API REST PARA GERENCIAMENTO DE QUADRINHOS (COMICS) DE USUÁRIOS

Nome: José Denes Lima Araújo
Email: josedenes17@gmail.com
github: https://github.com/josedenes/desafio-marvel

Vamos implementar uma API REST para o gerenciamento de comics de usuários. Inicialmente deve-se construir um cadastro de usuário com os seguintes dados: Nome, Email, CPF e Data de nascimento. Como restrição todos os dados são obrigatórios, e além disso o CPF e email devem ser válidos e únicos.

A segunda etapa é a construção de um cadastro de comics para um determinado usuário. Deve-se receber um id de um quadrinho no endpoint da nossa API, o Id do usuário ao qual ele será associado e consumir a API da MARVEL (disponível em https://developer.marvel.com/) para buscar os dados referentes ao quadrinho identificado por esse Id. Para um comic os dados de comicID, Título, Preço, Autores, ISBN e Descrição devem ser cadastrados. Todos os dados são obrigatórios e o ISBN deve ser único.

A terceira etapa é a listagem de usuários cadastrados na aplicação. A lista de comics de cada usuário não deve ser apresentada.

A quarta etapa é a listagem de comics de um determinado usuário. Para essa listagem devem seguir as seguintes regras de negócio:

- Cada quadrinho possui um desconto de 10% que pode ser aplicado ao seu preço de acordo com o dia da semana em que a consulta está sendo realizada.
- O dia da semana que corresponde a aplicação do desconto varia de acordo com o dígito final do ISBN, conforme abaixo:
    - Final 0-1: segunda-feira
    - Final 2-3: terça-feira
    - Final 4-5: quarta-feira
    - Final 6-7: quinta-feira
    - Final 8-9: sexta-feira
- Deverá haver um atributo em cada quadrinho indicando se um desconto está ou não está sendo aplicado ao seu preço.
- Caso o quadrinho possua desconto, o mesmo deverá ser aplicado ao seu valor quando retornar os dados do quadrinho.

Por fim, as respostas retornadas pelos endpoints da API construída devem seguir os requisitos:

- Caso os cadastros estejam corretos, é necessário retornar o HTTP Status Code 201. Caso haja erros de preenchimento de dados, o

HTTP Status Code retornado deve ser 400 e deve ser apresentada uma mensagem indicando qual campo foi preenchido incorretamente.
- Caso as buscas estejam corretas, é necessário retornar o HTTP Status Code 200. Caso haja erro na busca, retornar o status adequado (atenção a erros do cliente ou do servidor) e uma mensagem de erro amigável.

Desafios extras implementados:
- Realizar a integração a API da MARVEL usando Spring-Cloud-Feign;
- Construir um Exception Handler para o tratamento das exceções que podem ocorrer na execução da aplicação;

O código da API encontra-se no github:
https://github.com/josedenes/desafio-marvel

1 - Linguagem e Tecnologias Utilizadas

Para o desenvolvimento da API REST foram utilizadas as seguintes tecnologias:

- **Linguagem java 11:** Linguagem de programação orientada a objetos. É uma linguagem multiplataforma e também possui uma comunidade de usuários grande e além disso é gratuita.
- **Spring web:** Framework que facilita o desenvolvimento de aplicações web. Possui as funcionalidades que precisamos que são: atender as requisições HTTP, delegar responsabilidades de processamento de dados para outros componentes e preparar a resposta que precisa ser dada.
- **Spring data JPA:** É um framework para mapeamento objeto-relacional e persistência de dados. Esse framework facilita a criação e manipulação dos repositórios pois nos libera de ter que implementar as interaces referentes aos nossos repositórios (ou DAOs). Assim, agilizando o processo de desenvolvimento devido as funcionalidades comuns (buscar, salvar, atualizar e remover) em aplicações já implementadas, prontas para serem utilizadas.
- **Spring-boot-starter-validation:** Implementa java bean que é o padrão de para implementar a lógica de validação no ecossistema Java. Funciona definindo através de anotações que definem restrições aos campos de uma classe. Facilita o desenvolvimento das aplicações pois validações como campo não nulo, email e CPF válidos, entre outros estão implementadas, sendo acessíveis através das anotações.

- **Spring-Cloud-Feign:** O Spring-Cloud-Feign é um projeto que faz parte das soluções do Spring Cloud e ele basicamente é utilizado para integração com serviços Rest. O Feign é uma boa alternativa para implementar clients HTTPs em Java de forma fácil e prática, sendo necessário escrever apenas uma definição de interface para chamar o serviço.

2 - Modelo de Desenvolvimento

Para o desenvolvimento da aplicação inicialmente vamos entender os conceitos de front end e back end. A Figura 1 apresenta essa divisão. O usuário ao acessar a aplicação pelo navegador ou dispositivo móvel visualiza o front end, ou seja, a aplicação que está rodando no lado cliente. Já o backend é a parte do sistema que está rodando no servidor. E assim, para o funcionamento do sistema existe a troca de dados entre o front end e o back end. Essa troca de dados é feita comumente através de requisições HTTP.

O padrão MVC separa a comunicação direta entre as camadas de visualização com as camadas de modelo ou negócio. Essas camadas são isoladas e se comunicam por uma camada de controle intermediária. Assim, existe um desacoplamento que melhora a qualidade e manutenção da aplicação. O nome MVC representa as camadas Model, View e Controller. O Model representa os dados e não deve incluir detalhes de implementação. Já a camada View representa um componente de interface de usuário como as páginas HTML. E por último, a camada Controller, que é responsável por trocar e interpretar mensagens entre a View e o Model. A Figura 1 apresenta o front end (View) e o back end (model e controller) para uma aplicação implementada no spring.

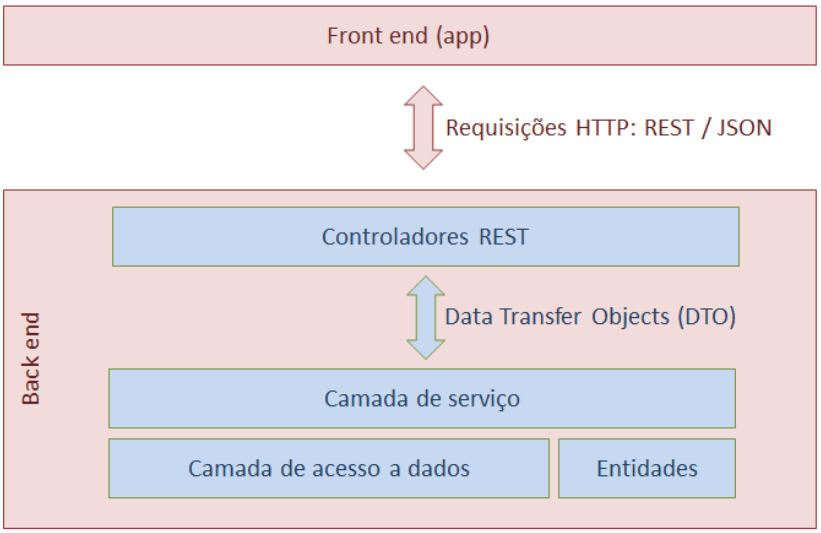


Figura 1 - Modelo padrão camadas

O backend é dividido em camadas lógicas. Na Figura 1 o backend é organizado nas camadas:

- Controladores Rest: Responsável por receber as requisições da aplicação cliente (interações que o usuário faz) e encaminhar as chamadas para os serviços responsáveis por realizar aquela ação do usuário;
- Camada de serviço: Possui a lógica de negócio do sistema. Nessa camada são feitas as verificações, cálculos e operações;
- Camada de acesso a dados: Possui as operações básicas de acessar os dados no banco como salvar, atualizar, deletar e buscar;
- DTO: Objetos para carregar os dados entre o controlador Rest e o serviço;
- Entidade: Objeto que é monitorado para manter a integridade dos dados sendo utilizado entre as camadas de serviço e de acesso a dados.

Essa organização é importante para deixar o sistema flexível e de fácil manutenção.

3 - Diagrama de Classes

Ao analisar as especificações para o desenvolvimento da api apresentadas no inicio do blog é possível definir os dados das classes Usuário e Comic que devem ser persistidos no banco de dados. A Figura 2 apresenta o diagrama de classes utilizado como base para a implementação da API.

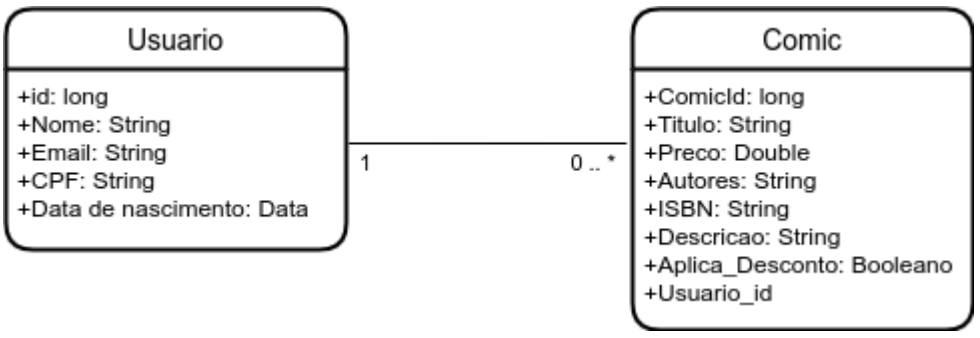


Figura 2 - Diagrama de classes

O relacionamento entre Usuário e Comic é de 1 para n. Uma vez que, um usuário pode possuir várias comics e uma comic só pode pertencer a um usuário, pois a regra especificada no início do blog só permite o cadastro de um isbn uma única vez.

4 - Implementando o Acesso ao Banco

Para o desenvolvimento da API foi utilizado H2 como SGBD. No entanto, o JPA abstrai toda a conexão com o banco de dados. Sendo necessário apenas configurar alguns parâmetros básicos. Assim, toda a camada de acesso a dados e as consultas são gerados automaticamente pela ferramenta ORM para o SGBD H2. O que deixa a aplicação menos dependente do SGBD. Para utilizar outro SGBD é necessário apenas a mudança da string de conexão.

Para conectar nossa aplicação ao H2 é necessário configurar o arquivo src/pom.xml para instalar as dependências necessárias. A Figura 3 apresenta todas as dependências do projeto (JPA, spring web, spring validation, spring cloud openfeign) e entre elas a h2 database que é a dependência do banco de dados de teste que será utilizado para o desenvolvimento da aplicação.

```
20    <dependencies>
21        <dependency>
22            <groupId>org.springframework.boot</groupId>
23            <artifactId>spring-boot-starter-web</artifactId>
24        </dependency>
25
26        <dependency>
27            <groupId>org.springframework.boot</groupId>
28            <artifactId>spring-boot-starter-data-jpa</artifactId>
29        </dependency>
30
31        <dependency>
32            <groupId>com.h2database</groupId>
33            <artifactId>h2</artifactId>
34            <scope>runtime</scope>
35        </dependency>
36
37        <dependency>
38            <groupId>org.postgresql</groupId>
39            <artifactId>postgresql</artifactId>
40            <scope>runtime</scope>
41        </dependency>
42
43        <dependency>
44            <groupId>org.springframework.boot</groupId>
45            <artifactId>spring-boot-starter-validation</artifactId>
46        </dependency>
47        <dependency>
48            <groupId>org.springframework.boot</groupId>
49            <artifactId>spring-boot-starter-test</artifactId>
50            <scope>test</scope>
51        </dependency>
52        <dependency>
53            <groupId>org.springframework.cloud</groupId>
54            <artifactId>spring-cloud-starter-openfeign</artifactId>
55            <version>3.0.0</version>
56        </dependency>
```

Figura 3 - Pom.xml

Após a inclusão da dependência do h2 é necessário criar o arquivo application-test-properties na pasta src/main/resources. A Figura 4 apresenta as configurações que devem estar nesse arquivo para o funcionamento do banco H2.

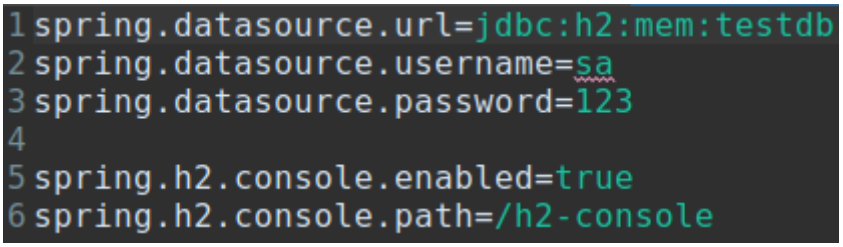


Figura 4 - src/main/resources/application-test-properties

4.1 - Criando Entities

Após as configurações de dependências e conexões, iremos iniciar o processo de mapeamento objeto-relacional. Como possuímos duas tabelas (usuário e comic) iremos mapeá-las para objetos java, para que o JPA saiba como transformar linhas tabela no banco de dados em objetos java e vice-versa.

Cada classe é transformada em uma entidade quando é usada a anotação **@Entity** da JPA antes do nome da classe. A anotação **@Table** é usada para definir o nome da tabela a ser gerada no banco de dados. Com essas anotações (**@Entity** e **@Table**) cada atributo das classes representa os campos das tabelas no banco de dados.

Agora vamos definir as nossas entidades. Em nossa aplicação iremos criar duas classes: Usuário e Comic. O Usuário tem id, nome, email, CPF e data de nascimento (Figura 5). Uma Comic possui comicId, título, preço, autores, isbn, descrição e um boolean que determina se um desconto é aplicado ou não (Figura 6). Além dos atributos, são criados seus respectivos métodos getters e setters e também são criados os construtores das classes.

Os outros atributos das classes podem receber anotações da JPA que devem vir antes do nome do atributo e podem ser mais de uma para um mesmo atributo. A anotação **@Id** determina a chave primária da entidade e **@GeneratedValue**: faz com que o valor do id seja gerado e controlado automaticamente de acordo com o banco de dados. A anotação **@Column** é utilizada para especificar os detalhes da coluna que um campo será mapeado e possui alguns parâmetros. **@Column**(unique = true) com esse parâmetro é utilizado para especificar colunas a serem marcadas como possuindo valores únicos. **@Column**(columnDefinition = "TEXT") com esse parâmetro é utilizado para setar um atributo como do tipo TEXT. A anotação **@DateTimeFormat**(pattern = "dd/MM/yyyy") formata o padrão de entrada para a data.

```java
@Entity
@Table(name = "tb_usuario")
public class Usuario implements Serializable{
    private static final long serialVersionUID = 1L;

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

    @Column(unique = true)
    private String email;

    @Column(unique = true)
    private String cpf;

    @DateTimeFormat(pattern = "dd/MM/yyyy")
    private Date dataNascimento;

    @OneToMany(mappedBy = "usuario", cascade = CascadeType.ALL)
    private List <Comic> comics;

    public Usuario() {

    }

    public Usuario(Long id, String nome, String email, String cpf, Date dataNascimento) {
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }
```

Figura 5 - Usuario.java

Finalmente, note que uma relação foi criada, usando as anotações **@OneToMany** na entidade Usuário e **@ManyToOne** na entidade Comic. A anotação **@OneToMany** permite que uma linha em uma tabela seja mapeada para várias linhas em outra tabela. A anotação **@ManyToOne** está associada à variável da classe Usuário. A anotação **@JoinColumn** faz referência à coluna mapeada.

```java
@Entity
@Table(name = "tb_comic")
public class Comic implements Serializable {
    private static final long serialVersionUID = 1L;

    @Id
    private Long comicId;
    private String titulo;
    private Double preco;

    @Column(columnDefinition = "TEXT")
    private String autores;

    @Column(unique = true)
    private String isbn;

    @Column(columnDefinition = "TEXT")
    private String descricao;

    private Boolean aplicaDesconto;

    @ManyToOne
    @JoinColumn(name="usuario_id")
    private Usuario usuario;

    public Comic() {

    }

    public Comic(Long comicId, String titulo, Double preco, String autores, String isbn, String descricao,
            Boolean aplicaDesconto, Usuario usuario) {
        this.comicId = comicId;
        this.titulo = titulo;
        this.preco = preco;
        this.autores = autores;
        this.isbn = isbn;
        this.descricao = descricao;
        this.aplicaDesconto = aplicaDesconto;
        this.usuario = usuario;
    }
```

Figura 6 - Comic.java

4.2 - Criando Repositories

Após a definição das entidades é necessário criar a camada de acesso aos dados. Para isso são criados os *repositories* que são as interfaces responsáveis por fazer as transações com o banco de dados, como as operações recuperar, inserir, alterar ou deletar. As Figuras 7 e 8 apresentam os repositórios de usuário e comic. Para definir uma interface como repository é necessário inserir a annotation **@Repository** antes do nome da interface.

```java
@Repository
public interface UsuarioRepository extends JpaRepository<Usuario, Long>{

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

Figura 7 - UsuarioRepository.java

```java
@Repository
public interface ComicRepository extends JpaRepository<Comic, Long>{

    List<Comic> findByUsuarioId(Long usuarioId);
}
```

Figura 8 - ComicRepository.java

É necessário também estender a interface *JpaRepository* onde deve ser informada a entidade que ela vai representar e tipo do id. Com isso temos a implementação das funcionalidades de CRUD em um banco de dados para cada entidade. E seguindo o padrão de nomenclatura podemos chamar métodos de busca também já implementados pela JPA, como os apresentados nas Figuras 7 e 8.

5 - Implementando as Regras de Negócio (Camada de Serviço)

A camada de serviço é responsável por se comunicar com a camada de acesso a dados. Nessa camada estão implementadas as regras de negócio e a comunicação com os repositórios e com o controlador. O tráfego de dados entre a camada de serviço e o controlador é feito usando Data Transfer Object (DTO).

5.1 - Criando DTOs

As entidades criadas anteriormente são utilizadas na camada de acesso a dados. Na camada de serviços com os DTOs podemos controlar quais dados serão transferidos para o controlador e assim o que a api vai disponibilizar para as aplicações. Essas classes implementam a interface Serializable, identificando que são objetos serializáveis, mostrando assim para o spring como os dados são transformados em JSON, XML ou qualquer que seja o formato que a API irá se comunicar, no nosso caso iremos usar JSON. As Figuras 9 e 10 apresentam os DTOs de Usuario e Comic.

```java

public class UsuarioDTO implements Serializable{
    private static final long serialVersionUID = 1L;

    private Long id;

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

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

    @NotBlank(message = "Campo cpf obrigatório")
    @CPF(message = "Por favor entrar com cpf válido")
    private String cpf;

    @DateTimeFormat(pattern = "dd/MM/yyyy")
    @NotNull(message = "Campo data de nascimento obrigatório")
    private Date dataNascimento;


    public UsuarioDTO() {

    }

    public UsuarioDTO(Long id, String nome, String email, String cpf, Date dataNascimento) {
        this.id = id;
        this.nome = nome;
        this.email = email;
        this.cpf = cpf;
        this.dataNascimento = dataNascimento;
    }

    public UsuarioDTO(Usuario entity) {
        this.id = entity.getId();
        this.nome = entity.getNome();
        this.email = entity.getEmail();
        this.cpf = entity.getCpf();
        this.dataNascimento = entity.getDataNascimento();
    }
```

Figura 9 - UsuarioDTO.java

```java

public class ComicDTO implements Serializable{
    private static final long serialVersionUID = 1L;

    @NotBlank(message = "Campo comicId obrigatório")
    private Long comicId;

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

    @NotBlank(message = "Campo preço obrigatório")
    private Double preco;

    private String autores;

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

    private String descricao;

    private Boolean aplicaDesconto;

    public ComicDTO() {

    }

    public ComicDTO(Long comicId, String titulo, Double preco, String autores, String isbn, String descricao,
            Boolean aplicaDesconto) {
        this.comicId = comicId;
        this.titulo = titulo;
        this.preco = preco;
        this.autores = autores;
        this.isbn = isbn;
        this.descricao = descricao;
        this.aplicaDesconto = aplicaDesconto;
    }

    public ComicDTO(Comic entity) {
        this.comicId = entity.getComicId();
        this.titulo = entity.getTitulo();
        this.preco = entity.getPreco();
        this.autores = entity.getAutores();
        this.isbn = entity.getIsbn();
        this.descricao = entity.getDescricao();
        this.aplicaDesconto = entity.getAplicaDesconto();
    }
```

Figura 10 - ComicDTO.java

Os DTOs apresentados possuem os mesmos atributos das entidades Usuario e Comic. São criados dois construtores, um que recebe os atributos e outro que recebe uma entidade. O construtor que recebe a entidade é importante para conversão entre as entidades e os DTOs que deve ocorrer na camada de serviço. Pois essa camada disponibiliza DTOs para o controlador e entidades para a camada de acesso a dados.

Nos DTOs estão presentes anotações de validação do Bean validation necessárias para validação de alguns atributos de acordo com as especificações da API. A anotação **@NotBlank** impede do campo ter um valor nulo ou vazio. **@Email** verifica se um email é válido e **@CPF** verifica se um CPF válido.

5.2 - Criando Services

Com os DTOs implementados é possível criar a camada de serviço. Inicialmente vamos implementar o serviço referente ao usuário. A Figura 11 apresenta o serviço do usuário. A classe UsuarioService possui a anotação **@Service** que registra a classe como um componente que participa do sistema de injeção de dependência automatizado do Spring, indicando que a classe pertence a camada de serviço e o spring é responsável por gerenciar suas instâncias. A anotação **@Autowired** injeta automaticamente a dependência com os repositórios de usuário e de comic, que ficam responsáveis pelas operações CRUD com o banco de dados.

```
 3 import java.time.DayOfWeek;
28
29 @Service
30 public class UsuarioService {
31
32     @Autowired
33     private UsuarioRepository repository;
34
35     @Autowired
36     private ComicRepository comicRepository;
```

Figura 11 - Construção do Serviço na classe UsuarioService.java

Na classe UsuarioService estão implementados os métodos para a classe usuário que são: *insert*, *findAll*, *findById*, delete, *update* e *findComicsPorIdUsuario*. O método *insert* (Figura 12) permite cadastrar um usuário, para isso recebe um dto repassado pela camada do controlador e então cria uma entidade do tipo usuário. Essa entidade é repassada para o repositório que a salva no banco de dados usando o método *save* já existente na JPA. Esse método *insert* é precedido pela anotação **@Transactional** que garante a integridade da transação com o banco de dados.

```java
    @Transactional
    public UsuarioDTO insert(UsuarioInserirDTO dto) {
        Usuario entity = new Usuario();
        entity.setNome(dto.getNome());
        entity.setEmail(dto.getEmail());
        entity.setCpf(dto.getCpf());
        entity.setDataNascimento(dto.getDataNascimento());
        entity = repository.save(entity);
        return new UsuarioDTO(entity);
    }
```

Figura 12 - Método insert na classe UsuarioService.java

Os métodos *findAll* e *findById,* apresentados na Figura 13, são responsáveis respectivamente pela listagem de todos os usuários cadastrados e pela recuperação de um usuário pelo seu id. Ao serem chamados pelo controlador de usuário os dois métodos utilizam funções já implementadas pelo repository e retornam respectivamente uma lista de DTO de usuário e um DTO de usuário que possui o id recebido por parâmetro. Os dois métodos possuem a anotação **@Transactional** com o parâmetro de somente leitura ativado, indicando que não há o travamento do banco de dados, pois as operações são apenas de leitura e não modificam os dados.

```java
    @Transactional(readOnly = true)
    public List<UsuarioDTO> findAll(){
        List<Usuario> list = repository.findAll();
        return list.stream().map(x -> new UsuarioDTO(x)).collect(Collectors.toList());
    }

    @Transactional(readOnly = true)
    public UsuarioDTO findById(Long id) {
        Optional<Usuario> obj = repository.findById(id);
        Usuario entity = obj.orElseThrow(() -> new ResourceNotFoundException("Usuario nao encontrada"));
        return new UsuarioDTO(entity);
    }
```

Figura 13 - Métodos *findAll* e *findById* na classe UsuarioService.java

O método *delete* apresentado na Figura 14 tem a função de deletar um usuário cadastrado. Para isso é necessário o id do usuário que é recebido por parametro. Esse id é repassado para a função *deleteById* do repositório JPA que realiza a deleção do usuário que possui esse id.

```java
    public void delete(Long id) {
        try {
            repository.deleteById(id);
        }
        catch(EmptyResultDataAccessException e) {
            throw new ResourceNotFoundException("Id nao encontrado " + id);
        }
        catch(DataIntegrityViolationException e) {
            throw new DatabaseException("Violacao de integridade");
        }
    }
```

Figura 14 - Método *delete* na classe UsuarioService.java

O método *update* atualiza os dados de um determinado usuário. A Figura 15 apresenta o método update que recebe como parâmetros o id do usuário e os novos dados em um objeto DTO. Com o método *getOne* do repository a entidade usuário a ser atualizada é encontrada e em seguida os novos dados são atribuídos e ao final a entidade é salva.

```
159●    @Transactional
160     public UsuarioDTO update(Long id, UsuarioAtualizarDTO dto) {
161         try {
162             Usuario entity = repository.getOne(id);
163             entity.setNome(dto.getNome());
164             entity.setEmail(dto.getEmail());
165             entity.setCpf(dto.getCpf());
166             entity.setDataNascimento(dto.getDataNascimento());
167             entity = repository.save(entity);
168             return new UsuarioDTO(entity);
169         }
170         catch(EntityNotFoundException e) {
171             throw new ResourceNotFoundException("Id nao encontrado " + id);
172         }
173     }
```

Figura 15 - Método *update* na classe UsuarioService.java

O método *findComicsPorIdUsuario* (Figura 16) recebe como parâmetro um id de um usuário. Primeiramente o método *findById* do *repository* de usuário encontra o usuário pelo id repassado. Após verifica a existência do usuário é usado o método *findByUsuarioId* do *comicRepository* para retornar a lista de objetos comic do usuário. Essa lista é percorrida para ser transformada em DTO e aplicar a regra de desconto de 10% se o dia em que a consulta for realizada coincidir com o digito final do atributo isbn. Após isso, a lista de Comic DTOs é retornada para o controlador de Comic.

```
70●    @Transactional(readOnly = true)
71     public List<ComicDTO> findComicsPorIdUsuario(Long id){
72         Optional<Usuario> obj = repository.findById(id);
73         Usuario entity = obj.orElseThrow(() -> new ResourceNotFoundException("Usuario nao encontrada"));
74
75         List<Comic> list = comicRepository.findByUsuarioId(id);
76         List<ComicDTO> listComicDTO = new ArrayList<>();
77
78         for (Comic comic: list) {
79             comic.setAplicaDesconto(descontoAtivado(comic.getIsbn()));;
80             if(comic.getAplicaDesconto()) {
81                 comic.setPreco(comic.getPreco()*0.9);
82             }
83             listComicDTO.add(new ComicDTO(comic.getComicId(), comic.getTitulo(), comic.getPreco(), comic.getAutores(),
84                     comic.getIsbn(), comic.getDescricao(), comic.getAplicaDesconto()));
85         }
86         return listComicDTO;
87     }
```

Figura 16 - Método *findComicsPorIdUsuario* na classe UsuarioService.java

A verificação do desconto de 10% é realizado pelo método descontoAtivado, apresentado na Figura 17. Esse método recebe um isbn e compara o ultimo digito desse isbn com o dia da consulta, retornando verdadeiro o falso dependendo das regras explicadas no início desse blog post.

```java
    public Boolean descontoAtivado(String isbn){
        DayOfWeek data = LocalDate.now().getDayOfWeek();
        int len = isbn.length();
        char lastChar = isbn.charAt(len - 1);
        String lastCharString = ""+lastChar;

        int digitoFinalIsbn = Integer.parseInt(lastCharString);
        int diaDaSemanaIsbn;

        if (digitoFinalIsbn == 0 || digitoFinalIsbn == 1) {
            diaDaSemanaIsbn = 1;
        } else if (digitoFinalIsbn == 2 || digitoFinalIsbn == 3) {
            diaDaSemanaIsbn = 2;
        } else if (digitoFinalIsbn == 4 || digitoFinalIsbn == 5) {
            diaDaSemanaIsbn = 3;
        } else if (digitoFinalIsbn == 6 || digitoFinalIsbn == 7) {
            diaDaSemanaIsbn = 4;
        } else {
            diaDaSemanaIsbn = 5;
        }

        int diaDaRequisicao = data.getValue();
        if(diaDaRequisicao == diaDaSemanaIsbn) {
            return true;
        }else {
            return false;
        }
    }
```

Figura 17 - Método descontoAtivado na classe UsuarioService.java

A camada de serviço referente a Comic implementado na classe ComicService.java possui as anotações **@Service** e **@Autowired** para respectivamente transformar a classe em serviço e injetar dependência com os repositórios. A classe ComicService possui apenas o método *insert* apresentado na Figura 18. Esse método recebe comicId que é o id a ser requisitado a API da Marvel e o id do Usuário a qual esse comic será cadastrado. Primeiramente o usuário é encontrado usando a função *findById* do *usuarioRepository*. Após isso, o método *buscaComicPorId*, recebe um comicId para buscar um comic na API da Marvel. A comic retornanda é cadastrada relacionada ao usuário correspondente. O método *buscaComicPorId* pertence a classe SolicitarComic que consome a API da Marvel usando o Spring-Cloud-Feign e será mostrado na sessão 8.

```java
    @Transactional
    public ComicDTO insert(Long comicId, Long usuarioId) {
        Optional<Usuario> obj = usuarioRepository.findById(usuarioId);
        Usuario entityUsuario = obj.orElseThrow(() -> new ResourceNotFoundException("Usuario nao encontrada"));

        ResultsResponse resposta;

        try {
            resposta = solicitarComic.buscaComicPorId(comicId);
        }
        catch(FeignException e) {
            throw new ResourceNotFoundException("Id da comic nao encontrado: " + comicId);
        }

        Comic entity = new Comic();
        entity.setComicId(resposta.getComicId());
        entity.setTitulo(resposta.getTitle());
        entity.setPreco(resposta.getPrice());
        entity.setAutores(resposta.getCreators());
        entity.setIsbn(resposta.getIsbn());
        entity.setDescricao(resposta.getDescription());
        entity.setAplicaDesconto(false);
        entity.setUsuario(entityUsuario);
        entity = repository.save(entity);
        return new ComicDTO(entity);
    }
```

Figura 18 - Método *insert* na classe ComicService.java

6 - Criando a API REST

Após a criação das entidades, repositórios, DTOs e serviços é necessário implementar os controladores REST. As classes REST são responsáveis por receber as requisições, executar uma ação (repassar para a camada de serviço) e devolver a resposta para o cliente.

6.1 - Usuário

A Figura 19 a seguir apresenta o controlador responsável pelo usuário. Para configurar a classe para ser um controlador REST para responder requisições devemos colocar uma anotação chamada **@RestController** antes do nome da classe. O spring já possui toda a estrutura necessária para transformar uma classe em um recurso REST. Essa anotação é uma forma de configurar e usar essa estrutura que já está implementada.

A anotação **@RequestMapping** define qual a rota REST do recurso. Ou seja, temos um controlador que irá responder com a URL **/usuarios**. Assim, podemos criar nossos *endpoints* que são as rotas possíveis que respondem as solicitações. Antes de criar nossos *endpoints* é necessário criar uma dependência com a camada de serviço. Isso é feito usando a variável *service* do tipo UsuarioService precedida pela anotação **@Autowired** que injeta automaticamente a dependência através do Spring.

```
27 @RestController
28 @RequestMapping(value = "/usuarios")
29 public class UsuarioResource {
30
31     @Autowired
32     private UsuarioService service;
33
```

Figura 19 - Construção do Controlador na classe UsuarioResource.java

O primeiro *endpoint* da nossa API é o cadastro de usuário apresentado na Figura 20 abaixo. O método *insert* deve ter a anotação **@PostMapping** que indica por padrão em uma API REST que o método é um endpoint da nossa API e que será inserido um novo recurso. O método insert tem um retorno do tipo *ResponseEntity<UsuarioDTO>* e recebe como parâmetro os dados a serem inseridos na forma de um objeto. A anotação **@RequestBody** mapeia o corpo HttpRequest para um objeto de domínio, permitindo a desserialização automática do corpo HttpRequest de entrada em um objeto Java, que no caso é do tipo UsuarioDTO. Já a anotação **@Valid** verifica se os atributos estão de acordo com anotações *Bean validation* definidas nas classes DTOs. Após essa validação o objeto é então repassado para a camada de serviço pela chamada ao método service.insert(dto).

```java
    @PostMapping
    public ResponseEntity<UsuarioDTO> insert(@Valid @RequestBody UsuarioInserirDTO dto){
        UsuarioDTO newDto = service.insert(dto);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(newDto.getId()).toUri();
        return ResponseEntity.created(uri).body(newDto);
    }
```

Figura 20 - Método *insert* do UsuarioResource.java

Para testar se nosso endpoint de cadastro de usuário está funcionando será utilizado o Postman. A Figura 21 apresenta a requisição feita no Postman. Para o teste foi utilizado o método POST com uma requisição para a URL **http://localhost:8080/usuarios** enviando um JSON com os dados do usuário a serem inseridos. Na parte inferior da figura é possível observar que o cadastro retornou **HTTP Status Code 201**, indicando que o usuário foi cadastrado. Também foi retornado o objeto cadastrado com o id atribuído a esse usuário.

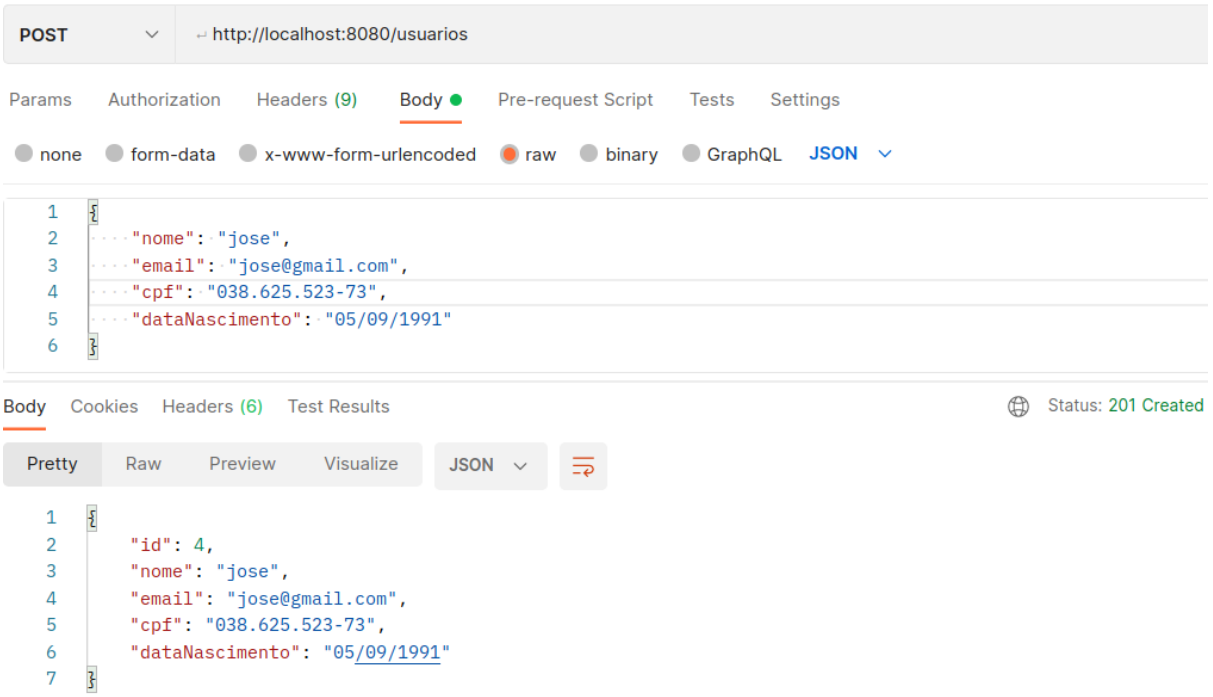


Figura 21 - Requisição de Cadastro de usuário no Postman

O segundo *endpoint* da nossa API é o listagem de usuários cadastrados, apresentado na Figura 22 abaixo, representado pelo método *findAll*. Esse método possui a anotação **@GetMapping** que indica em uma API REST que o método é um endpoint e que mapeia solicitações HTTP GET. O método *findAll* não recebe parâmetros e na sua execução chama o método service.*findAll* da camada de serviço de usuário, e ao final retorna uma lista de UsuarioDTO.

```java
    @GetMapping
    public ResponseEntity<List<UsuarioDTO>> findAll(){
        List<UsuarioDTO> list = service.findAll();
        return ResponseEntity.ok().body(list);
    }
```

Figura 22 - Método *findAll* do UsuarioResource.java

A Figura 23 apresenta a requisição feita no Postman para o endpoint de listagem de usuários. Esse endpoint é testado utilizando o método GET com uma requisição para a URL **http://localhost:8080/usuarios**. Na parte inferior da figura é possível observar que a listagem retornou **HTTP Status Code 200**, indicando que o listagem deu certo. Também foi retornado a lista de usuários cadastrados.

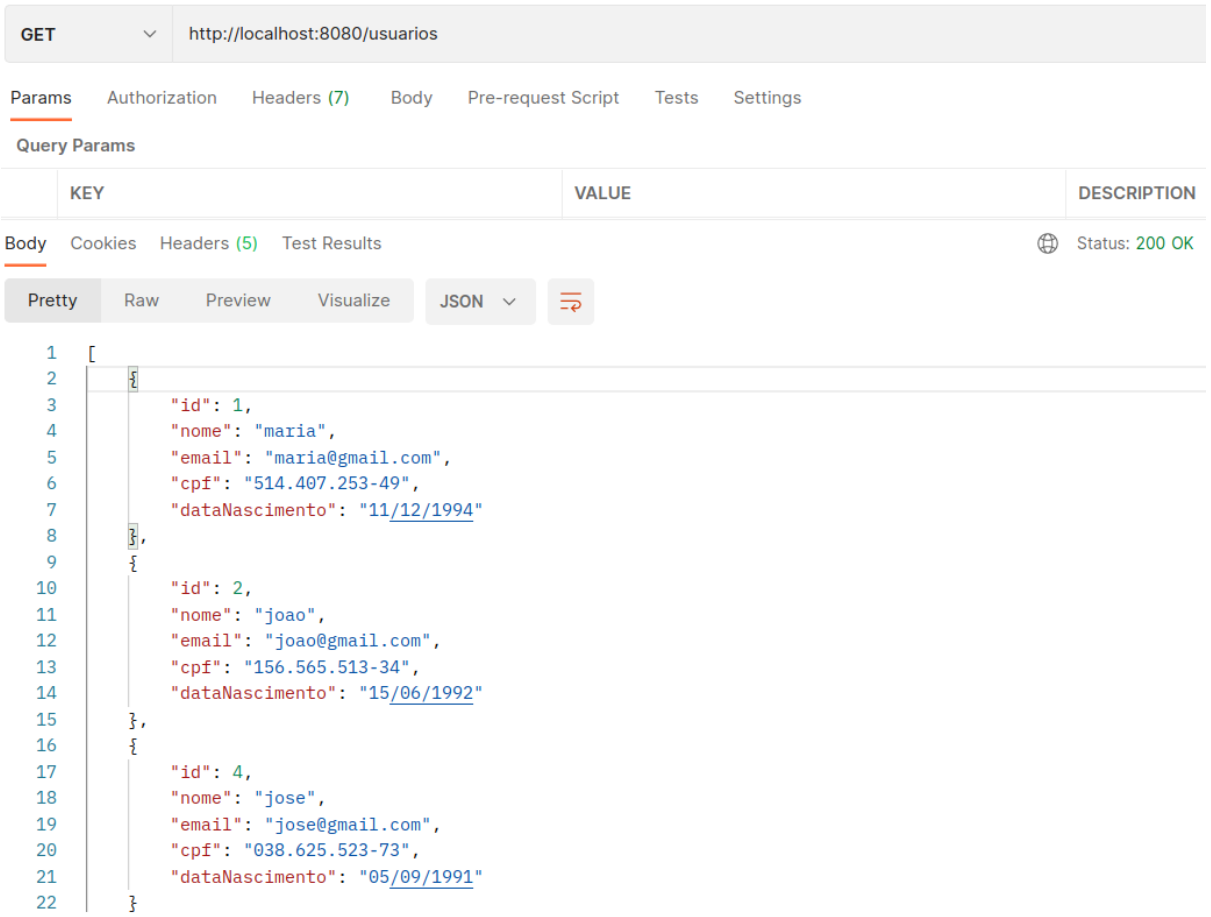


Figura 23 - Requisição de Listagem de usuários no Postman

O terceiro *endpoint* da nossa API é recuperação de usuário por id, apresentado na Figura 24 abaixo, representado pelo método *findById*. Assim como o endpoint anterior, o método *findById* possui a anotação **@GetMapping** porém com o acréscimo do parâmetro *value = "/{id}"* que é acrescentado a rota básica definida na anotação **@RequestMapping**. O método *findById* recebe como parâmetro o id do usuário através do *path* e na sua execução chama o método service.*findById* da camada de serviço de usuário, e ao final retorna o UsuarioDTO corresponde ao id.

```java
    @GetMapping(value = "/{id}")
    public ResponseEntity<UsuarioDTO> findById(@PathVariable Long id){

        UsuarioDTO dto = service.findById(id);
        return ResponseEntity.ok().body(dto);
    }
```

Figura 24 - Método *findById* do UsuarioResource.java

A Figura 25 apresenta a requisição feita no Postman para o endpoint de recuperação de usuário por id. Esse endpoint é testado utilizando o método GET com uma requisição para a URL **http://localhost:8080/usuarios/1** indicando que será recuperado o usuário que tem o id 1. Na parte inferior da figura é possível observar que foi retornado **HTTP Status Code 200**, indicando que o recuperação deu certo. Também foi retornado o usuário cadastrado que possui o id passado como parâmetro.

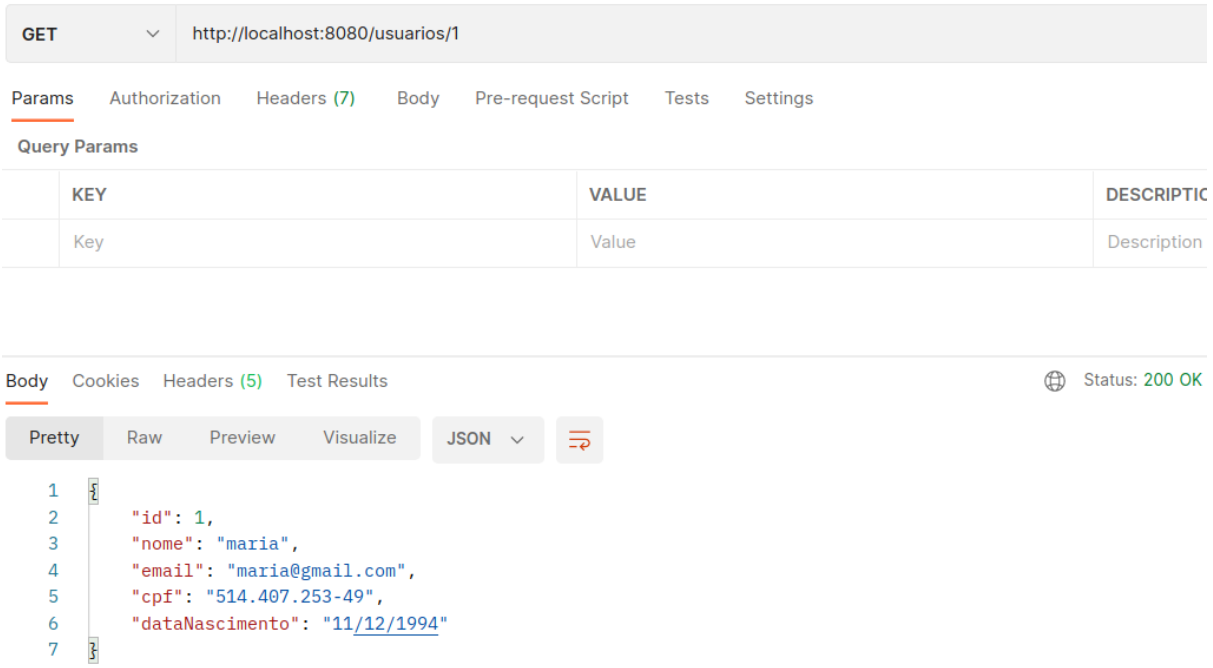


Figura 25 - Requisição de recuperação de usuário por id no Postman

O quarto *endpoint* da nossa API é atualização de usuário por id, apresentado na Figura 26 abaixo, representado pelo método *update*. Para esse endpoint é usada a anotação **@PutMapping** que mapeia solicitações HTTP PUT. Esse endpoint necessita de um parâmetro id pelo *path,* para encontrar o usuário a ser atualizado, e também necessita dos dados a serem atualizados que são passados com a anotação **@RequestBody** e transformados em objetos java. O método *update* chama o método *service.update* da camda de serviço de usuário, e ao final retorna o UsuarioDTO atualizado.

```java
    @PutMapping(value = "/{id}")
    public ResponseEntity<UsuarioDTO> update(@PathVariable Long id, @Valid @RequestBody UsuarioAtualizarDTO dto){
        UsuarioDTO newDto = service.update(id, dto);
        return ResponseEntity.ok().body(newDto);
    }
```

Figura 26 - Método *update* do UsuarioResource.java

A Figura 27 apresenta a requisição feita no Postman para o endpoint de atualização de usuário por id. Esse endpoint é testado utilizando o método PUT com uma requisição para a URL **http://localhost:8080/usuarios/1** indicando que será atualizado o usuário que tem o id 1, também é enviando um JSON com os dados do usuário a serem atualizados. Na parte inferior da figura é possível observar que foi retornado **HTTP Status Code 200**, indicando que a atualização deu certo. Também foi retornado o usuário atualizado que possui o id passado como parâmetro.

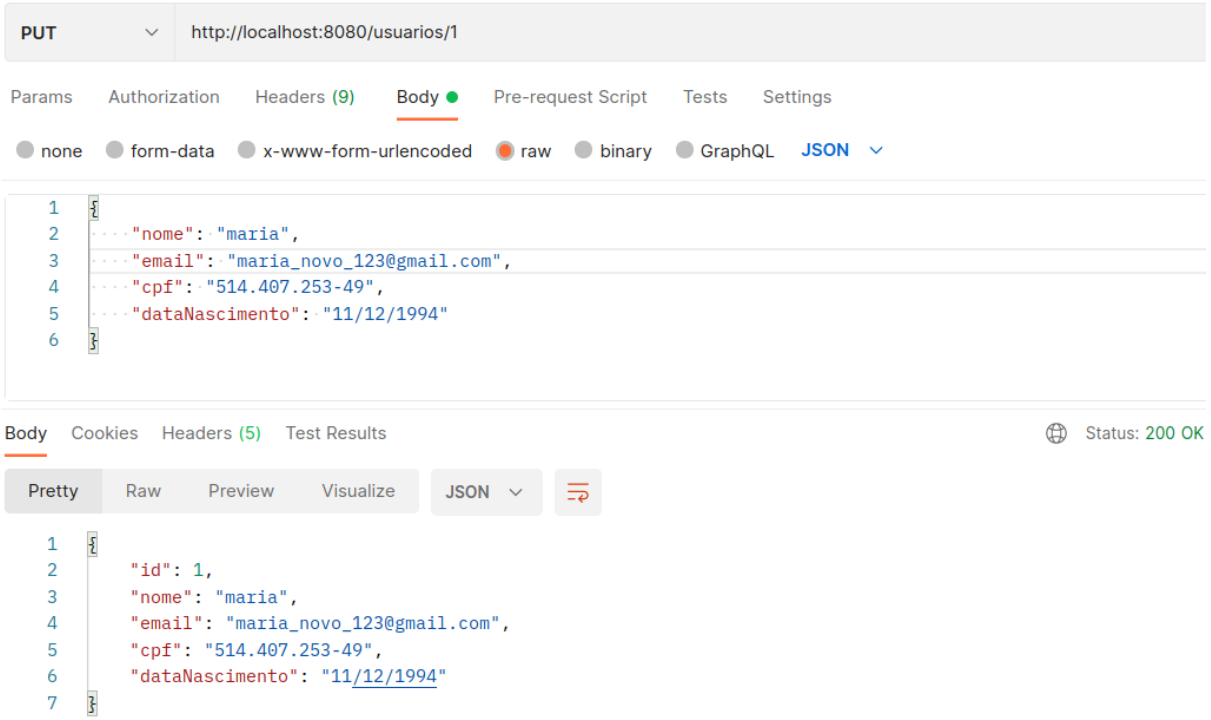


Figura 27 - Requisição de atualização de usuário por id no Postman

O quinto *endpoint* da nossa API é deleção de usuário por id, apresentado na Figura 28 abaixo, representado pelo método *delete*. Para esse endpoint é usada a anotação **@DeleteMapping** que mapeia solicitações HTTP DELETE. Esse endpoint necessita de um parâmetro id pelo *path,* para encontrar o usuário a ser deletado. O método *delete* chama o método *service.delete* da camada de serviço de usuário, e ao final retorna o status de usuário deletado.

```java
    @DeleteMapping(value = "/{id}")
    public ResponseEntity<UsuarioDTO> delete(@PathVariable Long id){
        service.delete(id);
        return ResponseEntity.noContent().build();
    }

```

Figura 28 - Método *delete* do UsuarioResource.java

A Figura 29 apresenta a requisição feita no Postman para o endpoint de deleção de usuário por id. Esse endpoint é testado utilizando o método DELETE com uma requisição para a URL **http://localhost:8080/usuarios/1** indicando que será deletado o usuário que tem o id 1, também é enviando um JSON com os dados do usuário a serem atualizados. Na parte inferior da figura é possível observar que foi retornado **HTTP Status Code 200**, indicando que a atualização deu certo. Também foi retornado o usuário atualizado que possui o id passado como parâmetro.

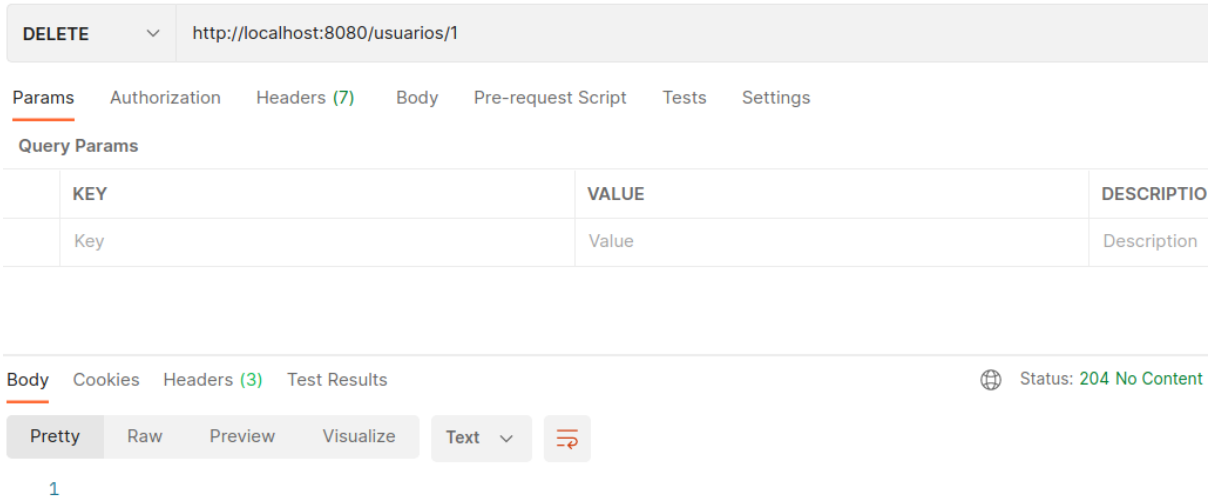


Figura 29 - Requisição de atualização de usuário por id no Postman

O sexto *endpoint* da nossa API é a listagem de comics de um usuário, apresentado na Figura 30 abaixo, representado pelo método *findComicsByIdUsuario*. Esse método possui a anotação **@GetMapping** que indica em uma API REST que o método é um endpoint e que mapeia solicitações HTTP GET. O método *findComicsByIdUsuario* recebe como parâmetro o id do usuário através do *path* e na sua execução chama o método *service.findComicsPorIdUsuario* da camada de serviço de usuário, e ao final retorna uma lista de ComicDTO.

```java
    @GetMapping(value = "/{id}/comics")
    public ResponseEntity<List<ComicDTO>> findComicsByIdUsuario(@PathVariable Long id){

        List<ComicDTO> dto = service.findComicsPorIdUsuario(id);
        return ResponseEntity.ok().body(dto);
    }
}
```

Figura 30 - Método *findComicsByIdUsuario* do UsuarioResource.java

A Figura 31 apresenta a requisição feita no Postman para o endpoint de listagem de comics de um usuário. Esse endpoint é testado utilizando o método GET com uma requisição para a URL **http://localhost:8080/usuarios/2/comics** indicando que será recuperado os comics do usuário que tem o id 2. Na parte inferior da figura é possível observar que a listagem retornou **HTTP Status Code 200**, indicando que a listagem deu certo. Também foi retornado a lista de comics cadastradas para o usuário com id 2. O dia dessa consulta é uma segunda-feira e para o comicId 1003 foi aplicado um desconto, mudando o atributo aplicaDesconto para true, pois o dígito final do isbn é 1.

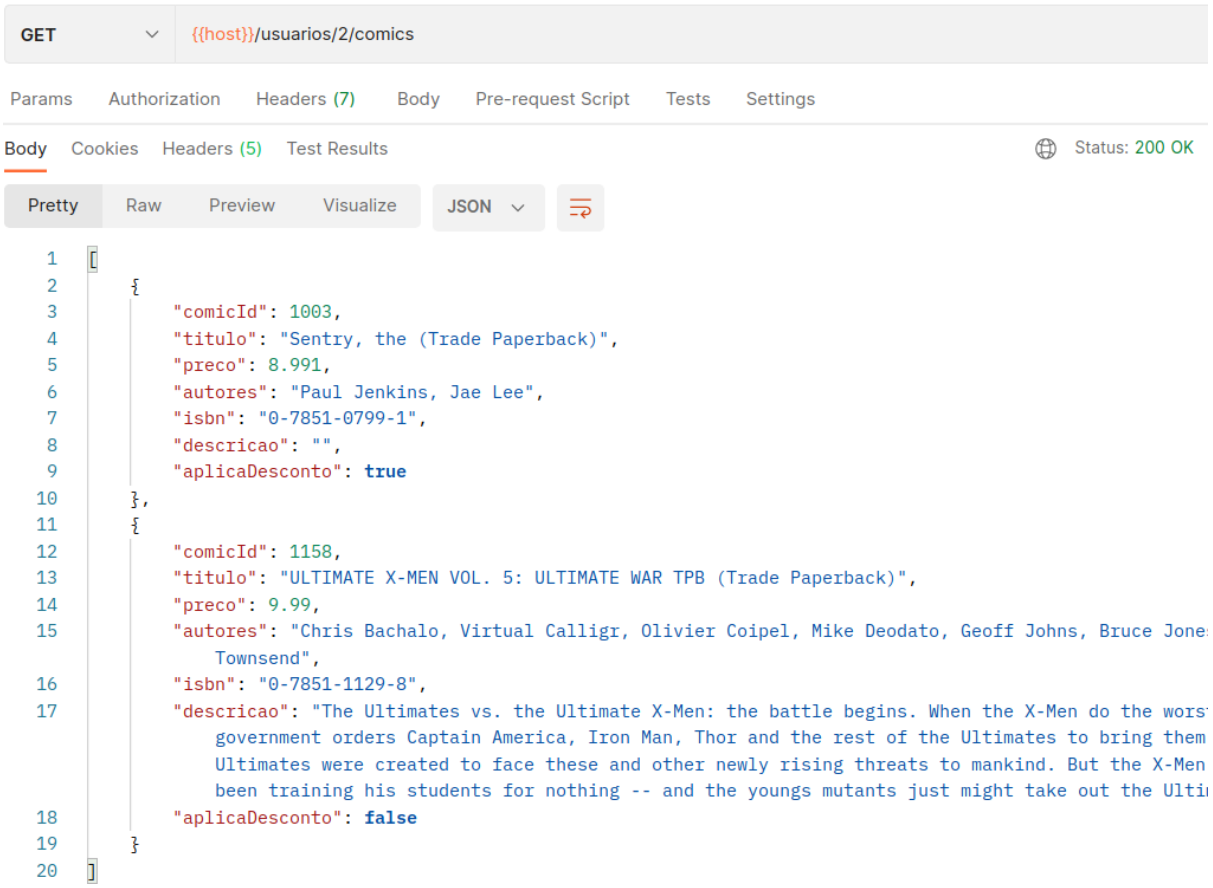


Figura 31 - Requisição de listagem de comic por usuário no Postman

6.1 - Comic

A camada de controlador responsável pelo comic é implementado pela classe ComicResource.java. Essa classe também faz uso da anotação **@RestController** para a transformá-la em um controlador REST e da **@RequestMapping** que mapeia a rota do recurso para **/comics**. O controlador possui apenas um endpoint que é o sétimo endpoint da nossa API. A Figura 32 apresenta o endpoint criado. O método *insert* recebe no corpo da requisição apenas dois parâmetros que são o comicId do comic a ser cadastrado e o usuarioId para relacionar a qual usuário o comic será cadastrado. Os valores são repassados para o método *service.insert* da camada de serviço de comic.

```
60    @PostMapping
61    public ResponseEntity<ComicDTO> insert(@RequestBody ObjectNode objectNode){
62        Long comicId = objectNode.get("comicId").asLong();
63        Long usuarioId = objectNode.get("usuarioId").asLong();
64
65        ComicDTO dto = service.insert(comicId, usuarioId);
66
67        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
68                .buildAndExpand(dto.getComicId()).toUri();
69        return ResponseEntity.created(uri).body(dto);
70    }
```

Figura 32 - Método *insert* do ComicResource.java

A Figura 33 apresenta a requisição feita no Postman para o endpoint de cadastro de comic. Esse endpoint é testado utilizando o método POST com uma requisição para a URL **http://localhost:8080/comics** enviando um JSON com os dados comicId e usuarioId. a parte inferior da figura é possível observar que o cadastro retornou **HTTP Status Code 201**, indicando que o comic foi cadastrado. Também foi retornado o objeto cadastrado.

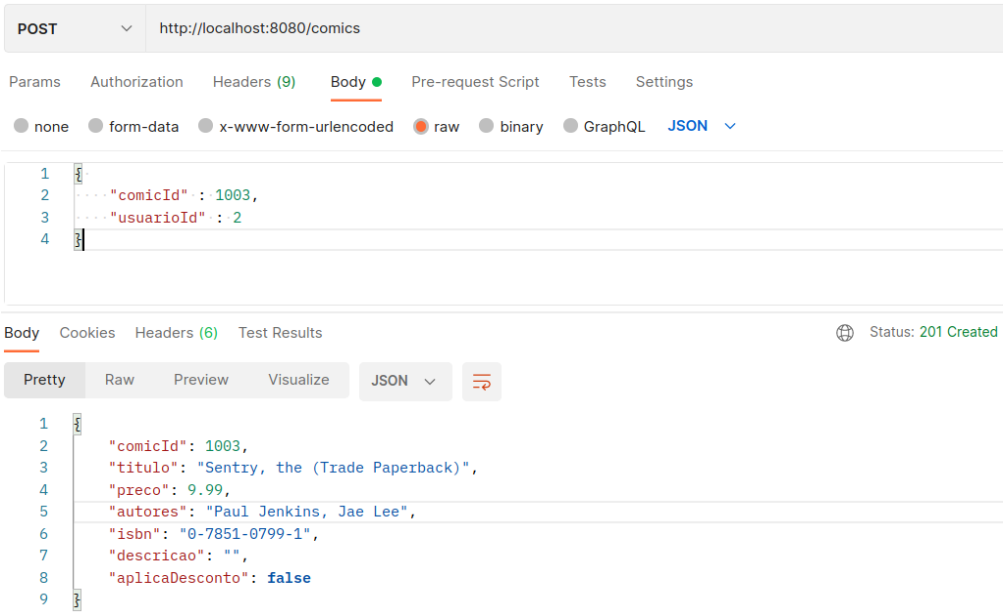


Figura 33 - Requisição de cadastro de comic no Postman

7 - Tratando as Exceções (Exception Handler)

As requisições feitas a API devem retornar erros que possuem códigos apropriados. Como por exemplo o código 400 para informar que algum campo não foi preenchido de maneira correta.

Para tratar as exceções da nossa API REST foi criado uma classe chamada ResourceExceptionHandler.java (Figura 34). Essa classe intercepta as exceções que podem ocorrer. As exceções que podem ser interceptadas podem ser de 3 tipos: ResourceNotFound (quando alguma entidade não é encontrada), DatabaseException (quando alguma restrição do banco de dados é violada) e MethodArgumentNotValidException (quando algum atributo não está de acordo com a validação).

```java
@ControllerAdvice
public class ResourceExceptionHandler {

    @ExceptionHandler(ResourceNotFoundException.class)
    public ResponseEntity<StandardError> resourceNotFound(ResourceNotFoundException e, HttpServletRequest request){
        HttpStatus status = HttpStatus.NOT_FOUND;
        StandardError erro = new StandardError();
        erro.setTimestamp(Instant.now());
        erro.setStatus(status.value());
        erro.setError("Recurso nao encontrado");
        erro.setMessage(e.getMessage());
        erro.setPath(request.getRequestURI());
        return ResponseEntity.status(status).body(erro);
    }

    @ExceptionHandler(DatabaseException.class)
    public ResponseEntity<StandardError> database(DatabaseException e, HttpServletRequest request){
        HttpStatus status = HttpStatus.BAD_REQUEST;
        StandardError erro = new StandardError();
        erro.setTimestamp(Instant.now());
        erro.setStatus(status.value());
        erro.setError("Exception na base de dados");
        erro.setMessage(e.getMessage());
        erro.setPath(request.getRequestURI());
        return ResponseEntity.status(status).body(erro);
    }

    @ExceptionHandler(MethodArgumentNotValidException.class)
    public ResponseEntity<ValidationError> validation(MethodArgumentNotValidException e, HttpServletRequest request){
        HttpStatus status = HttpStatus.BAD_REQUEST;
        ValidationError erro = new ValidationError();
        erro.setTimestamp(Instant.now());
        erro.setStatus(status.value());
        erro.setError("Exception da validacao");
        erro.setMessage(e.getMessage());
        erro.setPath(request.getRequestURI());

        for (FieldError f :  e.getBindingResult().getFieldErrors()) {
            erro.addError(f.getField(), f.getDefaultMessage());
        }
        return ResponseEntity.status(status).body(erro);
```

Figura 34 - ResourceExceptionHandler.java

A classe possui a anotação **@ControllerAdvice** que permite a classe interceptar alguma exceção que ocorra no controlador REST. Além disso, cada método dessa classe possui a anotação **@ExceptionHandler** onde é especificada qual exceção cada método deve tratar.

As figuras a seguir mostram algumas exceções captadas pelo ResourceExceptionHandler.java. A Figura 34 apresenta uma requisição para cadastro de usuário onde o nome do usuário não foi informado e o email informado é inválido. A parte inferior da figura mostra que a requisição retornou **HTTP Status Code 400**, indicando que o cadastro de usuário deu errado, e também foram

apresentadas as mensagens que o campo nome é obrigatório e que o campo email foi preenchido de maneira errada.

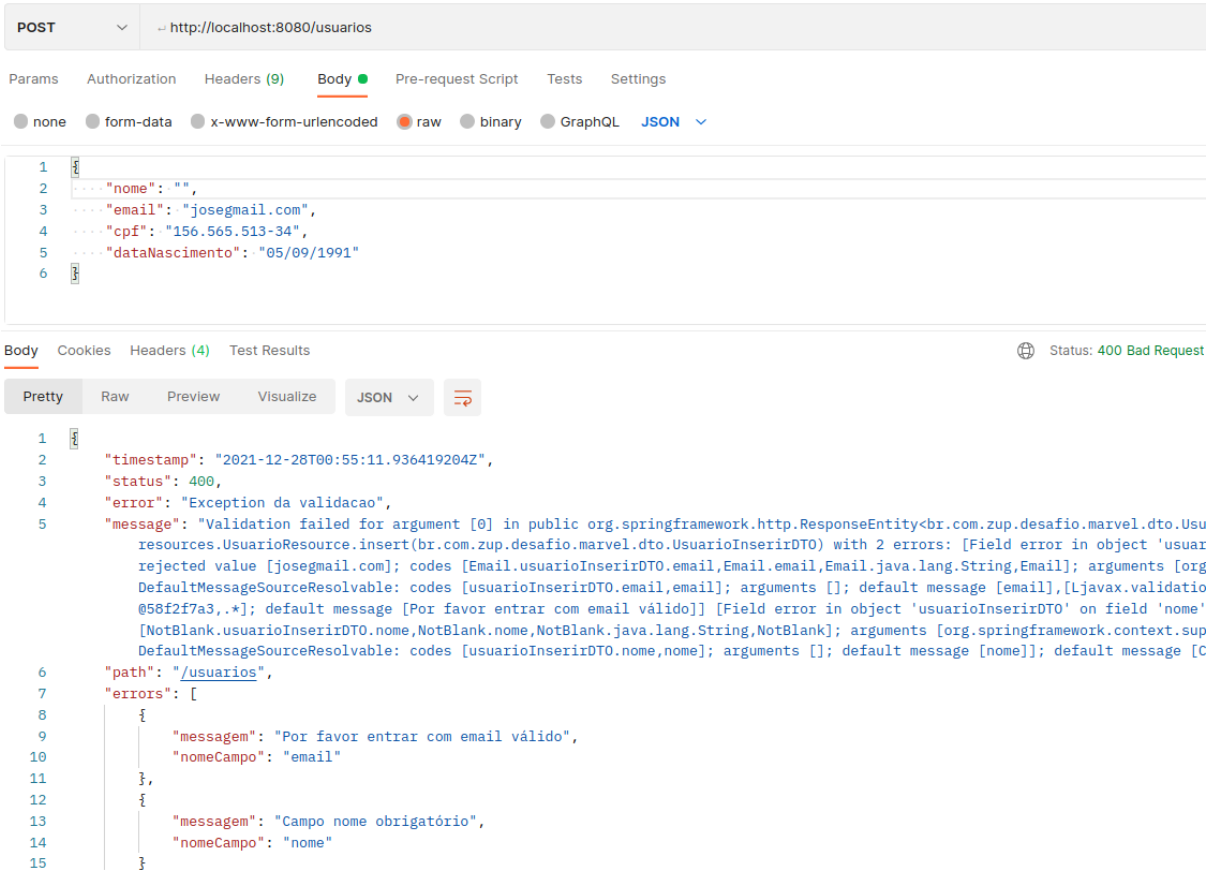


Figura 34 - Requisição de cadastro de usuário com erros no Postman

A Figura 34 apresenta uma requisição para recuperação de usuário por id onde o id informado não está cadastrado no banco. A parte inferior da figura mostra que a requisição retornou **HTTP Status Code 404**, indicando que o recurso não foi encontrado, e também foi apresentada a mensagem que o usuário não foi encontrado.

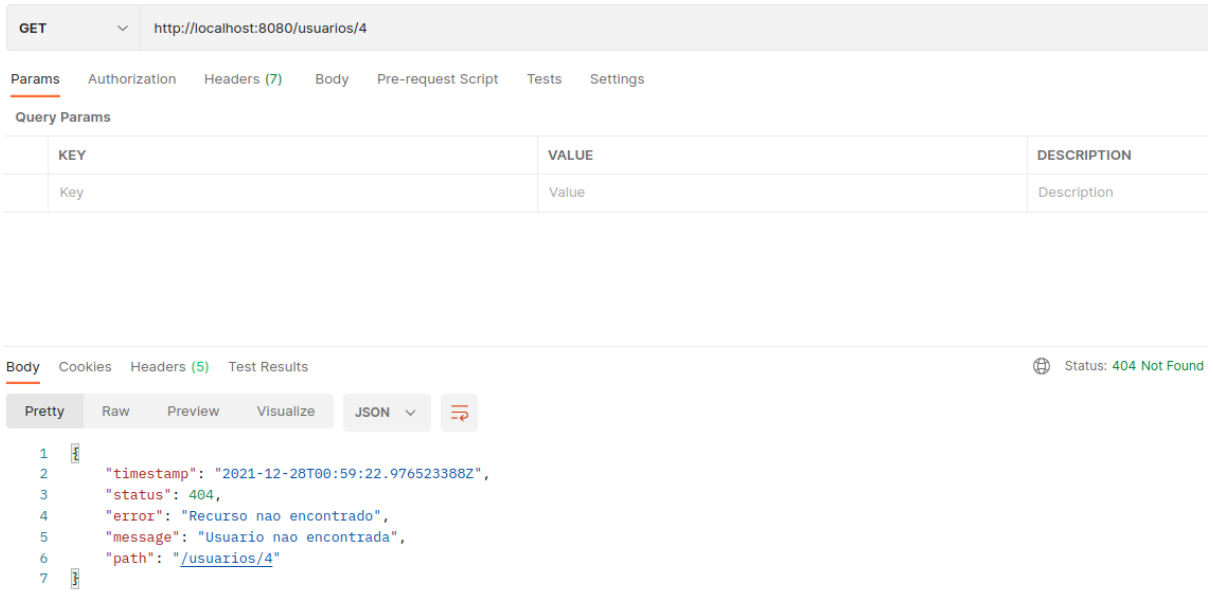


Figura 35 - Requisição de recuperação de usuário por id com erro no Postman

8 - Consumo da API da marvel com Spring-Cloud-Feign

O Spring-Cloud-Feign implementa clientes HTTP java de forma prática através de uma definição de interface. A API da marvel foi consumida pela requisição ao endpoint **GET /v1/public/comics**. A qual retorna uma resposta em formato JSON, mais detalhes da API, como geração da chave de autenticação e outros endpoints podem ser consultados na documentação oficial na URL https://developer.marvel.com/docs#!/. Para consumo desse endpoint foi necessário a implementação das classes seguintes classes para mapeamento dos atributos que queremos filtrar da API da marvel.

As primeiras classes implementadas tratam da transformação dos objetos JSON retornados em objetos Java, funcionando de maneira similar ao mapeamento ORM, onde para cada entrada no retorno teremos a classe java correspondente.

O retorno da API mostrado na Figura 36, mostra parte desse retorno, foram mapeados apenas os itens que nos interessam no retorno. Como a entrada data, a entrada results e as propriedades definidas no problema, como id, title, isbn.

```
{
    "code": 200,
    "status": "Ok",
    "copyright": "© 2021 MARVEL",
    "attributionText": "Data provided by Marvel. © 2021 MARVEL",
    "attributionHTML": "<a href=\"http://marvel.com\">Data provided by Marvel. © 2021 MARVEL</a>",
    "etag": "2d523f9f58a575052e9bb99600af14a75dcbe285",
    "data": {
        "offset": 0,
        "limit": 20,
        "total": 50661,
        "count": 20,
        "results": [
            {
                "id": 82967,
                "digitalId": 0,
                "title": "Marvel Previews (2017)",
                "issueNumber": 0,
                "variantDescription": "",
                "description": "",
                "modified": "2019-11-07T08:46:15-0500",
                "isbn": "",
                "upc": "75960608839302811",
                "diamondCode": "",
                "ean": "",
                "issn": "",
                "format": "",
                "pageCount": 112,
                "textObjects": [],
                "resourceURI": "http://gateway.marvel.com/v1/public/comics/82967",
                "urls": [
                    {
                        "type": "detail",
                        "url": "http://marvel.com/comics/issue/82967/marvel_previews_2017?utm_campaign=apiRef&utm_source=b7c88768f7ef9604a4615d0d3e4153d9"
                    }
                ],
                "series": {
                    "resourceURI": "http://gateway.marvel.com/v1/public/series/23665",
                    "name": "Marvel Previews (2017 - Present)"
                },
                "variants": [
```


Figura 36 - Retorno da API da marvel

Após entendimento do retorno do json, da API da marvel foram necessárias três classes de mapeamento, a classe ComicsResponse, onde a entrada “data” é mapeada (Figura 37), dentro da entrada “data”, possuímos a entrada “result” que trás nossa lista de resultados, Essa entrada foi mapeada na classe “DataResponse”, na sequência, cada item individual de um comic foi mapeado na classe ResultsResponse (Figura 38), que trazem os dados de cada comic individualmente.

```
5  public class ComicsResponse {
6
7      @JsonProperty("data")
8      private DataResponse data;
9
10     public ComicsResponse(@JsonProperty("data") DataResponse data) {
11         this.data = data;
12     }
13
14     public ResultsResponse getResult() {
15         return data.getResult();
16     }
17
18 }
```

Figura 37 - ComicsResponse.java

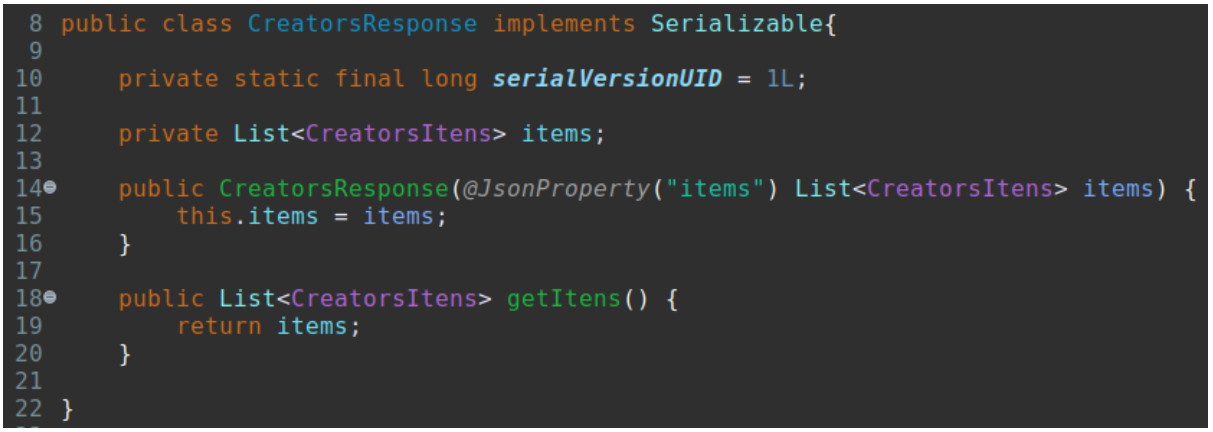
```java
public class DataResponse {

    @JsonProperty("results")
    private List<ResultsResponse> results;

    public DataResponse(@JsonProperty("results") List<ResultsResponse> results) {
        this.results = results;
    }

    public DataResponse() {

    }

    public ResultsResponse getResult() {
        return results.get(0);
    }
```

Figura 38 - DataResponse.java

```java
public class ResultsResponse {

    @JsonProperty("id")
    private Long comicId;
//  private Integer comicId;

    @JsonProperty("title")
    private String title;

    @JsonProperty("isbn")
    private String isbn;

    @JsonProperty("description")
    private String description;

    @JsonProperty("prices")
    private List<PriceResponse> prices;

    @JsonProperty("creators")
    private CreatorsResponse creators;
```

Figura 39 - DataResponse.java

Por fim, foram mapeados os autores das comics, nas classes CreatorsResponde (Figura 40) e CreatorsItens (Figura 41), onde de maneira similar aos resultados e comics, são mapeadas a lista de autores e as propriedades individuais de cada autor, respestivamente.



```java
public class CreatorsResponse implements Serializable{

    private static final long serialVersionUID = 1L;

    private List<CreatorsItens> items;

    public CreatorsResponse(@JsonProperty("items") List<CreatorsItens> items) {
        this.items = items;
    }

    public List<CreatorsItens> getItens() {
        return items;
    }

}
```

Figura 40 - CreatorsResponse.java

```java
public class CreatorsItens {

    private String name;


    public CreatorsItens(@JsonProperty("name") String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

}
```

Figura 41 - CreatorsItens.java

Na nossa classe principal é necessário adicionar a anotação **@EnableFeignClients** para habilitar o Feign para a nossa aplicação. A Figura 42 mostra a anotação colocada antes da nossa classe principal.

```java
@EnableFeignClients
@SpringBootApplication
public class MarvelApplication {

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

}

```

Figura 42 - Classe principal do projeto

Após isso é necessário criar uma interface onde o Feign chama o serviço.

```java
@FeignClient(url = "${marvel.url}/comics", name = "marvel-url")
public interface MarvelComicsClient {

    @GetMapping("/{id}")
    ResponseEntity<ComicsResponse> findComicsPorId(
            @PathVariable Long id,
            @RequestParam String ts,
            @RequestParam String apikey,
            @RequestParam String hash);
}
```

Figura 40 - MarvelComicsClient.java

O próximo passo é criar o serviço e então implementar as regras necessárias para montar os parâmetros e enviar para a interface. Nessa classe é onde fica o mapeamento da requisição em si.

```java
@Service
public class SolicitarComic {

    @Autowired
    private MarvelComicsClient marvelComicsClient;

    @Value("${marvel.public.key}")
    private String publicKey;

    @Value("${marvel.private.key}")
    private String privateKey;


    public ResultsResponse buscaComicPorId(Long id) {
        var dataAtual = String.valueOf(System.currentTimeMillis());
        var response = marvelComicsClient.findComicsPorId(id, dataAtual, publicKey, hash(dataAtual)).getBody();
        return response.getResult();
    }

    private String hash(String dataAtual) {
        try {
            MessageDigest md = MessageDigest.getInstance("MD5");
            var Stringhash = dataAtual + privateKey + publicKey;
            BigInteger hash = new BigInteger(1, md.digest(Stringhash.getBytes(StandardCharsets.UTF_8)));
            return hash.toString(16);
        } catch (NoSuchAlgorithmException e) {
            throw new RuntimeException();
        }
    }
```

Figura 43 - SolicitarComic.java

O serviço implementado na Figura 43 é chamado no controlador de Comic para consumir a API da marvel e assim cadastrar as comics retornadas, a figura 18 mostra como o serviço é instanciado e os dados carregados no momento da criação de um objeto comic.

9 - Estrutura Final do Projeto

A Figura 44 apresenta a maneira como está organizado o projeto da API. A organização do projeto segue os padrões apresentados na Figura 1, onde cada classe/interface fica organizada organizada em um pacote junto com as demais classes do mesmo escopo:

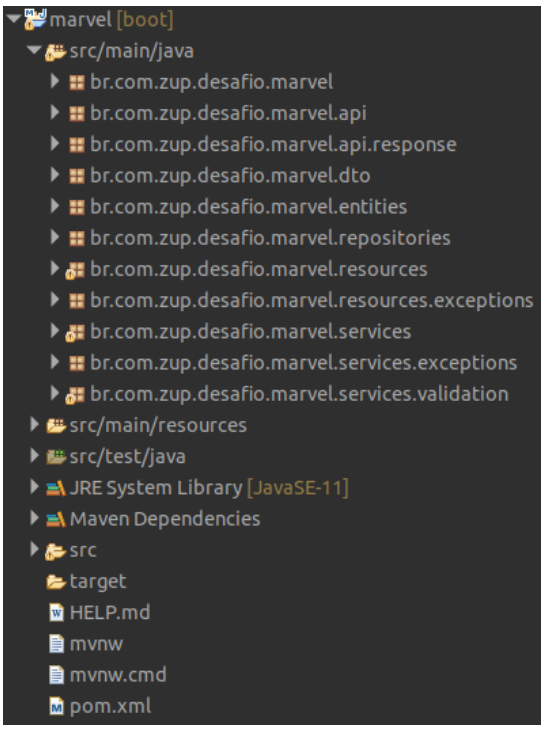


Figura 44 - Estrutura do projeto

Cada pacote apresentado na Figura 44 contém classes que possuem funções relacionadas:

api: Possui os códigos responsáveis pelo consumo da api da marvel;

dto: Possui as classes de abstração de dados do tipo dto que fazem a comunicação entre a camada de controladores e a camada de serviço;

entities: Possui as classes de entidades que definem os dados a serem persistidos no banco;

repositories: Possui as classes responsáveis pelas interfaces JPA que fazem transição dos dados com o banco de dados;

resources: Possui as classes responsáveis por receber as requisições e encaminhar as chamadas para os serviços responsáveis por realizar aquela ação;

services: Possui as classes que implementam as regras de negócio.