RestControllerAdvice](#) que

podemos adicionar a uma classe que interceptará todas as exceptions lançadas e a classe abstrata [ResponseEntityExceptionHandler](#) que fornece vários métodos que capturam uma exception específica. Então vamos criar um pacote separado chamado `erros`, e dentro dele a classe `RestExceptionHandler` que entenderá de `ResponseEntityExceptionHandler` e será anotada com `@RestControllerAdvice`.

Toda vez que falha a validação de um argumento anotado com `@Valid` como o nosso parâmetro no nosso controller é lançada a exception `MethodArgumentNotValidException` então precisamos de criar um método que intercepta exatamente essa exception, para nossa sorte a `ResponseEntityExceptionHandler` já possui esse método então basta sobrescrevê-lo.

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

A partir daí já conseguimos pegar o erro, tratá-lo e retornar o que desejamos. Para isso temos que criar uma classe que represente nosso erro assim como Spring faz, podemos tomar como base o objeto lançado por ele e criar um objeto que define a hora da requisição, o endpoint acessado, uma lista de erros globais da requisição, uma lista indicando cada campo, qual erro está sobre ele e uma mensagem. Vamos definir isso em nosso código.

Classe que representa os campos que geram erros:

```
public class FielErrorOutputDTO {
    private String field;
    private String message;
    private Object rejectedValue;
}
```

Classe que representa o objeto de erro:

```
public class ValidationErrorsOutputDTO {

    private String path;
    private LocalDateTime timestamp;
    private List<String> globalErrorsMessages = new ArrayList<>();
    private List<FielErrorOutputDTO> fieldErrors = new ArrayList<>();

    public void addGlobalError(String message) { this.globalErrorsMessages.add(message); }

    public void addFieldError(String field, String message, Object parameter){
        FielErrorOutputDTO fielErrorOutputDTO = new FielErrorOutputDTO(field, message, parameter);
        fieldErrors.add(fielErrorOutputDTO);
    }
}
```

Além dos campos que falamos anteriormente, temos dois métodos para adicionar erros globais e campos de erros.

Agora que temos definido nosso objeto de erro vamos construí-lo para ser retornado. Vamos pegar os erros globais e os dos campos através da exception e temos que pegar apenas os campos que nos interessa e precisamos de gerar uma mensagem de erro para adicionar nos campos. Para a mensagem de erro podemos adicionar as mensagens que queremos dentro de um arquivo *messages.properties* no diretório do application e vamos pegar essas mensagens através da interface *MessageSource* que será injetada em nossa classe.

```
@Override
protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
                                                                HttpHeaders headers, HttpStatus status, WebRequest request) {
    List<ObjectError> globalErrors = ex.getBindingResult().getGlobalErrors();
    List<FieldError> fieldErrors = ex.getBindingResult().getFieldErrors();

    ValidationErrorsOutput validationErrorsOutput = buildValidationErrors(globalErrors, fieldErrors);
    validationErrorsOutput.setPath(request.getDescription(Boolean.FALSE));

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

```
private ValidationErrorsOutput buildValidationErrors(List<ObjectError> globalErrors, List<FieldError> fieldErrors) {
    ValidationErrorsOutput validationErrors = new ValidationErrorsOutput();

    globalErrors.forEach( error -> validationErrors.addGlobalError(getErrorMessage(error)));

    fieldErrors.forEach(error -> {
        String errorMessage = getErrorMessage(error);
        validationErrors.addFieldError(error.getField(), errorMessage, error.getRejectedValue());
    });
    return validationErrors;
}
```

```
private String getErrorMessage(ObjectError error) {
    return messageSource.getMessage(error, LocaleContextHolder.getLocale());
}
```

O método *buildValidationErrors* cria um objeto e preenche os listas com base na que veio da requisição. O método *getErrorMessage* parece misterioso, mas ele pega o código de erro do campo e verifica se ele está no *messages.properties* caso tiver ele retorna a mensagem definida senão retorna a *defaultMessage*. Por exemplo, pegamos o código de erro da data de nascimento e atribuímos uma mensagem:

```
NotNull.novaPessoaRequest.dataNascimento = Data de Nascimento não pode ser nula
```

Anteriormente ela tinha cometido este erro.


```
{
  "codes": [
    "NotNull.novaPessoaRequest.dataNascimento",
    "NotNull.dataNascimento",
    "NotNull"
  ],
  "arguments": [
    {
      "codes": [
        "novaPessoaRequest.dataNascimento",
        "dataNascimento"
      ],
      "arguments": null,
      "defaultMessage": "dataNascimento",
      "code": "dataNascimento"
    }
  ],
  "defaultMessage": "não deve ser nulo",
  "objectName": "novaPessoaRequest",
  "field": "dataNascimento",
  "rejectedValue": null,
  "bindingFailure": false,
  "code": "NotNull"
}
```

Agora após o tratamento dos erros temos a seguinte resposta:

```
{
  "field": "dataNascimento",
  "message": "Data de Nascimento não pode ser nula.",
  "rejectValue": null
},
```

Observe que sempre estamos retornando o status 400 Bad Request, já que foi realizada uma requisição mal feita. Veja agora o resultado final da request.

400 Bad Request

538 ms

434 B

Preview ▾

Header 4

Cookie

Timeline

```

1 {
2   "path": "uri=/api/cadastrar-pessoa",
3   "timestamp": "2021-01-03T21:23:48.227",
4   "globalErrorsMessages": [],
5   "fieldErrors": [
6     {
7       "field": "cpf",
8       "message": "não deve estar em branco",
9       "rejectValue": null
10    },
11   ],
12   {
13     "field": "email",
14     "message": "não deve estar em branco",
15     "rejectValue": null
16   },
17   {
18     "field": "dataNascimento",
19     "message": "Data de Nascimento não pode ser nula",
20     "rejectValue": null
21   },
22   {
23     "field": "nome",
24     "message": "não deve estar em branco",
25     "rejectValue": null
26   }
27 ]

```

Criando nossa entidade e persistindo-a

Precisamos agora transformar nosso objeto de entrada em um objeto que será persistido no banco de dados, mas vamos antes criar nossa entidade que terá os mesmos dados da request porém com o identificador.

```
@Entity
@Table(name = "TB_PESSOA")
public class Pessoa {

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

    @NotBlank
    @Column(nullable = false)
    private String nome;

    @NotBlank
    @Email
    @Column(nullable = false, unique = true)
    private String email;

    @NotBlank
    @CPF
    @Column(nullable = false, unique = true, length = 14)
    private String cpf;

    @NotNull
    @Past
    @Column(nullable = false)
    private LocalDate dataNascimento;
```

Além das verificações que já tínhamos foram colocadas algumas a nível de banco, para o MySQL criar as tabelas com as constraints necessárias. Essas validações foram colocadas novamente para garantir que nenhum dado chegue inconsistente ao banco de dados e que ela independe de outras classes para funcionar corretamente.

O método criado dentro da classe de request para transformar-lo para a entidade:

```
public Pessoa toEntity(){
    return new Pessoa(this.nome, this.email, this.cpf, this.dataNascimento);
}
```


Nosso controller fica assim;

```
@PostMapping
ResponseEntity<Void> cadastrarDadosPessoais(@RequestBody @Valid NovaPessoaRequest novaPessoaRequest){
    Pessoa pessoa = novaPessoaRequest.toEntity();
```

Agora é só persistir o objeto, para isso vamos criar uma interface que será um repositório Spring Data JPA que poderá manipular os objetos.

```
public interface PessoaRepository extends JpaRepository<Pessoa, Long> {
}
```

Com apenas essa interface já é possível salvar um objeto. Basta injetá-la em nosso controller e utilizar o método save que retornará o objeto persistido com o ID gerado pelo banco.

Por ser um método POST vamos retornar um corpo vazio mas porém indicando o local onde se encontra o objeto criado ou seja o caminho que se deve acessar para buscá-lo. Para isso vamos setar o Header Location com o caminho que será definido através de uma URI que será criada pegando a URI corrente e concatenando com o ID da pessoa salva. Veja como ficou nosso método finalizado.

```
@PostMapping
ResponseEntity<Void> cadastrarDadosPessoais(@RequestBody @Valid NovaPessoaRequest novaPessoaRequest){
    Pessoa pessoa = novaPessoaRequest.toEntity();
    Pessoa pessoaSalva = pessoaRepository.save(novaPessoaRequest.toEntity());

    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri().path("/{id}")
        .buildAndExpand( pessoaSalva.getId() ).toUri();

    return ResponseEntity.created(uri).build();
}
```

Ao testar nosso método pelo Insomnia veja a resposta:

POST http://localhost:8080/api/cadastrar-pessoa Send		201 Created	543 ms	0 B	Just Now	
JSON		Auth	Query	Header	Docs	Preview
1 { 2 "nome": "João", 3 "email": "joão@gmail.com", 4 "cpf": "05862173048", 5 "dataNascimento": "2000-10-12" 6 }				Header		
				Cookie		
				Timeline		
				NAME		
				VALUE		
				Location		
				http://localhost:8080/api/cadastrar-pessoa/1		
				Content-Length		
				0		
				Date		
				Mon, 04 Jan 2021 01:03:43 GMT		

Ele indica o status 401 Created e preencheu o Header Location com a URI a ser acessada. Agora vamos criar um método que retorne o objeto criado.

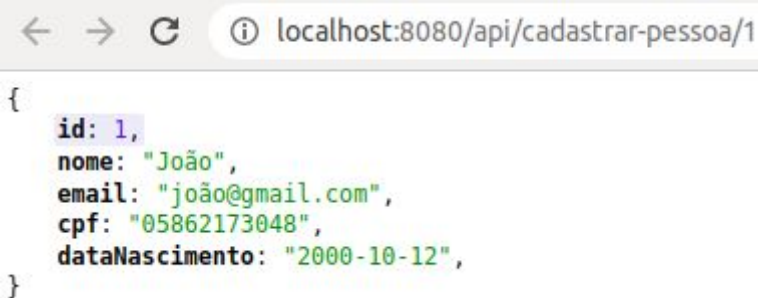
```
@GetMapping(value =("/{id}")
ResponseEntity<NovaPessoaResponse> obterPessoaPorId(@PathVariable("id") Long id){
    NovaPessoaResponse response = pessoaRepository.findById(id)
        .orElseThrow( () -> new ResponseStatusException(HttpStatus.NOT_FOUND))
        .toResponse();
    return ResponseEntity.ok(response);
}
```

Retornando uma pessoa

Da mesma forma que criamos um objeto que represente a entrada na nossa API criamos um objeto para resposta da requisição, isso se faz necessário para protegermos nossas classes de domínio e retornarmos exatamente o que o cliente deseja. Olhe como ficou o método que transforma nossa entidade vinda do banco na response:

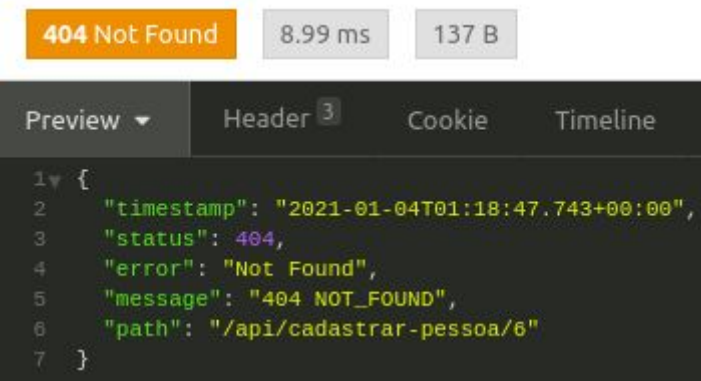
```
public NovaPessoaResponse toResponse() {
    return new NovaPessoaResponse(this.id, this.nome, this.email, this.cpf, this.dataNascimento);
}
```

Agora caso a gente clique no link retornado veremos o objeto que acabamos de criar. Caso não exista o objeto procurado retornamos um 404 NOT FOUND.



```
{
  id: 1,
  nome: "João",
  email: "joão@gmail.com",
  cpf: "05862173048",
  dataNascimento: "2000-10-12",
}
```

Caso não exista:



```
404 Not Found 8.99 ms 137 B

Preview ▾ Header 3 Cookie Timeline
1 {
2   "timestamp": "2021-01-04T01:18:47.743+00:00",
3   "status": 404,
4   "error": "Not Found",
5   "message": "404 NOT_FOUND",
6   "path": "/api/cadastrar-pessoa/6"
7 }
```

Validando valores iguais

No momento que definirmos nossa entidade colocamos que o email e o CPF seriam valores únicos na tabela, porém não estamos verificando isso em nenhum momento. Isso está gerando erro 500.

Da mesma forma que utilizamos as anotações já existentes para validar os dados de entrada, podemos criar uma anotação para fazer essa validação. Veja por exemplo a anotação `@Past`.

```
@Constraint(validatedBy = { })  
public @interface Past {
```

Ela é uma constraint, porém não é definido por ela por quem ela será validada. Isso depende de quem vai validar, mas podemos criar uma anotação e definirmos um validador para ela.

```
@Target({ElementType.FIELD, ElementType.PARAMETER})  
@Retention(RetentionPolicy.RUNTIME)  
@Documented  
@Constraint(validatedBy = UniqueValueValidator.class)  
public @interface UniqueValue {  
  
    String fieldName();  
  
    String message() default "{com.jeanpandolfi.validation.uniquevalue}";  
  
    Class<?>[] groups() default {};  
  
    Class<? extends Payload>[] payload() default {};  
}
```

- `@Target` define onde a anotação vai ser utilizada
- `@Retention` define quando que ocorrerá a validação
- `@Documented` é utilizado para documentação.
- `@Constraint` indica que a anotação é uma constraint e que pode ser indicada qual classe será responsável pela validação, nesse caso pela `UniqueValueValidator`.

O atributo `fieldName` vai indicar para nós qual campo que será de valor único. Os outros atributos são obrigatórios apenas por ser uma anotação constraint. Veja agora a classe validadora.

```
public class UniqueValueValidator implements ConstraintValidator<UniqueValue, String> {  
  
    private String fieldName;  
  
    @PersistenceContext  
    private EntityManager entityManager;  
  
    @Override  
    public void initialize(UniqueValue constraintAnnotation) {  
        this.fieldName = constraintAnnotation.fieldName();  
    }  
  
    @Override  
    public boolean isValid(String value, ConstraintValidatorContext context) {  
        Query query = entityManager.createQuery( String.format("select 1 from Pessoa where %s =: value", fieldName));  
        List<?> list = query.setParameter( String.format("%s", "value"), value).getResultList();  
        return list.isEmpty();  
    }  
}
```

Veja que ela implementa uma interface que exige como parâmetro a anotação que ela vai validar e qual o tipo do parâmetro vai vim para ela validar, ou seja o tipo de valor do campo de quem será anotado por ela. Como essa anotação será utilizada no atributos cpf e email que são Strings o valor foi definido como String.

Além disso é obrigatório implementar os métodos *initialize* e *isValid*. Um vai pegar o valor passado ao atributo *fieldName* e o outro vai fazer a validação. Na validação é realizada uma query no banco para saber se o valor já existe, caso exista a validação vai falhar.

Agora basta anotarmos qual campo será realizada a validação, no nosso caso o cpf e o email da classe NovaPessoaRequest. Veja:

```
@NotBlank
@email
@UniqueValue(fieldName = "email")
private String email;

@NotBlank
@CPF
@UniqueValue(fieldName = "cpf")
private String cpf;
```

Estamos indicando que queremos validar se existe um único valor para o campo de email e cpf. Precisamos também de definir uma mensagem no *messages.properties*.

```
UniqueValue = O valor já existe e não pode ser duplicado
```

Basta reiniciarmos a aplicação para funcionar.

The screenshot shows a REST client interface. The top bar indicates a POST request to `http://localhost:8080/api/cadastrar-pessoa` with a status of `400 Bad Request`, a response time of `674 ms`, and a body size of `322 B`. The left pane shows the request body in JSON:

```
1 {
2   "nome": "João",
3   "email": "joão@gmail.com",
4   "cpf": "05862173048",
5   "dataNascimento": "2000-10-12"
6 }
```

The right pane shows the response body in JSON:

```
1 {
2   "path": "uri=/api/cadastrar-pessoa",
3   "timestamp": "2021-01-03T23:11:29.9",
4   "globalErrorsMessages": [],
5   "fieldErrors": [
6     {
7       "field": "cpf",
8       "message": "O valor já existe e não pode ser duplicado",
9       "rejectValue": "05862173048"
10    },
11    {
12      "field": "email",
13      "message": "O valor já existe e não pode ser duplicado",
14      "rejectValue": "joão@gmail.com"
15    }
16  ]
17 }
```

Foi realizada a validação corretamente. Mas veja um caso interessante, se da mesma forma que adicionamos as anotações na classe da entidade e simularmos comentando na request que a validação não funcionou veremos que dará um *NullPointerException* para o *EntityManager*. Isso acontece pelo fato que tinha dito antes, depende de quem vai realizar a validação, com a anotação na classe da request quem faz a validação é o Spring e ele sabe

de uma implementação o *EntityManager* , já quando anotamos na entidade quem faz a validação é o Hibernate que não sabe qual é a implementação do *EntityManager* tornado ele nulo ao injetar na classe.

Bom pessoal esse foi o post de hoje e espero que tenha gostado e o código do projeto está no [GitHub](#). Abraços!