API REST para gerenciar quadrinhos (Comics) de Usuários
Desafio ZUP

Nome: Marcelino Mendes da Silva Neto
E-mail: marcelinoneto34@gmail.com
github: https://github.com/marcelinoNet/Desafio-ZUP.git

O projeto que vamos desenvolver é uma API REST para gerenciamento de quadrinhos de usuários. Para tal, utilizaremos a linguagem de programação Java 11, o framework Spring boot na versão 2.4.4 e o gerenciamento de dependência Maven.

A API REST possui os seguintes requisitos:

1. O Usuário deve possuir os seguintes dados cadastrados:
    1. **Nome**: obrigatório;
    2. **E-mail**: obrigatório, deve ser um e-mail valido (ex: formato usuario@email.com) e por fim, deve ser único (apenas um usuário cadastrado pode possuir um determinado endereço de e-mail);
    3. **CPF**: obrigatório, deve ser valido e único;
    4. **Data de nascimento**: obrigatório e deve estar no formato dd/MM/yyyy.
2. O Quadrinho (Comic) possui os seguintes dados cadastrados:
    1. **ComicId**: obrigatório;
    2. **Título**: obrigatório;
    3. **Preço**: obrigatório;
    4. **Autores**: obrigatório;
    5. **ISBN**: obrigatório e único (apenas um Comic cadastrado pode possuir um determinado ISBN);
    6. Para cadastrar o Comic deve-se consumir a API da MARVEL (disponível em https://developer.marvel.com/).
    7. Ao cadastrar um Comic deve associá-lo a um usuário cadastrado no sistema;
    8. O endpoint de cadastro da Comic deve receber um ID de uma Comic, o id do usuário ao qual ele será associado. Desse modo, o procedimento de consumir da API da Marvel e o cadastro da Comic associado com usuário devem ocorrer em uma única requisição;
3. Construir um endpoint para retornar a lista de todos os usuários cadastrados. Para essa consulta, não deve ser retornada a lista de quadrinhos de cada usuário;
4. Construir um endpoint para retornar uma consulta da lista de Comics de um determinado Usuário. Para esta listagem, devemos obedecer algumas regras de negócio da aplicação:
    1. Casa quadrinho possui um desconto de 10% que pode ser aplicado ao seu preço de acordo com o dia da semana em que a consulta foi realizada;
    2. O dia da semana que corresponde a aplicação do desconto varia de acordo com o dígito final do ISBN, conforme abaixo:
        1. FINAL 0-1: segunda-feira;

2. FINAL 2-3: terça-feira;
3. FINAL 4-5: quarta-feira;
4. FINAL 6-7: quinta-feira;
5. FINAL 8-9: sexta-feira.

3. Deverá haver um atributo em cada quadrinho indicando se o desconto está ou não sendo aplicado;
4. Caso o quadrinho possua desconto, o mesmo deverá ser aplicado ao seu valor quando retornar os dados do quadrinho;

5. As respostas retornadas pelos endpoints da API construída devem seguir os requisitos:

    1. 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 apresentar um mensagem indicando qual campo foi preenchido incorretamente;
    2. Caso as buscas estejam corretas, é necessário retornar o HTTP 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.
6. Desafio Extra
    1. Integrar a API da MARVEL usando Spring-Cloud-Feign;
    2. Construir um Exception Handler para o tratamento das exceções que podem ocorrer na execução da aplicação.

1. Estrutura Geral do projeto

Para o desenvolvimento desse projeto vamos utilizar o seguinte padrão de camadas, pode ser visto na figura 1:

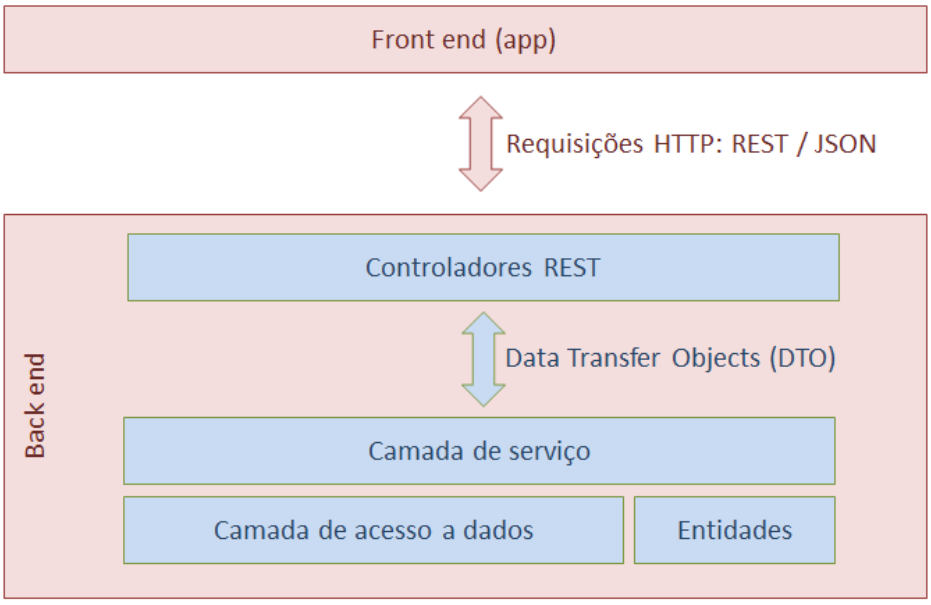

Figure 1: Estrutura geral do projeto

- Repositórios (Acesso aos dados no banco de dados);

- DTO's (Objetos que carregam os dados entre as camadas de serviço e controladores);
- Serviços (Manipula os dados do banco e converte para DTO);
- Controladores (Interface REST).

1.1. Diagrama de classe da API

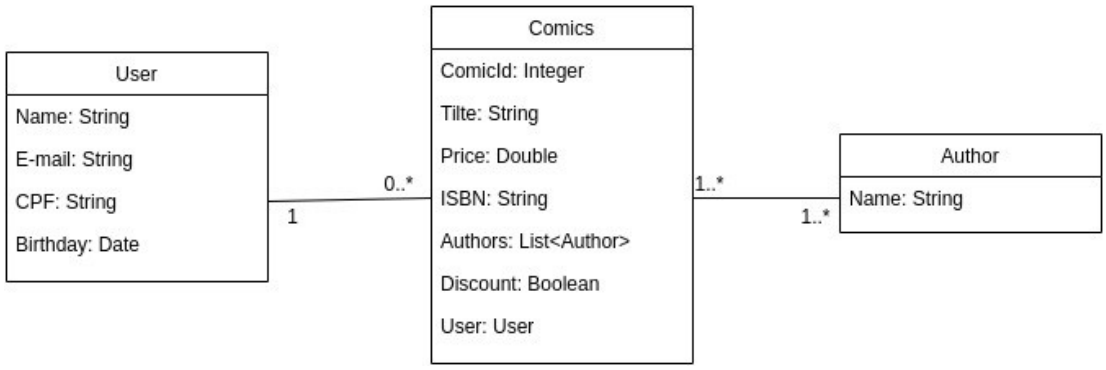

Figure 2: Diagrama de Classes

Para este projeto iremos utilizar o diagrama de classes, como visto na figura 2. Os relacionamentos entre as classes são:

- User pode ter zero ou várias Comics;
- Uma Comic só pode ser associado a um User;
- Um Comic pode ter um ou vários authors;
- Um author pode estar associado a um ou várias Comics.

1.3. Criando o projeto REST Spring Boot

Para criar o aplicativo Spring Boot, usaremos https://start.spring.io/, para gerar nosso projeto base. Precisamos dos pacotes da DevTools, Web, H2 Database, Validation e o JPA, como pode ser visto na figura 3:

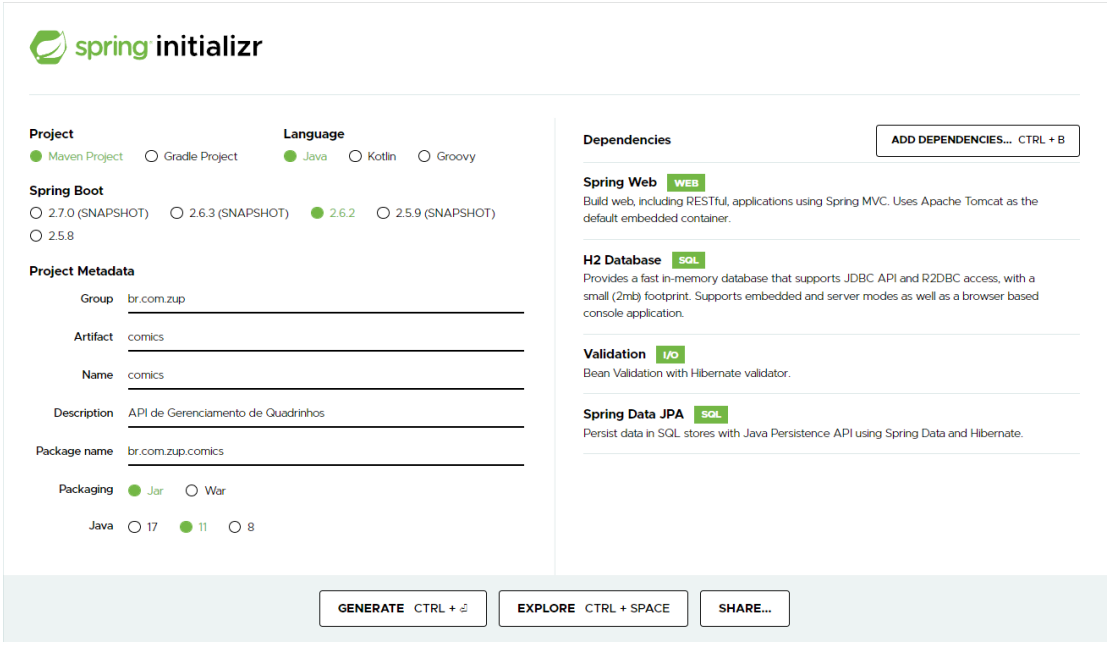

Figure 3: Spring Initializr

Clique no botão Generate, baixe o arquivo zip, extraia seu conteúdo para sua área de trabalho e abra-o em seu IDE favorito.

2. Usuário(User)

A primeira classe que vamos criar é a Entidade **User** (figura 4) dentro do pacote **entities**. A classe ***User*** que conterá as especificações definidas para usuários.

```java
@Entity
@Table(name="tb_users")
public class User implements Serializable{

    private static final long serialVersionUID = 1L;
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;

    @Column(unique = true)
    private String email;

    @Column(unique = true)
    private String cpf;

    @DateTimeFormat(pattern = "dd/MM/yyyy")
    private LocalDate birthday;

    // Construct, Getters and Setters....
```

Figure 4: Entidade User

Para a entidade **User** utilizamos as seguintes anotações:

- **@Entity** é utilizada para informar que uma classe também é uma entidade. A partir disso, a JPA estabelecerá a ligação entre a entidade e uma tabela de mesmo nome no banco de dados, onde os dados de objetos desse tipo poderão ser persistidos;
- **@Table** fornece a tabela que mapeia esta entidade;
- **@Id** é para a chave primária;
- **@GeneratedValue** é usada para definir a estratégia de geração para a chave primária. Os valores a serem atribuídos ao identificador único serão gerados pela coluna de auto incremento do banco de dados;
- **@Column** é usada para definir a coluna no banco de dados que mapeia o campo anotado. O paramentro **UNIQUE**, significa que os valores deste campo devem ser únicos;
- **@DateTimeFormat**(pattern = “dd/MM/yyy”) declara que um parâmetro de campo ou método deve ser formatado como uma data ou hora. No nosso caso, está formatando a data para o formato definido.

2.1. Criando o Repositório JPA

Para acessar facilmente os métodos de manipulação da tabela **User**, precisamos apenas criar uma interface que estenda JpaRepository passando a classe que representa nossa Entidade e o tipo da chave primária como argumentos genéricos, desse modo vamos criar a **UserRepository** (figura 5) dentro do pacote **repositories**:

```java
@Repository
public interface UserRepository extends JpaRepository<User, Long>{

    User findByEmail(String email);
    User findByCpf(String cpf);

}
```

Figure 5: UserRepository

A interface JpaRepository fornece uma maneira simples e fácil de acessar todas as operações CRUD. Podemos usar os métodos de JpaRepository: save (), findOne (), findById (), findAll (), count (), delete (), deleteById () ... sem implementar esses métodos.

Também podemos definir métodos personalizados:

- findByEmail: retorna o usuário que possua o e-mail passado no parametro;
- findByCpf: retorna o usuário que possua o CPF passado no parametro;

2.2. Criando o UserDTO

O ***Data Transfer Object*** (DTO) ou simplesmente ***Transfer Object*** é um padrão de projetos bastante usado em Java para o transporte de dados entre diferentes componentes de um sistema, diferentes instâncias ou processos de um sistema distribuído ou diferentes sistemas via serialização. A ideia consiste basicamente em agrupar um conjunto de atributos numa classe simples de forma a otimizar a comunicação. Muitas vezes os dados usados na comunicação não refletem exatamente os atributos do seu modelo. Então, um DTO seria uma classe que provê exatamente aquilo que é necessário para um determinado processo.

Agora que sabemos o que é o DTO e para que serve, vamos ver como está o nosso **UserDTO** (figura 6) dentro do pacote **dtos**.

```java
public class UserDTO implements Serializable{

    private static final long serialVersionUID = 1L;

    private Long id;

    @NotBlank(message = "O Nome é obrigatório!!!")
    private String name;

    @NotBlank(message = "O E-mail é obrigatório!!!")
    @Email(message = "O e-mail deve ser um e-mail válido!!!")
    private String email;

    @NotBlank(message = "O CPF é obrigatório!!!")
    @CPF(message = "O CPF deve ser válido")
    private String cpf;

    @NotNull(message = "A Data de Nascimento é obrigatório!!!")
    private LocalDate birthday;
```

Figure 6: UserDTO

Para o **UserDTO** utilizamos as seguintes anotações:

- @NotBlank: não permite valor nulo e o comprimento (sem considerar espaços em branco) deve ser maior que zero.
- @NotNull: Não permite um valor nulo, porém permite um valor vazio.
- @Email: usado para validar o e-mail;
- @CPF: usado para validar o CPF.

```
public UserDTO() {}

public UserDTO(Long id, String name, String email, String cpf, LocalDate birthday) {
    this.id = id;
    this.name = name;
    this.email = email;
    this.cpf = cpf;
    this.birthday = birthday;
}
public UserDTO(User entity) {
    id = entity.getId();
    name = entity.getName();
    email = entity.getEmail();
    cpf = entity.getCpf();
    birthday = entity.getBirthday();
}
```

Figure 7: UserDTO - Contrutores

No **UserDTO**, figura 7, utilizamos 3 construtores: O primero e o segundo construtor são contrutores padrão. O terceiro construtor é usado para facilitar a conversão da entide **User** para o **UserDTO**, passado a entidade por paramentro.

Vamos precisar de mais um DTO para a entidade **User**: **UserInsertDTO** (figura 8). Iremos falar mais sobre esta classe quando falarmos sobre as validações e tratamento de exceções. Mas agora vamos apresenta-lo.

```
@UserInsertValid
public class UserInsertDTO extends UserDTO{

    private static final long serialVersionUID = 1L;

}
```

Figure 8: UserInsertDTO

O **UserInsertDTO** herda da classes **UserDTO**. E utiliza uma anotação personalizada, que iremos discurir no decorrer deste blog post.

2.3. Criando UserService

```
@Service
public class UserService {

    @Autowired
    private UserRepository userRepository;
```

Figure 9: UserService

A anotação **@Service**, como pode ser visto na figura 9, indica que a classe está na camada de Serviço. Anotação **@Autowired** é usado para injetar as dependência da classe **UserRepository**. O **UserService** possui os seguintes métodos, figura 10:

```
@Transactional(readOnly = true)
public List<UserDTO> findAll(){
    List<User> list = userRepository.findAll();
    return list.stream().map((x) -> new UserDTO(x)).collect(Collectors.toList());
}

@Transactional
public UserDTO insert(UserInsertDTO dto) {
    User entity = new User();
    convertDtoToEntity(dto, entity);
    entity = userRepository.save(entity);
    return new UserDTO(entity);
}


private void convertDtoToEntity(UserDTO dto, User entity) {
    entity.setName(dto.getName().trim());
    entity.setEmail(dto.getEmail().toLowerCase().trim());
    entity.setCpf(dto.getCpf());
    entity.setBirthday(dto.getBirthday());
}
```

Figure 10: UserService - métodos

O método *findAll()* returna a lista de todos os usuários cadastrados no sistema. Esse método faz uso dos métodos providos do **userRepository.** A anotação @Transactional(readonly = true) informa que determinada transação não pode realizar operações de escrita ou alterações, apenas leitura.

O método *insert()* é responsavel por salvar as informaçãos do usuário no banco de dados. Ele recebe com entrada um **UserInsertDTO.** O método *convertDtoToEntity()* é utilizado para converter o DTO que veio da camada do Controller para a entidade para que possa ser salvo no banco de dados. Possui dois parametros, o DTO e o entidade **User**.

2.4. Criando o UserController

A classe **UserController**, figura 11, dentro do pacote **controllers** é bastante simples, primeiro é adicionada a anotação **@RestController** para indicar que a classe é um controlador, e a anotação **@RequestMapping**, que indica o caminho para acessar esse controlador. Precisamos injeta a dependência de **UserService**. Além disso, existem dois métodos: um para listar os livros de um

determinado autor e que será acessado quando a requisição para esse controlador for do tipo GET, e o método para adicionar um usuário, que será acessado quando a requisição for do tipo POST.

A partir dos requisitos apresentados, iremos criar os seguintes entpoints:

- Retornar a lista de todos os usuários cadastrados: @GetMapping(“/users”);
- Cadastrar usuário: @PostMapping(“/users”);

```java
@RestController
@RequestMapping(value="/users")
public class UserController {

    @Autowired
    private UserService userService;

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



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

Figure 11: UserController

O método *findAll()* retorna a lista de usuários cadastrados no banco de dados e o HTTP Status Code. Caso a busca esteja correta, retorna o HTTP Code 200, representado pelo “*ResponseEntity.ok()*”.

O método *insert()* é responsavel por pegar as informações do Usuário informadas pelo cliente da aplicação e a enviar para o Service para salvar as informações no banco de dado, caso as informações recebidas estejam corretas. A anotação **@RequestBody** indica que um parâmetro do método deve ser vinculado ao corpo da solicitação da web. Isso significa que o método espera o conteúdo da solicitação (no formato JSON). O **@Valid** que irá executar a validação de todas as anotações da nossa classe **UserInsertDTO**. O método retorna o HTTP Status Code 201, por meio do “*ResponseEntoty.created()*”;

2.5. Testando a API de Usuários

Nós podemos testar nossa API de Usuário com o Postman.

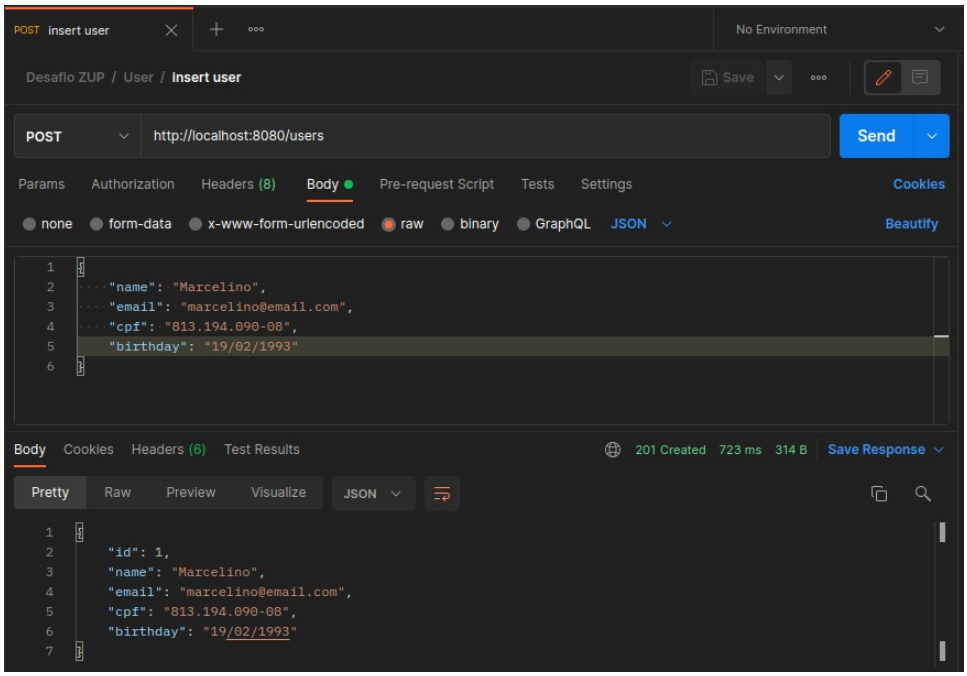

Figure 12: Testando o endpoint de User

Como podemos ver na figura 12, o nosso endpoint para inserir o usuário está funcionando corretamente e retornando o HTTP Status code 201.

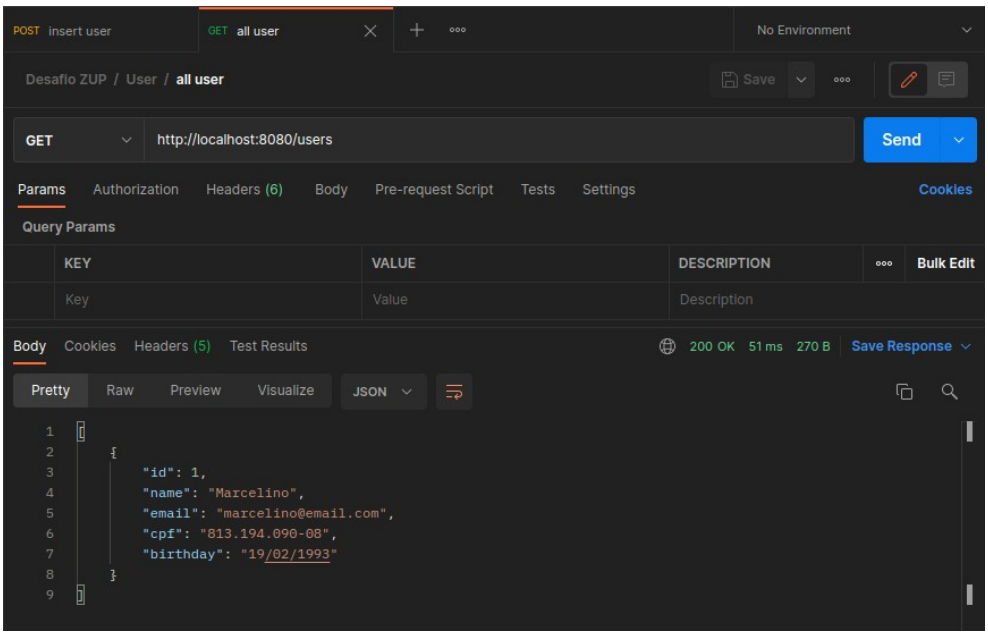

Figure 13: Testando endpoint de User

O endpoint para listar todos os usuários, figura 13, está funcionado e retornando o HTTP Status code 200.

3.1. Quadrinhos (Comics)

Agora vamos criar as Entidades **Comics** (figura 14) e **Author** (figura 15) dentro do pacote **entities**:

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

    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    private Integer comicId;
    private String title;
    private Double price;
    @Column(unique = true)
    private String isbn;
    @Column(columnDefinition = "TEXT")
    private String description;
    private boolean discount;

    @ManyToMany(cascade = CascadeType.PERSIST)
    @JoinTable(name = "tb_comic_author",
               joinColumns = @JoinColumn(name = "comic_id"),
               inverseJoinColumns = @JoinColumn(name = "author_id"))
    Set<Author> authors = new HashSet<>();

    @ManyToOne
    @JoinColumn(name = "user_id")
    private User user;
```

Figure 14: Entidade Comic

Para a entidade **Comic** utilizamos as mesmas anotações que utilizamos na entidade **User.** A entidade está de acordo com o diagrama de classes (figura 2) para satisfazer esse projeto. Para o atributo **autores** criamos uma entidade. Desse modo, podemos agregar mais informações, caso necessário, ao atributo autores sem mudar a estrutura da entidade **Comic**. Por exemplo: idade do autor, atividade desenvolvida no desenvolvimento da Comic: ilustrador, escritor, pintor e etc.

O atributo user do tipo **User** foi acrescentado a entidade **Comic** para associar o quadrinho ao usuário, para estar de acordo com os requisitos deste projeto. Por fim, o atributo discount é utilizado para informar se o quadrinho possui desconto ou não.

Acrescentamos as anotações **@ManyToMany**, **@JoinTable**, **@ManyToOne** e **@JoinColumn**.

- **@ManyToMany**: informa o relacionamento entre duas entidades de forma múltipla, ou seja, a entidade **Comic** possui várias entidades **Author**, assim como a entidade **Author** possui várias entidades **Comic**. O parametro CascateType.PERSIST propaga a operação de persistir um objeto Pai para um objeto Filho, assim quando salvar a Entidade **Comic**, também será salvo todas as Entidades **Author** associadas;
- **@JoinTable**: como se trata de uma relação N-N, será necessário criar uma nova tabela que fará o relacionamento entre **Comic** e **Author**. O nome dessa tabela foi definido pelo **@JoinTable**. E o nome dos campos usando o **@JoinColumn**. O parametro do joinColumns está informando a

quem é o lado “dominante”, ou seja, o lado que pode afetar diretamente a tabela criada pelo **@JoinTable**. E por fim, o inverseJoinColumns está definindo o lado “dominado”;

- **@ManyToOne**: faz o mapeamento de muitos para um de **User.** Ou seja, um User pode ter muito **Comics**, mas um **Comics** só pode estar associado a um **User**.

```java
@Entity
@Table(name="tb_authors")
public class Author {

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

    @ManyToMany(mappedBy = "authors")
    private Set<Comic> comics = new HashSet<>();
```

Figure 15: Entidade Author

A figura 15 apresenta a entidade **Author.** Contento apenas os atributos id e o name. A anotação @ManyToMany faz o mapeamento do relacionamento do lado da entidade **Author**.

3.2. Criando o ComicRepository

Vamos criar o respositorio da **Comic**, figura 16, dentro do pacote de repositories. Como já dito anteriormente, o repository é usando para acessar facilmente os métodos de manipulação da tabela **Comic.**

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

    List<Comic> findComicsByUser(User user);
}
```

Figure 16: ComicRepository

Também podemos definir métodos personalizados para o **ComicRepository**:

- FindComicsByUser: retorna uma lista de quadrinhos associados ao usuário informado.

3.3. Criando o ComicDTO e o AuthorDTO

Vamos utilizar o **ComicDTO** e **AuthorDTO** para receber e para retornar os dados do **Comic** do/para o cliente da API.

```java
public class ComicDTO implements Serializable{

    private static final long serialVersionUID = 1L;
    private Long id;

    @NotNull(message = "ComicId é obrigatório")
    private Integer comicId;
    @NotBlank(message = "O Título é obrigatório")
    private String title;
    @PositiveOrZero
    @NotNull(message = "O Preço é obrigatório")
    private Double price;
    @NotBlank(message = "O ISBN é obrigatório")
    private String isbn;
    private String description;

    private boolean discount;

    List<AuthorDTO> authors = new ArrayList<>();
```

Figure 17: ComicDTO

O **ComicDTO**, figura 17, possui as mesmas informações da entidade **Comic** exceto o atribuito User. Nessa classe estamos utilizando uma nova anotação: @PositiveOrZero. Essa anotação é usado para verificar se o valor numérico informado é positivo incluindo o zero. Desse modo, evita-se que o preço seja negativo.

```java
public ComicDTO() {
}

public ComicDTO(Long id,Integer comicId, String title, Double price, String isbn,
        String description, boolean discount) {
    this.id = id;
    this.comicId = comicId;
    this.title = title;
    this.price = price;
    this.isbn = isbn;
    this.description = description;
    this.discount = discount;
}

public ComicDTO(Comic entity) {
    id = entity.getId();
    comicId = entity.getComicId();
    title = entity.getTitle();
    price = entity.getPrice();
    isbn = entity.getIsbn();
    description = entity.getDescription();
    discount = entity.isDiscount();

    authors = entity.getAuthors().stream()
            .map(x-> new AuthorDTO(x)).collect(Collectors.toList());

}
```

Figure 18: ComicDTO - contrutores

Da mesma forma do **UserDTO**, vamos criar 3 contrutores na classe **ComicDTO**, figura 18. Um construtor sem argumentos, um com todos os atributos, exceto o atributo authors. Pois como é uma

lista, podemos inserir suas informação utilizando os métodos oferecidos pela lista. E o ultimo construtor recebe a própria entidade Comic.

```java
public class AuthorDTO implements Serializable{

    private static final long serialVersionUID = 1L;
    private Long id;

    @NotBlank(message = "O Nome do autor é obrigatório")
    private String name;
```

Figure 19: AuthorDTO

O DTO **AuthorDTO**, figura 19, possui as mesmas informações da entidade **Author**.

Antes de falarmos sobre o **ComicService** e o **ComicController**. Vamos ver a API de quadrinhos da Marvel, pois um dos requisitos da nossa API é consumir a API da Marvel para que possamos inserir quadrinhos na API que estamos construindo. Para isso iremos utilizar o Feign.

3.4. Consumindo API da Marvel utilizando o Feign

O Feign é uma boa alternativa para implementar clientes HTTPs de forma fácil e pratica, não sendo necessário escrever nenhum código para chamar o serviço, a não ser uma definição de interface. Para entender a estrutura de resposta da API, figura 20, vamos consumir a API da Marvel, especificamente a requisição GET /v1/public/comics. A API devolve em formato JSON a seguinte resposta:


```json
{
  "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": "91b14b11eff9fc398a573e52039c72ab4f0d77fb",
  "data": {
    "offset": 0,
    "limit": 20,
    "total": 1,
    "count": 1,
    "results": [
      {
        "id": 1308,
        "digitalId": 0,
        "title": "Marvel Age Spider-Man Vol. 2: Everyday Hero (Digest)",
        "issueNumber": 0,
        "variantDescription": "",
        "description": "\"The Marvel Age of Comics continues! This time around,Spidey meets his match against
        "modified": "2018-01-22T15:42:11-0500",
        "isbn": "0-7851-1451-3",
        "upc": "5960611451-00111",
        "diamondCode": "",
        "ean": "",
        "issn": "",
        "format": "Digest",
        "pageCount": 96,
        "textObjects": [
          {
            "type": "issue_solicit_text",
```


Figure 20: Resposta da API da Marvel

Nesse exemplo podemos ver que o payload de resposta é muito grande, com muitas informações que não tem interesse em consumir. Para filtrar as informações relevantes para nós, criaremos classes para mapear apenas os atributos que desejamos.

Criaremos o pacote MarvelApi. Dentro deste pacote vamos criar mais dois pacotes: client e response. Iremos fazer isso para organizar melhor nosso código. Dentro do pacote response iremos criar as seguintes classes: **ComicsResponse** (figura 21), **DataResponse** (figura 22), **ResultsResponse** (figura 23), **PriceResponse** (figura 24), **CreatorsResponse** (figura 25), **CreatoresItems** (figura 26).

```java
public class ComicsResponse {

    @JsonProperty("data")
    private DataResponse data;

    public ComicsResponse(@JsonProperty("data") DataResponse data) {
        this.data = data;
    }

    public ResultsResponse getResult() {
        return data.getResult();
    }
}
```

Figure 21: ComicResponse

A anotação @JsonProperty é usada para mapear nomes de propriedades com chaves JSON durante a serialização e desserialização. Nessa classe estamos mapeando a propriedade “data” da resposta da API da Marvel.

```java
public class DataResponse {

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

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

    @Deprecated
    public DataResponse() {}

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

}
```

Figure 22: DataResponse

A DataResponse mapea a propriedade “results” da resposta da API. Ele possui uma lista de results do tipo **ResultsResponse**.

```java
public class ResultsResponse {

    @JsonProperty("id")
    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;

    // constructor and getters
```

Figure 23: ResultsResponse

Na classe **ResultsResponse** são mapeadas as propriedades relevantes para a nossa API: comicId, título, isbn, descrição, preço e authores. O atributo prices é uma lista do tipo **PriceResponse** e mapeia o preço do quadrinho, e o atributo creators é do tipo **CreatorsResponse** e mapeia a lista de autores de um determinado quadrinho.

```java
public class PriceResponse {

    private Double price;

    public PriceResponse(@JsonProperty("price") Double price) {
        this.price = price;
    }

    public Double getPrice() {
        return price;
    }
```

Figure 24: PriceResponse

A classe **CreatorsResponse** mapeia a lista de autores do tipo **CreatoresItems**, por meio da @JsonProperty(“items”).

```java
public class CreatorsResponse {;

    private List<CreatoresItems> items;

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

    public List<CreatoresItems> getItems() {
        return items;
    }
```

Figure 25: CreatorsResponse

A classe **CreatoresItems** mapeia o nome do autor, por meio da @JsonProperty(“name”).

```java
public class CreatoresItems {

    private String name;


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

Figure 26: CreatoresItems

Pronto, nossa estrutura de resposta esta finalizada, com as informações que desejamos. Agora, para consumirmos a API com Feign, devemos adicionar duas dependências em nosso POM.XML, figura 27.

- spring-cloud-starter-openfeign
- spring-cloud-dependencies

```xml
        <dependency>
            <groupId>org.springframework.cloud</groupId>
            <artifactId>spring-cloud-starter-openfeign</artifactId>
        </dependency>
    </dependencies>

    <dependencyManagement>
        <dependencies>
            <dependency>
                <groupId>org.springframework.cloud</groupId>
                <artifactId>spring-cloud-config-dependencies</artifactId>
                <version>3.0.1</version>
                <type>pom</type>
                <scope>import</scope>
            </dependency>
```

Figure 27: Adcionando as dependencias do Feign

Com as dependências adicionadas, será necessário mapear, como pode ser visto na figura 28, em nossa classe principal a sequinte anotação @EnableFeignClients, para que o Feign seja habilitado.

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

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

}
```

Figure 28: Habilitando o Feign

Após esta anotação devemos criar nossa interface **MarvelComicsClient** (figura 29), no pacote que nós criamos anteriormente /MarvelApi/client, onde o @FeignClient irá fazer a chamada do serviço.

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

    @GetMapping("/{id}")
    public ResponseEntity<ComicsResponse> findComicsById(
            @PathVariable Integer id,
            @RequestParam(value = "ts") Long ts,
            @RequestParam(value="apikey") String apikey,
            @RequestParam(value="hash") String hash
            );
```

Figure 29: MarvelComicsClient

Nessa interface temos alguns pontos que devemos destacar:

- Temos dentro de @FeignClient o parametro **url** que contém a url da api externa, e **name** para referenciar o bean do client.
- ${marvel.url} é utilizado para uma melhor organização e por ser uma boa pratica de manter as urls das integrações no arquivo de propriedades, e pode ajudar na manutenção caso venham a ser alteradas no futuro.

```
#Marvel Api
marvel.public.key=${MARVEL_PUBLIC_KEY:c83c3d8189a0187ec8869e0ed5305c50}
marvel.private.key=${MARVEL_PRIVATE_KEY:b29405ed18ea29bc9ff0917d759e52dcda506662}
marvel.url=${MARVEL_URL:http://gateway.marvel.com/v1/public}
```

Figure 30: application.properties

Na figura 30, também temos as chaves publica e privadas que são necessárias para realizar a chamada na Api da Marvel.

No método findComicsById() na Interface **MarvelComicsClient** temos o @RequestParam que usamos para mapear juntamente com a URL da Marvel {URL + ts + apiKey + hash/{id}}.

- Ts: é um carimbo de data/hora
- apiKey (publicKey e privateKey): são únicas, adquiridas ao ciar o cadastro na plataforma da Marvel.
- Hash: é um MD5 gerado dos seguintes atributos (ts + privateKey+publicKey).

Próximo passo, criar nosso **MarvelComicsService.**

```java
@Service
public class MarvelComicsService {

    @Autowired
    private MarvelComicsClient marvelClient;

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

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

    public ResultsResponse findComicsById(Integer id) {
        Long currentDate = new Date().getTime();
        ComicsResponse response =
                marvelClient.findComicsById(
                        id,
                        currentDate,publicKey ,
                        hash(currentDate)).getBody();
        return response.getResult();
    }

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

Figure 31: MarvelComicsService

Na classe **MarvelComicsService**, figura 31, injetamos a dependência da classe **MarvelComicsClient** para que possamos fazer as requisições na Api da Marvel. A anotação @Value pega as informação na **applications.properties**. O método hash gera o hash que é necessário para realizar a requisição na API externa. Recebi como parametro a data atual. E a utiliza para gerar o hash MD5 juntamento com a privateKey e publicKey.

Pronto, agora toda a estrutura necessária para consumir uma api externa está pronta. A estrutura de diretorios no nosso projeto para lidar com a Api da Marvel deve estar como na figura 32.

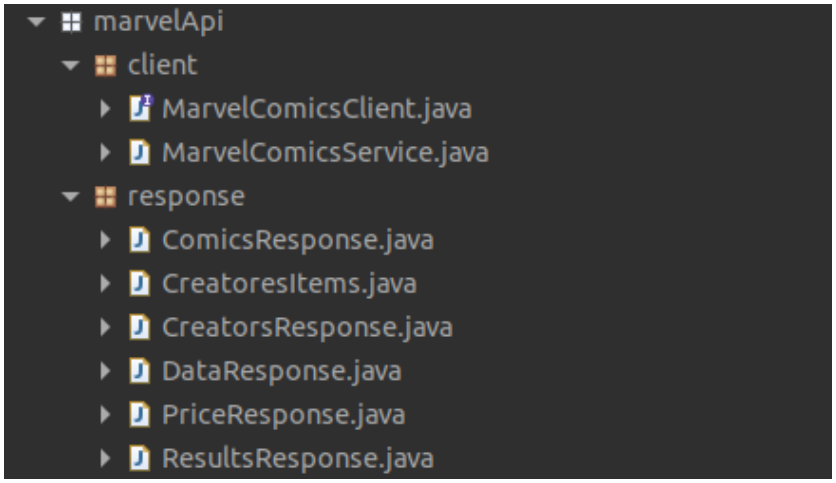

Figure 32: Estrutura da marvelApi

Agora que toda a estrutara para receber os dados da Api externa está pronta, vamos fazer o **ComicsService** (figura 33) e o **ComicController.**

```
@Service
public class ComicsService {

    @Autowired
    private ComicRepository comicRepository;

    @Autowired
    private UserRepository userRepository;
```

Figure 33: ComicsService

Na classe **ComicsService** vamos injetar as dependências do **ComicRepository** e da **UserRepository**. Proximo passo é criar os métodos para retornar uma lista de quadrinhos para um determinado usuário e o método para inserir o quadrinho que veio da Api da Marvel e associa-lo a um usuário.

```
@Transactional(readOnly = true)
public List<ComicDTO> findByUser(Long userId) {
    Optional<User> obj = userRepository.findById(userId);
    User user = obj.orElseThrow(()-> new
            ResourceNotFoundException("Usuário não encontrado: {id = "+ userId+"}"));
    List<Comic> list = comicRepository.findComicsByUser(user);
    return list.stream().map(x -> new ComicDTO(x)).collect(Collectors.toList());
}

public ComicDTO insertComic(Long userId, ResultsResponse response) {
    Comic entity = new Comic();

    Optional<User> obj = userRepository.findById(userId);
    User user = obj.orElseThrow(()->
        new ResourceNotFoundException("Usuário não encontrado: {id = "+ userId+"}"));
    entity = convertResponseToEntity(user, response);
    for(CreatoresItems item: response.getCreators().getItems()) {
        entity.getAuthors().add(new Author(item.getName()));
    }
    entity =  comicRepository.save(entity);
    return new ComicDTO(entity);
}
```

Figure 34: ComicsSevice - métodos

O primeiro método findByUser(), visto na figura 34, recebe como argumento o id do Usuário. Primeiro vamos verificar se o usuário existe, por meio do finById do **userRepository**. Em seguida, verificamos se o usuário existir, atribuimos a variavel user do tipo **User**, caso contrario lançamos uma exceção (vamos falar sobre as validações e exceções no decorrer do texto). O método findComicsByUser() do **ComicRepository** pega todos os quadrinhos de um determinado usuário. E Por sim, retornamos a lista.

O segundo método insertComic() recebe dois argumentos, o id do Usuário e o ResultResponse que é a resposta da nossa requisição a Api da Marvel. Da mesmo forma do método anterior, vamos vefiricar se o usuário existe, senão existir teremos que lançar uma exceção. Em seguida, vamos converter o ResultsResponse para a entidade para que a gente possa salvar no banco de dados. Por meio dos métodos disponibilzados pela List, vamos pegar os autores do ResultsResponse e adciona-los na entidade por meio do add(). E por fim, salvamos a entidade no banco de dados e retornamos o **ComicDTO**.

```
private Comic convertResponseToEntity(User user,ResultsResponse response) {

    return new Comic(
            response.getComicId(),
            response.getTitle(),
            response.getPrices(),
            response.getIsbn(),
            response.getDescription(),
            false,
            user
            );
}
```

Figure 35: ComicsService - método convertResponseToEntity()

Ainda na classe da **ComicsService**, temos o método convertResponseToEntity(), que recebe dois argumentos, o usuário para que possamos vincular o quadrinho a ele, e o ResultsResponse para pegar as informações do quadrinho que veio da Api da Marvel. O método retorna a entidade Comic com os dados do usuário e do quadrinho, exceto as informações dos autores.

Agora vamos criar a classe **ComicController**, como visto na figura 36. A partir dos requisitos apresentados, iremos criar os seguintes entpoints:

- @GetMapping("/comics/user/{id}"): retornar a lista de quadrinhos para um determinado usuário cadastrado;
- @PostMapping("/comics/{comicId}/{userId}"): cadastrar um Comic vinculado a um usuário.

```
@RestController
@RequestMapping(value = "/comics")
public class ComicController {

    @Autowired
    private ComicsService comicService;

    @Autowired
    private MarvelComicsService marvelService;

    @GetMapping(value = "/user/{id}")
    public ResponseEntity<List<ComicDTO>> findByUser(@PathVariable Long id){
        List<ComicDTO> list = comicService.findByUser(id);
        return ResponseEntity.ok().body(list);
    }

    @PostMapping(value="/{id}/{userId}")
    public ResponseEntity<ComicDTO> insertComicWithUser(@PathVariable Integer id,
            @PathVariable Long userId) {
        ResultsResponse response = marvelService.findComicsById(id);
        ComicDTO dto = comicService.insertComic(userId,response);
        URI uri = ServletUriComponentsBuilder.fromCurrentRequest().path("/{id}")
                .buildAndExpand(dto.getId()).toUri();
        return ResponseEntity.created(uri).body(dto);
    }
```

Figure 36: ComicController

Como de praste, vamos injetar as dependências necessárias, **ComicsService** e **MarvelComicsService**. Nesta classe temos dois métodos:

- findByUser: método GET que retorna a lista de quadrinhos de um determinado usuário e retorna o HTTP Status Code 200, caso a consulta estaja correta;
- insertComicWithUser: método POST que consulta a Api da Marvel para pegar o quadrinho por meio do comidId. E passa o retorna da consula da Api para insertComic do ComicService juntamente do id do usuário.

3.5. Testando a API de Quadrinhos (Comics)

Nós podemos testar nossa API de Quadrinhos (Comics) com o Postman.

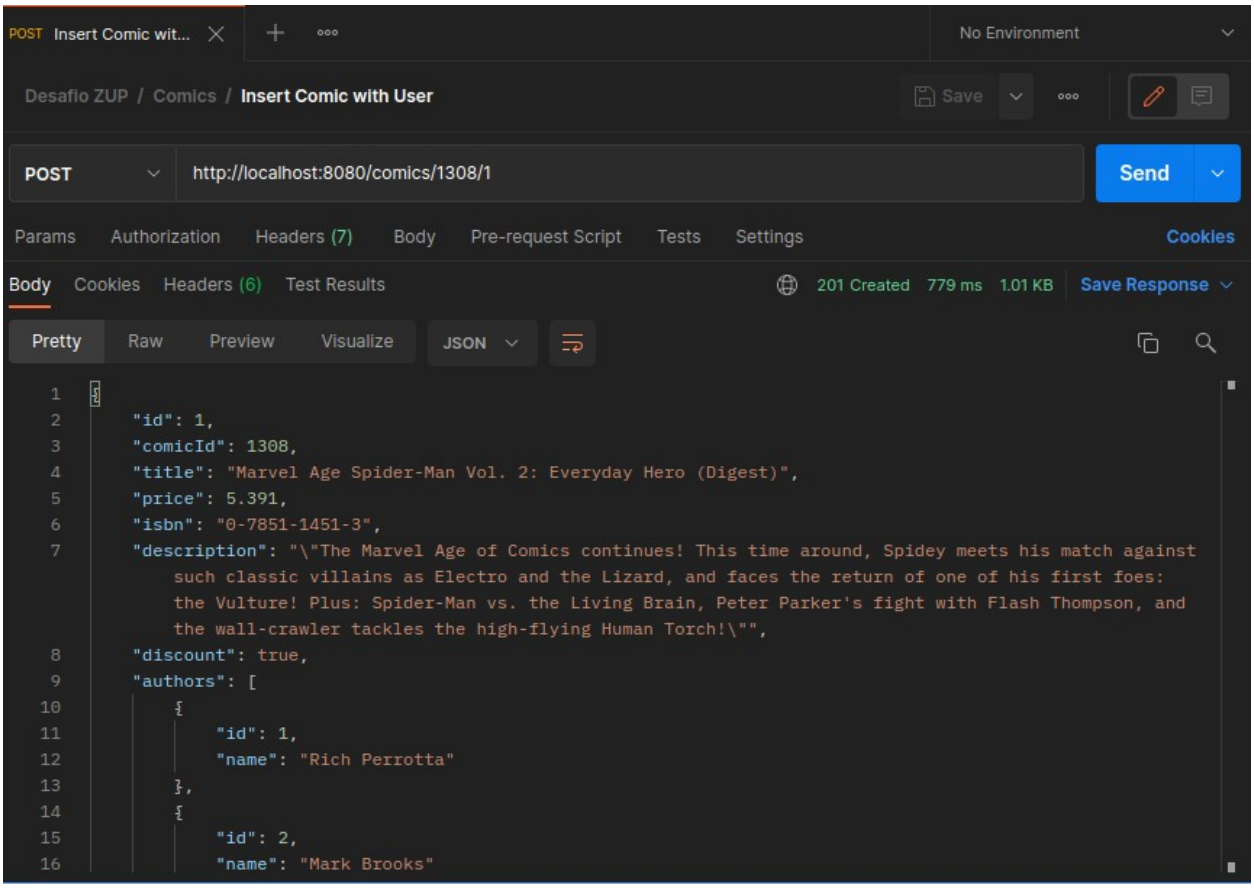

Figure 37: Inserir um Comic

Como podemos ver na figura 37, o nosso endpoint para inserir o Comic associado com o Usuário está funcionando corretamente e retornando o HTTP Status code 201.

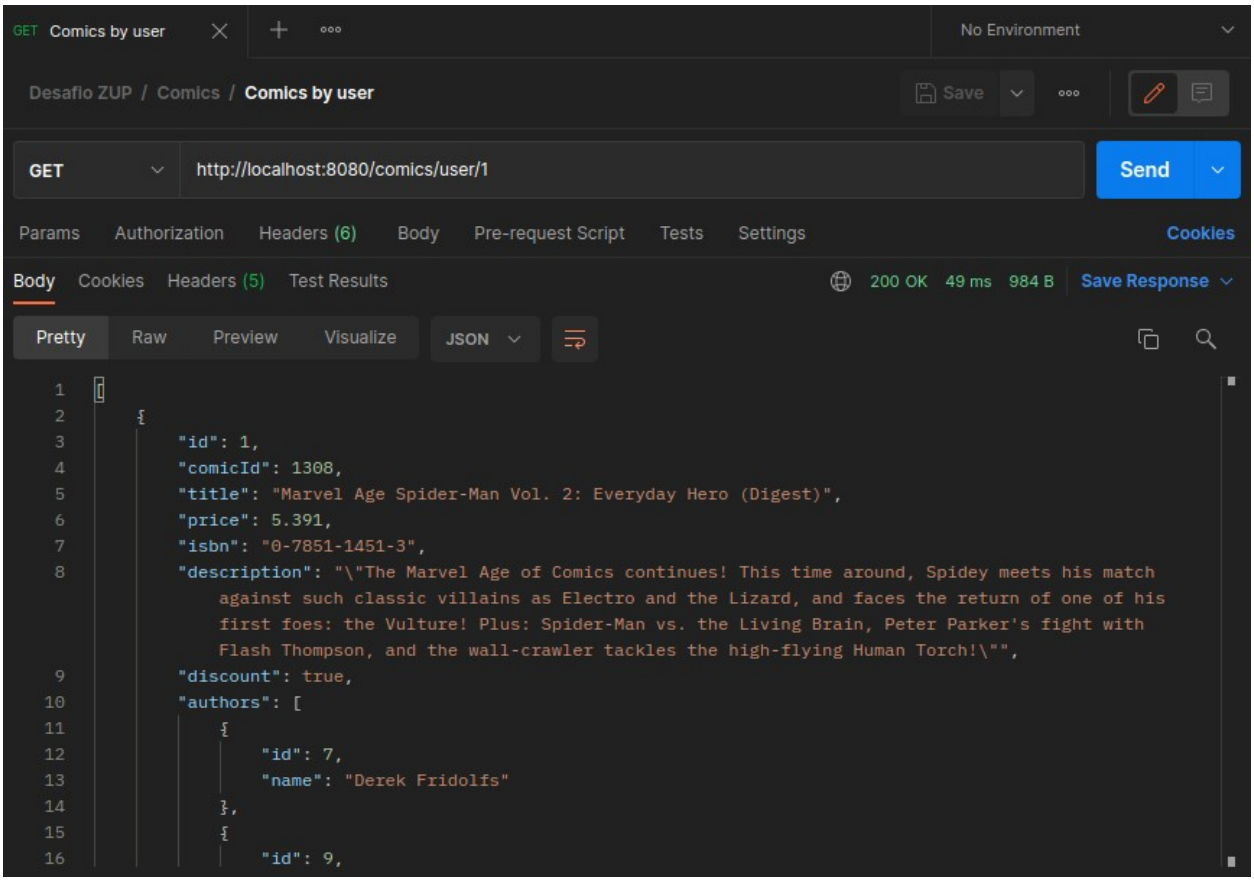

Figure 38: Listar Comics para um determinado User

O endpoint para listar os quadrinhos de um determinado usuário está funcionado e retornando o HTTP Status code 200, como pode ser visto na figura 38.

4.1. Tratamento de Exceções e Validações

Vamos criar um pacote chamado **exceptions** no diretorio raiz da nossa aplicação, como na figura 39:

Figure 39: Estrutura do projeto

Dentro do pacote exceptions, iremos criar quatro classes: **StandarError** (figura 40), **ResourceExceptionHandler** (figura 41), **FieldMessage** (figura 42), **ValidationError** (figura 43). A classe **StandarError** possui as informações necessárias para informar os erros que possam ocorrer.

```
public class StandardError implements Serializable{

    private static final long serialVersionUID = 1L;

    private Instant timestamp;
    private Integer status;
    private String error;
    private String message;
    private String path;


    public StandardError() {}

    //Getters and Setters
```

Figure 40: StandarError

Temos o timestamp para informar o momento que houve o erro, status para o código do erro, error para o informar qual erro aconteceu, message para passarmos uma mensagem personalizada e por fim, o path que nos diz o caminho para o recurso onde ocorreu o erro.

```
public class FieldMessage implements Serializable{

    private static final long serialVersionUID = 1L;

    private String fieldName;
    private String message;

    public FieldMessage() {}

    public FieldMessage(String fieldName, String message) {
        this.fieldName = fieldName;
        this.message = message;
    }
```

Figure 41: FieldMessage

A classe **FieldMessage** tem dois atributos: fieldName para informar o campo que ocorreu o erro, e message para passarmos uma mensagem personalizada.

```
public class ValidationError extends StandardError{
    private static final long serialVersionUID = 1L;
    private List<FieldMessage> errors = new ArrayList<>();

    public List<FieldMessage> getErrors() {
        return errors;
    }
    public void addError(String fieldName, String message) {
        errors.add(new FieldMessage(fieldName, message));

    }

}
```

Figure 42: ValidationError

A classe **ValidationError** herda de **StandarError**. E possui uma lista de erros do tipo FieldMessage, para que possamo retornar uma lista de erros de uma vez.

```
@ControllerAdvice
public class ResourceExceptionHandler {
```

Figure 43: ResourceExceptionHandler

Agora vamos criar o **ResourceExceptionHandler** e anota-la com @ControllerAdvice. @ControllerAdvice é usado para tratamento global de erros no aplicativo Spring. Ele também tem controle total sobre o corpo da resposta e o código de status.

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

@ExceptionHandler(MethodArgumentNotValidException.class)
public ResponseEntity<ValidationError> Validation(MethodArgumentNotValidException e,
            HttpServletRequest request){
    HttpStatus status = HttpStatus.BAD_REQUEST;
    ValidationError err = new ValidationError();
    err.setTimestamp(Instant.now());
    err.setStatus(status.value());
    err.setError("Excessão na base de dados");
    err.setMessage(e.getMessage());
    err.setPath(request.getRequestURI());
    for(FieldError f: e.getBindingResult().getFieldErrors()) {
        err.addError(f.getField(), f.getDefaultMessage());
    }
    return ResponseEntity.status(status).body(err);
}
```

Figure 44: ResourceExceptionHandler - métodos

Temos dois método: entityNotFound e Validation anotados com @ExceptionHandler. A anotação @ExceptionHandler é usada para manipular exceções em classes específicas e/ou métodos expecíficos.

O primeiro método é utilizado para tratar as exceções do tipo **ResourceNotFoundException** (falaremos sobre esta classe no decorrer do texto). Nesse método nos inserimos as informações do erro, por exemplo: o momento que o erro ocorreu, o HTTP Status Code NOT_FOUND, a mensagem do erro, uma mensagem mais detalhada do erro e o endereço do recurso que ocorreu o erro. E, por fim, retorna um **StandarError**.

O segundo método é utilizado para tratar as exceções de validação. Lembra das anotações que inserimos nas Entidades e nos Dtos, mas não apenas para essas validações. Vamos fazer mais para frente outra forma de como validar as informações.

Continuando, o método assim como o primeiro coleta as informações do erro, adcionando o campo que houve o erro por meio da classe FieldError. Por fim, retorna o error com o HTTP Status code BAD_REQUEST.

4.2. Testando as validações na inserção de Usuário com o Postman.

Vamos enviar uma requisição post sem informar o nome e o e-mail, campos que são obrigatórios, figura 45.

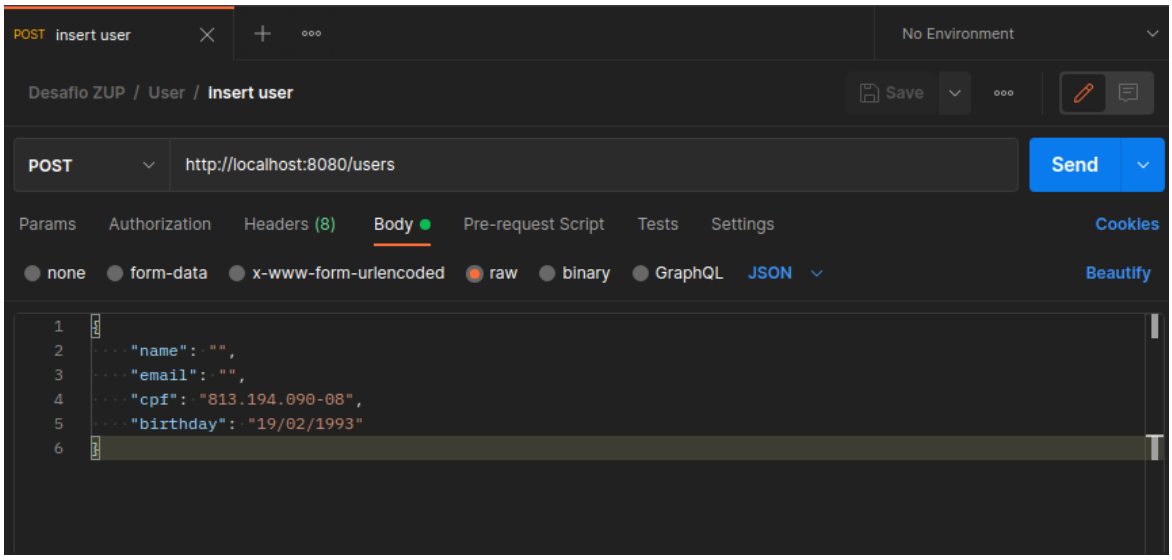

Figure 45: Teste de validação

Mensagem de returno na figura 46.


```
{
    "timestamp": "2021-12-28T17:10:56.783518782Z",
    "status": 400,
    "error": "Exceção na base de dados",
    "message": "Validation failed for argument [0] in public org.springframework.http.ResponseEntity<
    br.com.zup.desafio.comics.dtos.UserDTO> br.com.zup.desafio.comics.controllers.UserController.insert(
    br.com.zup.desafio.comics.dtos.UserInsertDTO) with 2 errors: [Field error in object 'userInsertDTO' on
    field 'email': rejected value []; codes [
    NotBlank.userInsertDTO.email,NotBlank.email,NotBlank.java.lang.String,NotBlank]; arguments [
    org.springframework.context.support.DefaultMessageSourceResolvable: codes [userInsertDTO.email,email];
    arguments []; default message [email]]; default message [O E-mail é obrigatório!!!]] [Field error in
    object 'userInsertDTO' on field 'name': rejected value []; codes [
    NotBlank.userInsertDTO.name,NotBlank.name,NotBlank.java.lang.String,NotBlank]; arguments [
    org.springframework.context.support.DefaultMessageSourceResolvable: codes [userInsertDTO.name,name];
    arguments []; default message [name]]; default message [O Nome é obrigatório!!!]] ",
    "path": "/users",
    "errors": [
        {
            "fieldName": "email",
            "message": "O E-mail é obrigatório!!!"
        },
        {
            "fieldName": "name",
            "message": "O Nome é obrigatório!!!"
        }
    ]
}
```


Figure 46: Resposta do teste

Lembram-se da classe **ResourceNotFoundException**, vamos cria-la agora. Crie um pacote exceptions dentro do pacote de services.

```java
public class ResourceNotFoundException extends RuntimeException{

    private static final long serialVersionUID = 1L;

    public ResourceNotFoundException(String msg) {
        super(msg);
    }


}
```

Figure 47: ResourceNotFoundException

Essa classe ficará responsável por gerar as exceções quando buscarmos por um recurso que não existe no banco de dados.

4.3. Testando inserir um Quadrinho associado a um Usuário não cadastrado.

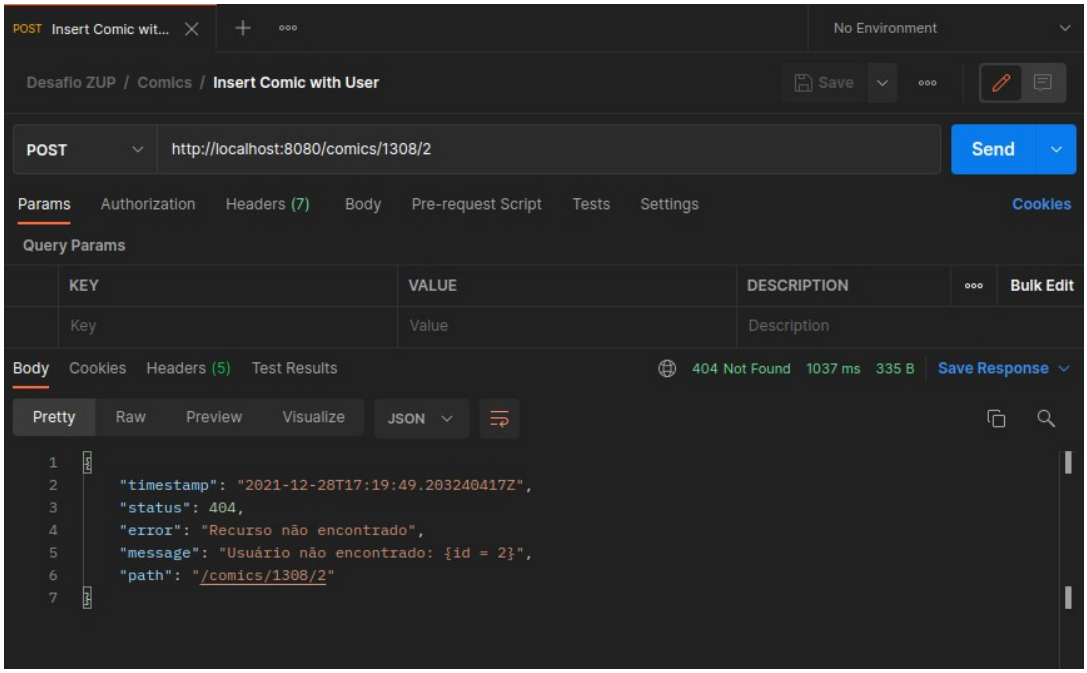

Figure 48: Teste de inserção de Comics sem User cadastrado

Como podemos ver na figura 48, foi gerada uma exceção com HTTP Status Code 404 Not Found. Temos também o momento, o HTTP Status, o tipo, a mensangem detalhada do erro e, por fim o endereço do recurso onde houve o erro.

4.4. Criando Validações para tratar atributos únicos das entidades.

No pacote service vamos criar mais um pacote chamado validations. Nesse pacote vamos criar duas classes: **UserInsertValidator** (figura 49) e um classe do tipo Annotation **UserInsertValid** (figura 51).

```java
public class UserInsertValidator implements ConstraintValidator<UserInsertValid, UserDTO>{

    @Autowired
    private UserRepository userRepository;

    @Override
    public void initialize(UserInsertValid ann) {
    }
```

Figure 49: UserInsertValidator

Na classe **UserInsertValidator** precisaremos injetar a dependência de UserRepository. A classe **UserInsertValidator** implementa a interface ConstraintValidator e também deve implementar o método isValid, como pode ser visto na figura 50. E é neste método que definimos nossas regras de validação.

```java
@Override
public boolean isValid(UserDTO dto, ConstraintValidatorContext context) {

    List<FieldMessage> list = new ArrayList<>();

    User user = userRepository.findByEmail(dto.getEmail());
    if(user != null) {
        list.add(new FieldMessage("email", "Email já existe"));
    }
    User user2 = userRepository.findByCpf(dto.getCpf());
    if(user2 != null) {
        list.add(new FieldMessage("cpf", "CPF já existe"));
    }

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

Figure 50: UserInsertValidator - método isValid

Nesse método basicamente verificamos se há algum algum usuário com o e-mail e CPF que passamos por paramento. Caso haja, retorna um erro de validação com o campo e um mensagem personalizado.

```java
@Constraint(validatedBy = UserInsertValidator.class)
@Target({ ElementType.TYPE })
@Retention(RetentionPolicy.RUNTIME)
public @interface UserInsertValid {

    String message() default "Erro de Validação";

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

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

}
```

Figure 51: UserInsertValid

A classe **UserInsertValid** possui um código chamado de *boilerplate*, ou seja, codicos incluídos com pouca ou nenhuma alteração. Para esta classe basta alterar o @Constraint e colocar o nome da classe de validação, que no nosso caso é a **UserInsertValidator**.

```
@UserInsertValid
public class UserInsertDTO extends UserDTO{

    private static final long serialVersionUID = 1L;

}
```

Figure 52: UserInsertValid - aplicando a validação

Para que essa validação funcione, basta anotar nossa classe com a @UserInsertValid, como pode ser visto na figura 52.

4.5. Testando Validação dos atributos únicos de Usuário: Tentando inserir outro usuário com o mesmo e-mail e CPF – figura 53, 54.

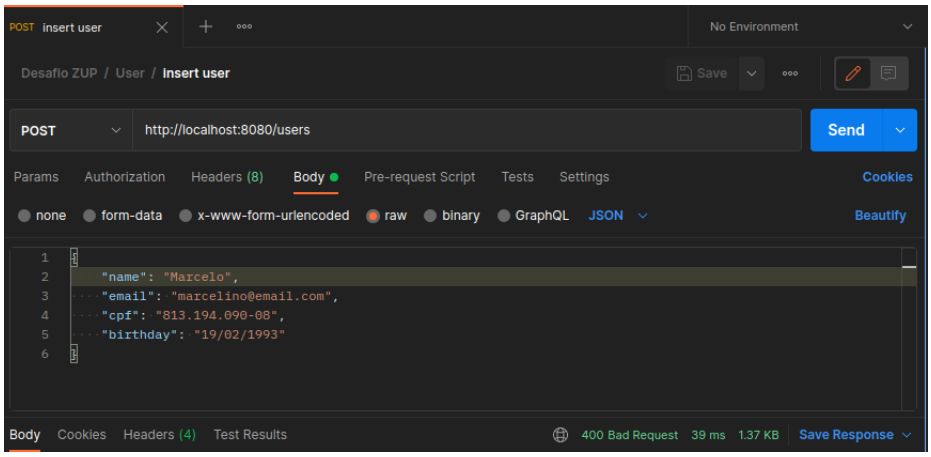

Figure 53: Testando inserção de user com mesmo email e cpf


```
{
    "timestamp": "2021-12-28T17:47:26.664477520Z",
    "status": 400,
    "error": "Exceção na base de dados",
    "message": "Validation failed for argument [0] in public org.springframework.http.ResponseEntity<
    br.com.zup.desafio.comics.dtos.UserDTO> br.com.zup.desafio.comics.controllers.UserController.insert(
    br.com.zup.desafio.comics.dtos.UserInsertDTO) with 2 errors: [Field error in object 'userInsertDTO' on
    field 'email': rejected value [marcelino@email.com]; codes [UserInsertValid.userInsertDTO.email,UserInser
    tValid.email,UserInsertValid.java.lang.String,UserInsertValid]; arguments [
    org.springframework.context.support.DefaultMessageSourceResolvable: codes [userInsertDTO.email,email];
    arguments []; default message [email]]; default message [Email já existe]] [Field error in object
    'userInsertDTO' on field 'cpf': rejected value [813.194.090-08]; codes [
    UserInsertValid.userInsertDTO.cpf,UserInsertValid.cpf,UserInsertValid.java.lang.String,UserInsertValid];
    arguments [org.springframework.context.support.DefaultMessageSourceResolvable: codes [
    userInsertDTO.cpf,cpf]; arguments []; default message [cpf]]; default message [CPF já existe]] ",
    "path": "/users",
    "errors": [
        {
            "fieldName": "email",
            "message": "Email já existe"
        },
        {
            "fieldName": "cpf",
            "message": "CPF já existe"
        }
    ]
}
```


Figure 54: Resposta de teste de validação

5. Aplicando Desconto no Comic

Vamos agora aplicar o desconto na quadrinho de acordo com as regras definidas no escopo do projeto. Teremos que alterar algumas coisas na Entidade **Comic**.

```
private boolean apllyDiscount(String isbn) {
    LocalDate localDate = LocalDate.now();
    DayOfWeek dayOfWeek = DayOfWeek.from(localDate);
    int digito = Integer.valueOf(isbn.substring(isbn.length() - 1));

    if (dayOfWeek.name().toString().equals("MONDAY")
                && (digito == 0 || digito == 1)) {
        return true;
    } else if (dayOfWeek.name().toString().equals("TUESDAY")
                && (digito == 2 || digito == 3)) {
        return true;
    } else if (dayOfWeek.name().toString().equals("WEDNESDAY")
                && (digito == 4 || digito == 5)) {
        return true;
    } else if (dayOfWeek.name().toString().equals("THURSDAY")
                && (digito == 6 || digito == 7)) {
        return true;
    } else if (dayOfWeek.name().toString().equals("FRIDAY")
                && (digito == 8 || digito == 9)) {
        return true;
    }
    return false;
}
```

Figure 55: Método de desconto

Vamos adcionar esse método apllyDiscount(), como pode ser visto na figura 55, para verificar o dia da semana e se valor do ultimo digito do atributo ISBN. E retorna true caso se adeque as regras de desconto estabelecida. Também precisaremos alterar o método getPrice.

```
public Double getPrice() {
    if (apllyDiscount(getIsbn())) {
        setDiscount(true);
        return price - (price * 0.1);
    }
    setDiscount(false);
    return price;
}
```

Figure 56: Método getPrice()

O método getPrice(), figura 56, agora vai retornar o valor com desconto caso o método applyDiscount() retorne true e altera o valor do atributo discount para true. Caso applyDiscount retorne false, altera o valor do altributo discount para false e retorna o preço sem desconto.

5.1. Data de Nascimento no formato dd/MM/yyyy

Vamos agora formatar a data de nascimento para que ele retorne no formato dd/MM/yyyy. Para isso, vamos alterar o getBirthday da entidade User e do dto UserDTO adcionando a anotação `@JsonFormat(shape=JsonFormat.Shape.STRING, pattern="dd/MM/yyyy")`. E desse modo a data de nascimento vai retornar no formato dd/MM/yyyy.