

Criando uma API CRUD simples em Rest utilizando JAVA + Spring + Hibernate + MySql + Bean Validation

Nesse projeto, iremos construir uma aplicação CRUD simples utilizando JAVA versão 8 como linguagem de programação, Maven para gerenciamento de dependências, Spring Boot como framework de gerenciamento geral, Spring Web para criação de aplicações REST, Spring Data JPA com Hibernate e MYSQL para persistência de dados e o Bean Validation para validação dos dados. Essas tecnologias serão usadas para facilitar a codificação e por ser as tecnologias mais utilizados atualmente, atingindo um excelente nível de maturidade e robustez tanto para aplicações de pequeno porte até as de grande porte e complexas.

Iremos criar as seguintes classes em nosso projeto:

- **Usuario.java**: Classe Model responsável por criar a entidade, os atributos, getters e setters, além das validações necessárias. A mesma será persistida no Banco de Dados sendo convertida via ORM pelo Spring JPA.
- **UsuarioRepository.java**: Classe responsável por realizar as operações do Banco de Dados. O Spring Data é muito poderoso e já possui toda a lógica de implementação e conversão necessárias se tornando muito simples o controle de nossas entidade e conversão de Objeto para Entidade de Banco de Dados, sendo necessário, apenas, estender alguma de suas interfaces que em nosso caso utilizaremos JpaRepository. Toda a lógica de acesso, inserção, deleção e paginação entre outras é abstraída.
- **UsuarioController.java**: Classe Controller do pacote Spring Web onde iremos mapear nossos endpoints, receber as requisições de dados, efetuar a chamada adequada ao Banco de Dados e retornar as respostas adequadas. Como estamos criando uma aplicação REST, o padrão de requisição resposta será via JSON, um padrão criado para integração entre diversos sistemas e aplicações mesmo que utilizem linguagem de programação e infraestrutura diferentes.
- **MyExceptionHandler.java**: Classe responsável pelo tratamento de exceções lançadas pelo nosso sistema, como por exemplo falha de Banco de Dados e falha na validação dos dados. Para manter o padrão de respostas de retorno do Spring, iremos criar uma função `createBodyResponse` para retornar as mensagens de erro via JSON da mesma forma que o Spring retornaria.

Criação do Projeto via Spring Initializr

Antes de mais nada, precisamos criar um projeto no site Spring Initializr pelo link: <https://start.spring.io/>. No site selecionamos Project "Maven Project", Language "Java", Spring Boot "2.4.1" (devido ser a versão recente mais estável), Packaging "War" e Java "8". Então preencheremos o Project Metadata conforme nosso projeto e incluiremos as seguintes dependências necessárias:

Project

☒ Maven Project
☐ Gradle Project

Language

☒ Java ☐ Kotlin ☐ Groovy

Spring Boot

☐ 2.5.0 (SNAPSHOT) ☐ 2.4.2 (SNAPSHOT) ☒ 2.4.1
☐ 2.3.8 (SNAPSHOT) ☐ 2.3.7

Project Metadata

Group

Artifact

Name

Description

Package name

Packaging ☐ Jar ☒ War

Java ☐ 15 ☐ 11 ☒ 8

Dependencies ADD DEPENDENCIES... CTRL + B

MySQL Driver SQL
MySQL JDBC and R2DBC driver.

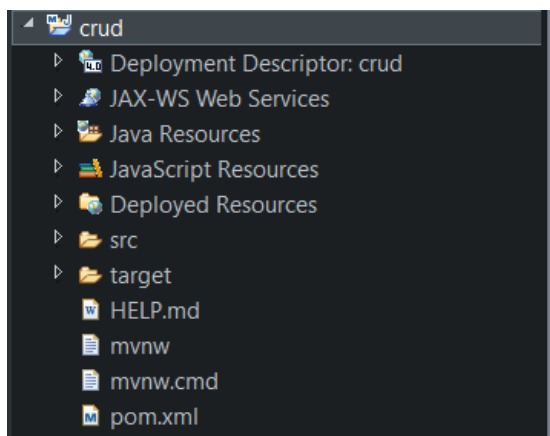
Spring Data JPA SQL
Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.

Spring Web WEB
Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.

Validation I/O
Bean Validation with Hibernate validator.

GENERATE CTRL + G EXPLORE CTRL + SPACE SHARE...

Após isso, clicamos em GENERATE, baixamos o projeto gerado em formato zip e extraímos para nosso workspace. Em seguida abrimos uma IDE de preferência que no nosso caso será o Eclipse IDE, importamos o projeto clicando em FILE -> IMPORT -> Selecionamos Existing Maven Projects -> NEXT -> Em root directory selecionamos a pasta do projeto que acabamos de extrair -> FINISH. Aguardamos até que o Eclipse reconheça e monte as pastas do nosso projeto, pode demorar um pouco e então o mesmo gera a seguinte estrutura básica:



Application.properties

Vamos definir os dados de nosso Banco de Dados no arquivo application.properties que geralmente fica na pasta main/resources, note que devemos informar a Timezone nessa nossa versão do Driver MySQL, usamos, também, `spring.jpa.hibernate.ddl-auto=update` para que o spring crie as tabelas no Banco de Dados automaticamente e `server.error.include-message=Always` para que as mensagens de exceção personalizadas sejam exibidas via JSON:

```
1 spring.datasource.url=jdbc:mysql://localhost:3306/crud?serverTimezone=America/Sao_Paulo
2 spring.datasource.username=root
3 spring.datasource.password=
4 spring.jpa.hibernate.ddl-auto=update
5
6 server.error.include-message=always
```

Classe Model

Vamos começar criando a nossa classe model referente a Usuario. Para fins de organização vamos criar uma pasta(package) Model e inserir nossa classe Usuario dentro da mesma. Nossa classe vai receber os atributos CPF, Nome, E-mail e Data de nascimento. Devemos anotar nossa classe como @Entity para ser uma classe de persistência no Banco de Dados, anotamos o atributo CPF como @Id para ser a nossa chave primária além de definirmos que CPF e E-mail serão atributos únicos no Banco de Dados para não permitirmos registros duplicados com esses atributos. Definimos as validações utilizando as anotações do Bean Validation e JsonFormat para serialização e desserialização automáticas para resposta da aplicação. Geramos os getters and setters e finalmente nossa classe terá a seguinte estrutura:

```
@Entity
public class Usuario {
    @Id
    @NotEmpty(message = "CPF é obrigatório")
    @CPF(message = "CPF Inválido")
    @Column(unique = true)
    String cpf;
    @NotEmpty(message = "Nome é obrigatório")
    String nome;
    @NotEmpty(message = "E-mail é obrigatório")
    @Email(message = "E-mail inválido")
    @Column(unique = true)
    String email;
    @DateTimeFormat(pattern = "MM/dd/yyyy")
    @JsonFormat(pattern = "dd/MM/yyyy")
    private LocalDate dataNascimento;

    public String getCpf() {
        return cpf;
    }
    public void setCpf(String cpf) {
        this.cpf = cpf;
    }
    public String getNome() {
        return nome;
    }
    public void setNome(String nome) {
        this.nome = nome;
    }
    public String getEmail() {
        return email;
    }
    public void setEmail(String email) {
        this.email = email;
    }
    public LocalDate getDataNascimento() {
        return dataNascimento;
    }
    public void setDataNascimento(LocalDate dataNascimento) {
        this.dataNascimento = dataNascimento;
    }
}
```

Classe Repository

Agora criaremos nossa Classe de acesso ao Banco de Dados, o Spring facilita muito sendo necessário apenas de uma interface que estenda JpaRepository:

```
1 package br.com.banco.crud.Repository;
2
3 import org.springframework.data.jpa.repository.JpaRepository;
4
5 import br.com.banco.crud.Model.Usuario;
6
7 public interface UsuarioRepository extends JpaRepository<Usuario, String>{
8
9 }
```

Classe Controller

Nesse momento já podemos criar nosso Controller, que irá receber as requisições e devolver as respostas no formato JSON. Anotamos nossa classe com @RestController e injetamos a dependência do nosso repositório utilizando @Autowired, criamos os endpoints responsáveis pela criação, alteração, exclusão e lista dos usuários:

```
@RestController
public class UsuarioController {
    @Autowired
    private UsuarioRepository repository;

    @GetMapping("/usuarios")
    public List<Usuario> Get() {
        return repository.findAll();
    }

    @GetMapping("/usuarios/{cpf}")
    public Usuario GetById(@PathVariable String cpf) {
        return repository.findById(cpf).get();
    }

    @PostMapping("/usuarios")
    public Usuario Post(@Valid @RequestBody(required = false) Usuario usuario) {
        return repository.save(usuario);
    }

    @PutMapping("/usuarios/{cpf}")
    public ResponseEntity<Usuario> Put(@Valid @RequestBody Usuario novoUsuario, @PathVariable String cpf) {
        return repository.findById(cpf)
            .map(usuario -> {
                usuario.setNome(novoUsuario.getNome());
                usuario.setEmail(novoUsuario.getEmail());
                usuario.setDataNascimento(novoUsuario.getDataNascimento());
                repository.save(usuario);
                return new ResponseEntity<>(usuario, HttpStatus.OK);
            })
            .orElseThrow(() -> new IllegalArgumentException());
    }

    @DeleteMapping("/usuarios/{cpf}")
    public void Delete(@PathVariable String cpf) {
        repository.deleteById(cpf);
    }
}
```

Classe de tratamento de exceções

O framework Spring possui um comportamento padrão de tratamento de exceções, mas em nosso caso, iremos sobrescrever alguns de seus comportamentos para devolver respostas personalizadas para o cliente que consumir a aplicação. Conforme dito anteriormente, criaremos uma função privada na classe (createBodyResponse) para retornar a mensagem no mesmo padrão do Spring. Nossa classe “MyExceptionHandler” irá estender a classe padrão do Spring “ResponseEntityExceptionHandler” e sobrescrever alguns de seus métodos, além de tratar algumas exceções não tratadas pela superclasse, ficando da seguinte forma:

```
@RestControllerAdvice
public class MyExceptionHandler extends ResponseEntityExceptionHandler {

    • @Override
    protected ResponseEntity<Object> handleMethodArgumentNotValid(MethodArgumentNotValidException ex,
        HttpHeaders headers, HttpStatus status, WebRequest request) {
        List<String> erros = new ArrayList<String>();
        for(ObjectError oe : ex.getBindingResult().getAllErrors())
            erros.add(oe.getDefaultMessage());
        return new ResponseEntity<>(createBodyResponse(headers, status, erros), headers, status);
    }

    • @Override
    protected ResponseEntity<Object> handleHttpMessageNotReadable(HttpMessageNotReadableException ex,
        HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<>(createBodyResponse(headers, status,
            ex.getMostSpecificCause().getMessage()), headers, status);
    }

    • @Override
    protected ResponseEntity<Object> handleHttpRequestMethodNotSupported(
        HttpRequestMethodNotSupportedException ex,
        HttpHeaders headers, HttpStatus status, WebRequest request) {
        return new ResponseEntity<>(createBodyResponse(headers, status, ex.getMessage()), headers, status);
    }

    • private Map<String, Object> createBodyResponse(
        HttpHeaders headers, HttpStatus status, Object erros) {
        Map<String, Object> body = new LinkedHashMap<>();
        body.put("timestamp", new Date());
        body.put("status", status.value());
        body.put("error", status.getReasonPhrase());
        body.put("message", erros);
        body.put("path", ServletUriComponentsBuilder.fromCurrentRequest().toUriString());
        return body;
    }

    • @ExceptionHandler({ConstraintViolationException.class})
    public void constraintViolationException(
        HttpServletResponse response, ConstraintViolationException ex) throws IOException {
        response.sendError(HttpStatus.INTERNAL_SERVER_ERROR.value(), "E-mail ou CPF já registrado");
    }

    • @ExceptionHandler({IllegalArgumentException.class})
    public void illegalArgumentException(HttpServletResponse response) throws IOException {
        response.sendError(HttpStatus.INTERNAL_SERVER_ERROR.value(), "Falha ao buscar/salvar o usuário");
    }

    • @ExceptionHandler({EmptyResultDataAccessException.class, NoSuchElementException.class})
    public void emptyResultDataAccessException(HttpServletResponse response) throws IOException {
        response.sendError(HttpStatus.INTERNAL_SERVER_ERROR.value(), "Usuário não localizado");
    }

}
```

Testando a aplicação

Nossa aplicação já está pronta para funcionar, iremos utilizar o Tomcat como servidor para rodar nossa aplicação. Iremos iniciar diretamente pelo Tomcat sem utilizar nenhuma classe para isso e seu log de saída irá se apresentar da seguinte forma ao iniciar:

```

Spring Boot (v2.4.1)
:: Spring Boot ::

2021-01-08 14:47:33.131 INFO 18088 --- [main] br.com.banco.crud.ServletInitializer : Starting ServletInitializer v0.0.1-SNAPSHOT using Java 1.8.0_271 on DESKTOP-8F1F005 with PID 18088 (C:\ec\
2021-01-08 14:47:33.151 INFO 18088 --- [main] br.com.banco.crud.ServletInitializer : No active profile set, falling back to default profiles: default
2021-01-08 14:47:36.361 INFO 18088 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data JPA repositories in DEFAULT mode.
2021-01-08 14:47:36.504 INFO 18088 --- [main] s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in 151 ms. Found 1 JPA repository interfaces.
2021-01-08 14:47:37.758 INFO 18088 --- [main] o.a.c.c.C.[Catalina].[localhost].[/] : Initializing Spring embedded WebApplicationContext
2021-01-08 14:47:37.759 INFO 18088 --- [main] w.s.c.ServletWebServerApplicationContext : Root WebApplicationContext: initialization completed in 4309 ms
2021-01-08 14:47:38.794 INFO 18088 --- [main] o.hibernate.jpa.internal.util.LogHelper : HH0000204: Processing PersistenceUnitInfo [name: default]
2021-01-08 14:47:39.182 INFO 18088 --- [main] org.hibernate.Version : HH0000412: Hibernate ORM core version 5.4.25.Final
2021-01-08 14:47:39.917 INFO 18088 --- [main] o.hibernate.annotations.common.Version : HCANN000001: Hibernate Commons Annotations (5.1.2.Final)
2021-01-08 14:47:40.534 INFO 18088 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2021-01-08 14:47:41.071 INFO 18088 --- [main] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2021-01-08 14:47:41.313 INFO 18088 --- [main] org.hibernate.dialect.Dialect : HH0000400: using dialect: org.hibernate.dialect.MySQL55Dialect
2021-01-08 14:47:44.445 INFO 18088 --- [main] o.h.e.t.j.p.i.JtaPlatformInitiator : HH0000490: Using JtaPlatform implementation: [org.hibernate.engine.transaction.jta.platform.internal.NoJta
2021-01-08 14:47:44.485 INFO 18088 --- [main] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFactory for persistence unit 'default'
2021-01-08 14:47:46.443 WARN 18088 --- [main] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enabled by default. Therefore, database queries may be performed during view re
2021-01-08 14:47:46.670 INFO 18088 --- [main] o.s.s.concurrent.ThreadPoolTaskExecutor : Initializing ExecutorService 'applicationTaskExecutor'
2021-01-08 14:47:48.291 INFO 18088 --- [main] br.com.banco.crud.ServletInitializer : Started ServletInitializer in 16.973 seconds (JVM running for 28.691)
2021-01-08 14:47:48.464 INFO 18088 --- [main] org.apache.catalina.startup.Catalina : Server startup in [26365] milliseconds
2021-01-08 14:47:48.753 INFO 18088 --- [nio-8080-exec-1] o.a.c.c.C.[Catalina].[localhost].[/] : Initializing Spring DispatcherServlet 'dispatcherServlet'
2021-01-08 14:47:48.754 INFO 18088 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Initializing Servlet 'dispatcherServlet'
2021-01-08 14:47:48.757 INFO 18088 --- [nio-8080-exec-1] o.s.web.servlet.DispatcherServlet : Completed initialization in 2 ms
```

Após iniciar, vamos efetuar os testes de nossa aplicação, utilizaremos o Postman Desktop para isso.

Teste HTTP POST

- Utilizando uma data errada:

POST

http://localhost:8080/usuarios

Params

Authorization

Headers (8)

Body

Pre-request Script

Tests

Settings

none

form-data

x-www-form-urlencoded

raw

binary

GraphQL

JSON

1

{

2

"cpf": "10020030080",

3

"nome": "Teste",

4

"dataNascimento": "41/01/1970",

5

"email": "teste@email.com.br"

6

}

Body

Cookies

Headers (4)

Test Results

Pretty

Raw

Preview

Visualize

JSON

1

{

2

"timestamp": "2021-01-08T19:16:17.934+00:00",

3

"status": 400,

4

"error": "Bad Request",

5

"message": "Invalid value for DayOfMonth (valid values 1 - 28/31): 41",

6

"path": "http://localhost:8080/usuarios"

7

}

- Utilizando formato de data fora do padrão dd/MM/yyyy:

POST http://localhost:8080/usuarios

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL **JSON** ▼

```
1 {
2   "cpf": "10020030080",
3   "nome": "Teste",
4   "dataNascimento": "01/1970",
5   "email": "teste@email.com.br"
6 }
```

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON ▼

```
1 {
2   "timestamp": "2021-01-08T19:17:09.556+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "message": "Text '01/1970' could not be parsed at index 5",
6   "path": "http://localhost:8080/usuarios"
7 }
```

- Utilizando CPF, Nome e E-mail fora do padrão:

POST http://localhost:8080/usuarios

Params Authorization Headers (8) **Body** Pre-request Script Tests Settings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQL **JSON** ▼

```
1 {
2   "cpf": "10020030080",
3   "nome": "",
4   "dataNascimento": "01/01/1970",
5   "email": "testeemail.com.br"
6 }
```

Body Cookies Headers (4) Test Results

Pretty Raw Preview Visualize JSON ▼

```
1 {
2   "timestamp": "2021-01-08T19:17:55.058+00:00",
3   "status": 400,
4   "error": "Bad Request",
5   "message": [
6     "Nome é obrigatório",
7     "CPF Inválido",
8     "E-mail inválido"
9   ],
10  "path": "http://localhost:8080/usuarios"
11 }
```

- Inserindo os dados corretamente o mesmo retorna a entidade criada:

The screenshot displays the Postman interface for a POST request. The URL bar shows 'http://localhost:8080/usuarios'. The 'Body' tab is selected, showing a JSON payload:

```
{  "cpf": "10020030088",  "nome": "Teste",  "dataNascimento": "01/01/1970",  "email": "teste@email.com.br"}
```

. The 'Headers' tab shows 5 headers. The 'Test Results' tab is also visible. The response body is shown in the 'Pretty' view, displaying the same JSON object:

```
{  "cpf": "10020030088",  "nome": "Teste",  "email": "teste@email.com.br",  "dataNascimento": "01/01/1970"}
```

Adicionamos mais dois usuários gerados aleatoriamente para testes.

Teste HTTP GET

- Listar todos os usuários:

GET

▼

http://localhost:8080/usuarios

ParamsAuthorizationHeaders (8)Body ●Pre-request ScriptTestsSettings

BodyCookiesHeaders (5)Test Results

PrettyRawPreviewVisualizeJSON ▼

1

[

2

{

3

"cpf": "10020030088",

4

"nome": "Teste",

5

"email": "teste@email.com.br",

6

"dataNascimento": "01/01/1970"

7

},

8

{

9

"cpf": "30941165663",

10

"nome": "Allana Isabella das Neves",

11

"email": "allanaisabelladasneves@email.com.br",

12

"dataNascimento": "08/05/1952"

13

},

14

{

15

"cpf": "71307541682",

16

"nome": "Oliver Anthony Nunes",

17

"email": "oliveranthonymunes@email.com.br",

18

"dataNascimento": "23/06/1950"

19

}

20

]

- Buscar usuário específico:

GET

▼

http://localhost:8080/usuarios/10020030088

ParamsAuthorizationHeaders (8)Body ●Pre-request ScriptTestsSettings

BodyCookiesHeaders (5)Test Results

PrettyRawPreviewVisualizeJSON ▼

1

{

2

"cpf": "10020030088",

3

"nome": "Teste",

4

"email": "teste@email.com.br",

5

"dataNascimento": "01/01/1970"

6

}

- Buscar usuário inexistente:

GET

http://localhost:8080/usuarios/111111111111

ParamsAuthorizationHeaders (8)Body ●Pre-request ScriptTestsSettings

BodyCookiesHeaders (4)Test Results

PrettyRawPreviewVisualizeJSON

```
1 {
2   "timestamp": "2021-01-08T19:20:50.604+00:00",
3   "status": 500,
4   "error": "Internal Server Error",
5   "message": "Usuário não localizado",
6   "path": "/usuarios/111111111111"
7 }
```

Teste HTTP PUT

- Alterar usuário corretamente:

PUT

http://localhost:8080/usuarios/10020030088

ParamsAuthorizationHeaders (8)Body ●Pre-request ScriptTestsSettings

● none ● form-data ● x-www-form-urlencoded ● raw ● binary ● GraphQLJSON

1 {
2 "cpf": "10020030088",
3 "nome": "Teste da Silva",
4 "dataNascimento": "01/01/2000",
5 "email": "testesilva@email.com.br"
6 }

BodyCookiesHeaders (5)Test Results

PrettyRawPreviewVisualizeJSON

```
1 {
2   "cpf": "10020030088",
3   "nome": "Teste da Silva",
4   "email": "testesilva@email.com.br",
5   "dataNascimento": "01/01/2000"
6 }
```

- Utilizando CPF, Nome e E-mail fora do padrão, mesmas validações do POST:

The screenshot shows a REST client interface. At the top, a PUT request is configured for the URL `http://localhost:8080/usuarios/10020030088`. The 'Body' tab is selected, showing a JSON payload:

```
{  "cpf": "00000000000",  "nome": "",  "dataNascimento": "01/01/2000",  "email": "testesilva"}
```

. Below the request, the 'Test Results' tab is active, displaying the response in JSON format:

```
{  "timestamp": "2021-01-08T19:22:39.733+00:00",  "status": 400,  "error": "Bad Request",  "message": [    "CPF Inválido",    "Nome é obrigatório",    "E-mail inválido"  ],  "path": "http://localhost:8080/usuarios/10020030088"}
```

Teste HTTP DELETE

- Utilizando CPF incorreto ou não existente:

The screenshot shows a REST client interface. At the top, a DELETE request is configured for the URL `http://localhost:8080/usuarios/11111111111`. The 'Body' tab is selected, showing an empty JSON object:

```
{}
```

. Below the request, the 'Test Results' tab is active, displaying the response in JSON format:

```
{  "timestamp": "2021-01-08T19:23:20.328+00:00",  "status": 500,  "error": "Internal Server Error",  "message": "Usuário não localizado",  "path": "/usuarios/11111111111"}
```

- Utilizando CPF existente no Banco de Dados, retorna HTTP Status 200 (Ok):

DELETE http://localhost:8080/usuarios/10020030088

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

Body Cookies Headers (4) Test Results Status: 200 OK

Pretty Raw Preview Visualize Text

1

- Após utilizar o DELETE, verificamos que o usuário realmente foi apagado do Banco de Dados:

GET http://localhost:8080/usuarios

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 [
2   {
3     "cpf": "30941165663",
4     "nome": "Allana Isabella das Neves",
5     "email": "allanaisabelladasneves@email.com.br",
6     "dataNascimento": "08/05/1952"
7   },
8   {
9     "cpf": "71307541682",
10    "nome": "Oliver Anthony Nunes",
11    "email": "oliveranthonymunes@email.com.br",
12    "dataNascimento": "23/06/1950"
13  }
14 ]
```

- Por último, ao efetuar qualquer requisição com método diferente do mapeado pelo endpoint, será retornado a seguinte mensagem de erro:

DELETE http://localhost:8080/usuarios

Params Authorization Headers (8) Body ● Pre-request Script Tests Settings

Body Cookies Headers (5) Test Results

Pretty Raw Preview Visualize JSON

```
1 {
2   "timestamp": "2021-01-09T01:10:05.610+00:00",
3   "status": 405,
4   "error": "Method Not Allowed",
5   "message": "Request method 'DELETE' not supported",
6   "path": "http://localhost:8080/usuarios"
7 }
```