Criando uma API Rest com Java e Spring

Por Fernanda Moran Menezes Pereira

femmpereira@gmail.com

O termo REST vem do inglês *Representational State Transfer* e é traduzido para o português como Transferência Representacional de Estado. Trata-se de um estilo de arquitetura de software que define a forma como os componentes de um *web service* devem se comportar.

Uma API (*Application Programming Interface*) que possibilita que duas aplicações se comuniquem sem que uma conheça os detalhes de implementação da outra, é definida como RESTful se for desenvolvida seguindo todos os conceitos da arquitetura REST.

O funcionamento da API em um *web service* segue um padrão cliente-servidor, em que um cliente envia uma requisição em HTTP e o servidor devolve uma resposta ao cliente. A resposta do servidor pode ser apresentada em diferentes formatos como XML e JSON, por exemplo.

Para o desenvolvimento dessa API um conjunto de ferramentas podem ser usadas, tendo como base a linguagem Java com o Spring Framework, o qual oferece módulos como o Spring Boot que possibilita o desenvolvimento com mínimas configurações. A ferramenta Apache Maven também é importante para gerenciar as dependências da aplicação.

O primeiro passo para projetar nosso *web service* é decidir o que queremos que o serviço realize. Nesse caso, é preciso criar um cadastro de clientes e um cadastro de quadrinhos, com possibilidade de consulta de clientes cadastrados e lista de quadrinhos de um cliente. Assim, a API deve ser capaz de criar, atualizar, deletar e listar tanto os clientes quanto os quadrinhos.

Portanto, um bom ponto de partida é a definição de um cliente e de um quadrinho, o que pode ser feito usando objetos JSON (*Javascript Object Notation*). Assim, no padrão cliente-servidor serão usados objetos JSON.

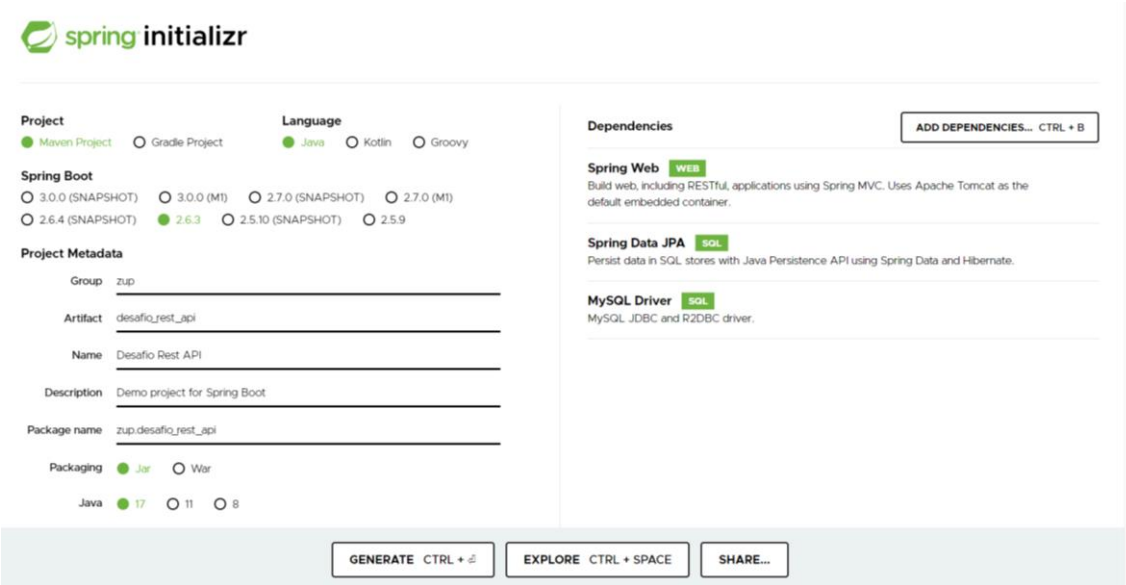
Definidos todos os *endpoints* da API, os quais farão a conexão entre o serviço e o cliente, podemos partir para a criação do projeto! Nesse caso, vamos criar um projeto Spring Boot no Spring Initializr. Criaremos um Maven Project em linguagem Java (versão 17), indicando como dependências, as seguintes ferramentas:

* Spring Web (Spring MVC): que ajuda no desenvolvimento de aplicações web no padrão MVC;

*Spring Data JPA: permite fazer o acesso aos bancos de dados MySQL;

*MySQL Driver: permite a conexão com o banco de dados MySQL.

A tela do *Spring Initialzr* aparece conforme figura a seguir:

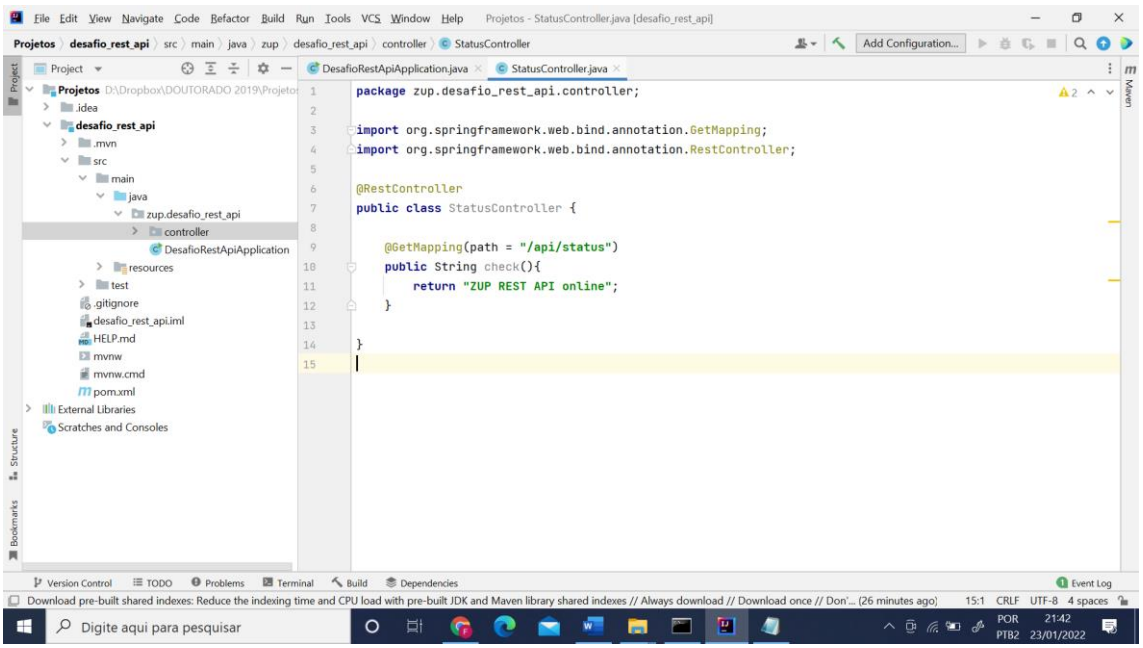


Ao clicar em GENERATE é feito download de um arquivo zip que contém os arquivos da aplicação com as configurações realizadas.

Os arquivos criados pelo Spring Boot representam a base do nosso projeto, o qual será importado para o IntelliJ para iniciarmos a implementação das funcionalidades desejadas para nosso sistema. Obviamente, qualquer outra IDE poderia ser usada nessa tarefa, mas o IntelliJ fornece recursos que facilitam a codificação e maximizam a produtividade no desenvolvimento.

Uma vez aberto no IntelliJ um passo importante e informar à IDE que se trata de um projeto externo em Java. Para tal, escolha no menu a opção ‘File >> New >> Project From Existing Sources’.

Antes de iniciarmos a criação e conexão com o banco de dados e a criação das nossas classes de usuários e quadrinhos, convém definir uma classe controladora que possibilite verificar se nossa aplicação está ou não online. Para tal, vamos criar uma pacote (Package) chamado ‘controller’. Dentro do novo pacote vamos incluir uma classe para controlar o status da aplicação, e que terá um método para retornar o status da aplicação quando uma determinada requisição for realizada. É importante definir a notação ‘@RestController’ e ‘@GetMapping’ para indicar ´para o Java que esse método deverá ser executado em caso de requisições. A codificação da classe é ilustrada na figura a seguir:



Assim, sempre que o caminho definido na variável path for requisitado junto ao endereço da aplicação, a string “ZUP REST API online” será retornada. Esse resultado é ilustrado na figura a seguir:

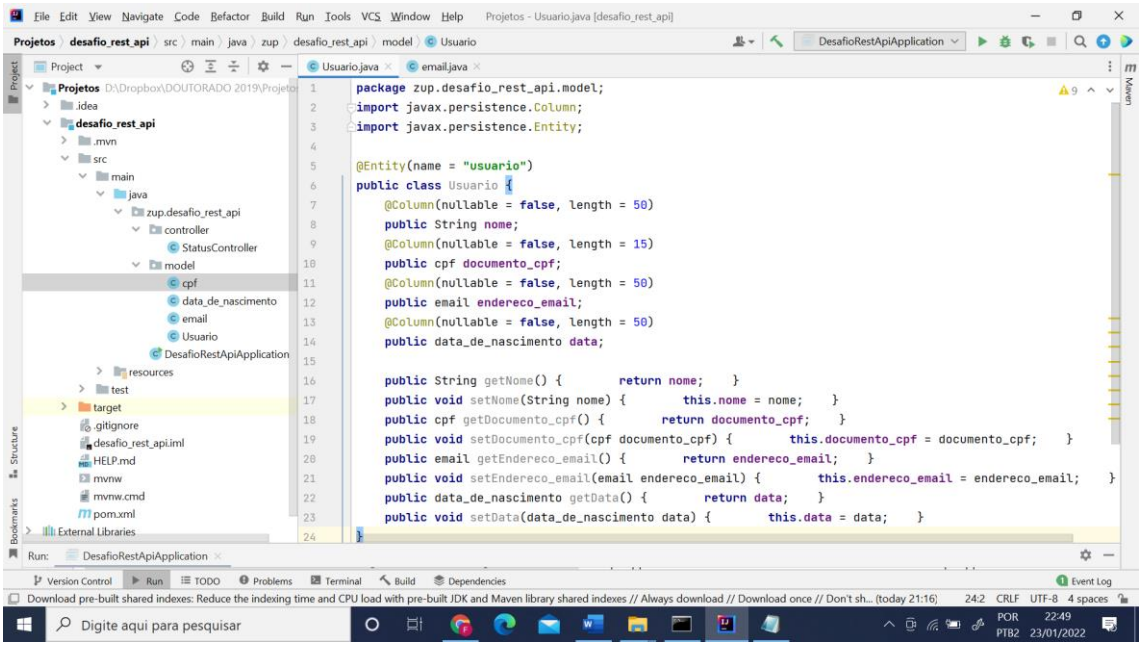
Esse simples exemplo já é suficiente para ilustrar o padrão cliente-servidor mencionado no início deste artigo. Uma consulta foi feita pelo cliente ao digitar o endereço da aplicação e a aplicação, por sua vez, retornou um resultado indicando que está online.

Configurando acesso ao banco de dados e criando classes

Para garantir a persistência dos dados de usuários e quadrinhos é necessário configurar acesso ao banco de dados. Nessa etapa devemos prover as operações básicas (CRUD) de manipulação das informações no banco. Ou seja, a aplicação deve ser capaz de criar (Create), ler (Read), atualizar(Update) e remover (Delete) um usuário e um quadrinho da nossa base de dados. Obviamente, é necessário definir as classes para essas duas entidades da nossa aplicação.

Iniciando com a criação da classe para o usuário, vamos seguir o mesmo procedimento adotado na criação do controlador. Assim, criamos um pacote e uma nova classe dentro do pacote. Definimos a notação '@Entity' para indicar que essa classe é uma entidade de banco de dados. A classe usuário precisa ter todos os atributos definidos na descrição da API, a saber: nome, e-mail, CPF e data de nascimento. Enquanto o primeiro atributo (nome) pode ser definido usando uma string, os demais atributos da classe 'usuario' podem ser objetos definidos também por meio de classes. Assim, além da classe 'usuario', devemos criar outras três classes para os atributos e-mail, CPF e data de nascimento. Essas três classes devem possuir um construtor que recebe as dados informados e métodos para validar e exibir esses dados.

Como a classe 'usuario' é do tipo 'Entity' seus membros representam campos nas tabelas do banco de dados. É importante que isso seja definido por meio da notação '@Column', indicando que os campos são obrigatórios (nullable = false). Uma ilustração da classe 'usuario' e é apresentada na figura a seguir. Vale observar que as classes cpf, email e data_de_nascimento foram criadas dentro do mesmo pacote 'model'.



Definida a classe de usuários devemos criar os métodos para as operações de CRUD dessa entidade no banco de dados, o que é feito por meio de uma classe controladora. Esses métodos são criados dentro de uma classe controladora da entidade chamada, por exemplo, ‘UsuarioController’. Um exemplo dessa classe UsuarioController com métodos para consultar e salvar um usuário no banco de dados pode ser visto na Figura a seguir:

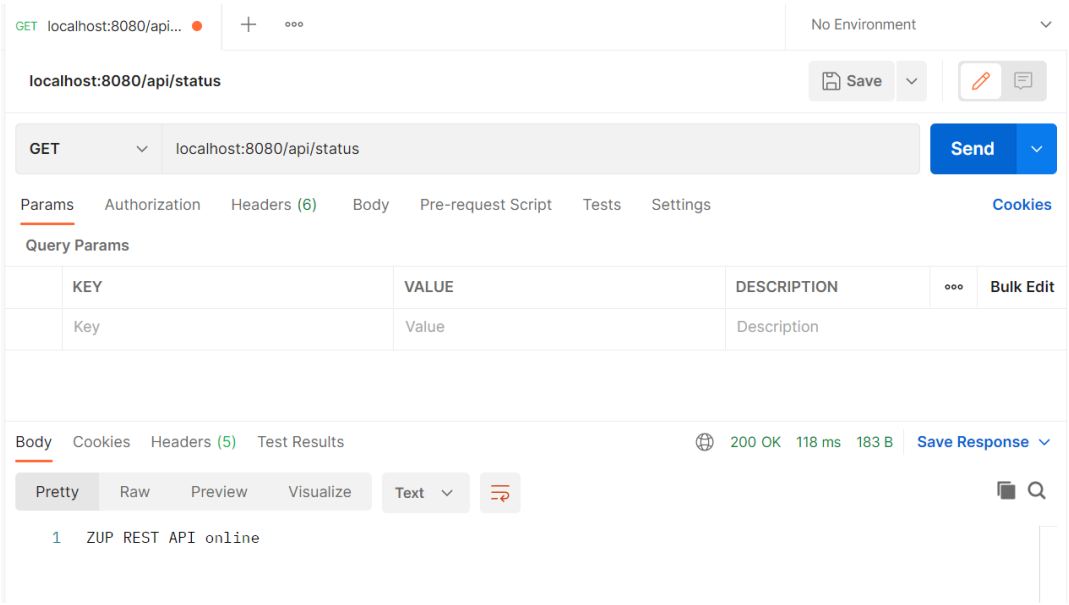
```java
package zup.desafio_rest_api.controller;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.ResponseEntity;
import org.springframework.web.bind.annotation.*;
import zup.desafio_rest_api.model.Usuario;
import zup.desafio_rest_api.repository.UsuarioRepository;

@RestController
public class UsuarioController {

    @Autowired
    private UsuarioRepository repository;

    @GetMapping(path = "/api/usuario/{codigo}")
    public ResponseEntity consultar(@PathVariable("codigo") Integer
codigo){
        return repository.findById(codigo)
                .map(record -> ResponseEntity.ok().body(record))
                .orElse(ResponseEntity.notFound().build());
    }

    @PostMapping(path = "/api/usuario/salvar")
    public Usuario salvar(@RequestBody Usuario usuario){
        return repository.save(usuario);
    }
}
```

A figura ilustra a além dos métodos inserir e salvar, uma interface repository que possibilita a realização das operações de CRUD no banco de dados. A interface repository possui diversos métodos pré-definidos para manipulação do banco. No método consultar, por exemplo, usamos o método ‘FindById’ para um exemplo de consulta a um cliente do banco. Essa consulta também pode ser realizada utilizando o CPF como chave primária. Se o resultado da busca é positivo, então o método retorna o corpo do registro encontrado na base (por exemplo, em formato JSON).

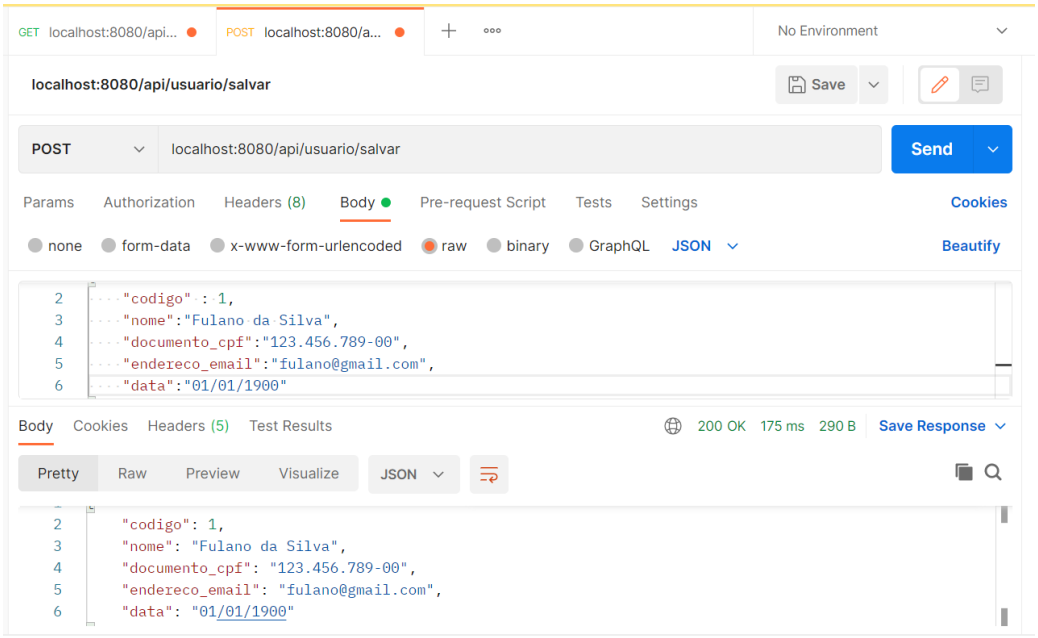
O método salvar, por sua vez, também é oriundo da interface repository (repository.save). O método da classe retorna o Usuário inserido no banco e recebe como parâmetros, obrigatoriamente, as informações do usuário (@RequestBody).

Os demais métodos da classe de Usuários e da classe para os quadrinhos, podemos verificar o funcionamento da API com o uso da ferramenta PostMan. Por meio da ferramenta podemos realizar consultar e inserir usuários em nosso banco usando objetos JSON, por exemplo. As próximas três figuras a

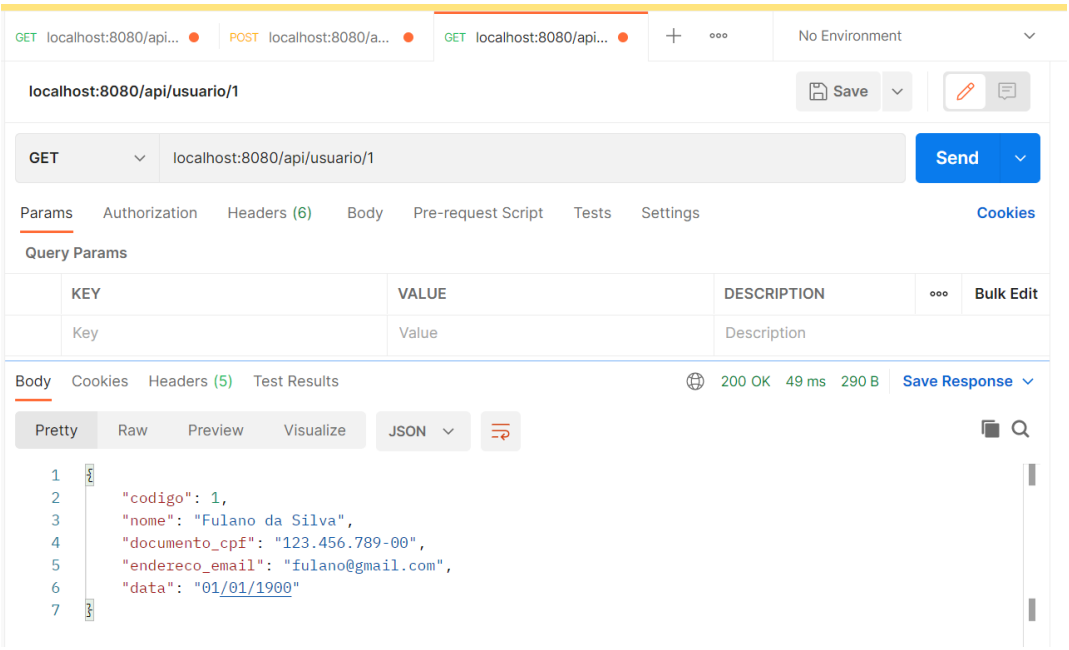
seguir ilustram essa verificação. Na primeira figura a seguir é possível observar o resultado de uma requisição para verificar se a aplicação está online.



Como o resultado da verificação mostra que a aplicação está online, podemos realizar a inserção de um novo usuário. Trata-se de uma operação de POST, conforme figura a seguir. Vale lembrar que o método ‘salvar’ requer um corpo com as informações do usuário a ser inserido, o que pode ser feito por meio de uma JSON.



Por fim, até este momento do desenvolvimento, é possível fazer uma consulta ao banco buscando um usuário por seu Id, conforme figura.



Para continuar nosso projeto precisamos agora criar uma classe para a entidade quadrinhos, a qual também deverá ter acesso ao banco de dados para armazenamento das informações. Nesse caso, porém, as informações devem ser obtidas a partir da API da Marvel.

Vamos lembrar que assim como qualquer API, a API da Marvel opera enviando respostas às requisições dos usuários. As requisições são feitas passando alguns parâmetros para a API como, por exemplo, uma chave pública criada no momento do cadastro na Marvel e um valor de hash criado por um algoritmo de sintetização de mensagem md5, um timestamp, e as chaves públicas e privadas. As requisições têm a seguinte aparência, na qual os valore da apikey e do hash foram substituídos por ‘X’.

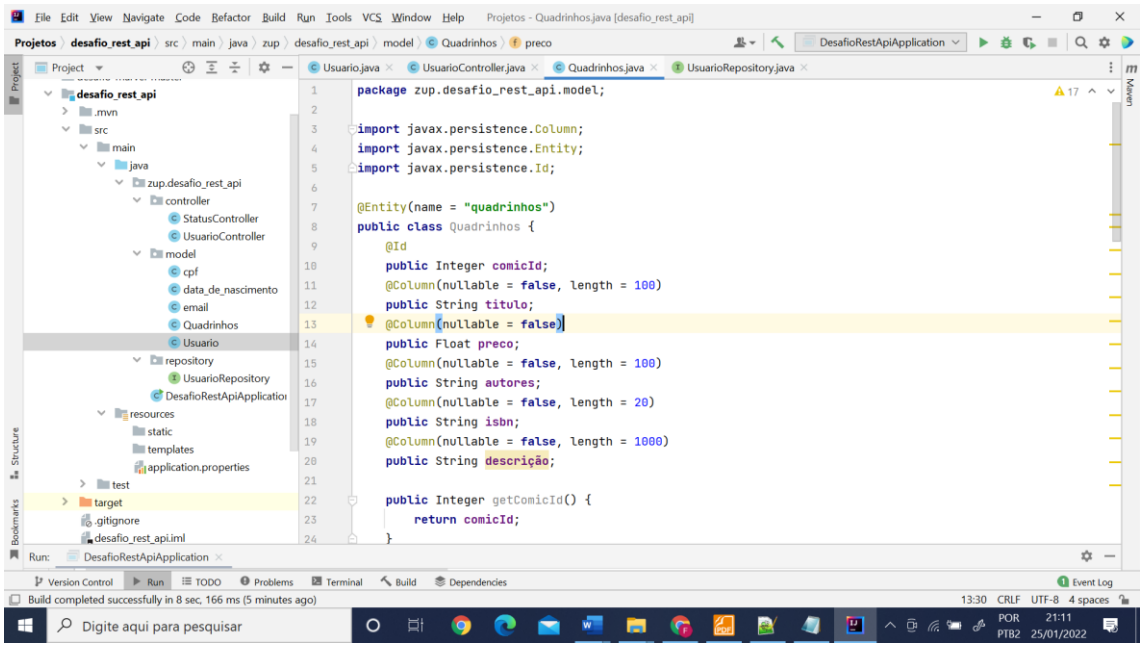
https://gateway.marvel.com/v1/public/comics/2?ts=1&apikey=XXXXXXXXXXXXXXX&hash=XXXXXXXXXXXXXXXXX

O resultado dessa requisição, obtida apenas colando essa requisição na barra de pesquisa do navegador, é ilustrado na figura a seguir:

{"code":200,"status":"Ok","copyright":"© 2022 MARVEL","attributionText":"Data provided by Marvel. © 2022 MARVEL","attributionHTML":"Data provided by Marvel. © 2022 MARVEL","etag":"4374aa19340774706920e0fca992900a85469b027","data":{"offset":0,"limit":20,"total":1,"count":1,"results":
[{"id":2,"digitalId":3855,"title":"Pulse (2004) #6","issueNumber":6,"variantDescription":"","description":"PART 1 (OF 5)\r
Jessica Jones and Luke Cage's lives have been destroyed by the events of the Secret War, so what is Jessica going to do about it? Fans of Secret War - feel The Pulse as it pounds out more gritty intrigue even as it welcomes a new artist: award-winning comics superstar Brent (RISING STARS, ASTRO CITY) Anderson! \r
Guest-starring: Captain America, Luke Cage, Iron Fist, Nick Fury and many, many more.\r
","modified":"2015-11-23T17:27:26-0500","isbn":"","upc":"5960605483-00611","diamondCode":"","ean":"","issn":"","format":"Comic","pageCount":0,"textObjects":[{"type":"issue_solicit_text","language":"en-us","text":"PART 1 (OF 5)\r
Jessica Jones and Luke Cage's lives have been destroyed by the events of the Secret War, so what is Jessica going to do about it? Fans of Secret War - feel The Pulse as it pounds out more gritty intrigue even as it welcomes a new artist: award-winning comics superstar Brent (RISING STARS, ASTRO CITY) Anderson! \r
Guest-starring: Captain America, Luke Cage, Iron Fist, Nick Fury and many, many more.\r
"}],"resourceURI":"http://gateway.marvel.com/v1/public/comics/2","urls":
[{"type":"detail","url":"http://marvel.com/comics/issue/2/pulse_2004_6?utm_campaign=apiRef&utm_source=0ec177c45ec3a0440d67bb3bc171e9ee"},
{"type":"purchase","url":"http://comicstore.marvel.com/The-Pulse-6/digital-comic/3855?utm_campaign=apiRef&utm_source=0ec177c45ec3a0440d67bb3bc171e9ee"},
{"type":"reader","url":"http://marvel.com/digitalcomics/view.htm?iid=3855&utm_campaign=apiRef&utm_source=0ec177c45ec3a0440d67bb3bc171e9ee"},
{"type":"inAppLink","url":"https://applink.marvel.com/issue/3855?utm_campaign=apiRef&utm_source=0ec177c45ec3a0440d67bb3bc171e9ee"}],"series":
{"resourceURI":"http://gateway.marvel.com/v1/public/series/674","name":"Pulse (2004 - 2006)"},"variants":[],"collections":
[{"resourceURI":"http://gateway.marvel.com/v1/public/comics/2166","name":"THE PULSE VOL. 2: SECRET WAR TPB (Trade Paperback)"}],"collectedIssues":[],"dates":
[{"type":"onsaleDate","date":"2004-11-10T00:00:00-0500"},{"type":"focDate","date":"-0001-11-30T00:00:00-0500"},{"type":"unlimitedDate","date":"2007-11-13T00:00:00-0500"},
{"type":"digitalPurchaseDate","date":"2014-09-02T00:00:00-0400"}],"prices":[{"type":"printPrice","price":2.99},{"type":"digitalPurchasePrice","price":1.99}],"thumbnail":
{"path":"http://i.annihil.us/u/prod/marvel/i/mg/1/30/56538fd257915","extension":"jpg"},"images":
[{"path":"http://i.annihil.us/u/prod/marvel/i/mg/1/30/56538fd257915","extension":"jpg"}],"creators":
{"available":0,"collectionURI":"http://gateway.marvel.com/v1/public/comics/2/creators","items":[],"returned":0},"characters":
{"available":7,"collectionURI":"http://gateway.marvel.com/v1/public/comics/2/characters","items":
[{"resourceURI":"http://gateway.marvel.com/v1/public/characters/1009220","name":"Captain America"},
{"resourceURI":"http://gateway.marvel.com/v1/public/characters/1010776","name":"Danny Rand"},
{"resourceURI":"http://gateway.marvel.com/v1/public/characters/1009378","name":"Jessica Jones"},
{"resourceURI":"http://gateway.marvel.com/v1/public/characters/1010363","name":"Logan"},{"resourceURI":"http://gateway.marvel.com/v1/public/characters/1009215","name":"Luke Cage"},{"resourceURI":"http://gateway.marvel.com/v1/public/characters/1009471","name":"Nick Fury"},
{"resourceURI":"http://gateway.marvel.com/v1/public/characters/1009718","name":"Wolverine"}],"returned":7},"stories":
{"available":2,"collectionURI":"http://gateway.marvel.com/v1/public/comics/2/stories","items":[{"resourceURI":"http://gateway.marvel.com/v1/public/stories/2334","name":"1 of 5 - Secret War arc","type":"interiorStory"},{"resourceURI":"http://gateway.marvel.com/v1/public/stories/127841","name":"cover to The Pulse (2004) #6","type":"cover"}],"returned":2},"events":{"available":0,"collectionURI":"http://gateway.marvel.com/v1/public/comics/2/events","items":[],"returned":0}}]}}

Portanto, nossa aplicação precisa saber fazer requisições para a API Marvel e, posteriormente, tratar as respostas recebidas.

Primeiramente precisamos de uma classe que represente os dados que queremos como resposta. Como vimos, a API da Marvel trás no JSON diversas propriedades, mas queremos apenas as informações ComicId, Título, Preço, Autores, ISBN e Descrição, conforme solicitado no enunciado do desafio. A criação de uma classe com esses atributos pode ser feita de forma semelhante àquela adotada na criação da classe para os usuários. A figura a seguir ilustra um exemplo:



De maneira semelhante, definimos uma classe para controlar essa entidade, chamada ‘QuadrinhosController’, bem como uma interface para operações CRUD no banco de dados. A interface ‘QuadrinhosRepository’ podem ser definida conforme figura a seguir:

```java
package zup.desafio_rest_api.repository;

import org.springframework.data.repository.CrudRepository;
import zup.desafio_rest_api.model.Quadrinhos;

public interface QuadrinhosRepository extends CrudRepository<Quadrinhos, Integer> {
}
```

Vale lembrar que as informações dos quadrinhos devem ser obtidas da API Marvel, realizando as requisições e tratando o retorno dessa API . Para tal, uma boa maneira é criar uma classe Service para encapsular essa lógica de acesso à API da Marvel, e instanciar um objeto da classe RestTemplate, para chamar essa API externa. Adicionalmente, para representar o retorno da API podemos criar *Data Transfer Object* (DTO), que basicamente é uma classe com atributos simples que será usada para otimizar a comunicação entre o cliente e o servidor.

Assim, as informações retornadas pela API da Marvel ficarão salvas no objeto DTO. Como o método salvar da classe QuadrinhosController precisa de um objeto do tipo quadrinho, precisamos criar um construtor na classe e adicionar um método que transforme o objeto DTO em um objeto do tipo quadrinho, o qual será salvo em nosso banco de dados.

Em resumo, portanto, o controller chama a classe Service que encapsula a chamada à API da Marvel com o RestTemplate. Os dados retornados pela API externa são armazenados como um objeto DTO que, por sua vez, é transformado em um objeto quadrinho e inserido no banco de dados.

Por fim, precisamos gerenciar os quadrinhos de um determinado usuário. Uma forma simples de fazer esse gerenciamento é a inclusão de uma atributo na classe usuário como um array de quadrinhos (array de números inteiros com

os IDs dos quadrinhos daquele usuário), além de métodos para incluir, excluir, imprimir essa lista de IDs. Dessa forma, a API possibilitará a consulta da lista de usuários cadastrados, retornando as informações de um usuário. Entre as informações do usuário estará o array com os IDs dos seus quadrinhos que possibilitará uma consulta à lista de quadrinhos que esse usuário possui.

Por fim, essa não é, obviamente, a única forma de realizar esse desenvolvimento. Para o consumo da API Marvel, por exemplo, poderíamos usar também o Feign que é um projeto do Spring Cloud que permite a integração e consumo de serviços rest.