

Para validações utilizaremos o **Bean Validation** que é uma especificação que permite validar elementos de uma forma prática e fácil. As restrições ficam inseridas nas classe do pacote model.

Incluir a dependência do spring-boot-starter-validation no pom.xml

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

Qual código será retornado caso o valor da descrição for nulo ou não preenchido o campo?

The screenshot shows a REST client interface with a POST request to `http://localhost:8080/produtos/`. The request body is a JSON object: `{ "descricao": null, "dataCadastro": "2021-03-25", "valor": "255" }`. The response status is `500 Internal Server Error`, highlighted with a red box. The response body is a JSON object: `{ "timestamp": "2021-03-03T18:50:40.277+00:00", "status": 500, "error": "Internal Server Error", "message": "", "path": "/produtos/" }`.

POST `http://localhost:8080/produtos/` Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   ... "descricao": null,
3   ... "dataCadastro": "2021-03-25",
4   ... "valor": "255"
5 }
```

Body Cookies Headers (4) Test Results Status: 500 Internal Server Error Time: 499 ms Size: 277 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   ... "timestamp": "2021-03-03T18:50:40.277+00:00",
3   ... "status": 500,
4   ... "error": "Internal Server Error",
5   ... "message": "",
6   ... "path": "/produtos/"
7 }
```

Vamos inserir a anotação **NotBlank** do pacote **javax.validation.constraints** inserindo para o atributo **descrição** e também a anotação **@Size** para definir o tamanho máximo do atributo.

```
@Not  
@Colu  
niva @ NotBlank - javax.validation.constraints
```

```
@NotBlank  
@Size(max = 40)  
@Column(name = "descricao", nullable = false, length = 40)  
private String descricao;
```

Para ativar a validação devemos inserir a anotação **@Valid** para os métodos **inserir** e **atualizar**

```
@PostMapping  
@ResponseStatus(HttpStatus.CREATED)  
public Produto inserir(@Valid @RequestBody Produto produto) {  
    return produtoRepository.save(produto);  
}
```

```
@PutMapping("/{id}")  
public ResponseEntity<Produto> atualizar(@Valid @PathVariable Long id, @RequestBody Produto produto) {  
    if (!produtoRepository.existsById(id)) {  
        return ResponseEntity.notFound().build();  
    }  
    produto.setId(id);  
    produto = produtoRepository.save(produto);  
    return ResponseEntity.ok(produto);  
}
```

Testando no Postman com o atributo descrição nulo e também com tamanho superior a quarenta. O erro 400 indica que o servidor não pode processar a requisição devido ao erro do cliente.

POST http://localhost:8080/produtos/

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "descricao": null,
3   "dataCadastro": "2021-03-25",
4   "valor": "255"
5 }
```

Body Cookies Headers (4) Test Results

Status: 400 Bad Request Time: 89 ms Size: 257 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "timestamp": "2021-03-04T01:28:52.535+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "",
6   "path": "/produtos/"
7 }
```

POST http://localhost:8080/produtos/

Params Authorization Headers (9) Body Pre-request Script Tests Settings

none form-data x-www-form-urlencoded raw binary GraphQL JSON

```
1 {
2   "descricao": "Pendrive Usb 3.0 - 256gb - Corsair Flash Voyager",
3   "dataCadastro": "2021-03-25",
4   "valor": "255"
5 }
```

Body Cookies Headers (4) Test Results

Status: 400 Bad Request Time: 53 ms Size: 257 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   "timestamp": "2021-03-04T01:36:36.163+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "",
6   "path": "/produtos/"
7 }
```

Usando a validação para quando o cliente inserir atributos desconhecidos. Por padrão o campo **tamanho** é ignorado e requisição é atendida.

The screenshot shows a REST client interface with a POST request to `http://localhost:8080/produtos/`. The request body is a JSON object with the following fields: `descricao`, `dataCadastro`, `valor`, and `tamanho`. The `tamanho` field is highlighted with a red box. The response is a JSON object with the following fields: `id`, `descricao`, `valor`, and `dataCadastro`.

Request:

```
POST http://localhost:8080/produtos/
{
  "descricao": "Tablet",
  "dataCadastro": "2021-03-25",
  "valor": "255",
  "tamanho": "7 polegadas"
}
```

Response:

```
{
  "id": 8,
  "descricao": "Tablet",
  "valor": 255.0,
  "dataCadastro": "2021-03-25T00:00:00.000+00:00"
}
```

Se inserirmos a linha destacada abaixo no arquivo **application.properties**, teremos como retorno o erro 400 bad request.

```
*application.properties
```

```
spring.datasource.url=jdbc:postgresql://localhost:5432/aula
spring.datasource.username=postgres
spring.datasource.password=postgres
spring.jpa.show-sql=true
spring.jpa.hibernate.ddl-auto= update
spring.jackson.deserialization.fail-on-unknown-properties=true
```

deserialização - Transformar de um objeto JSON para Java

POST http://localhost:8080/produtos/ Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   ... "descricao": "Tablet",
3   ... "dataCadastro": "2021-03-25",
4   ... "valor": "255",
5   ... "tamanho": "7 polegadas"
6 }
```

Body Cookies Headers (4) Test Results Status: 400 Bad Request Time: 346 ms Size: 257 B Save Response

Pretty Raw Preview Visualize JSON

```
1 {
2   ... "timestamp": "2021-03-04T02:13:42.035+00:00",
3   ... "status": 400,
4   ... "error": "Bad Request",
5   ... "message": "",
6   ... "path": "/produtos/"
7 }
```

Para manipular uma exceção lançada pela falha na validação devemos criar uma classe que será responsável por capturar e tratar esses erros.

Vamos criar a classe **ControllerExceptionHandler** no pacote **exception**

```
package org.serratec.java2backend.exercicio02.exception;

import org.springframework.web.bind.annotation.ControllerAdvice;
import org.springframework.web.servlet.mvc.method.annotation.ResponseEntityExceptionHandler;

@ControllerAdvice
public class ControllerExceptionHandler extends ResponseEntityExceptionHandler {

}
```

Podemos herdar como base a classe **ResponseEntityExceptionHandler**, ela já possui vários métodos que tratam exceções para que o usuário tenha uma resposta mais completa a respeito do erro lançado, sendo cada método para uma exceção específica.

@RestControllerAdvice - Com esta anotação estamos dizendo que a classe é um componente especializado do Spring para tratar exceções. Qualquer controlador que lançar uma **exceção** vai entrar em um métodos desta classe.

Testando no Postman vamos ter um tratamento simplificado de erros pela exceção gerada com corpo de resposta vazio.

The screenshot shows the Postman interface for a POST request to `http://localhost:8080/produtos/`. The request body is a JSON object: `{ "descricao": null, "dataCadastro": "2021-03-25", "valor": "255", "tamanho": "7 polegadas" }`. The status bar at the bottom indicates a **400 Bad Request** error, which is highlighted with a red rectangle. The response time is 423 ms and the size is 103 B. The response body is empty.

POST `http://localhost:8080/produtos/` Send

Params Authorization Headers (9) Body Pre-request Script Tests Settings Cookies

none form-data x-www-form-urlencoded raw binary GraphQL JSON Beautify

```
1 {
2   ... "descricao": null,
3   ... "dataCadastro": "2021-03-25",
4   ... "valor": "255",
5   ... "tamanho": "7 polegadas"
6 }
```

Body Cookies Headers (3) Test Results Status: 400 Bad Request Time: 423 ms Size: 103 B Save Response

Pretty Raw Preview Visualize Text

1

Quando alguma validação feita pelas anotações do Bean Validation falha é lançada uma exceção do tipo **MethodArgumentNotValidException**

A captura dessa exceção só é possível graças a anotação **@ExceptionHandler** que está na classe do Spring que herdamos. Essa anotação prove ao método a capacidade de tratar uma exceção quando ela for lançada. Para isso precisamos passar a classe da exceção como parâmetro da anotação e passar um objeto do tipo da exceção como parâmetro do método.

Vamos sobrescrever o método `handleMethodArgumentNotValid` e alterar o retorno para o método **`handleExceptionInternal`**

```
@ControllerAdvice
public class ControllerExceptionHandler extends ResponseEntityExceptionHandler {

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

Inserir a classe **ErroResposta** no pacote **exception** com construtor com todos argumentos, getter e setter.

```
public class ErroResposta {  
    private Integer status;  
    private String titulo;  
    private LocalDateTime dataHora;  
  
    public ErroResposta(Integer status, String titulo, LocalDateTime dataHora) {  
        super();  
        this.status = status;  
        this.titulo = titulo;  
        this.dataHora = dataHora;  
    }  
  
    public Integer getStatus() {  
        return status;  
    }  
  
    public void setStatus(Integer status) {  
        this.status = status;  
    }  
  
    public String getTitulo() {  
        return titulo;  
    }  
  
    public void setTitulo(String titulo) {  
        this.titulo = titulo;  
    }  
  
    public LocalDateTime getDataHora() {  
        return dataHora;  
    }  
  
    public void setDataHora(LocalDateTime dataHora) {  
        this.dataHora = dataHora;  
    }  
}
```

Criar a instância de **ErroResposta** na classe **ControllerExceptionHandler** e passar para o argumento que retorna o corpo da requisição.

```
@ControllerAdvice  
public class ControllerExceptionHandler extends ResponseEntityExceptionHandler {  
  
    @Override  
    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,  
        HttpHeaders headers, HttpStatus status, WebRequest request) {  
        ErroResposta erroResposta = new ErroResposta(status.value(), "Existem Campos Inválidos. Confira o preenchimento.",  
            LocalDateTime.now());  
  
        return super.handleExceptionInternal(ex, erroResposta, headers, status, request);  
    }  
}
```

Testar no Postman

The screenshot displays the Postman interface for a POST request to `http://localhost:8080/produtos/`. The request body is a JSON object with the following fields: `descricao` (null), `dataCadastro` ("2021-03-25"), `valor` ("255"), and `tamanho` ("7 polegadas"). The response status is 400 Bad Request, with a message: "Existem Campos Inválidos. Confira o preenchimento.".

Request Details:

- Method: POST
- URL: `http://localhost:8080/produtos/`
- Body Type: JSON
- Body Content:

```
1 {
2   ... "descricao": null,
3   ... "dataCadastro" : "2021-03-25",
4   ... "valor" : "255",
5   ... "tamanho": "7 polegadas"
6 }
```

Response Details:

- Status: 400 Bad Request
- Time: 542 ms
- Size: 258 B
- Save Response: [checked]
- Body Type: Pretty
- Body Content:

```
1 {
2   ... "status": 400,
3   ... "titulo": "Existem Campos Inválidos. Confira o preenchimento. ",
4   ... "dataHora": "2021-03-07T09:44:48.35"
5 }
```

Para sabermos quais campos estão gerando a exceção precisamos fazer algumas alterações no código.

```
@Entity
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    @NotBlank(message = "Preencha a descrição")
    @Size(max = 40, message = "Tamanho máximo 40")
    @Column(name = "descricao", nullable = false, length = 40)
    private String descricao;

    @DecimalMax(value = "5000", message = "O preço não pode ser maior que R${value}.00")
    @DecimalMin(value = "10", message = "O preço não pode ser menor que R${value}.00")
    @Column
    private BigDecimal valor;

    @Column(name = "data_cadastro")
    @Temporal(TemporalType.DATE)
    private Date dataCadastro;

    public Produto() {
    }

    public Long getId() {
        return id;
    }

    public String getDescricao() {
        return descricao;
    }

    public Date getDataCadastro() {
        return dataCadastro;
    }

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

Inserir a propriedade **message** que permite personalizar as mensagens para o usuário.

Inserir a anotação **@DecimalMax** e **@DecimalMin** responsável pelos valores máximos e mínimos para o atributo **valor**.

Vamos criar uma classe com o nome **ErroResposta** no pacote **exception** com os principais atributos de erro para resposta.

```
package org.serratec.java2backend.exercicio02.exception;

import java.time.LocalDateTime;
import java.util.List;

public class ErroResposta {
    private Integer status;
    private String titulo;
    private LocalDateTime dataHora;
    private List<String> erros;

    public ErroResposta(Integer status, String titulo, LocalDateTime dataHora, List<String> erros) {
        this.status = status;
        this.titulo = titulo;
        this.dataHora = dataHora;
        this.erros = erros;
    }

    public Integer getStatus() {
        return status;
    }

    public void setStatus(Integer status) {
        this.status = status;
    }

    public String getTitulo() {
        return titulo;
    }

    public void setTitulo(String titulo) {
        this.titulo = titulo;
    }

    public LocalDateTime getDataHora() {
        return dataHora;
    }
}
```

Inserir construtor com argumentos, getter e setter

Vamos alterar o método `handleMethodArgumentNotValid`

```
@ControllerAdvice
public class ControllerExceptionHandler extends ResponseEntityExceptionHandler {

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

        List<String> errors = new ArrayList<String>();
        for (FieldError error : ex.getBindingResult().getFieldErrors()) {
            errors.add(error.getField() + ": " + error.getDefaultMessage());
        }

        ErroResposta erroResposta = new ErroResposta(status.value(), "Existem campos inválidos", LocalDateTime.now(),
            errors);

        return super.handleExceptionInternal(ex, erroResposta, headers, status, request);
    }
}
```

O bind **BindingResult** reúne informações sobre erros que resultam da validação de uma instância de classe.

Obtemos uma coleção de instâncias do tipo **FieldError**, percorremos a coleção e recuperamos o nome do campo e a mensagem de erro para cada campo

Instanciamos a classe **ErroResposta** e passamos os argumentos