Desenvolvimento de APIs REST

04 - IoC, Injeção de Dependências e Tratamento de Exceções

- IoC e Injeção de Dependências
- Validação
- Exceções
- Personalização de Exceções

Revisão - Correção do Exercício

Classe **Cliente** no pacote **domain**

```java
@Entity
@Table(name="cliente")
public class Cliente {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(name="id_cliente")
   private Long id;
   @Column
   private String nome;
   @Column
   private String cpf;

   @Column
   private String email;

   public Long getId() {
       return id;
   }

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

   // gets e sets para os atributos restantes
```

Interface **ClienteRepository** no pacote **Repository**

```java
@Repository
public interface ClienteRepository extends JpaRepository<Cliente,Long> {


}
```

Criação da tabela **cliente**

```sql
create table cliente (
   id_cliente integer NOT NULL GENERATED BY DEFAULT AS IDENTITY,
   nome varchar(60),
   cpf varchar(11),
   email varchar(50),
   data_nascimento date,
   PRIMARY KEY (id_cliente)
);
```

Alterar a propriedade spring.jpa.hibernate.ddl-auto para none para que o hibernate não crie as tabelas automaticamente no banco de dados

```
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=none
```

Revisão - Correção do Exercício

Criar a classe **ClienteController**

Declaração da classe, repositório e métodos buscar e listar

```
@RestController
@RequestMapping("/clientes")
public class ClienteController {

    @Autowired
    private ClienteRepository clienteRepository;

    @GetMapping
    public List<Cliente> listar(){
        return clienteRepository.findAll();
    }

    @GetMapping("/{id}")
    public ResponseEntity<Cliente> buscar(@PathVariable Long id) {
        Optional<Cliente> cliente = clienteRepository.findById(id);
        if (cliente.isPresent()) {
            return ResponseEntity.ok(cliente.get());
        }
        return ResponseEntity.notFound().build();
    }
```

Métodos inserir, atualizar e remover

```
    @PostMapping
    @ResponseStatus(HttpStatus.CREATED)
    public Cliente inserir(@RequestBody Cliente cliente) {
        return clienteRepository.save(cliente);
    }

    @PutMapping("/{id}")
    public ResponseEntity<Cliente> atualizar(@PathVariable Long id, @RequestBody Cliente cliente) {
        if (!clienteRepository.existsById(id)) {
            return ResponseEntity.notFound().build();
        }
        cliente.setId(id);
        cliente=clienteRepository.save(cliente);
        return ResponseEntity.ok(cliente);
    }

    @DeleteMapping("/{id}")
    public ResponseEntity<Void> remover(@PathVariable Long id) {
        if(!clienteRepository.existsById(id)){
            return ResponseEntity.notFound().build();
        }
        clienteRepository.deleteById(id);
        return ResponseEntity.noContent().build();
    }
```

Ioc e Injeção de Dependência

Vamos criar um exemplo para explicar os conceitos de inversão de controle. Na classe **Exame** o método **calcularExame** retorna o valor do exame mais uma taxa de administração de 5% de acréscimo no valor do exame. Na classe consulta o método **calcularConsulta** possui uma taxa de administração de 10% de acréscimo no valor da consulta.

```
public class Exame {
    public Double calcularExame(Double valor) {
        return valor = valor * 0.05;
    }
}
```

```
public class Consulta {
    public Double calcularConsulta(Double valor) {
        return valor = valor + valor * 0.1;
    }
}
```

```
public class Pagamento {
    private Consulta consulta = new Consulta();
    private Exame exame = new Exame();
    public Double calcularProcedimento(Double valorConsulta, Double valorExame) {
        return consulta.calcularConsulta(valorConsulta) + exame.calcularExame(valorExame);
    }
}
```

No exemplo acima foi utilizada da forma acoplada onde criamos instâncias de **Consulta** e **Exame** dentro da classe **Pagamento** o que torna difícil a manutenção futura do código. A classe **Pagamento** depende de **Consulta** e **Exame**.

Ioc e Injeção de Dependência

Para executar o exemplo anterior no Spring Boot vamos implementar na classe de execução do Spring Boot o método **CommandLineRunner**, implementar o método **run** e testar executando como Java Application

```
@SpringBootApplication
public class Exercicio01Application implements CommandLineRunner{

   public static void main(String[] args) {
       SpringApplication.run(Exercicio03Application.class, args);
   }

   @Override
   public void run(String... args) throws Exception {
       Pagamento pagamento = new Pagamento();
       System.out.println("Total a pagar:" + pagamento.calcularProcedimento(200.0, 80.0));
   }

}
```

```
2022-09-03 17:41:11.197  INFO 1312061 --- [  restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer  : Tomcat started on port(s): 8080 (http) with context path ''
2022-09-03 17:41:11.200  INFO 1312061 --- [  restartedMain] o.s.b.e.Exercicio01Application           : Started Exercicio01Application in 5.861 seconds (JVM running for 7.454)
Total a pagar:224.0
```

Ioc e Injeção de Dependência

Melhorando o código aplicando a inversão de controle

Vamos modificar a classe de **Pagamento** para retirar as instâncias desta classe, agora quem vai controlar as injeções de dependências será a classe com **main** e iremos passar os objetos através do construtor da classe **Pagamento**.

```
public class Pagamento {
   private Consulta consulta ;
   private Exame exame ;

   public Pagamento(Consulta consulta, Exame exame){
       this.consulta = consulta;
       this.exame = exame;

   }
   public Double calcularProcedimento(Double valorConsulta,
                                      Double valorExame) {
       return consulta.calcularConsulta(valorConsulta) +
               exame.calcularExame(valorExame);
   }


}
```

```
@SpringBootApplication
public class Exercicio03Application implements CommandLineRunner{

   public static void main(String[] args) {
       SpringApplication.run(Exercicio01Application.class, args);
   }

   @Override
   public void run(String... args) throws Exception {
       Consulta consulta = new Consulta();
       Exame exame = new Exame();
       Pagamento pagamento = new Pagamento(consulta,exame);
       System.out.println("Total a pagar:" +
                          pagamento.calcularProcedimento(200.0, 80.0));
   }

}
```

Ioc e Injeção de Dependência

Realizando a injeção de dependência pelo Spring

Vamos anotar nossas classes **Pagamento**, **Consulta** e **Exame** com a anotação **@Component** para que elas possam ser gerenciadas pelo Spring. Podemos utilizar outras anotações como **@Service** ou **@Configuration** por exemplo.

```
@Component
public class Pagamento {
   private Consulta consulta ;
   private Exame exame ;

   public Pagamento(Consulta consulta, Exame exame){
       this.consulta = consulta;
       this.exame = exame;

   }
   public Double calcularProcedimento(Double valorConsulta,
                                      Double valorExame) {
       return consulta.calcularConsulta(valorConsulta) +
               exame.calcularExame(valorExame);
   }


}
```

Vamos deixar o construtor da forma que estava e o próprio Spring realizará a injeção de dependência através do construtor definido na classe Pagamento.

IoC - Inversão de Controle

Nossa aplicação não é mais responsável por criar componentes, o Spring passa a ser o responsável

- Controllers
- Services
- Beans
- Etc..

Ioc e Injeção de Dependência

Realizando a injeção de dependência pelo Spring

Vamos modificar nossa classe principal definindo um atributo para **Pagamento** e anotando com **@Autowired**

```
@SpringBootApplication
public class Exercicio03Application implements CommandLineRunner{

    @Autowired
    private Pagamento pagamento;

    public static void main(String[] args) {
        SpringApplication.run(Exercicio01Application.class, args);
    }

    @Override
    public void run(String... args) throws Exception {
        System.out.println("Total a pagar:" +
                pagamento.calcularProcedimento(200.0, 80.0));
    }
}
```

```
2022-09-03 17:51:41.758  INFO 1316628 --- [  restartedMain] o.s.b.w.embedded.
2022-09-03 17:51:41.767  INFO 1316628 --- [  restartedMain] o.s.b.e.Exercicio
Total a pagar:224.0
```

~~new~~

Injeção de Dependência

Os componentes são “injetados” onde são necessários

- Atributos com anotação **@Autowired**
- Valores (componentes) requisitados no construtor de componentes

O que são Beans?

São objetos gerenciados pelo próprio Spring. O Spring fornece algumas anotações como @Component, @Repository, @Service entre outras. Com o @Autowired não precisaremos fazer new. O autowired serve para dizer que queremos usar uma instância de uma determinada propriedade.

Ioc e Injeção de Dependência

Realizando a injeção de dependência pelo Spring sem o construtor na classe Pagamento

Outra forma de realizar a injeção de dependência é anotando os atributos com a anotação **@Autowired**. Vamos retirar o construtor da classe **Pagamento** e usar a anotação **@Autowired** para os atributos **consulta** e **exame**.

```
@Component
public class Pagamento {
   @Autowired
   private Consulta consulta ;
   @Autowired
   private Exame exame ;

   public Double calcularProcedimento(Double valorConsulta, Double valorExame) {
       return consulta.calcularConsulta(valorConsulta) +
               exame.calcularExame(valorExame);
   }
}
```

Validação

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.

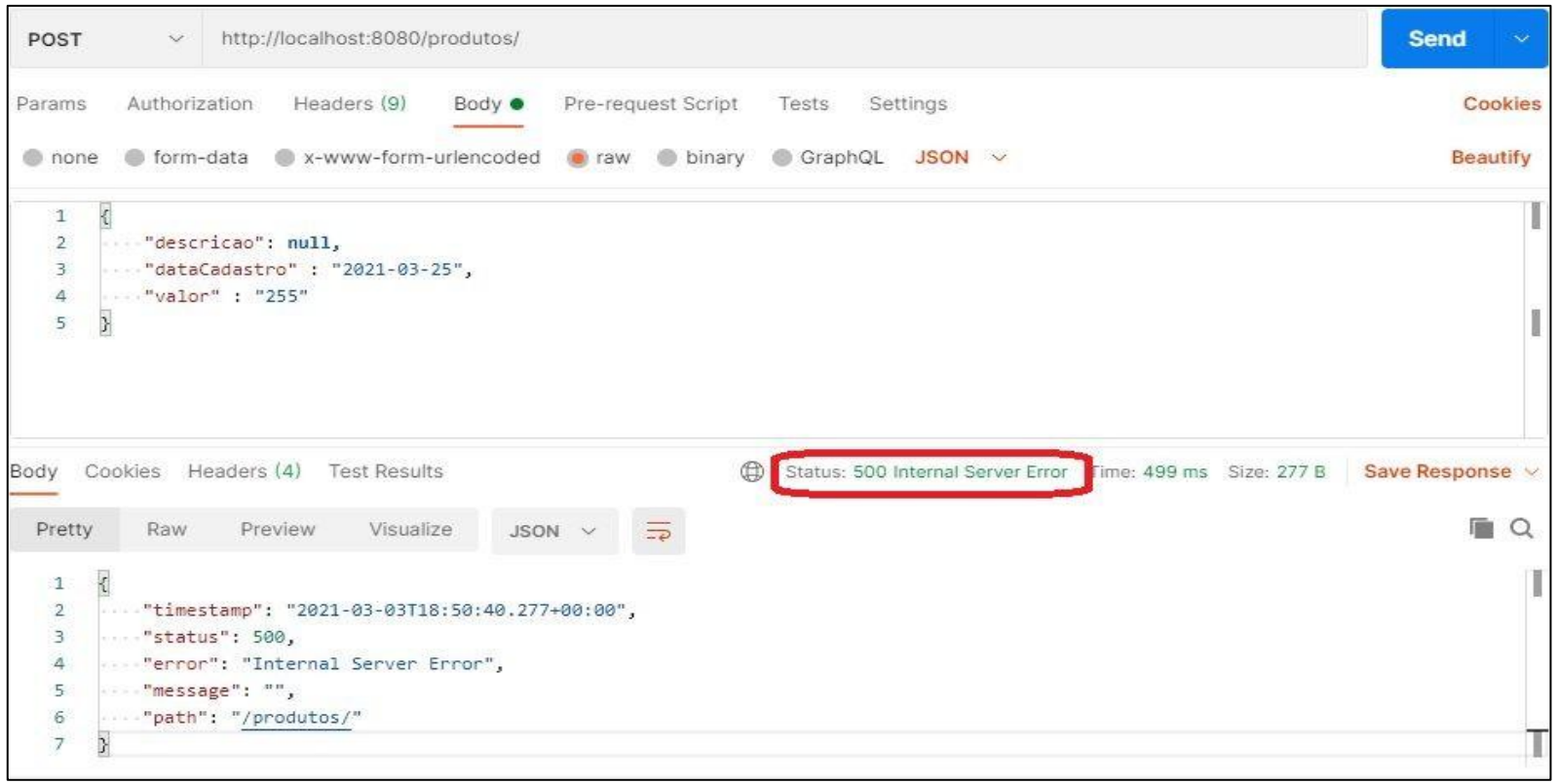
Incluir a dependência do spring-boot-starter-validation no pom.xml

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

Quando adicionamos uma nova dependência, eventualmente ocorrem erros no **pom.xml**. Estes podem ser corrigidos fazendo uma atualização dos repositórios. Clique com o botão direito sobre o projeto M**aven – Update Project** selecione o projeto e antes de clicar em ok, verifique se a opção **Force Update of Snapshot/Releases** está selecionada**.**

Validação

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



serratec

Validação

Vamos inserir a anotação **Notblank** do pacote **jakarta.validation.constraints** para o atributo **descrição** e também a anotação **@Size** do mesmo pacote para definir o tamanho máximo do atributo.

```
import javax.validation.constraints.NotBlank;
import javax.validation.constraints.Size;

// ...

   @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**

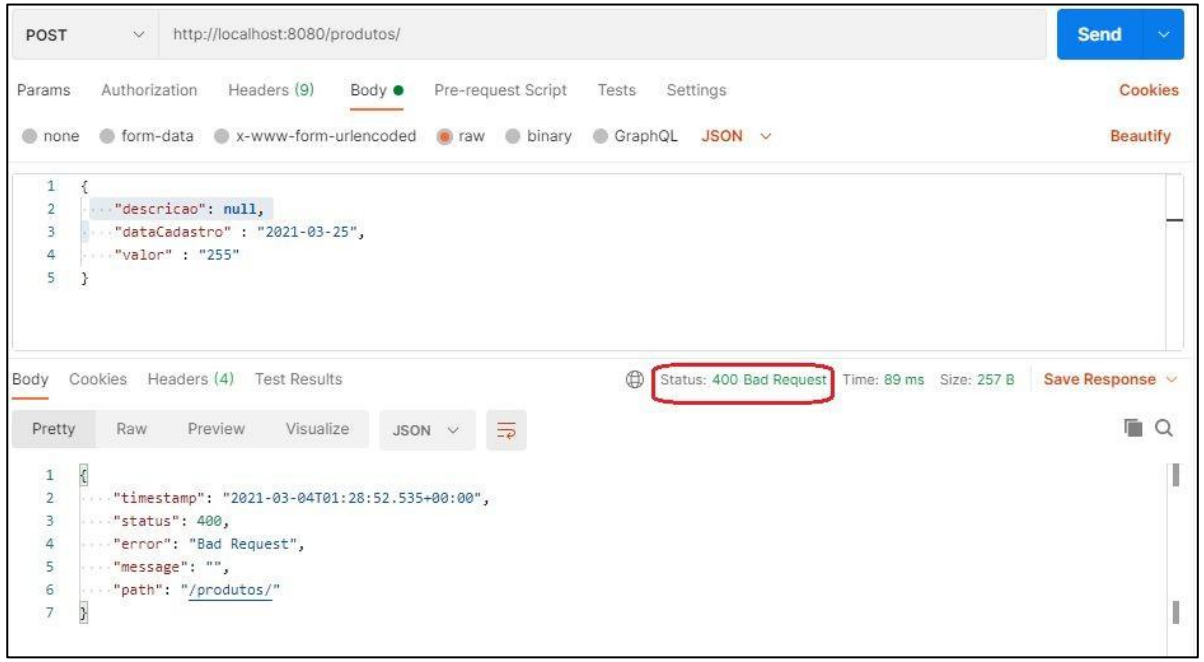
```
@PutMapping("/{id}")
public ResponseEntity<Produto> atualizar(@Valid @RequestBody Produto produto, @PathVariable Long id) {
    Optional<Produto> produtoOptional = produtoRepository.findById(id);
    if (!produtoOptional.isPresent()) {
        return ResponseEntity.notFound().build();
    }
// ...
```

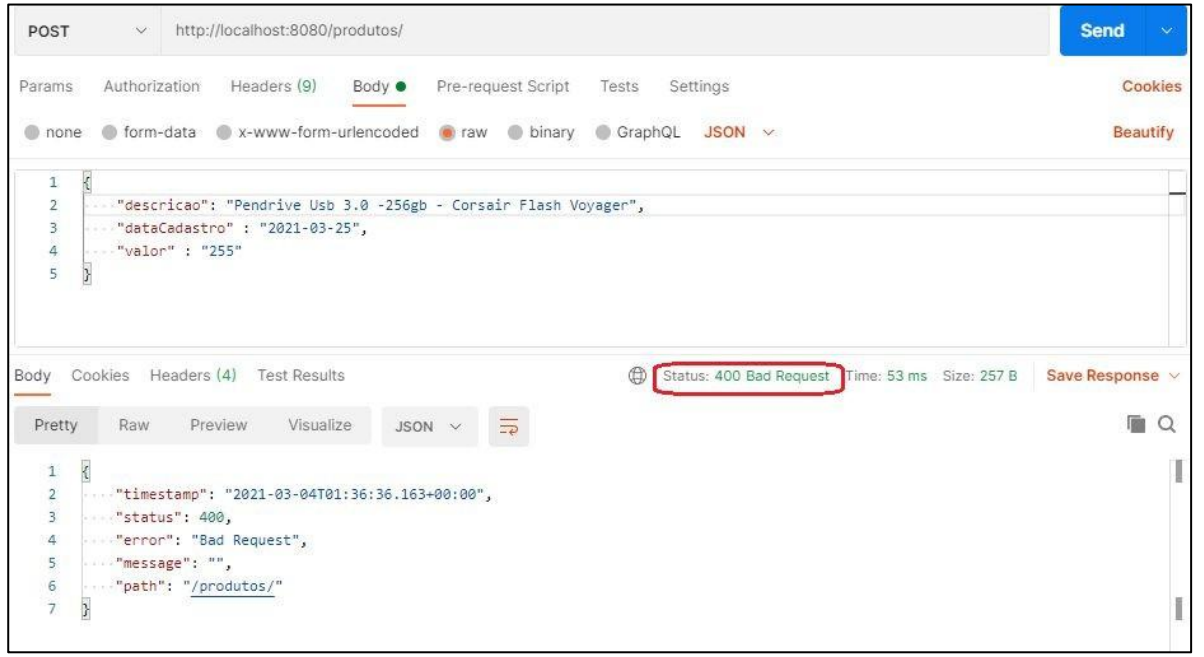
```
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Produto inserir(@Valid @RequestBody Produto produto) {
    return produtoRepository.save(produto);
}
```

serratec

Validação

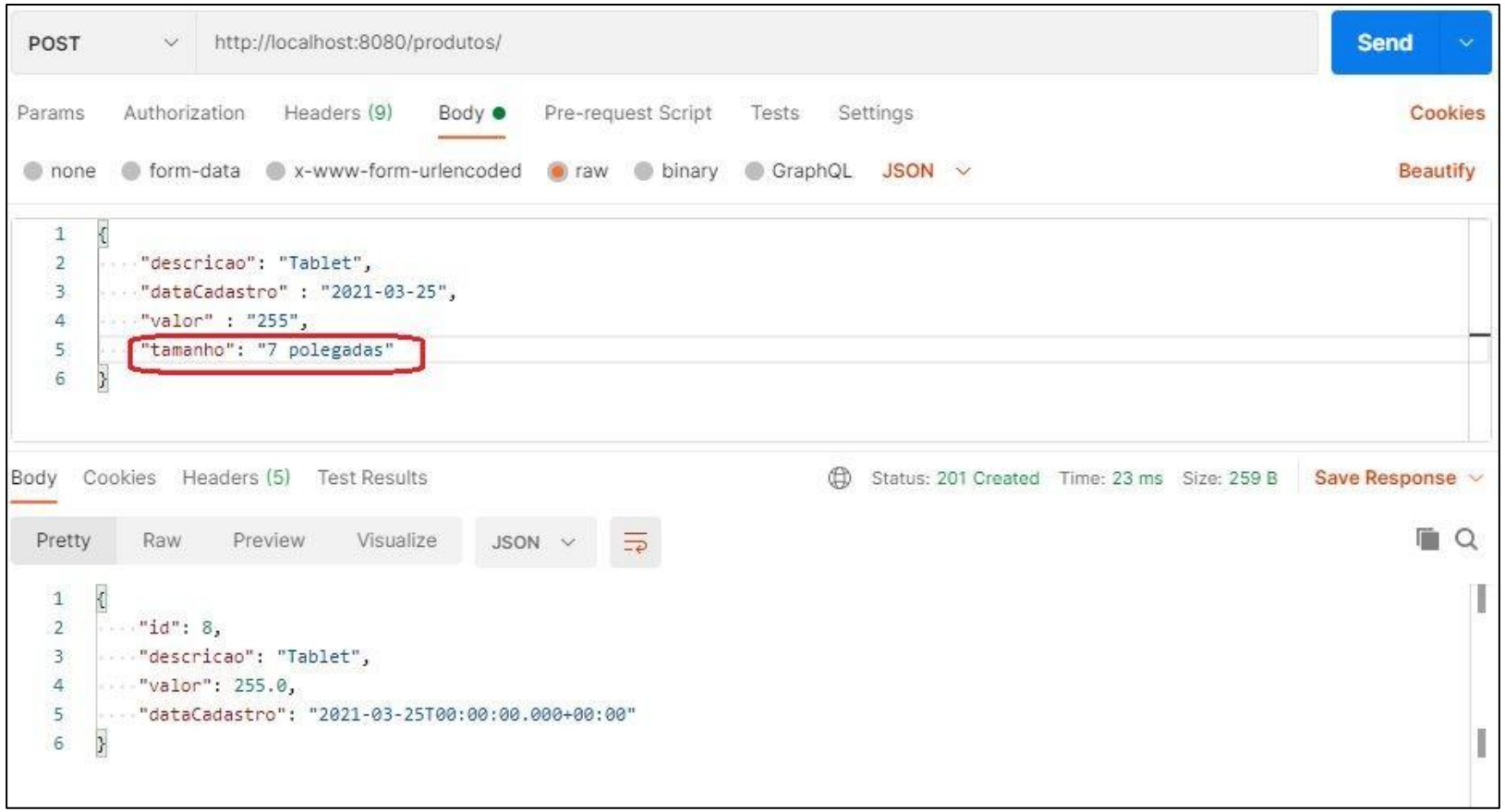
Testando no Postman com o atributo descrição nulo e também com tamanho superior a quarenta. O erro 400 indica que o servidor não pode processar a requisição devido ao erro do cliente.





Validação

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



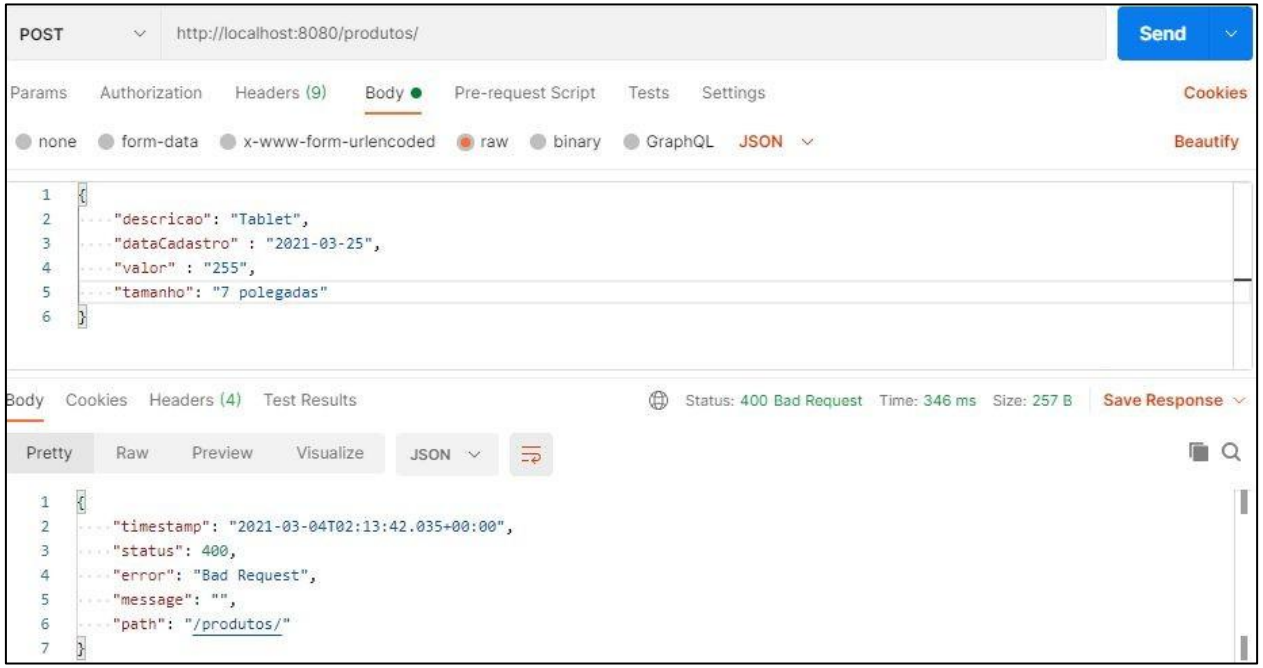
serratec

Validação

Se inserirmos a pripriedade **spring.jackson.deserialization.fail-on-unknow-properties** (falhar para propriedades desconhecidas), no arquivo **application.properties**, teremos como retorno o erro 400 bad request.

```
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=none
spring.jackson.deserialization.fail-on-unknown-properties=true
```

deserialização - Transformar de um objeto JSON para Java



serratec

Exceções

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**

```
@ControllerAdvice
public class ControllerExeptionHandler extends ResponseEntityExceptionHandler{

}
```

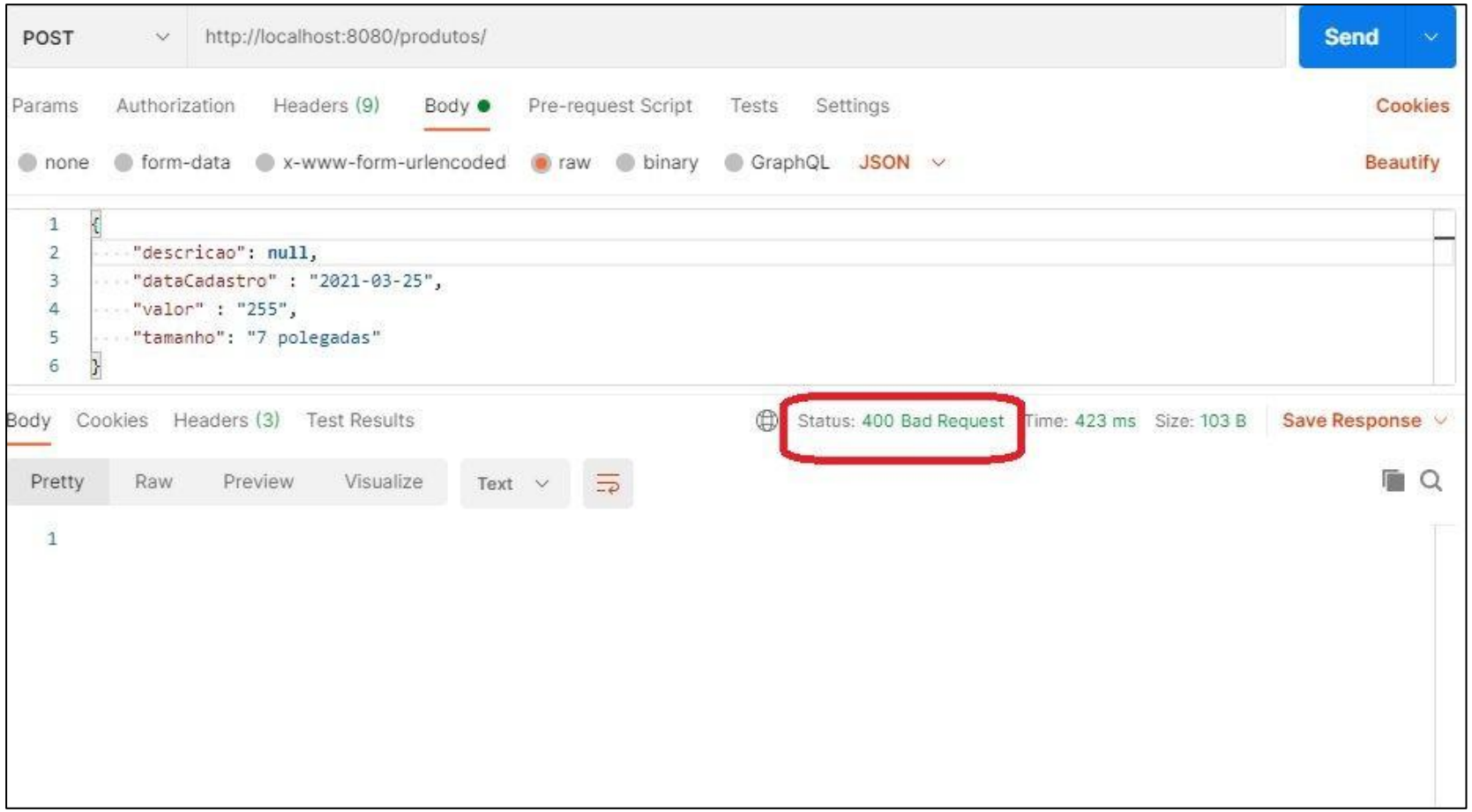
Podemos herdar como base a classe **ResponseEntityExceptionHandler**, ela já possui vários métodos que tratam exceções para que o usuário tenha uma resposta mais completa a respeito do erro lançado, sendo cada método para uma exceção específica.

- Com esta anotação **@ControllerAdvice** 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.

Ela intercepta todas as exceções que foram geradas a partir de um recurso da sua aplicação.

Exceções

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



Exceções

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

Exceções

```java
public class ErroResposta {
   private Integer status;
   private String titulo;
   private LocalDateTime dataHora;

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

  //... gets e sets
```

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

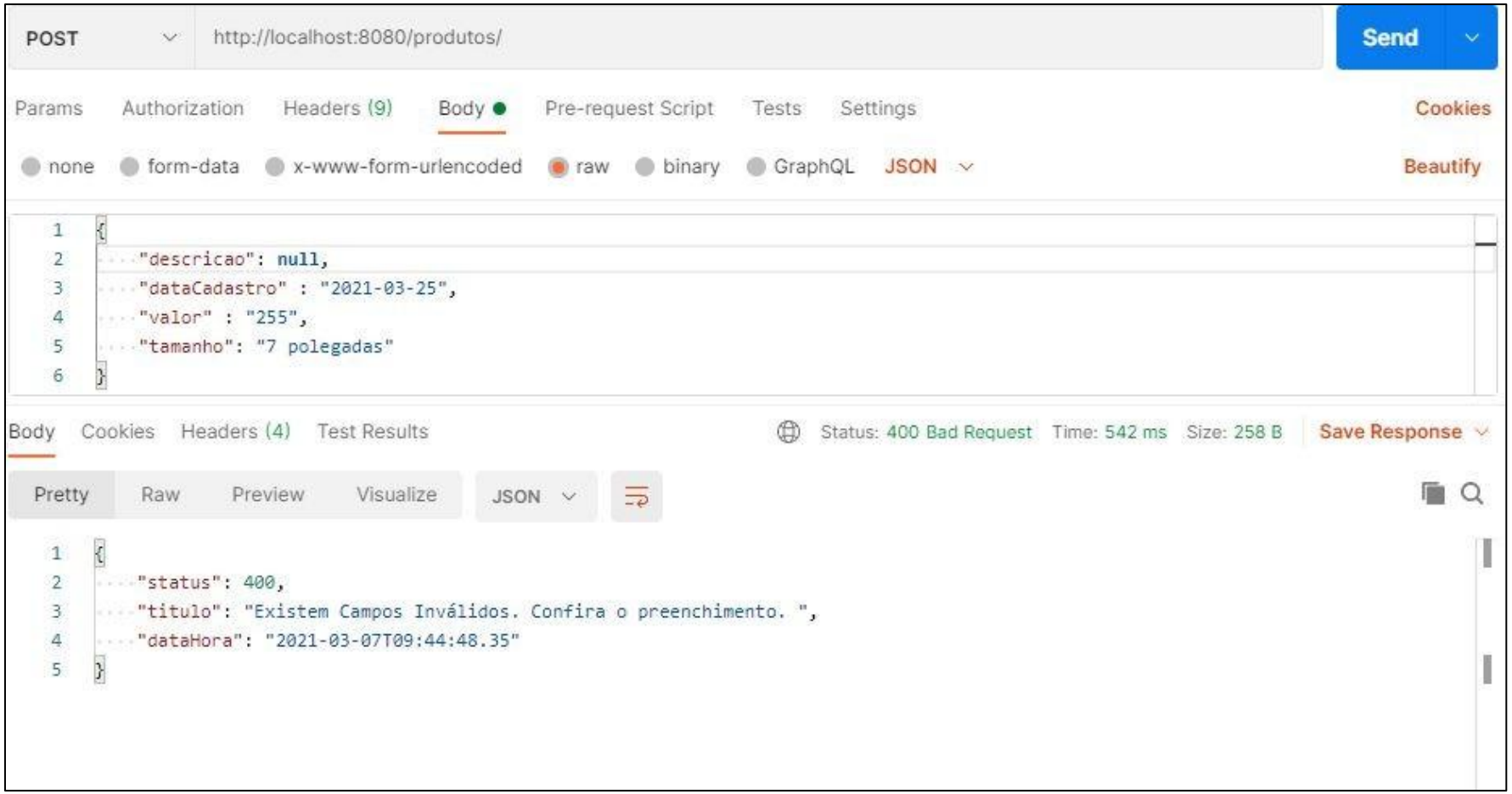
Criar a instância de **ErroResposta** na classe **ControllerExceptionHandler** e passar para o argumento que retorna o corpo da requisição.

```java
@ControllerAdvice
public class ControllerExeptionHandler 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 preechimento", LocalDateTime.now());
       return super.handleExceptionInternal(ex, erroResposta, headers, status, request);
   }
}
```

Exceções

Testar no Postman



Personalizar Exceção

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

```
@Entity
@Table(name = "produto")
public class Produto {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id_produto")
    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${value}.00")
    @DecimalMin(value="10", message="O preço não pode ser melhor que R${value}.00")
    @Column
    private BigDecimal valor;

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

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**.

Personalizar Exceção

Vamos adicionar o atributo **erros** na classe **ErroResposta**

```
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;

   }
```

Inserir construtor com argumentos, getter e setter para a lista de **erros**

Personalizar Exceção

Vamos alterar o método **handleMethodArgumentNotValid**

```
@ControllerAdvice
public class ControllerExeptionHandler extends ResponseEntityExceptionHandler{
   @Override
   protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
           HttpHeaders headers, HttpStatus status, WebRequest request) {



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



       ErroResposta erroResposta = new ErroResposta(status.value(), "Existem Campos Inválidos, Confira o preechimento", LocalDateTime.now(),
erros);
       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

Exercício

Criar validações na classe **Cliente** do exercício da aula passada:

- O atributo **nome** poderá ser nulo e tamanho máximo de 60.
- Utilizar a anotação para validar o **cpf.**
- Utilizar a anotação para validar o **email.**
- Adicionar a propriedade **message** com uma mensagem personalizada de validação para cada atributo.

Fazer o teste no Postman para verificar se as validações estão funcionando.

Exercício

Inserir as anotações na classe **Cliente**

```java
@Entity
@Table(name="cliente")
public class Cliente {

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(name="id_cliente")
    private Long id;

    @NotBlank(message="Preencha o nome")
    @Size(max=60)
    @Column
    private String nome;

    @CPF(message="CPF Inválido")
    @Column
    private String cpf;

    @Email(message="Email inválido")
    @Column
    private String email;
```

Inserir **@Valid** no argumento antes do **@RequestBody** dos métodos **inserir** e **atualizar**.

```java
@PostMapping
@ResponseStatus(HttpStatus.CREATED)
public Cliente inserir(@Valid @RequestBody Cliente cliente) {
    return clienteRepository.save(cliente);
}

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