

# API REST ABERTURA DE CONTA COM SPRINGBOOT

## INTRODUÇÃO

Nesse post será abordado uma forma de construir uma API REST de cadastro de conta bancária. Apesar do tema ser conta bancária, os mesmos princípios aqui podem ser utilizados para qualquer aplicação *REST* utilizando o *Spring* e algumas ferramentas do seu ecossistema.

A aplicação completa com *CRUD* funcional, tratamentos de erros personalizados e validações com *java bean validation* estará disponível no link do *github* fornecido no final do artigo. Aqui serão abordados somente algumas funcionalidades básicas para que o leitor com pouco ou nenhuma experiência obtenha mais familiaridade com o *Spring Framework* e seu rico ecossistema.

## CONFIGURAÇÕES BÁSICA DO PROJETO

Os materiais aqui utilizados foram qualquer versão java 8+, neste projeto foi utilizado a versão 11, IDE IntelliJ *Community version* 2020.3, qualquer versão superior ou igual a 2.4.1 do *Spring Boot*, tipo de empacotamento jar. O banco de dados relacional escolhido foi o *H2*. Por ser um banco em memória toda vez que a aplicação é reiniciada o banco também é, apagando todo o seu conteúdo. Isso ajuda na questão de não precisar gerenciar os dados no banco.

As dependências utilizadas aqui foram o *Spring Web* que oferece suporte para aplicações *web MVC*, disponibilizando um container *web apache tomcat* para aceitar as requisições e dar suporte para aplicações *REST*. A dependência *Spring Data JPA* que tem as *annotations* necessárias para persistências de dados e a dependência do banco *H2*. Como esse é um projeto introdutório foi utilizado o *maven* para gerenciamento de dependências, mas quem preferir pode utilizar o *gradle* bastando mudar a opção antes de gerar o projeto. Agora com tudo definido o próximo passo é colocar a mão na massa.

## TRATAMENTOS DE EXCEÇÃO

Esse tópico é extremamente importante e deve ser explicado com calma, pois em qualquer projeto básico que exija fazer um *CRUD*, é necessário ter tratamentos adequados para que não quebre a aplicação e não viole princípios básicos do banco de dados.

As exceções devem existir para prevenir que erros como pesquisar ou deletar um campo que não existe, prevenir que campos obrigatórios sejam deixados como nulo, prevenir que haja campos repetidos quando não é permitido. Em suma exceções são utilizadas para que não quebre a regra de negócio do sistema travando a aplicação. Depois de tratadas o código toma outro fluxo e mantém o sistema funcionando.

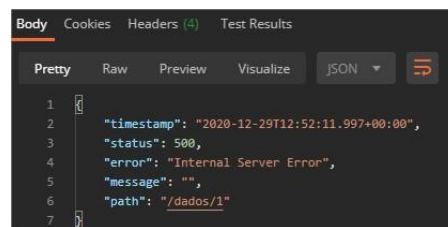
Há diversas formas de tratar um erro em *java*, desde usando a clausula *try-catch*, desde criando tratamento de exceções personalizadas. Apesar de não utilizada aqui no projeto base, um exemplo será apresentado aqui para que o leitor tenha ideia de como

esse procedimento é feito. Mais exemplos serão disponibilizados na versão final no *github*.

```
1 package com.pedromateus.dscatalog.exceptions;
2
3 public class DataBaseException extends RuntimeException{
4
5     private static final long serialVersionUID = 1L;
6
7     public DataBaseException(String msg) {
8         super(msg);
9     }
10 }
11
```

Esse modelo de exceção personalizada é simples e o que faz é passar a mensagem para a classe pai dela. Para utilizar essa classe basta colocar a clausula *try-catch* e se o código cair em *catch* uma exceção é lançada com a mensagem personalizada pelo desenvolvedor.

Sempre que uma requisição *http* não obtém sucesso algum tipo de erro é retornado. Ferramentas para realizar requisições como o *postman* frequentemente são utilizadas para realizar testes nos *endpoints* do *back-end*. Abaixo um erro comum retornado pela ferramenta de requisição *postman*.



The screenshot shows the Postman interface with the 'Body' tab selected. The response is displayed in 'Pretty' format as a JSON object. The JSON contains a timestamp, status 500, error message 'Internal Server Error', an empty message, and a path '/dados/1'.

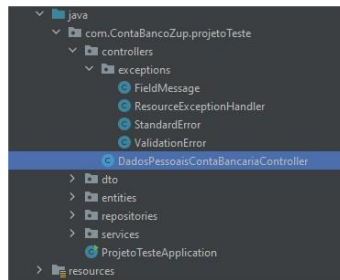
```
1 {
2   "timestamp": "2020-12-29T12:52:11.997+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "message": "",
6   "path": "/dados/1"
7 }
```

Esse tipo de erro frequentemente acontece para requisições que retornam erros e não são tratados. Há formas de tratar estes erros e retornar respostas mais adequadas que contém mais informação, tanto para o desenvolvedor que for trabalhar no código quanto para o usuário. Erros do tipo 500 são deslegantes e devem ser evitados ao máximo.

Uma das formas utilizadas em *java* é criar uma classe que contém as variáveis com os mesmos nomes daquelas retornadas dentro do corpo *json* pela ferramenta *postman* e utiliza-las em uma classe específica para capturar qualquer tipo de exceção que ocorra no código.

```
public class StandardError {
    private Instant timestamp;
    private Integer status;
    private String error;
    private String message;
    private String path;
}
```

É uma classe comum com construtores, *getters* e *setters*. Esse tipo de erro é tratado no controlador. A classe que captura os erros possui uma anotação especial em cima de sua declaração *@ControllerAdvice*. Classes com essa anotação vão capturar as exceções em classes ou métodos assinado com *@RequestMapping*. Ou seja, é mais comum ver esse tipo de classe de exceção dentro de pacotes na camada de controle.



Nesse exemplo a classe *StandardErrors* é a que tem as variáveis iguais as da resposta do *postman*. A classe *ResourceExceptionHandler* é a que captura as exceções ocorridas no controlador marcado com *@RequestMapping*.

É interessante observar que há um sufixo *Handler* no nome da classe. Não é obrigatório usar, mas deixa a classe mais sugestiva para futuros desenvolvedores. Abaixo o exemplo da classe *Handler* com as anotações explicadas.

```
@ControllerAdvice
public class ResourceExceptionHandler {

    @ExceptionHandler({DataIntegrityViolationException.class})
    public ResponseEntity<ValidationErrors>
    validation(DataIntegrityViolationException e, HttpServletRequest request){
        HttpStatus status = HttpStatus.UNPROCESSABLE_ENTITY;
        ValidationErrors err = new ValidationErrors();
        err.setTimestamp(Instant.now());
        err.setStatus(status.value());
        err.setError("Erro ao gravar, cpf já existe ou está em branco");
        err.setMessage(e.getMessage());
        err.setPath(request.getRequestURI());
        return ResponseEntity.status(status).body(err);
    }
}
```

Nessa classe para cada exceção é construído um método que é anotado com *@ExceptionHandler*, passando como parâmetro o nome da exceção mais ".class". Essa anotação vai garantir que o método capture e trate qualquer exceção daquele tipo. O tipo *HttpStatus*, retorna o estado de qualquer requisição que o desenvolvedor quiser. A classe *ValidationErrors* é filha de *StandardErrors* sendo apenas uma forma utilizada para capturar muitos erros e construir listas com erros personalizados.

Nos parâmetros do método também são passadas as exceções. O *HttpServletRequest* é utilizado caso o desenvolvedor queira retornar a *URI*. Apesar de esse ser um modelo genérico o desenvolvedor pode implementar da forma que for mais conveniente para sua aplicação.

Os meios apresentados são apenas para captura de erros e tratamentos personalizados utilizando parte da liberdade que o *spring framework* fornece. Deve ficar claro que apesar das diversas exceções que já existem no *java* o desenvolvedor pode criar suas próprias exceções e associa-las a qualquer condição, dando ainda mais liberdade para manipular as variáveis a serem persistidas no banco.

Por exemplo utilizando um código para que uma determinada variável não se repita ou não seja nula. O desenvolvedor pode fazer isso utilizando código ou ele pode optar pelo uso de metadados ou pode usar as famosas anotações.

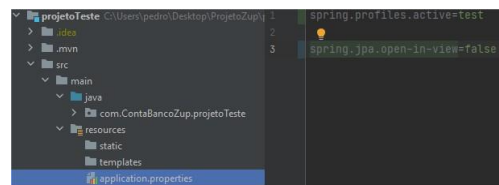
Para exemplificar o uso de exceções será adicionada uma anotação *@Column(unique=true)* da *API javax.persistence* no cpf e no email da classe de entidade, para garantir a não existência de valores repetidos para esses campos. O erro será capturado e tratado, retornando um erro personalizado como referido nessa mesma seção.

## CONFIGURAÇÃO BANCO DE DADOS H2

O banco de dados ou *database H2* é um banco em memória que todas as vezes que a aplicação é iniciada ele é apagado. Para começar a utiliza-lo basta colocar sua dependência no arquivo de configuração *pom.xml*.

```
<dependency>
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
  <scope>runtime</scope>
</dependency>
```

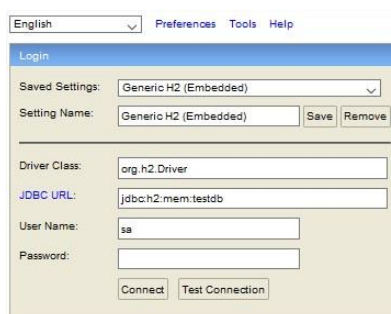
Outra coisa importante são os perfis de desenvolvimento. Normalmente se utiliza um perfil de teste para o banco de dados *h2*. Para os demais perfis utiliza-se um *prod* para produção e *dev* para desenvolvimento. Para ativar o banco específico é necessário colocar uma configuração no arquivo *application.properties* que fica na pasta *resource*.



A linha de código *spring.profiles.active=test* é onde será carregado o arquivo de configuração do banco que está sendo utilizado. O nome do arquivo para o banco *h2* é *application-test.properties*, para os outros perfis basta colocar o nome no lugar do *test*. A segunda linha da figura anterior garante que toda transação será feita na camada de serviço. Por fim basta colocar as configurações referente ao banco.

```
1 spring.datasource.url=jdbc:h2:mem:testdb
2 spring.datasource.username=sa
3 spring.datasource.password=
4
5 spring.h2.console.enabled=true
6 spring.h2.console.path=/h2-console
```

Essas configurações são colocadas nos respectivos campos do banco quando acessado o caminho da aplicação mais */h2-console*. A configuração *console.enable=true* mostra o seguinte console onde as configurações acima são utilizadas:



English Preferences Tools Help

Login

Saved Settings: Generic H2 (Embedded)

Setting Name: Generic H2 (Embedded) Save Remove

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:testdb

User Name: sa

Password:

Connected Test Connection

E se tudo estiver correto basta clicar no botão *Connect* para visualizar a seguinte janela.



```

<properties>
  <java.version>11</java.version>
</properties>

<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>

  <dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
  </dependency>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-test</artifactId>
    <scope>test</scope>
  </dependency>
</dependencies>

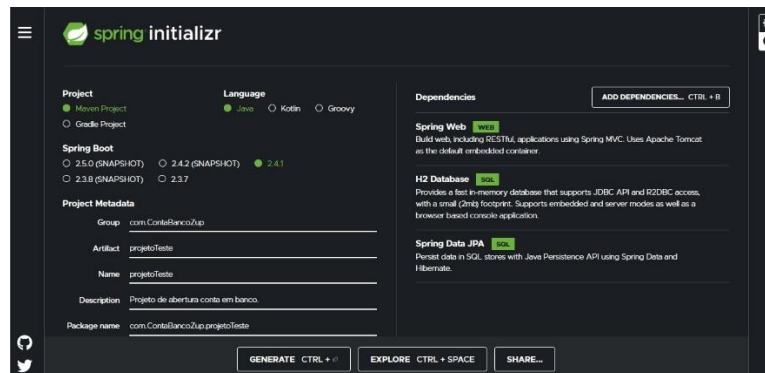
```

## INICIALIZANDO PROJETO

### BACK-END

#### Criando o projeto.

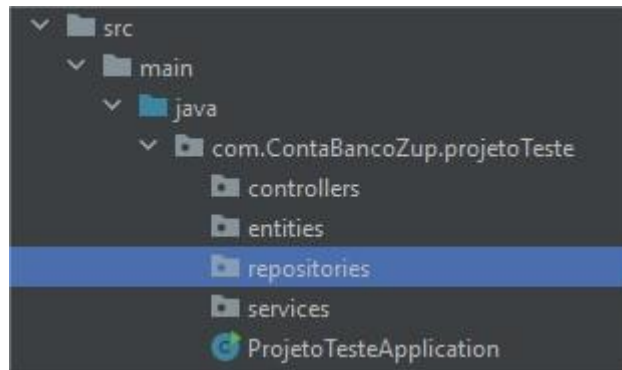
Uma forma rápida de inicializar um projeto é utilizando o *spring initializr* pelo link <https://start.spring.io/>. Nele é possível realiza todas as configurações citadas no início do projeto de forma simples.



Depois é só clicar em *generate*, baixar o zip, descompactar e abrir o projeto com a *IDE* que o desenvolvedor achar melhor.

#### Começando a criar a estrutura básica do modelo em camadas.

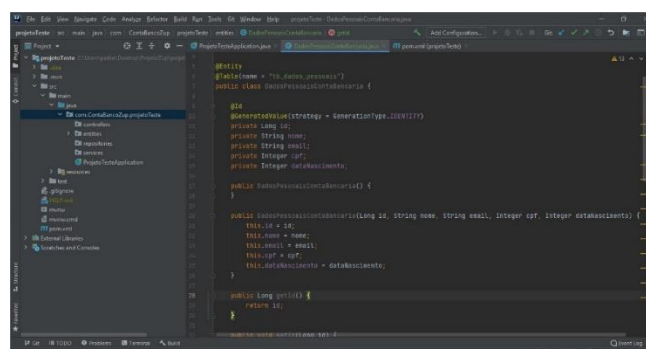
Agora o próximo passo é criar as classes e os pacotes necessário para organização do projeto. O modelo aqui utilizado é o modelo em camadas. A ideia básica é que cada funcionalidade tem seu próprio pacote, como por exemplo um pacote para os controladores, uma para as classes de entidade, um para as classes de serviços e por aí vai. Ao longo do projeto as imagens deixarão tudo mais claro. Abaixo um exemplo visual da estrutura básica de pacotes utilizados no projeto.



Essa é a estrutura inicial do projeto, sem tratamento de exceções. Projetos mais simples funcionam bem quando feitos dessa forma. Organizando o projeto com esse padrão fica mais fácil dar manutenção no projeto, já que as classes tem papéis bem definidos e separadas por pacotes. Para projetos maiores a complexidade pode aumentar muito deixando esse modelo inviável de ser praticado. Nesses casos outras abordagens podem ser melhores para o projeto, como por exemplo um sistema monolítico modular. Agora uma breve descrição de cada pacote.

- **Controllers:** também chamados de *resources*, aqui fica as classes com os *endpoints*. As requisições chegam do *front-end* e passam por aqui primeiro. Representa a camada *controller* do modelo MVC;
- **Entities:** aqui são as classes relacionadas as entidades, são os objetos gravados nas tabelas dos bancos de dados. As classes de entidades fazem parte da camada *model* do MVC;
- **Repositories:** é a camada mais baixa do código, a mais próxima do banco de dados. Ela não possui quase nada pois o *spring* encapsula a parte de *ORM* deixando muito simples de fazer as operações básicas com banco de dados;
- **Services:** As classes relacionadas a serviços são onde as regras de negócios ficam, é onde a parte relacionada a salvar, consultar e modificar as entidades estão.

Em cada pacote pode ter várias classes. Nesse projeto somente uma classe será utilizada, a classe com nome *DadosPessoaisContaBancaria*. Inicialmente a classe tem esse formato:



É uma classe genérica somente a título de demonstração. Na prática sempre é bom colocar *get* e *set* somente quando precisar. Atributos que não devem ser mudados podem ter somente *get*. Outra coisa que deve ser analisado também são quais atributos podem ou devem ser instanciados no construtor. Uma prática comum é colocar *hashCode* e *equals* para as classes de entidade. Uma dúvida comum é em relação a organização dos métodos e atributos nas



classes. Uma forma simples de organização é colocar os atributos da classe primeiro, depois os objetos de composições, listas e implementações de qualquer tipo de coleção, construtor vazio e com argumentos, *getters* e *setters*, métodos públicos e privados. Utilizando esse padrão as classes ficam mais organizadas e padronizadas.

O próximo passo é entender como as classes de entidade serão mapeadas no banco. Para isso são utilizadas as anotações ou *annotations*. Anotações são metadados que são processados de forma especial executando alguma tarefa específica. É uma forma enxuta de realizar uma configuração no código.

No caso de *@Entity*, essa anotação está sinalizando para o *spring jpa* que essa classe é um elemento de entidade e seus dados devem persistir em um banco de dados ou *database*, sendo os atributos as colunas da tabela. No momento de importar essa anotação haverá dois tipos de imports, o *javax.persistence* e do *hibernate*. É preferível usar a primeira porque ela é do *jpa* e o *hibernate* é uma implementação do mesmo. Então se houver necessidade de trocar a implementação não haverá a necessidade de modificar essas anotações.

Com a anotação *@Table* é possível modificar o nome da tabela referente aquela classe anotada, se não for colocada a tabela padrão será o próprio nome da classe. As duas últimas anotações *@Id* indica qual atributo deve ser a chave primária da tabela e *@GeneratedValue* é forma de incrementação dessa chave, *GenerationType.IDENTITY* indica que a chave é única e auto incrementável no banco.

Essas foram as anotações mais básicas para que uma classe precisa ter para persistir dados no banco de forma organizada. Lembrando que esse tipo de classe fica no pacote de entidades.

Não há uma forma padrão adotada para implementar as classes. Nesse projeto o caminho começado foi pelas entidades, repositórios, controladores e por último serviços. É sempre bom realizar testes com a ferramenta *postman* para verificar se todos os *endpoints* funcionam bem, testando os principais métodos do protocolo *http* utilizados na arquitetura *REST*. São oito métodos, mas os principais e mais usados são *GET*, *POST*, *DELETE*, *PUT* e *PATCH*.

```
1 package com.contabancozup.projetoeste.repositories;
2
3 import com.ContaBancoZup.projetoeste.entities.DadosPessoaisContaBancaria;
4 import org.springframework.data.jpa.repository.JpaRepository;
5 import org.springframework.stereotype.Repository;
6
7 @Repository
8 public interface DadosPessoaisContaBancariaRepository
9     extends JpaRepository<DadosPessoaisContaBancaria, Long> {}
10
```

O trecho de código mostrado na figura acima indica basicamente a implementação do repositório. Na verdade, ele é uma interface que herda de outra, da *JpaRepository*. Aqui tem todos os métodos para realizar operações com objetos do banco. O trecho *<DadosPessoaisContaBancaria, Long>* indica o tipo de objeto a persistir no banco e o tipo da chave primária. A anotação *@Repository* está sinalizando para o *spring framework* que essa classe é um componente elegível para se realizar a injeção de dependências e inversão de controle, similar a *@Component* só que com um pouco mais de abstração.

Seguindo agora para a criação da classe controladora. Geralmente denominada *resources* ou *controllers*. Aqui duas anotações são utilizadas *@RestController* para indicar que essa classe é um controlador *REST* e que vai ter os *endpoints*. Os *endpoints* são rotas que respondem alguma coisa. A outra anotação utilizada é *@RequestMapping*. Nesta última anotação recebe um valor indicando o nome da rota *REST*, normalmente sendo um nome no



plural e com todas letras minúsculas. Nesse projeto que um único controlador o nome da rota utilizado foi “/dados”, melhor visualizado na imagem abaixo .

```
1 package com.ContaBancoZup.projetoTeste.controllers;
2
3 import org.springframework.web.bind.annotation.RequestMapping;
4 import org.springframework.web.bind.annotation.RestController;
5
6 @RestController
7 @RequestMapping(value = "/dados")
8 public class DadosPessoaisContaBancariaController {
9
10 }
11
```

Para cadastro de uma conta bancária basta usar somente o método *POST*, mas foi realizado uma implementação de um *crud* um pouco mais completo. Fica mais fácil de explicar sobre controladores utilizando mais de um método do *http*.

Quando se utiliza o protocolo *http* é esperado algum código de retorno para indicar qual foi o resultado da operação utilizada, esses códigos são chamados de estado da resposta, normalmente referido como *status*. Por exemplo o retorno 200 indica que a requisição obteve sucesso, o 201 indica que algo foi criado com sucesso, o 400 indica que a requisição não obteve sucesso, o 401 indica que a requisição não foi concluída por falta de autorização. Uma forma de garantir que o *endpoint* retornará a resposta correta é utilizar um objeto do *spring* chamado de *ResponseEntity*. Esse objeto é do tipo *generics* e encapsula uma resposta *http*.

```
1 @GetMapping
2 public ResponseEntity<List<DadosPessoaisContaBancaria>> findAll(){
3     List<DadosPessoaisContaBancaria> list=new ArrayList<>();
4     //Lógica do método
5     return ResponseEntity.ok().body(list);
6 }
7
```

Como *ResponseEntity* é tipo genérico, o tipo de objeto que será encapsulado no corpo da resposta deve ser declarado entre os símbolos maior e menor. No retorno do método a lista é enviada no corpo da resposta *http* da requisição. Na linha de retorno é utilizado um *builder* que constrói um objeto *ResponseEntity* retornando um *ok* com estado de valor 200 e a lista no corpo. A anotação *@GetMapping* está indicando que esse método é um *endpoint* e será invocado quando uma requisição *GET* sem parâmetros for efetuada.

Depois das classes de repositórios e controladores prontos implementa-se as classes de serviços. Lembrando que classes de serviços são onde ficam as regras de negócio. Abaixo um exemplo de modelo de classe de serviços.

```
1 package com.ContaBancoZup.projetoTeste.services;
2
3 import org.springframework.stereotype.Service;
4
5 @Service
6 public class DadosPessoaisContaBancariaService {
7
8 }
9
```

A anotação *@Service* tem papel similar ao *@Repository*, marcando a classe como componente elegível para injeção de dependências. Para utilizar o componente injetável é necessário uma instancia na classe e marca-la com a anotação *@Autowired*. Essa notação pode ser colocada no objeto de interesse declarado, no construtor ou nos métodos *setters*. Muitos

desenvolvedores preferem utilizar no construtor por facilitar a construção e realização de testes unitários.

```
@Autowired
private DadosPessoaisContaBancariaRepository dadosPessoaisContaBancariaRepository;
```

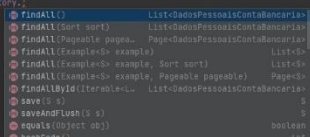
Essa anotação é usada para qualquer tipo de injeção e não somente nas classes de serviço. Para qualquer tipo de variável configurada como componente do *spring*, ou seja, classes marcadas como um componente pode ser auto injetada com essa anotação. Outro exemplo é na classe de controlador.

```
@Autowired
private DadosPessoaisContaBancariaService dadosPessoaisContaBancariaService;
```

O que acontece é que o objeto vai ser instanciado e todos os métodos daquela classe podem ser utilizados através do mesmo. As classes que tem a interface do *jpa* implementadas possuem diversos métodos já prontos para serem utilizados com banco de dados relacionais. Esses são apenas alguns exemplos.

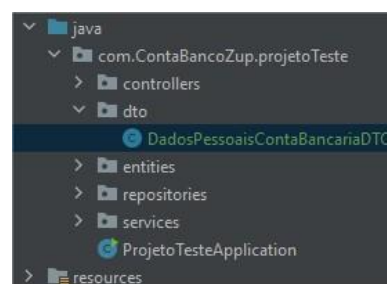
```
@Autowired
private DadosPessoaisContaBancariaRepository dadosPessoaisContaBancariaRepository;

private List<DadosPessoaisContaBancaria> findAll() {
    return dadosPessoaisContaBancariaRepository.findAll();
}
```

A screenshot of an IntelliJ IDEA autocomplete menu for the `findAll()` method. The menu lists several methods from the `Repository` interface, including `findAll()`, `findAll(Sort sort)`, `findAll(Pageable pageable)`, `findAll(Example example)`, `findAll(Example example, Sort sort)`, `findAll(Example example, Pageable pageable)`, `findAllById(Iterable ids)`, `save(S)`, `saveAndFlush(S)`, `equals(Object o)`, and `hashCode()`. Each method is preceded by a red circular icon with a white dot.

Agora é possível chamar qualquer método do repositório na classe de serviço. Também na classe do controlador é possível chamar qualquer método do serviço, graças as injeções de dependências.

Pelo padrão de camadas as classes do controlador não têm contato direto com os objetos da entidade, a forma mais correta seria utilizar um tipo de objeto apenas para transferir os dados da camada de serviço para camada de controle. Esses objetos recebem o nome de *DTO Data Transfer Object*. Então agora é adicionar um novo pacote para as classes *dto* e as respectivas classes que terão dados transportados da camada de serviço para a de controle.



As classes de criação *dto* são muito parecidas com as originais contendo as mesmas variáveis, os construtores vazio e com argumentos, *getters* e *setters*. É conveniente utilizar um construtor a mais onde é recebido o próprio objeto entidade nos parâmetros. Dentro dele cada variável *dto* recebe os respectivos atributos similares aos da entidade, isso garante uma transformação de objetos da entidade para *dto*.

```

public DadosPessoaisContaBancariaDTO(DadosPessoaisContaBancaria entity) {
    id = entity.getId();
    nome = entity.getNome();
    email = entity.getEmail();
    cpf = entity.getCpf();
    dataNascimento = entity.getDataNascimento();
}

```

Agora sim será possível utilizar o objeto da camada de serviço injetado no controlador apenas realizando as mudanças necessárias da entidade para *dto*. Agora é só implementar os métodos da camada de serviço e de controle e o *CRUD* estará pronto. Lembrando que para registrar a conta bancária basta utilizar apenas o *POST*, os outros métodos são apenas para o leitor entender melhor e estudar o código. Mais uma vez o *link* do projeto no *github* estará disponível no fim do artigo.

Agora uma explicação rápida de cada método utilizado nas camadas de controle e serviço. Começando pelo controlador.

```

@Autowired
private DadosPessoaisContaBancariaService dadosPessoaisContaBancariaService;

@GetMapping
public ResponseEntity<List<DadosPessoaisContaBancariaDTO>> findAll() {
    List<DadosPessoaisContaBancariaDTO> list = dadosPessoaisContaBancariaService.findAll();
    return ResponseEntity.ok().body(list);
}

@GetMapping(value =("/{id}")
public ResponseEntity<DadosPessoaisContaBancariaDTO> findById(@PathVariable Long id) {
    DadosPessoaisContaBancariaDTO dto = dadosPessoaisContaBancariaService.findById(id);
    return ResponseEntity.ok().body(dto);
}

```

Na figura pode ser visto que os objetos que são passados no parâmetro e retornados são do tipo *dto*. Como explicado antes o retorno sempre é uma *ResponseEntity* retornando algum método indicando o *status http* com mais alguma coisa podendo ser passada no corpo. O método *findById* recebe um valor de *id* como valor da anotação *@GetMapping*. Para que valor desse parâmetro seja passado nos parâmetros do método como valor aceitável uma anotação deve ser colocada antes do tipo da variável o *@PathVariable*. Essa anotação serve para apenas variáveis simples. Se tratando de variáveis compostas, como objetos outra anotação deve ser usada para converter os dados que vem do *front-end* para o tipo específico definido no parâmetro do método. A anotação *@RequestBody* antes do tipo do objeto faz essa tarefa. Nos métodos como *POST* e *PUT* são exemplos de utilização dessa anotação.

```

@PostMapping
public ResponseEntity<DadosPessoaisContaBancariaDTO>
insert(@Valid @RequestBody DadosPessoaisContaBancariaDTO dto) {
    dto = dadosPessoaisContaBancariaService.insert(dto);
    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri().path("/{id}")
        .buildAndExpand(dto.getId()).toUri();
    return ResponseEntity.created(uri).body(dto);
}

@PutMapping(value =("/{id}")
public ResponseEntity<DadosPessoaisContaBancariaDTO>
insert(@PathVariable Long id, @RequestBody DadosPessoaisContaBancariaDTO dto) {
    dto = dadosPessoaisContaBancariaService.update(dto, id);
    return ResponseEntity.ok().body(dto);
}

@DeleteMapping(value =("/{id}")
public ResponseEntity<DadosPessoaisContaBancariaDTO> delete(@PathVariable Long id) {
    dadosPessoaisContaBancariaService.delete(id);
    return ResponseEntity.noContent().build();
}

```

No método *PUT* as duas anotações são usadas. No *DELETE* não há necessidade de retornar nada. Os métodos *noContent* é usado quando não retorna nada no corpo do *ResponseEntity*.

O método *POST* retorna um http status 201 quando algo foi criado com sucesso. Nesses casos é muito conveniente criar um header chamado *location* com endereço da resposta. É exatamente o que essa linha de código faz:

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

Esse trecho de código constrói uma *URI* para retornar junto com objeto que vem no corpo. Para um registro de conta bancária com os dados da classe basta somente um método *POST*. Agora que os métodos do controlador foram explicados o próximo passo são os métodos de serviço.

Uma abordagem similar à da camada de controle será utilizada aqui.

```
@Autowired  
private DadosPessoaisContaBancariaRepository dadosPessoaisContaBancariaRepository;  
  
public List<DadosPessoaisContaBancariaDTO> findAll() {  
    List<DadosPessoaisContaBancaria> list=dadosPessoaisContaBancariaRepository.findAll();  
    List<DadosPessoaisContaBancariaDTO> listDTO=list.stream()  
        .map(elem->new DadosPessoaisContaBancariaDTO(elem))  
        .collect(Collectors.toList());  
    return listDTO;  
}  
  
public DadosPessoaisContaBancariaDTO findById(Long id) {  
    Optional<DadosPessoaisContaBancaria> obj=dadosPessoaisContaBancariaRepository.findById(id);  
    DadosPessoaisContaBancaria entity=obj.get();  
    return new DadosPessoaisContaBancariaDTO(entity);  
}
```

Se o leitor observar os nomes dos métodos de serviço são iguais aos de repositório. Como o repositório implementa a interface *JPARespository*, seus métodos já estão prontos. Essa estratégia de nomes iguais é proposital, uma dica que fica para o leitor é sempre nomear os métodos com nomes sugestivos. Comentários sempre ajudam, mas é melhor um nome que indica o papel do método do que um nome pouco sugestivo, levando a necessidade de acrescentar comentários ao código.

No método *findAll* não há nenhum segredo. Aqui é retornado uma lista de *dto* para camada de controle. Já o *findById* recebe um *id* da camada de controle. O objeto do tipo *Optional* é sugerido pelo próprio *spring framework* para caso aconteça do *id* pesquisado não exista e não ter perigo de retornar um valor nulo, pode ser usado essa abordagem ou pode realizar um tratamento mais adequado. O método *get* retorna o objeto do tipo da entidade tratada, depois o método retorna um *dto* para as classes de serviço.

O método *DELETE* apresentado na figura abaixo tem apenas uma linha que não retorna nada deletando o objeto. Nos métodos *UPDATE* e *POST* o objeto do tipo *dto* é convertido para o tipo da entidade, realizado o processo de atualizar e ou salvar e retornado o objeto *dto* para camada de controle.

```

public DadosPessoaisContaBancariaDTO insert(DadosPessoaisContaBancariaDTO dto){
    DadosPessoaisContaBancaria entity= converteDTOEntidade(dto);
    entity=dadosPessoaisContaBancariaRepository.save(entity);
    return new DadosPessoaisContaBancariaDTO(entity);
}

public DadosPessoaisContaBancariaDTO update(DadosPessoaisContaBancariaDTO dto, Long id) {
    DadosPessoaisContaBancaria entity=dadosPessoaisContaBancariaRepository.getOne(id);
    entity=converteDTOEntidade(dto);
    entity=dadosPessoaisContaBancariaRepository.save(entity);
    return new DadosPessoaisContaBancariaDTO(entity);
}

public void delete(Long id) { dadosPessoaisContaBancariaRepository.deleteById(id); }

```

Na explicação da classe *dto* foi dito que havia mais um construtor recebendo o objeto identidade. Esse construtor serve para converter entidade para *dto*. Essa conversão pode ser feita manualmente no próprio método ou utilizar um método para fazê-la. Aqui a estratégia adotada foi criar um método para realizar essa conversão.

```

private DadosPessoaisContaBancaria converteDTOEntidade(DadosPessoaisContaBancariaDTO dto) {
    DadosPessoaisContaBancaria entity=new DadosPessoaisContaBancaria();
    entity.setNome(dto.getNome());
    entity.setCpf(dto.getCpf());
    entity.setEmail(dto.getEmail());
    entity.setDataNascimento(dto.getDataNascimento());
    return entity;
}

```

Sobre essas conversões citadas há formas mais elegantes de realizá-las, como é o caso da interface *Mapper* que fornece diversos métodos para automatizar essas conversões além de outros métodos úteis. Basta apenas adicionar sua dependência no projeto. Como esse projeto tem público alvo desenvolvedores que possui pouca experiência no *Spring Framework*, foi adotada uma abordagem mais manual para construção do projeto. O *Lombok* também é uma dependência que ajuda muito na hora do desenvolvimento economizando muitas linhas de código, automatizando criação de *getters* e *setters*, construtores e outros.

Algo que faltou ser apresentado com mais detalhes aqui nesse artigo foi o tratamento das exceções que deve ser feito para que aplicação quebre e no caso disso acontecer o erro seja devidamente mapeado e mostrado ao desenvolvedor. No entanto a explicação básica sobre a técnica de tratamento de exceções no início do artigo serve para tratar qualquer exceção.

Aqui o *back-end* está finalizado. Agora só falta mostrar o teste da conta sendo cadastrada e em caso de erro o retorno fornecido pela ferramenta de teste.

Antes de iniciar os testes será adicionado abaixo uma sessão bônus sobre *java beans validation* e validações personalizadas. Elas não são de fato obrigatórias, mas adicionam mais regras de negócio deixando o sistema mais robusto em troca de um pouco mais de complexidade.

Os resultados serão divididos em duas subseções, a primeira apresenta os resultados somente com a *constraint @Column* e a segunda apresenta com validações do *beans validation*.

## JAVA BEANS VALIDATION E EXCEÇÕES PERSONALIZADAS

Nessa sessão será implementado melhorias para a *api*, utilizando o *java bean validation*. Duas variáveis em questão foram escolhidas para exemplificar o uso dessa biblioteca de validação. Ao final o *cpf* e e-mail vão possuir proteção contra valores repetidos e nulos. Nesse projeto é utilizado uma implementação do *Hibernate* para essa biblioteca. Outra coisa

importante que foi definida para o projeto é que todo erro de requisição retorna o código *http 422, UNPROCESSABLE\_ENTITY*.

Em qualquer projeto sempre é necessário proteger certos dados no memento de ser gravado no banco. Por exemplo e-mails inválidos, campos que não podem ser nulos ou vazios, quantidade mínima e máxima de caracteres que pode ser inserida em um campo. Esses são alguns dos exemplos. Citando outro exemplo, na abertura de conta não é permitido duas pessoas com *cpfs* iguais.

Uma solução simples foi apresentada nesse projeto, o uso da anotação *@Column(unique=true)*, que garante valores únicos no banco. Existem diversos outros parâmetros que podem ser passados para melhorar ainda mais a proteção dos dados. Essas anotações de validação e proteção são popularmente chamadas de *constraints*, que significa restrições do inglês.

Esse tipo de *constraint* utilizado no projeto é executada no momento de criação do banco, ou seja, ela adiciona mais regras de negócio ao banco de dados. Há outras *constraints* muito utilizada em *java* utilizando a biblioteca do *java beans validation*. Esse tipo de validação é feito em tempo de execução na camada de controle, onde chegam as informações do *front*. Por padrão a validação é feita antes mesmo dos objetos de entrada tocarem o banco, sendo feito a verificação no ponto de entrada. Uma analogia simples é de um segurança que verifica a idade das pessoas antes de entrar em uma festa.

Mesmo que por padrão as verificações são feitas apenas nos *endpoints*, a biblioteca fornece ferramentas para criar as próprias anotações e restrições personalizadas, podendo por exemplo buscar e verificar dados presente no banco.

Abaixo alguns exemplos dos tipos das *constraints* fornecidas pelo *java beans validation* disponível na documentação oficial.

- **@NotNull:** a propriedade não pode ser nula;
- **@AssertTrue:** verifica se a propriedade é verdadeira;
- **@Size:** verifica se a propriedade está dentro dos valores definidos em *@Max* e *@Min*;
- **@Min:** define um limite mínimo para o atributo;
- **@Max:** define um valor máximo para o atributo;
- **@Email:** verifica se o endereço de e-mail é válido.

Essas *constraints* estão no pacote *javax.validation.constraints*. O desenvolvedor que for implementar deve ter cuidado para usar as anotações de forma correta. Por exemplo não dá para colocar uma validação de números em um campo que tem só dígitos. Para ter sempre certeza é melhor consultar a documentação oficial.

Para começar a utilizar a biblioteca é necessário adicionar a sua dependência no arquivo *pom.xml*.

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

A primeira anotação a ser utilizada vai ser o *@Email*, colocada no campo e-mail. Já que as validações são realizadas na camada de controle, essas anotações devem ser colocadas nos



objetos *dto* ou qualquer outro tipo de objeto utilizado para transferir dados de controle para as camadas inferiores, em outras palavras, a verificação dos dados é feita no objeto que vai passar pela requisição. Para garantir que o campo não fique vazio pode ser utilizado o `@NotEmpty`. Além da funcionalidade de cada *constraint* também é possível retornar uma mensagem personalizada passada como parâmetro da própria anotação.

```
public class DadosPessoaisContaBancariaDTO {
    private Long id;
    private String nome;

    @Email(message = "Favor entrar com email válido")
    @NotBlank(message = "Campo obrigatório")
    private String email;

    @NotBlank(message = "Campo obrigatório")
    private String cpf;
    private String dataNascimento;
}
```

Essas anotações por si só não garantem que os dados já estão protegidos. Para que a *constraint* realmente surta efeito no código é necessário indicar ao *spring* que elas devem ser validadas. Isso é feito na camada de controle, colocando a anotação `@Valid` nos parâmetros dos métodos que tenham objetos com atributos a ser validados. Como é utilizado o *POST* para abertura de conta, o exemplo desse método com `@Valid` seria assim:

```
@PostMapping
public ResponseEntity<DadosPessoaisContaBancariaDTO>
insert(@Valid @RequestBody DadosPessoaisContaBancariaDTO dto) {
    dto = dadosPessoaisContaBancariaService.insert(dto);
    URI uri = ServletUriComponentsBuilder.fromCurrentRequestUri().path("/{id}")
        .buildAndExpand(dto.getId()).toUri();
    return ResponseEntity.created(uri).body(dto);
}
```

Apenas com essas configurações o código está protegido contra diversos dados inconsistentes. No entanto, para que seja retornado as mensagens relacionadas a cada erro é necessário tratar o tipo de exceção responsável quando a regra definida pela anotação não é respeitada. A exceção lançada é a *MethodArgumentNotValidException*. Um método para tratar essa exceção é então adicionado na classe *ResourceExceptionHandler*.

```
@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ValidationError>
validation(MethodArgumentNotValidException e, HttpServletRequest request){
    HttpStatus status = HttpStatus.UNPROCESSABLE_ENTITY;
    ValidationError err = new ValidationError();
    err.setTimestamp(Instant.now());
    err.setStatus(status.value());
    err.setError("Validation exception");
    err.setMessage(e.getMessage());
    err.setPath(request.getRequestURI());

    for(FieldError f: e.getBindingResult().getFieldErrors()) {
        err.addError(f.getField(), f.getDefaultMessage());
    }

    return ResponseEntity.status(status).body(err);
}
```

O método é similar aos outros dessa classe explicado na sessão de tratamentos de exceção. A maior diferença é que há um laço *for* percorrendo o objeto *e* do tipo *MethodArgumentNotValidException* e capturando os campos com erros e mensagens personalizadas, depois esses campos e mensagens são passadas para a função *addError* da classe *ValidationError*. Essa classe é uma extensão de *StandardError* já explicada. Abaixo uma imagem dela.



```

public class ValidationError extends StandardError{

    private static final long serialVersionUID = 1L;

    private List<FieldMessage> errors=new ArrayList<>();

    public List<FieldMessage> getErrors(){return errors;}

    public void addError(String fieldName, String message){errors.add(new FieldMessage(fieldName,message));}

}

```

Além das funcionalidades herdadas de *StandardError*, essa classe tem uma lista do tipo *FieldMessage*, uma função *get* para pegar a lista quando necessário e uma função para adicionar erros a lista, a *addError*. A classe *FieldMessage* tem apenas duas variáveis, *fieldName* que pega o nome do atributo que lançou erro e *message* que é a própria mensagem de erro.

```

public class FieldMessage{

    private String fieldName;

    private String message;

}

```

O restante da classe é apenas o padrão das classes *javas* com construtores e métodos assessores *getters* e *setters*. A explicação pode parecer mais complexa do que a funcionalidade, mas basicamente *MethodArgumentNotValidException* pode retornar várias exceções devido ao fato de ter várias anotações no código que lançam a mesma exceção. Então estes erros serão capturados e mostrados de forma mais amigável para o usuário em uma lista, ou seja, uma lista de *StandardErrors*.

Agora sim a aplicação está funcionando de forma adequada, retornando uma lista de erros personalizada pelas mensagens das *constraints* do *java bean validation* definidas pelo desenvolvedor. As imagens dos testes realizados estão disponíveis na sessão de testes do *back-end*.

### Criando validações e *constraints* personalizadas

O leitor pode ter notado que a lista de erros personalizadas, apresentada na sessão de testes, não inclui aquelas *constraints* de banco de dados. Lembrando que por padrão a biblioteca de *validation* não faz a validação de banco, apenas dos objetos de requisição, ou seja, validação no nível de sintaxe do valor por ele mesmo. Caso o desenvolvedor quiser que essa biblioteca também valide dados no banco e retorne mensagens personalizadas para esses tipos de erros, é necessário realizar algumas configurações especiais e criar anotações *constraints* personalizadas.

A primeira coisa a se fazer é definir o local que vai ser inserido essas novas exceções personalizadas. Elas são colocadas na pasta referente a camada de serviço, além de ser uma regra de negócio também se trata de verificar dados que tocam o banco de dados.

Aqui o objetivo não é se aprofundar na criação de anotações. Na verdade, é necessário entender apenas algumas linhas e não o código todo. O código padrão ou *boilerplate*, para criar a anotação é apresentado abaixo.

```

@Constraint(validatedBy = DadosBancariosValidator.class)
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface DadosBancariosValid {

    String message() default "Validation error";

    Class<?>[] groups() default {};

    Class<? extends Payload>[] payload() default {};

}

```

O nome da anotação escolhido foi *DadosBancariosValid*, visto na imagem acima. Essa anotação faz uso da classe *DadosBancariosValidator*, que é onde a lógica de negócio está, definido na primeira linha. O nome dessas classes pode ser qualquer um que o desenvolvedor desejar, mas os sufixos *Valid* e *Validator* é interessante manter para torna-las mais sugestivas. A maior parte da classe *DadosBancariosValidator* também é *boilerplate* não sendo necessária muitas explicações, somente a parte da lógica de negócio necessita ser explicada.

```
public class DadosBancariosValidator
    implements ConstraintValidator<DadosBancariosValid, DadosPessoaisContaBancariaDTO>{

    @Override
    public void initialize(DadosBancariosValid constraintAnnotation) {

    }

    @Override
    public boolean
    isValid(DadosPessoaisContaBancariaDTO dadosPessoaisContaBancariaDTO
        , ConstraintValidatorContext constraintValidatorContext) {
        List<FieldMessage> list = new ArrayList<>();

        // Coloque aqui seus testes de validação, acrescentando objetos FieldMessage à lista

        for (FieldMessage e : list) {
            constraintValidatorContext.disableDefaultConstraintViolation();
            constraintValidatorContext.buildConstraintViolationWithTemplate(e.getMessage())
                .addPropertyNode(e.getFieldName())
                .addConstraintViolation();
        }
        return list.isEmpty();
    }
}
```

A classe *Validator* implementa a interface *ConstraintValidator* que é um *generics*, implementando a anotação que vai ser criada, sendo a *DadosBancariosValid*. Como segundo parâmetro recebe a classe *dto*, sendo a *DadosBancariosDTO*.

O método *initializer* é onde ficam as implementações relacionadas quando o objeto é inicializado. Como a funcionalidade principal é não deixar valores repetidos de e-mail e *cpf* o método mais adequado aqui será o *isValid*. Apesar de parecer complexo, a lógica é bem simples. O método testa o objeto *dto* com qualquer condição desejada. As validações podem ser uma ou mais.

Aqui no projeto já foi implementada a lista de erros personalizadas, então podem ser colocado diversas condições de teste no corpo. No *for* é passado uma lista de *FieldMessage* para a lista padrão do *beans validation*.

Quando o método *isValid* for chamado, vão ter diversas condições para verificar os dados. Se o fluxo de código não entrar em nenhum teste de verificação a lista vai estar vazia e retornar *true* no comando *list.isEmpty*. Caso contrário, se a lista tiver elementos significa que houve algum erro e o retorno vai ser *false*.

Depois dessa breve explicação da lógica do método agora vai ser implementado as condições personalizadas. Por padrão não há métodos que buscam por e-mail ou *cpf* no *spring data JPA*, mas é muito simples de criar métodos de busca personalizado, mostrado na imagem abaixo.

```
@Repository
public interface DadosPessoaisContaBancariaRepository
    extends JpaRepository<DadosPessoaisContaBancaria, Long> {

    DadosPessoaisContaBancaria findByEmail(String email);
    DadosPessoaisContaBancaria findByCpf(String cpf);
}
```

Basta colocar o tipo de objeto registrado na classe *repository* e nomear a função com *findBy* mais o atributo com a primeira letra maiúscula que o desenvolvedor queira procurar.

Para utilizar estes novos métodos na classe *Validator* é necessário injetar um objeto do tipo *repository* nela.

```
public class DadosBancariosValidator
    implements ConstraintValidator<DadosBancariosValid,
        DadosPessoaisContaBancariaDTO>{

    @Autowired
    private DadosPessoaisContaBancariaRepository repository;
```

Agora basta fazer a lógica de verificação para o banco destes atributos.

```
@Override
public boolean
isValid(DadosPessoaisContaBancariaDTO dadosPessoaisContaBancariaDTO
    , ConstraintValidatorContext constraintValidatorContext) {
    List<FieldMessage> list = new ArrayList<>();
    DadosPessoaisContaBancaria testeEmail=
        repository.findByEmail(dadosPessoaisContaBancariaDTO.getEmail());

    DadosPessoaisContaBancaria testeCpf=
        repository.findByCpf(dadosPessoaisContaBancariaDTO.getCpf());

    if(testeEmail!=null){
        list.add(new FieldMessage
            ( fieldName: "email", message: "Já existe alguém cadastrado com esse email"));
    }
    if(testeCpf!=null){
        list.add(new FieldMessage
            ( fieldName: "cpf", message: "Já existe alguém cadastrado com esse cpf"));
    }
    for (FieldMessage e : list) {
        constraintValidatorContext.disableDefaultConstraintViolation();
        constraintValidatorContext.buildConstraintViolationWithTemplate(e.getMessage())
            .addPropertyNode(e.getFieldName())
            .addConstraintViolation();
    }
    return list.isEmpty();
}
```

Duas variáveis do tipo *DadosPessoaisContaBancaria* foram criadas. Utilizando a variável injetada do repositório nomeada *repository* as funções para realizar operações no banco ficam disponíveis. O *testeEmail* é um objeto da entidade utilizada no projeto e invoca a nova função criada *findByEmail* passando o e-mail do *dto* recebido no parâmetro do método.

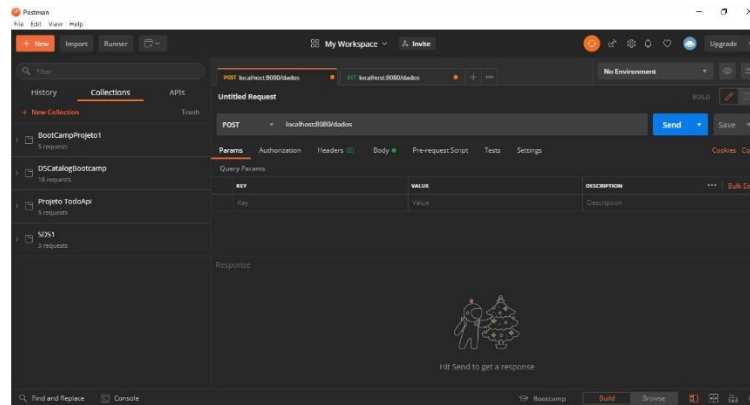
Por padrão o *repository* retorna nulo se não encontrar nada. Basta colocar um *if* verificando se o retorno é nulo, se não for é sinal de que há um objeto com aquele valor no banco de dados e o erro personalizado é adicionado na lista de *FieldMessage*, que é adicionado a lista de erro padrão do *beans validation*. Consequentemente o retorno do método não vai ser vazio e seu retorno será *false*, estourando o erro.

Seguindo essas práticas o desenvolvedor consegue criar não só as *constraints* apresentadas, mas também as que forem mais conveniente para aplicação.

## TESTES DO BACK-END

### Testes sem *bean validation*

A ferramenta utilizada aqui para realizar os testes e observar as respostas foi o *postman*. Nele é possível realizar requisições dos métodos *http* para o *back-end* e receber as respostas indicando algum tipo de falha ou sucesso.

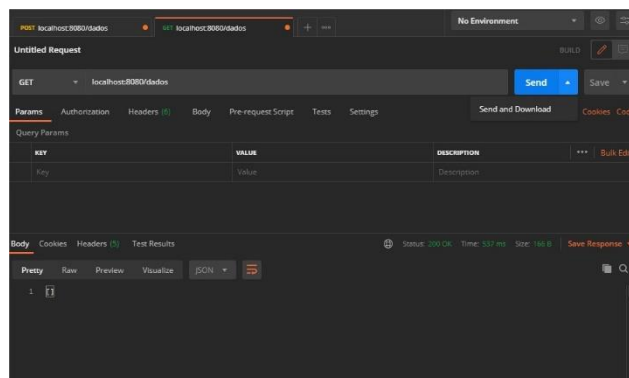


O *postman* é uma ferramenta simples de utilizar e basta adaptar seu método.



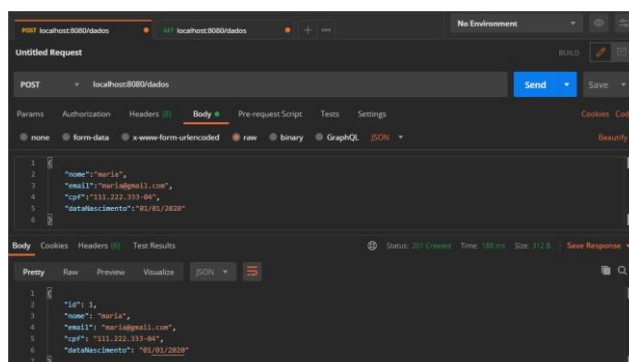
Basta colocar o método que desejar e fazer algumas alterações nas configurações. Nesse método *POST* é necessário enviar um *JSON* com variáveis iguais a utilizadas no projeto, ou as que não podem ser nulas. Um detalhe a ser observado é que está sendo utilizado um servidor local configurado pelo *tomcat* com porta 8080 seguido do caminho “/dados”, que é a rota configurada para o controlador. O desenvolvedor deve sempre lembrar que o projeto deve estar rodando e sem erros, portanto lembre-se de executar o projeto. O passo a passo dos testes serão da seguinte forma:

Fazer um *GET* recebendo um vetor vazio. Como no projeto foi utilizado o banco *h2* e não há nenhum tipo de sementeira prévia o banco sempre retorna uma coleção vazia.

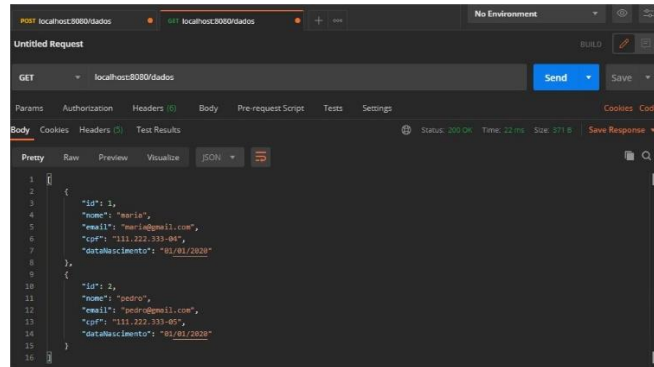


A resposta está na parte inferior na aba *body*.

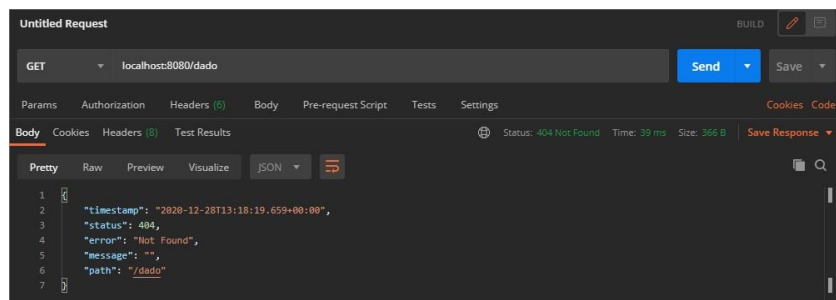
Na requisição *POST* é necessário enviar um corpo junto.



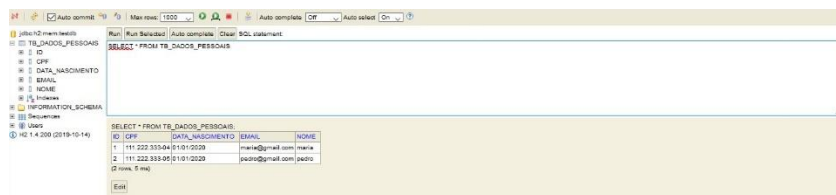
Aqui a requisição foi realizada com sucesso e o corpo recebeu o objeto gravado no banco de dados. Após persistir os objetos no banco é retornado um objeto *dto* igual ao salvo no banco. Aquela coleção vazia de antes agora está preenchido com o único objeto. Esse objeto parece muito sozinho nessa lista, portanto outro será salvo e uma requisição será realizada. O resultado da nova inserção é apresentado na imagem abaixo.



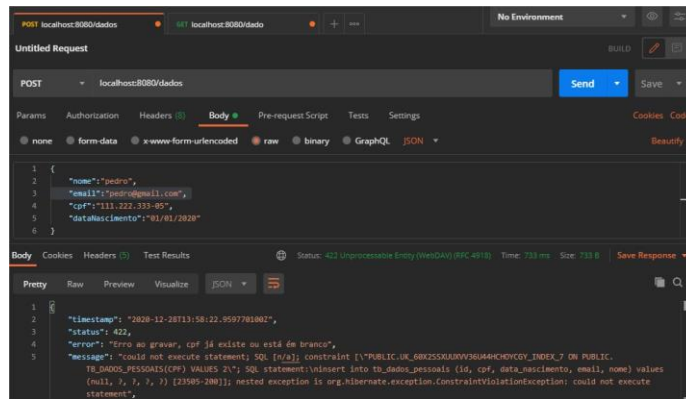
Logo abaixo da opção *SEND* está o *Status: 200 OK*, essa é a resposta *http* para requisições do tipo *GET* que obteve sucesso. No exemplo do *POST* a resposta foi *Status: 201 CREATED*. Quando a requisição não funciona a resposta é *Status: 404 NOT FOUND*. Na imagem abaixo para realizar esse teste foi colocado o nome errado do caminho do controlador.



No banco de dados agora há dois objetos persistidos.



Para encerrar estes testes mais básicos haverá uma tentativa de gravar um *cpf* repetido com intuito de demonstrar a validade da *constraint @Column(unique=true)*, explicada na sessão sobre tratamento de exceção. Será tentado gravar o mesmo usuário Pedro com mesmo *cpf*.



Nesse caso funcionou como esperado, para *cpfs* repetido a própria anotação utilizadas do *Spring framework* resolveu o problema deixando esse campo para valores únicos. O *back-end* está pronto e funcionando como esperado. Os próximos testes são com a utilização de *constraints* do *beans validation*.

### Teste com *beans validation* e sem *constraint* personalizada

Para realizar esse teste foi adicionado *constraints* na classe de *dto*. Pela imagem abaixo o e-mail deve ser um valor válido e não nulo e o *cpf* não nulo.

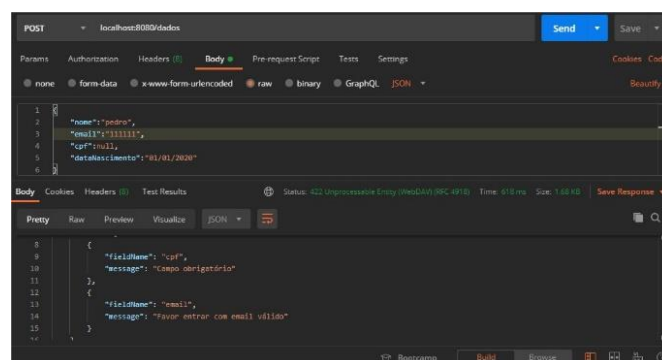
```

@Email(message = "Favor entrar com email válido")
@NotBlank(message = "Campo obrigatório")
private String email;

@NotBlank(message = "Campo obrigatório")
private String cpf;
private String dataNascimento;

```

Após adicionar as anotações, restrições ou *annotations* do *beans validation*, foi obtido a seguinte resposta pelo *postman* quando tentado gravar um e-mail invalido e deixado o campo de *cpf* nulo.



Na parte inferior, no corpo da resposta está sendo exibido apenas a lista de erro que foi criada na sessão de tratamentos de exceção no início desse tutorial. A título de ilustração o erro completo ficaria como na imagem abaixo.

```

8 {
9   "timestamp": "2021-01-01T10:26:48.414718200Z",
10  "status": 422,
11  "error": "Validation exception",
12  "message": "Validation failed for argument [0] in public org.springframework.http.ResponseEntity com.ContaBancoZup.projetoTeste.dto.DadosPessoaisContaBancariaDTO: com.ContaBancoZup.projetoTeste.controllers.DadosPessoaisContaBancariaController.insert(com.ContaBancoZup.projetoTeste.dto.DadosPessoaisContaBancariaDTO) with 2 errors: [Field error in object 'dadosPessoaisContaBancariaDTO' on field 'cpf': rejected value [null]; codes [NotBlank.dadosPessoaisContaBancariaDTO.cpf,NotBlank.cpf,NotBlank.java.lang.String,NotBlank]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [dadosPessoaisContaBancariaDTO.cpf,cpf]; arguments []; default message [cpf]]; default message [Campo obrigatório] [Field error in object 'dadosPessoaisContaBancariaDTO' on field 'email': rejected value [11111111]; codes [Email.dadosPessoaisContaBancariaDTO.email,Email.java.lang.String,Email]; arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [dadosPessoaisContaBancariaDTO.email,email]; arguments []; default message [email]]; (Ljava.validation.constraints.Pattern$Flag;010e547,*); default message [Favor entrar com email válido]] ",
13  "path": "/dados"
14 }

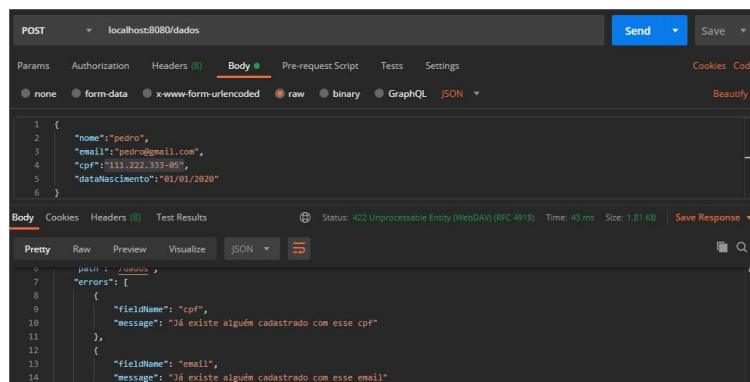
```

A segunda parte desse mesmo erro está apresentada na figura abaixo.

```
"path": "/dados",
"errors": [
  {
    "fieldName": "cpf",
    "message": "Campo obrigatório"
  },
  {
    "fieldName": "email",
    "message": "Favor entrar com email válido"
  }
]
```

### Teste com *constraint* personalizada do *beans validation*

Agora o resultado do último teste do *back-end*, vão garantir que as variáveis *cpf* e e-mail não podem ser nulas, utilizando a anotação criada e personalizada *@DadosBancariosValid*. Quando tentado gravar o mesmo *cpf* e e-mail duas vezes no banco a seguinte lista de erro é retornada.



O erro completo não vai ser apresentado aqui por ser similar ao do teste anterior. Somente a lista de erros é necessária. Agora com o *back-end* funcionando como esperado basta criar o *front-end* para retornar essa resposta ao usuário.

### FRONT-END

Nesse projeto foi escolhido para que o *front-end* fosse desenvolvido em *ReactJs*. O *react* é uma biblioteca *javascript* utilizada para desenvolver interfaces *SPA*(single page application), ou seja, interfaces de uma única página.

Para o desenvolvedor que quiser replicar a aplicação desenvolvida nesse artigo é necessário ter na máquina o *NodeJs* instalado. A *IDE* utilizada no desenvolvimento foi o *VSCode*. Para iniciar um projeto *react* um comando específico deve ser utilizado. No site <https://pt-br.reactjs.org/docs/create-a-new-react-app.html> ensina o passo a passo para iniciar uma aplicação *react*. Lá consta uma documentação completa desde o básico até assuntos mais avançados. Algo que todo desenvolvedor deve fazer é sempre consultar as documentações oficiais para sanar dúvidas.

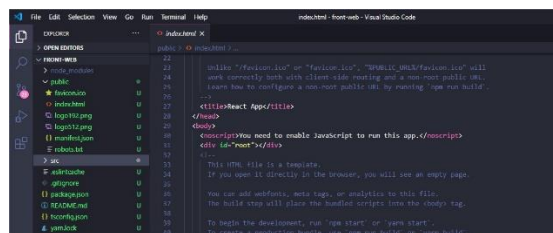
O comando *npx create-react-app my-app* cria inicialmente uma aplicação *react* pronta para o uso. Esse comando deve ser executado em algum terminal e na pasta que queira criar o projeto. Após a criação basta entrar na pasta do projeto e executar algum comando que execute a aplicação, no *node* vem o *npm start* por padrão. As aplicações *react* são previamente configuradas para utilizar a porta 3000. A pasta inicial do projeto fica da seguinte forma após executar o comando.



Após executar o comando `npm start` na porta certa no navegador, algo similar a imagem abaixo deve aparecer no navegador.



Para que uma mudança seja aplicada, basta salvar e a atualização é feita de forma automática. O arquivo *index.tsx* é o ponto de entrada da aplicação. Dentro dele há uma função que vai renderizar um arquivo *html* dentro de uma *div* com *id* de nome *root* de outro *html*. Esse arquivo fica dentro da pasta *public*.



O nome da função que renderiza o *html* dos componentes em *root* é a *render*.

```
ReactDOM.render(  
  <React.StrictMode>  
    <App />  
  </React.StrictMode>,  
  document.getElementById('root')  
);
```

O *react* é baseado em componentes sendo o *app* dentro da função *render* o principal da aplicação, o componente pai. As aplicações que utilizam essa biblioteca são construídas como se fossem peças de quebra cabeça, mas todas de alguma forma herdando de *app*, ou dentro dele. Assim todos os componentes fazem parte do componente pai e são carregados quando chamados.

Para a criação de um método básico *react* basta criar uma pasta com o nome do componente, colocar um arquivo *index.tsx*, pode ser o nome que quiser, mas *index* é mais sugestivo, depois inserir também um arquivo *css* para estilização do *html* do componente. Para poder usar qualquer componente, ou método dele, o mesmo deve ser exportado para que outros componentes o enxerguem quando for necessário utilizá-lo.

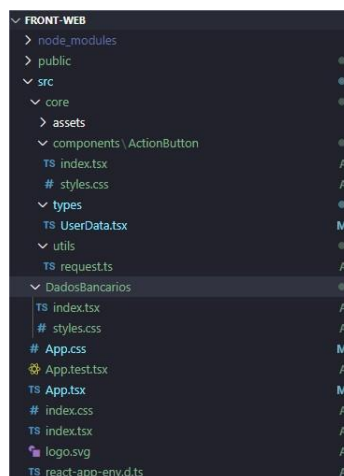
Para que um componente use outro é necessário que esse mesmo seja importado. Abaixo um exemplo do corpo de um componente.

```
1 import React from 'react';
2 import './App.css';
3
4 function App() {
5   return (
6     <div>Alguma coisa</div>
7   );
8 }
9
10 export default App;
```

O *import* da primeira linha é necessário para que o componente seja do tipo *react*. Outro *import* comum em todos componentes são seus estilos *css*. Com essas explicações básicas do *react* agora dá para começar a programar a aplicação.

O visual da aplicação será a penas um container com quatro campos e um botão para enviar a requisição.

Quando há componentes que podem ser utilizados em diversas partes da aplicação eles são normalmente colocados em pastas específicas. A pasta *core* é criada e utilizada para essa finalidade. Dentro dela há outras pastas utilitárias, como componentes que são reutilizados em diversas partes da aplicação, *types* que possuem os tipos específicos utilizados no *typescript*. A estrutura do projeto ficou da seguinte forma após adicionado pacotes com arquivos e componentes.



Iniciando as explicações, no pacote *assets* são colocadas as imagens utilizadas no projeto. Nesse caso nenhuma. No pacote abaixo *components* estão os componentes que são de uso geral, um botão, uma barra superior *header*, um rodapé personalizado, qualquer componente que for usado várias vezes pode ser colocado aqui. Na pasta *type* fica localizada os tipos definidos utilizados no código. Na pasta *utils* é colocado os utilitários, aqui por exemplo é colocado um arquivo responsável pelas requisições com nome de *request*.

Esse tipo de organização pode parecer um pouco confuso, mas acredito caro leitor que isso facilita muito a organização e leitura quando o desenvolvedor estiver mais familiarizado. Esse tipo de estrutura pode mudar de desenvolvedor para desenvolvedor, mas sempre haverá uma estrutura seguida para facilitar a organização do código.

As requisições são realizadas através do *http cliente axios* que é baseado em promessas, podendo utilizar dos recursos de *async-await*, sendo implementado dentro do arquivo *request*.

```
1 import axios, {Method} from 'axios';
2
3 type RequestParams={
4   url:string;
5   method?:Method;
6 }
7
8 const BASE_URL='localhost:8080/'
9
10 export const makeRequest=({url,method='GET'}:RequestParams)=>{
11   return axios ({
12     method,
13     url:`${BASE_URL}${url}`
14   })
15 }
```

O tipo *RequestParams* tem um tipo especial *Method* do próprio *axios* utilizado para armazenar os métodos *http*. A outra variável é apenas serve para armazenar a *url*. A variável *BASE\_URL* armazena o endereço do servidor onde está rodando o *back-end*. Então é criada uma função *makeRequest* recebendo como parâmetro um *RequestParams* e retorna a chamada do *axios*. Por fim resta o componente *DadosBancarios*.

A melhor forma de começar a explicar *DadosBancarios* é mostrar uma visão inicial de como ficou o *fron-end*.

Essa página basicamente é composta de quatro campos, um para cada dado do cliente de acordo com o tipo definido no *react* e com os atributos do *back-end*. Para salvar os dados dos campos em um objeto foi definido um tipo específico *UserData*. Para salvar basta clicar no botão salvar. Agora a definição do código por trás dessa página.

O *html* referente a cada campo é colocado no retorno do corpo da função principal do componente, como mostrado antes.

```
return (
  <div>
    <div className="container">
      <h1 className="title">Entre com os dados do cliente</h1>
      <input
        placeholder="Nome do Cliente"
        type="text"
        value={name}
        className="input-text"
        onChange={handleOnChangeName}
      />
      <ActionButton title="Salvar" onClick={handleOnClick} />
    </div>
    {status==201&&<h1 className="return-message">created</h1>}
    {status==422&&<h1 className="return-message">Error</h1>}
  </div>
);
```

Na imagem anterior é a função de retorno inteira, mas com apenas um campo, os outros são iguais mudando apenas o valor *value* para a variável desejada. É essa variável que será alterada toda vez que o campo for atualizado. Se o leitor observar aqui está sendo colocado código *javascript* com *html*. Há somente uma *div* que encapsula um título com os quatros campos de entrada. Cada *input* tem uma função associada que irá mudar o valor da varável associada aquele campo.

Aqui dois eventos são mapeados, *onChange* para mudanças nos campos *input* e *onClick* quando clicar no botão. As funções associadas a esses eventos são declaradas antes do retorno. Depois do evento de clique uma requisição é realizada e há dois retornos possíveis de mensagem. Se o código *http* de retorno for 201 quer dizer que obteve sucesso na criação do usuário, se for 422 significa que deu algum erro, como definido no *back-end*. O retorno com o *html* do componente ou da página fica no final da função principal do componente.

É importante observar linha “{status==201&&<h1 className=“return-message”>created</h1>}”. Aqui *tag html* a direita só vai ser renderizado caso a condição da esquerda for satisfeita. O nome dessa técnica é *conditiona rendering*. A forma de passar valores do *javascript* para o *html* é colocando-os entre chaves, isso é o que se chama de *bind*.

Para manipular o estado, ciclo de vida e funcionalidades dos componentes *react* eram utilizadas classes. Agora é possível fazer de forma mais simples com *reacts hooks*, que são funções. Para mudar o estado de uma variável associada a algum campo deve ser utilizado o *react hook useState*.

```
const [status, setStatus] = useState<number>(0);
const [name, setName] = useState("");
const [erro, setErro] = useState<string[]>();
const [cpf, setCpf] = useState("");
const [email, setEmail] = useState("");
const [dataNascimento, setDataNascimento] = useState("");
const [userData, setUserData] = useState<UserData>();
let listError: string[] = [];
```

Cada linha é um *useState* referente a cada atributo dos dados do cliente. Desse *react hook* é extraído uma variável que armazena o estado e uma função para modificar esse estado. Os nomes da variável e da função podem ser qualquer um, mas é comum que esses métodos tenha o prefixo *set*, *setCpf* por exemplo.

Como pode ser observado na figura acima o *useState* pode ser tipado colocando o tipo entre o sinal de maior e menor. Outra coisa que pode ser feita é inicializar o valor de um estado colocando nos parênteses do *useState*. Cada uma dessas variáveis estão associadas aos *inputs* dentro da função de retorno. O erro é para a lista de erros que vai ser pega quando a requisição retornar estado 422.

Quando acontece qualquer evento dentro dos *inputs* o *onChange* é chamado invocando a função associada, apresentada na imagem abaixo.

```
const handleChangeName = (event: React.ChangeEvent<HTMLInputElement>) => {
  setName(event.target.value);
};
const handleChangeEmail = (event: React.ChangeEvent<HTMLInputElement>) => {
  setEmail(event.target.value);
};
const handleChangeCpf = (event: React.ChangeEvent<HTMLInputElement>) => {
  setCpf(event.target.value);
};
const handleChangedataNascimento = (
  event: React.ChangeEvent<HTMLInputElement>
) => {
  setDataNascimento(event.target.value);
};
```

Cada uma das funções da imagem acima estão associadas ao respectivo evento de mudança *onChange* de cada atributo dos dados do cliente. Dentro de cada função há somente a mudança de estado de cada variável. Resumindo a história toda, cada mudança em um campo é percebida e gravada nas variáveis associadas. Depois de preenchido os campos basta o usuário clicar em salvar para realizar a requisição para o *beck-end*, consumindo a *api*.

Após clicado a um evento também associado ao clique com nome de *click*. Associado essa mudança a uma função que executa toda vez que esse clique acontece.

```
const handleClick = () => {
  setUserData({
    nome: name,
    email,
    cpf,
    dataNascimento,
  });

  makeRequest({ url: "dados", method: "POST", data: userData })
    .then((response) => {
      console.log(response.data, response.status),
      setStatus(response.status),
    })
    .catch(() => setStatus(422))
};
```

A função apresentada na imagem acima modifica os dados com as variáveis digitadas no campo e chama a função para realizar requisição no arquivo *request*, explicado anteriormente. Lembrando que a função *makeRequest* utiliza o *axios* para realizar requisições.

O *axios* retornando uma *promisse* deixa a possibilidade de encadear funções *then* e *catch*. Caso a requisição tenha sucesso os dados podem ser obtidos dentro de *then*, caso fracasse um erro é lançado pelo *catch*. Aqui o erro foi configurado para o erro padrão usado no *back-end*, retornando 422 em caso de falha na requisição. A clausula *catch* da linha da figura anterior é necessária para que aplicação não quebre caso a requisição dê errado. A lógica dela não precisa ser necessariamente colocando o estado para 422. Aqui foi utilizado dessa forma somente para retornar o erro igual do *back-end*.

Até aqui é possível gravar os nomes dos campos em um objeto, realizar requisições e obter o retorno de uma resposta *http* adequada dizendo se requisição obteve sucesso ou fracasso. O que falta para ficar pronto é em caso de erro capturar a resposta, pegar a lista definida de erros personalizados definida no *back-end* e exibir para o usuário.

A estratégia utilizada nesse projeto para capturar o erro foi modificar o *catch* do *makeRequest* para realizar algumas lógicas a mais.

```
makeRequest({ url: "dados", method: "POST", data: userData })
  .then((response) => {
    console.log(response.data, response.status),
    setStatus(response.status),
  })
  .catch((err) => {
    const { errors } = err.response.data; //Pega a lista de erro
    listError = [];
    if (errors !== undefined) {
      for (const { message } of errors) {
        //extraí mensagem de erro de dentro de cada elemento da lista
        listError.push(message);
      }
      setStatus(422);
      setErro(listError);
    }
  });
```

Na figura anterior a mensagem de erro é interceptada retornando um objeto na resposta. A lista de erros tem nome de *errors* capturada da resposta da requisição que deu errada, adquirida por desestruturação. Essa lista era uma coleção de objetos com dois atributos cada, sendo um a mensagem de erro. A lista foi percorrida extraindo a mensagem de erro de cada objeto do vetor e inserindo-as dentro de outra lista de *string*, criada apenas como variável auxiliar para agrupar as mensagens de erro.

Um estado a mais foi definido como *erro* e sua função de configuração de estado *setErro*. Ele foi usado para capturar a lista de erros *listError*. Todo esse procedimento foi feito para extrair do *catch* os dados desejados para exibir para o usuário. A mudança de *status* ainda continua pois é o valor de estado configurado na figura anterior que vai disparar a exibição. Mostrada na linha na imagem abaixo.

```
{status === 201 && <h1 className="return-message">Usuário criado com sucesso</h1>}
{status === 422 && erro?.map((elem) => <h2 key={elem} className="return-message">Erro: {elem}</h2>)}
```

Elas são um pouco diferentes da versão mostrada anteriormente, mas o funcionamento é o mesmo. Se a requisição for bem sucedida o retorno é 201 e a mensagem “*Usuário cadastrado com sucesso é exibida*”.

O tipo *ErrorData* foi criado apenas para capturar o objeto de resposta erro de forma mais elegante. Esse tipo foi adicionado dentro de *types*. Foi adicionado apenas um vetor de *string* para capturar os erros. Se o desenvolvedor quiser pode colocar todas as variáveis da resposta de erro aqui podendo criar tratamentos ainda mais personalizados.

```
1 export type ErrorData={
2   errors?:string[];
3 }
```

Com os erros devidamente capturados agora basta fazer o teste para ver se o *front-end* está retornando as respostas de forma adequada e analisar quais delas vai ser exibidas ao usuário.

## RESULTADOS DO FRONT-END

### Teste *cpf* e *email* vazios

Teste relacionado quando os campos *cpf* e *email* são deixados em branco. A resposta foi capturada do *back-end* e exibida ao usuário.

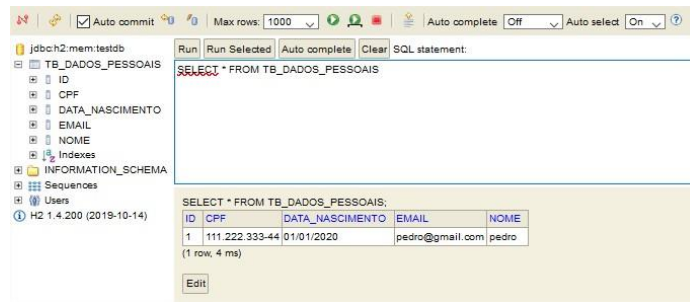


Além de exibir na tela para o usuário é possível ver o vetor de erros e a lista de erros na aba *response* da resposta da requisição.

### Teste de cadastro correto



Em caso de sucesso é exibido ao usuário que deu certo o cadastro. No *console* é possível ver a lista dos dados enviados para o *back-end* e o estado *http* 201, referente a cadastrado com sucesso. Abaixo uma imagem para mostrar como o banco *h2* ficou após o cadastro.



## Teste *cpf* e *email* repetido

Teste relacionado a *cpf* e *email* repetidos no banco, sendo o mesmo usuário cadastrado duas vezes.



Assim como as mensagens de erro foram exibidas ao usuário. Ao lado direito o console com o vetor das mensagens de erro.

## CONCLUSÃO

O projeto desenvolvido aqui serve como base para o leitor com pouco experiencia adquirir mais familiaridade com o *spring framework*. O *front-end* serviu apenas para pegar os dados fornecidos pelo usuário e retornar a resposta do *back-end*. Algumas partes podem parecer confusas, como no caso de criar as anotações personalizadas, mas com tempo fica simples fazendo da forma apresentada.

Com o que foi apresentado o leitor tem ferramentas o suficiente para continuar melhorar para um sistema de cadastro mais robusto e performático. Uma dica é que o desenvolvedor que for realizar esse tutorial tente, a partir deste ponto criar um sistema de autenticação utilizando o ecossistema *spring*.

## LINK DO PROJETO COMPLETO MAIS ALGUMAS MELHORIAS NO GIT HUB

<https://github.com/PedroMateus26/abertura-conta-teste-Zup>

## REFERÊNCIAS

[DevSuperior · GitHub](#) [Bootcamp DevSuperior]

<https://digitalinnovation.one> [Bootcamp everis FullStack Developer]

<https://medium.com/@jovannypcg/understanding-springs-controlleradvice-cd96a364033f>

<https://blog.rocketseat.com.br/axios-um-cliente-http-full-stack/>



<https://github.com/axios/axios>

<https://pt-br.reactjs.org/docs/forms.html>

<https://www.baeldung.com/javax-validation>

<https://gist.github.com/fgilio/230ccd514e9381fafa51608fcf137253>

<https://pt-br.reactjs.org/docs/create-a-new-react-app.html>