DESENVOLVIMENTO DE API REST COM SPRING BOOT

Edemilton Alcides Galindo Junior
Especialista de Desenvolvimento de Soluções Web
edemiltonjr@gmail.com

Romeu Dias Rocha
Graduando em Sistemas de Informação – UniRios.
romeu.d.rocha@gmail.com

Ronierison de Souza Maciel
Doutorando em Ciência da Computação - CIn UFPE
ronierison.maciel@gmail.com

RESUMO

Sistema modernos são construídos para dar suporte a vários tipos de aparelhos ou sistemas operacionais (SO), Android, IOS, Linux, Windows entre outros. Isso porque há uma variedade de equipamentos, cada um funcionando com seu sistema operacional, na maioria dos casos não é viável construir um sistema para cada aparelho ou SO, pois gera custos demasiados. Este trabalho consiste em descrever o processo de construção de back-end conforme *API REST* utilizando o ecossistema Spring, permitindo que um único sistema seja particionado conforme seus recursos e seja consumido em outros aparelhos e/ou sistemas operacionais. O estudo para elaboração deste trabalho, se deu através de pesquisa bibliográfica em trabalhos acadêmicos já publicados. Será usado pesquisa descritiva que de acordo com Gil (2002, p.42), "As pesquisas descritivas têm como objetivo primordial a descrição das características de determinada população ou fenômeno ou, então, o estabelecimento de relações entre variáveis". Nesse contexto, será descrito as características da arquitetura de desenvolvimento baseada em redes, a arquitetura *REST*. Somado a isso, a pesquisa também é explicativa que segundo (GIL, 2008, p. 28), "São aquelas pesquisas que têm como preocupação central identificar os fatores que determinam ou que contribuem para a ocorrência dos fenômenos. Este é o tipo de pesquisa que mais aprofunda o conhecimento da realidade, porque explica a razão, o porquê das coisas". Pois será explicado sobre o contexto de desenvolvimento de *software*, seus componentes e etapas.

Palavras-chave: API. Arquitetura REST. API REST. Desenvolvimento com Spring.

REST API DEVELOPMENT WITH SPRING BOOT

ABSTRACT

Modern systems are built to support several kids of tools or operational systems (OS) like Android, IOS, Linux, Windows, among others. This is necessary due to a variety of gadgets working through operational systems and it is often unviable to build a specific OS for each device, once that it generates great costs. This paper aims to describe the process of back-end development through REST API,

using the Spring system in such a way that a single system is portioned according to its resources and used in other devices and/or operational systems. This study was carried out through a bibliographic research on academic papers. Its approach is descriptive, which, according to Gil (2002, p.42), "have as primary goal the description of the characteristics of a certain group or phenomenon, or the establishment of relations between variables". In this context, this paper describes the architecture of development based on networks, the *REST* architecture. In addition, the research is also explicative, which, according to Gil (2008, p.28), "are the researches that are concerned about identifying factors that determine or that contribute to the occurrence of phenomena. This is the type of research that most deeply approaches the knowledge about reality, once that it explains the reasons why things happen". The study explains the context of the software development, its components and steps.

Keywords: API. REST Architecture. REST API. Development with Spring.

1 INTRODUÇÃO

Este estudo consiste em demonstrar o desenvolvimento de API REST utilizando o ecossistema Spring, ou seja, uma série de módulos construídos para auxiliar desenvolvedores Java em seu trabalho.

Alguns sistemas precisam compartilhar seus dados para mais de um dispositivo, dar acesso às informações a qualquer horar ou lugar, de forma remota para que o gestor ou colaborador consiga a partir de seu *notebook*, *tablet* ou até mesmo *smartphone*, ter acesso ao sistema de sua organização e tomar suas decisões.

API REST é uma solução baseada em redes para que os componentes que formam o sistema possam interagir entre si. O desacoplamento desse sistema proporciona vantagens sobre o desenvolvimento, pois haverá apenas um *back-end* se comunicando com vários front-ends, trazendo impacto positivo na engenharia de *software*, reduzindo o tempo e os custos de desenvolvimento.

A separação de responsabilidades em um sistema distribuído colabora para que os componentes possam evoluir de forma independente, ou seja, a camada de *back-end* da aplicação pode incluir novas funcionalidades sem que a camada do *front-end* seja prejudicada pelas alterações, o mesmo acontece com a camada do *front-end*, onde a interface gráfica pode

sofrer suas alterações e mudar a forma que as informações são exibidas, com o objetivo de torná-la mais compreensível para o usuário.

O estudo está dividido em três partes: *API,* Arquitetura *REST* e desenvolvimento com *Spring*. Os dois primeiros tópicos são de conceituação, para explicar as restrições conceituais que uma aplicação distribuída deve respeitar. O tópico de desenvolvimento é a prática de tais conceitos, utilizando o ecossistema Spring do Java.

2 API

Com o objetivo de reutilização, software pode ser construídos para compartilhar seus recursos com outros, por exemplo um software para emissão de boletos, pode compartilhar suas rotinas a diversos outros sistemas que necessite de suas funcionalidades, é uma forma de encapsular um módulo, e delegar a responsabilidade do sistema, imagine uma organização que oferece pagamentos de seus serviços via boleto, em que o usuário pode através do site ou do aplicativo fazer seu pagamento. Neste cenário fica evidente a boa prática de compartilharem o mesmo recurso de pagamento, pois desta forma não seria necessário o aplicativo e o site possuírem de forma separada seu próprio emissor de boletos, porque eles podem compartilhar um único. Uma forma de partilhar recursos é através de *API, Application Programming Interface* ou em português, Interface de Programação de Aplicações.

> O conceito de API nada mais é do que uma forma de comunicação entre sistemas. Elas permitem a integração entre dois sistemas, em que um deles fornece informações e serviços que podem ser utilizados pelo outro, sem a necessidade de o sistema que consome a API conhecer detalhes de implementação do software. (PEREIRA, 2019, n.p.).

API serve para interligar, fornecer suas funcionalidades ou rotinas, para mais de um sistema, sem que as partes saibam detalhes interno de sua implementação. Um grande exemplo é o serviço de geolocalização do Google Maps, que muitos sistemas (clientes) utilizam seu serviço, como aplicativos de delivery, transporte, trânsito e outros vários, e nenhum deles sabe detalhes de sua implementação. Vale destacar que *API* **não** está diretamente associado a uma linguagem de programação, pois uma API pode ser escrita em Java, e ser consumida por outro sistema construído em PHP por exemplo.

Sua utilização é abrangente, tem o propósito de facilitar rotinas de pessoas e sistemas, é construída a partir de necessidades cotidianas, para que outras pessoas ou sistemas consigam utilizar seus recursos. Uma definição de API utilizando exemplo cotidiano:

> [...] API é como um garçom de um restaurante. O cliente, neste caso a aplicação que deseja receber os serviços, recebe do garçom o menu com todos os itens daquele restaurante. Ao escolher uma opção o garçom leva este pedido até a cozinha, aplicação da API, onde por sua vez os cozinheiros, que são os serviços compartilhados pela aplicação, realizam o pedido como foi descrito pelo cliente. Ao concluir o pedido o cozinheiro avisa o garçom, este por sua vez entrega o pedido ao cliente completando o processo de exemplificação uma requisição de API. (GOUVEIA, 2016, n.p.).

Há algumas formas de interação entre cliente e *API*, as requisições podem ser através de mais de uma forma, esse estudo será direcionado para iterações utilizando a internet, isto é, através do protocolo *HTTP* e suas requisições. Quando falamos de *API* que utiliza o protocolo *HTTP* estamos falando de *web services*, ou traduzido, serviços *web*. A definição de web services segundo Augusto:

> [...] é uma tecnologia que permite a comunicação entre aplicações de maneira independente de linguagem de programação e de sistema operacional[...] Os Web Services são componentes que permite que as aplicações enviem e recebam dados geralmente em formato XML ou JSON. (AUGUSTO, 2019, n.p).

De modo prático, os recursos da API são compartilhados através de *Web Services* e utiliza *XML* ou *JSON* como formato de comunicação, ou seja, quando um cliente faz uma requisição, é retornado uma resposta no formato XML ou JSON, quando a API precisa de alguma informação do cliente, o formato enviado também precisa ser o mesmo. Conforme a figura 01, podemos ver o desacoplamento da linguagem de programação, independente do dispositivo ou a linguagem de programação que a aplicação está rodando, a API consegue se comunicar com todos eles, desde que a comunicação seja feita através de um formato em comum, XML ou JSON.

Figura 01 - Web Service

Fonte: (AUGUSTO,2019, n.p.)

Para exemplificar uma API de postagens, para uma requisição *HTTP* com o verbo *GET* na *URI* ‘https://localhost/autor’, o *Web Services* deverá retornar algo similar conforme a figura 02. os detalhes de implementação deste tipo de API estão no tópico de desenvolvimento.

Figura 02 - Resposta JSON para requisição de autores

```
[
    {
        "id": 1,
        "nome": "j.k rowling",
        "descricao": "escritora, roteirista ..."
    },
    {
        "id": 2,
        "nome": "Masashi Kishimoto",
        "descricao": " mangaká e escritor japonês ..."
    }
]
```

Fonte: Autores

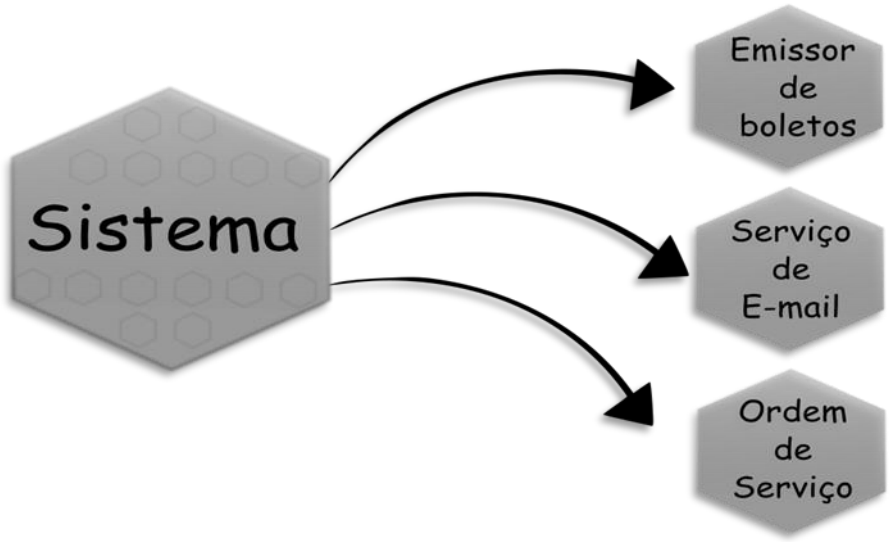
A informação retornada está pronta para ser usada em aplicações clientes, os dados são retornados no formato JSON, com isso, basta que outro sistema consuma suas informações, podendo ser um sistema web, mobile, ou mesmo sistema desktop. Existem muitos meios de implementar o consumo destas API's, as ferramentas mais comuns para esse fim na web são: *Angular*, *React*, Vue.JS e etc. Contudo o consumo desta API não será o foco deste estudo, apenas será apresentado no tópico de desenvolvimento.

3 ARQUITETURA REST

Transferência de Estado Representacional (*REST*) é uma arquitetura para sistemas baseados em redes (sistemas distribuídos), foi criada por Roy Thomas Fielding em sua tese de doutorado apresentado à universidade da Califórnia situada em Irvine. Fielding (2000) define *REST* como um conjunto de *constraints* (restrições ou princípios) com objetivo de definir a melhor forma de particionar um sistema, a forma como os componentes se identifica e se comunica entre si, como as informações são comunicadas, como os elementos podem evoluir de forma independente.

Fielding criou um modelo para organizar os recursos de um sistema *web*, permitindo uma comunicação padronizada, de modo que os recursos podem funcionar de forma independente. Conforme ilustra a **figura 03**, o sistema possui seus recursos ou componentes, estes funcionam de forma independente, permitindo a troca de informação através de *Web Services.*

Figura 03 - Particionamento de um sistema



Fonte: Autores

Esse modelo de particionamento permite algumas vantagens pois, as partes podem evoluir sem comprometer o funcionamento do todo, por exemplo, o recurso de ordem de serviço pode evoluir e implementar um algoritmo de agendamento, podendo inclusive, consumir o recurso de e-mail e enviar à ordem de serviço para o e-mail do cliente.

Fielding colaborou com Tim Berners-Lee na construção do protocolo *HTTP*, por isso o *REST* utiliza os recursos do protocolo de forma eficiente, Tilkov (2007, n.p., Trad. *Google Translate*) define REST como:

> [...] um conjunto de princípios que definem como os padrões da Web, como HTTP e URIs, devem ser usados (o que muitas vezes difere um pouco do que muitas pessoas realmente fazem). A promessa é que, se você aderir aos princípios do REST ao projetar seu aplicativo, terá um sistema que explora a arquitetura da Web em seu benefício.

Em termos práticos a utilização de *URI* no padrão *REST* tem um caráter especial, pois todos os recursos serão acessados pelos seus identificadores, usando a **figura 03** como exemplo, os recursos podem ser identificados pelas *URI's*:

- https://localhost/**boleto**;
- https://localhost/**emai**l;
- https://localhost/**ordemservico**;

Com a determinação das *URI's* dos recursos, é possível consumir as ações sobre eles, em *Web Services*, seu consumo é feito através do protocolo *HTTP*, de forma mais específica, através dos verbos. Conforme a documentação do protocolo *HTTP* os verbos são:

- *GET*: Solicita a representação de um recurso específico. Requisições utilizando o método GET devem retornar apenas dados.
- *POST*: É utilizado para submeter uma entidade a um recurso específico, frequentemente causando uma mudança no estado do recurso ou efeitos colaterais no servidor.
- *PUT*: Substitui todas as atuais representações do recurso de destino pela carga de dados da requisição.
- *DELETE*: Remove um recurso específico.

Existem outros cinco verbos, contudo neste estudo só será utilizado os citados acima, pois eles serão suficientes para consumir a aplicação. Por exemplo, se o sistema precisa da listagem de cliente, ele precisa fazer uma requisição na *URI* do recurso, com o verbo *GET*, desta forma a aplicação deverá retornar a listagem. Esse padrão adotado, é usado para todo tipo de requisição, conforme a **tabela 01**:

Tabela 01 - Requisições e utilização de verbos HTTP

Ação	URI	Verbo	Descrição da requisição
1	https://localhost/ordemservico	GET	Listar ordens de serviço
2	https://localhost/ordemservico	POST	Cadastrar uma nova ordem de serviço
3	https://localhost/ordemservico/**1**	GET	Buscar ordem de serviço de código 1
4	https://localhost/ordemservico/**1**	PUT	Editar ordem de serviço de código 1
5	https://localhost/ordemservico/**1**	DELETE	Excluir ordem de serviço de código 1

Fonte: Autores

Embora as ações 1 e 2 possuam a mesma *URI*, elas têm finalidades completamente diferentes, uma faz a consulta de dados, outra faz o cadastro de dados, a diferença estar no verbo utilizado. A mesma lógica acontece nas ações 3,4 e 5.

Analisando ainda a tabela, também é relevante o código de status de retorno do protocolo, para cada ação e verbo, o *HTTP* deve retornar seus status, de acordo com MOZILLA os status são:

- 200 *OK*: Esta requisição foi bem-sucedida. O significado do sucesso varia de acordo com o método HTTP.
- 201 *Created*: A requisição foi bem-sucedida e um novo recurso foi criado como resultado. Esta é uma típica resposta enviada após uma requisição *POST*.
- 204 *No Content*: Não há conteúdo para enviar para esta solicitação, mas os cabeçalhos podem ser úteis. O *user-agent* pode atualizar seus cabeçalhos em cache para este recurso com os novos.
- 400 *Bad Request*: Essa resposta significa que o servidor não entendeu a requisição pois está com uma sintaxe inválida.
- 404 *Not Found:* O servidor não pode encontrar o recurso solicitado. Este código de resposta talvez seja o mais famoso devido à frequência com que acontece na web.
- 500 *Internal Server Error*: O servidor encontrou uma situação com a qual não sabe lidar.

Existem outros códigos de status, contudo neste estudo só será aplicado os itens acima.

Tabela 02 - Status HTTP para cada requisição

Ação	URI	Verbo	HTTP Status
1	https://localhost/ordemservico	*GET*	*200 OK*
2	https://localhost/ordemservico/**1**	*GET*	*200 OK*
3	https://localhost/ordemservico/**99900**	*GET*	*404 Not Found*
4	https://localhost/ordemservico/vvv	*GET*	*400 Bad Request*
5	https://localhost/ordemservico	*POST*	*201 Created*
6	https://localhost/ordemservico/**1**	*PUT*	*204 No Content*
7	https://localhost/ordemservico/**1**	*DELETE*	*204 No Content*

Fonte: Autores

As ações representadas na tabela, demonstram o código de status retornado conforme o protocolo *HTTP*. A ação 3, simula uma consulta a um recurso que não existe, isto é, não existe ordem de serviço com o código 99900. A ação 4, serve para quando é feita uma requisição que o sistema não consegue identificar do que se trava, os caracteres “vvv”, não corresponde a nenhum parâmetro de *URI* identificável.

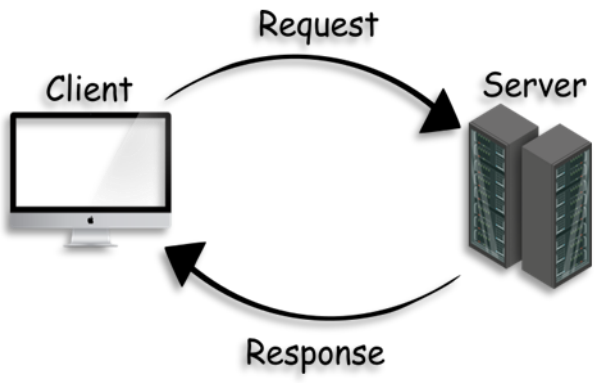
Conforme Fielding (2000) *REST* foi criado sobre 6 *constraint*, contudo uma delas é opcional por tanto foi omitida aqui.

3.1 Client-Server

A primeira *constraint* É uma arquitetura para aplicação distribuída, de acordo com Fielding, O *server* (servidor) é o componente responsável pelos serviços, então ele precisa “ouvir” os pedidos e responde-los, o *client* (cliente) é o componente que faz uso destes serviços, para isso o cliente faz requisições ao servidor.

Neste sentido, *client-server* é a comunicação entre dois componentes, os servidores são onde estão as informações, o papel do cliente é requisitar essas informações. Conforme a figura 02, podemos dividir então em dois componentes, client-side (lado do cliente) e server-side (lado do servidor). na web, são vários clientes fazendo requisições a servidores. Exemplificando de forma simples, ao acessar o *youtube*, nosso dispositivo faz uma requisição em seu servidor, “pedindo” o vídeo, o servidor “escuta” o pedido e estabelece uma comunicação, e envia uma cópia do vídeo para que o cliente possa reproduzi-lo.

Figura 04 - Cliente Servidor



Fonte: Autores

Na prática, cliente servidor é a base da web, a **figura 04** ilustra de os processos de um sistema distribuído. Fielding destaca que essa arquitetura tem como objetivo separar as funcionalidades, então o back-end de uma aplicação roda no lado do servidor, contendo regras de negócio, e todo *CRUD* para o banco de dados, enquanto o front-end, contendo a interface gráfica é executado apenas no dispositivo do cliente.

Fielding acrescenta que esse tipo de arquitetura permite que os componentes possam evoluir de forma independente, ou seja, novas funcionalidades podem ser incrementadas no sistema sem que o front-end seja obrigado a alterar seu funcionamento, bem como o front-end pode mudar sua interface gráfica a fim de tornar a informação mais compreensível para os usuários, sem a necessidade de alterar o back-end da aplicação.

3.2 Stateless

De acordo com Fielding, essa restrição diz que a comunicação entre cliente e servidor deve ser sem estado, toda requisição ao servidor deve conter todas as informações necessárias para que o servidor possa compreender a solicitação. Isso embora possa diminuir o empenho da rede, o sistema ganha em visibilidade, confiabilidade e escalabilidade. Visibilidade pois o sistema precisa olhar apenas a requisição para identificar sua natureza. Confiabilidade porque é ele facilita o processo de recuperação de falhas parciais. Escalável pois o fato de o servidor não armazenar estado entre as requisições permite que o servidor libere rapidamente o recurso.

Para demonstrar de forma clara observe a **figura 05**, simulando o pedido em um e-commerce, o cliente envia a lista contendo todos os produtos que deseja, seus dados e formas de pagamento. Com isso o servidor terá numa única requisição todas as informações necessárias para processar o pedido.

Figura 05 - Modelo Stateless

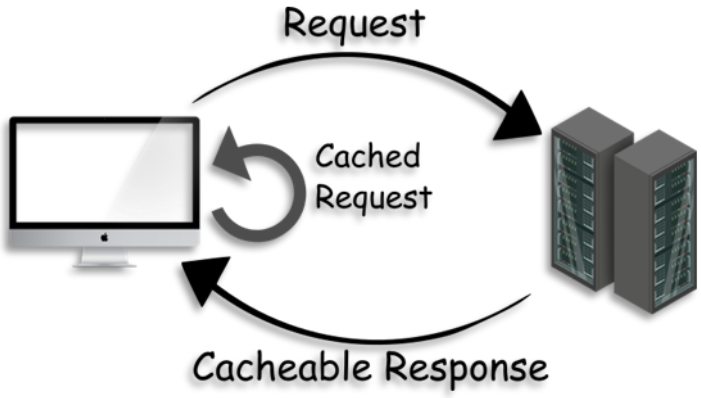
Fonte: Autores

Essa constraint, é o oposto de Stateful, que em um exemplo simples, ao visitar um site de vendas que utilize Stateful, o servidor irá "lembrar" da atividade do cliente enquanto ele estiver conectado, para posteriormente usar essas informações no carrinho de compra. Fielding propôs uma solução diferente, a arquitetura REST não permite reter informação no servidor, então cada requisição é tratada como a primeira que o servidor "ouviu" daquele cliente. No formato *Steteles* é utilizado *tokens* ou *cookies*, para armazenar estado no lado do cliente, fazendo parecer para o usuário que o servidor se lembra dele.

3.3 Cache

Conforme proposto por Fielding, essa constraint tem como objetivo melhorar o desempenho da rede, para isso, toda resposta do servidor deverá ser rotulada como armazenável com cache ou não armazenáveis em cache. Se uma resposta poder ser armazenada em cache, ela fica salva em cache, quando o cliente fizer a mesma requisição ou equivalente, a informação da resposta será utilizada novamente pelo cache, a menos que a informação no servidor tenha sofrido alterações, conforme ilustrado na **figura 06**.

Figura 06 - Cache

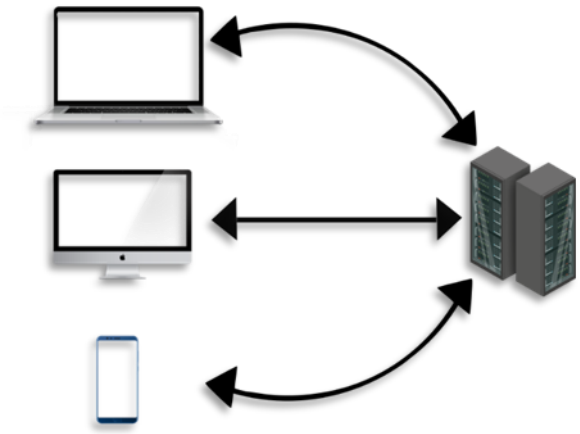


Fonte: Autores

3.4 Uniform Interface

Fielding pontua que a maior diferença da arquitetura REST para outros modelos arquitetônicos é sua ênfase em interfaces uniformes entre os componentes, visto que o servidor não deve diferenciar o acesso a dados pelo dispositivo ou sistema operacional do cliente, ou seja, o acesso aos endpoints do web service deve ser igual para qualquer dispositivo que acessar, seja um notebook, smartphone entre outros. Conforme a **figura 07**, o servidor deve se comunicar de formal igual e uniforme para qualquer dispositivo ou sistema operacional.

Figura 07 - Uniform Interface



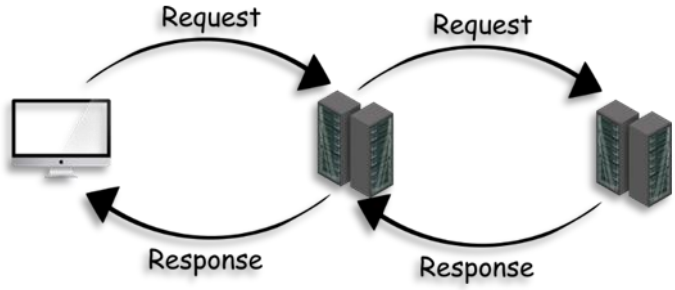
Fonte: Autores

Na prática isso é feito através de JSON ou XML, conforme ilustrado no tópico de API, então aquela informação, será manipulado em qualquer dispositivo ou sistema operacional.

3.5 Layered System

Layered System (sistema em camadas), de acordo com Fielding a arquitetura REST deve ser construída conforme o estilo em camadas, isto é, uma sequência hierárquica. Isso significa que um componente não pode ver além da camada imediata com a qual ele está interagindo. Para exemplificar isso de maneira prática, é como um sistema de delivery, em que os componentes clientes (aplicativo ou site) acessam o endpoint para fazer seu pedido, este endpoint possui integração com outro servidor, o de geolocalização do Google por exemplo. Fielding está dizendo que o usuário do aplicativo de delivery, não precisa diretamente acessar o serviço de geolocalização, pois ele é abstraído em camadas, em que o aplicativo acessa o endpoint do delivery e o delivery acessa o endpoint do serviço de geolocalização do google. Por isso o termo camadas, os recursos são dispostos em camadas, esta interação é ilustrada na **figura 08**.

Figura 08 - Sistema em camadas



Fonte: Autores

4 SOBRE O PROJETO

O projeto escolhido para aplicar todos esses conceitos será um blog, para publicação de postagens ou pequenos artigos, com esse sistema de exemplo será possível demonstrar de forma prática a maioria dos conceitos visto até aqui, contudo, existe uma curva de

aprendizagem, pois é necessário que o leitor saiba no mínimo programação orientada a objetos.

O projeto será feito no formato de *API REST*, ou seja, é uma aplicação que segue os conceitos de *API* e da arquitetura *REST*. Neste sentido *API REST* é uma aplicação que permite comunicação entre aplicações sem a dependência do sistema operacional e a linguagem de programação, além de respeitar todas as restrições propostas por Fielding.

Para o desenvolvimento será utilizado a linguagem de programação *Java* versão 11, *IDE Eclipse*, banco de dados H2 (fins de teste), *Lombok* e alguns módulos do *Spring Framework*. As dependências serão administradas pelo *Maven Apache*, então **não** será necessário baixar manualmente as bibliotecas que serão usadas no projeto.

4.1 Criação do projeto com Spring Boot

Há diversas formas de criar um projeto, foi escolhido uma ferramenta chamada *Spring Initializr*, em que selecionamos algumas dependências iniciais e ele constrói uma aplicação "base" e permite o download. Para o projeto selecione as dependências: *Spring web, Spring Data JPA, H2 Database e Lombok.*

Após descompactar o projeto baixado, é só importar o projeto na sua *IDE* de preferência, para importar no *Eclipse* faça: *File > Import > Existing Maven Projects* e selecione a pasta onde o projeto estiver contido. O projeto básico deverá conter a estrutura conforme a **figura 09.**

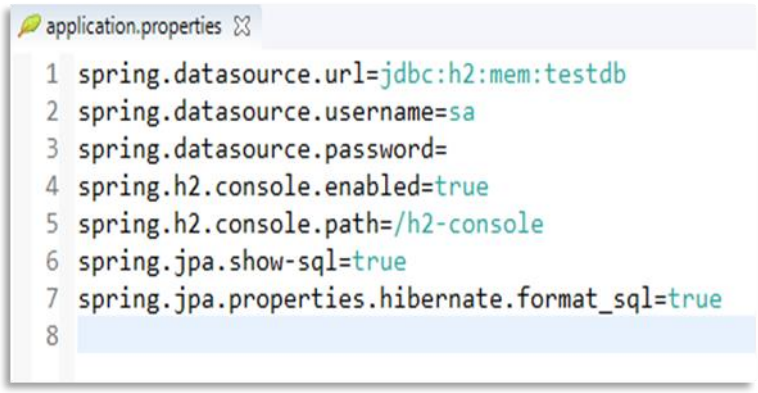
Figura 09 - Estrutura inicial de um projeto *Spring Boot*

Fonte: Autores

Na pasta de origem "**src/main/java"** são colocados código Java, isto é, nossas classes, a lógica do sistema e suas camadas de organização, por isso dentro da pasta, tem os pacotes onde serão colocados os códigos da aplicação. Na pasta "**src/main/resources"** são colocados os recursos do projeto, geralmente usado para arquivos de configuração com banco de dados, imagens e ícones entre outros. O arquivo **pom.xml** é muito importante em projetos *Maven,* é um arquivo de configuração do projeto, dentre as configurações são listadas as dependências do projeto, ou seja, as bibliotecas que o projeto irá utilizar, então o *Maven* faz o *download* automático para o desenvolvedor.

É necessário configurar o banco de dados, para isso no arquivo "aplication.properties" deverá ser colocada os comandos de configuração, conforme a **figura 10**.

Figura 10 - Configurações do banco de dados H2

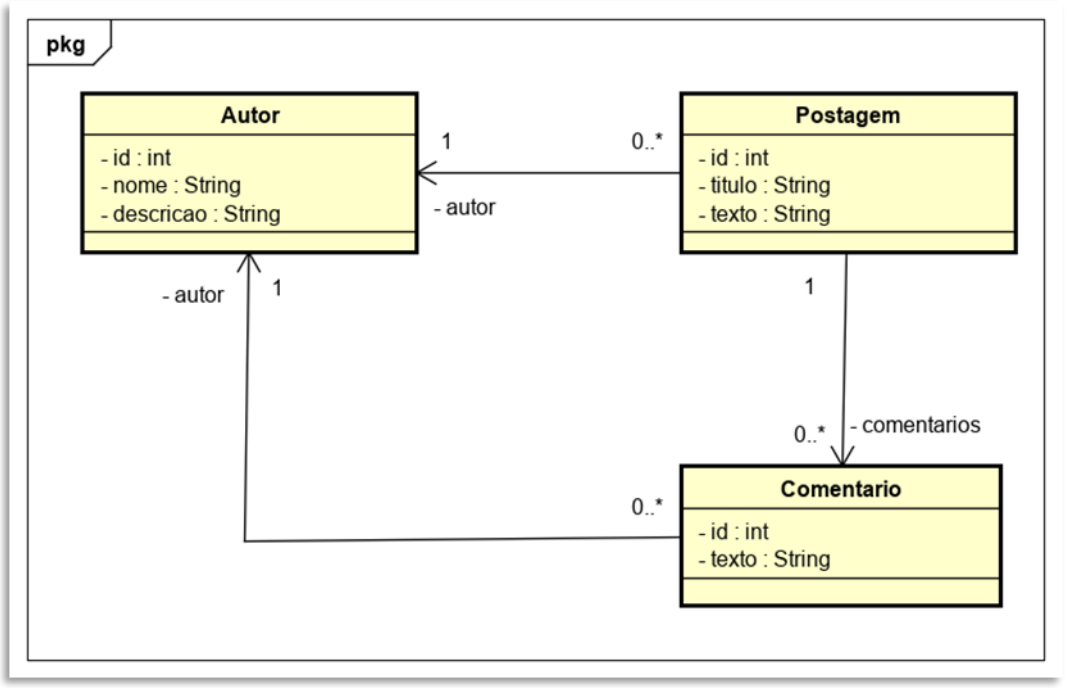


Fonte: Autores

4.2 Definindo e implementando as classes do sistema

Em um blog temos postagem que pertence a um autor, este possui nome e descrição. Cada postagem pode conter comentários feitos por outros autores. Conforme a **figura 11,** é ilustrado as classes do sistema.

Figura 11 - Diagrama de classes

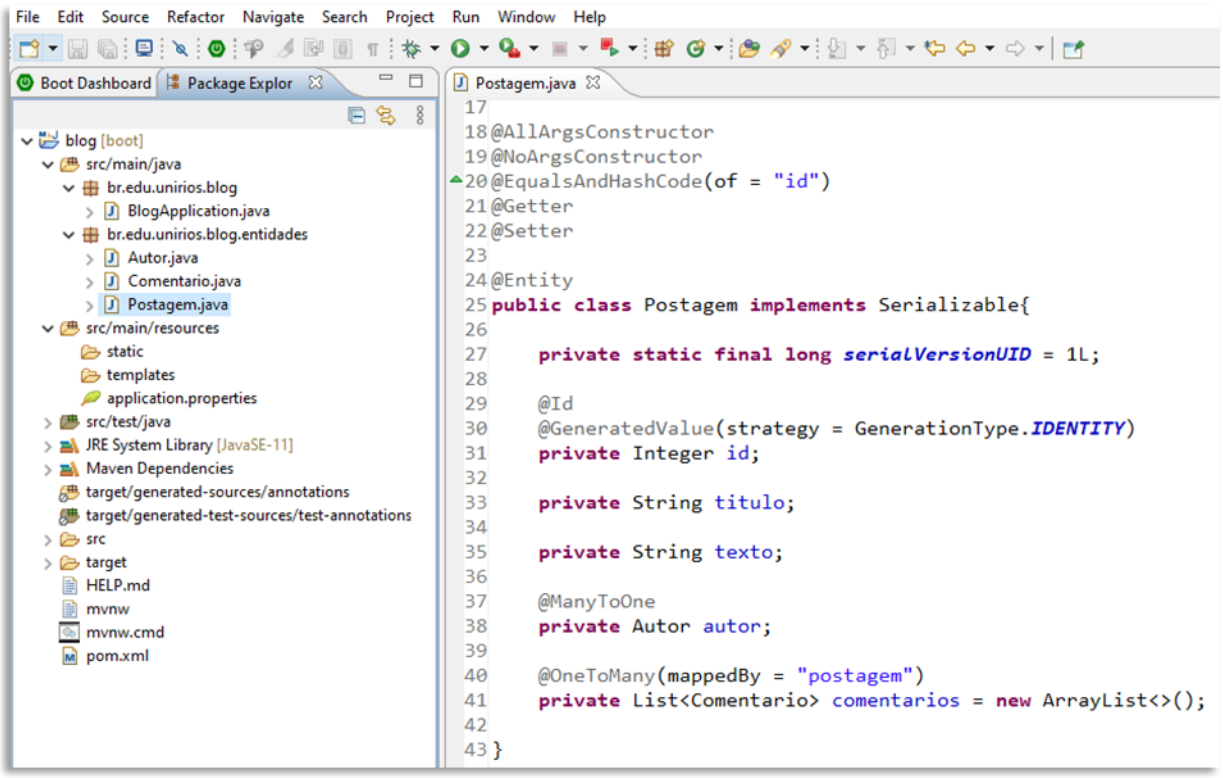


Fonte: Autores

Analisando o diagrama, podemos perceber que Postagem dois possui dois atributos de relacionamento, comentários e autor. O mesmo acontece com comentário que possui um atributo relacional do tipo Autor. Na **figura 12** será ilustrado na prática, a classe de Postagem.

Devemos implementar o sistema e organizar em camadas, para isso, dentro da pasta "src/main/java" e do pacote "br.edu.unirios.blog" deve ser criado outro pacote chamado "entidades", lá será colocado as classes do sistema. Observe as implementações na **figura 12**.

Figura 12 - Camada de entidades e suas relações



Fonte: Autores

Tem bastante informação na **figura 12,** é ilustrado seguindo o modelo orientado a objetos a representação física do diagrama, note que a classe de postagem possui explicitamente um atributo do tipo Autor e uma lista de comentários. Também é possível ver a organização dos pacotes até aqui, então toda vez que for criado um novo pacote, deverá seguir o mesmo padrão de nomes até aqui, com o prefixo "br.edu.unirios.blog" e o nome do novo pacote. Não é algo obrigatório, mas é uma boa prática, pois mantém o código organizado.

Os tributos de "id", "autor" e "comentarios" possuem anotações do *JPA* (*Java Persistence API*)[163], é um assunto a parte desta discussão, mas posso definir que sua função é sobre como as entidades se relacionam dentro de um banco de dados, é uma forma sofisticada de persistência em banco de dados, sobre a conversão entre as diferenças de paradigmas das linguagens de programação e bancos de dados. Para cada classe do sistema deverá ser implementado o seu modelo físico, conforme os seus atributos e relacionamentos definidos no diagrama de classes (**Figura 11**).

[163] JPA é um assunto muito abrangente, para saber mais é aconselhável consultar artigos ou livros só do tema.

4.3 Camada de repositórios

Para que nossas classes possam ser persistidas, precisamos usar *Spring Data*, um módulo do *Spring* que implementa o *JPA*[164]. De forma prática, cada classe deverá possuir sua interface de acesso ao banco de dados, a interface deve estender a classe "*JpaRepository"* e passar os dois parâmetros genéricos. O primeiro é a classe a ser persistida, o segundo é o tipo de sua chave primária. Com isso, crie um pacote chamado "repositorio" e crie as interfaces conforme a **figura 13**.

Figura 13: Acesso a dados com *Spring Data*

```
1  package br.edu.unirios.blog.repositorio;
2
3  import org.springframework.data.jpa.repository.JpaRepository;
4  import org.springframework.stereotype.Repository;
5
6  import br.edu.unirios.blog.entidades.Postagem;
7
8  @Repository
9  public interface RepositorioPostagem extends JpaRepository<Postagem, Integer>{
10
11 }
12
```

Fonte: Autores

Com esta simples interface podemos fazer as quatro operações (consulta, cadastro, edição e exclusão) na classe de Postagem no banco de dados. No *Spring*, devemos anotar com "@Repository", as interfaces de repositório.

4.4 Camada de serviço

Vamos definir uma camada chamada de "servico", nesse pacote são colocadas as regras de negócio do sistema, no blog não haverá muitas regras, contudo é importante seguir uma boa padronização, na prática, os *endpoints* da *API* não terão acesso direto a camada de repositório, não é uma boa prática, por isso é criada uma camada que faz essa ponte entre *endpoint* e camada de repositório.

Na camada de serviço, deve ser criada para cada classe sua classe de serviço, por exemplo, para classe Postagem a sua classe de serviço deverá se chamar "ServicoPostagem", as outras

[164] Spring Data, utiliza por padrão o Hibernate para sua implementação do JPA.

seguem o mesmo raciocínio, As classes de serviços terão 5 métodos: salvar, editar, deletar, buscar todos e busca por id. Poderia haver vários outros, contudo nesse estudo serão feitos apenas estes cinco, pois serão suficientes para a aplicação.

4.4.1 Serviço para buscar todos

A **figura 14** implementa um método para retornar a lista de postagens, ou seja, buscar todos, observe que ele utiliza a camada de repositório através da interface que definimos para Postagem "RepositorioPostagem", a anotação "@Autowired" faz com que o repositório seja instanciado automaticamente pelo Spring. Bastando apenas usar o método "findAll()" que foi implementado pelo *Spring Data*, não precisamos escrever nenhuma linha de *SQL* (linguagem de consulta estruturada)[166]. Nas classes de serviço, deve ser colocado a anotação "@Service".

Figura 14 - Camada de serviço - Método buscar todas postagens

Fonte: Autores

[165] SQL (sigla trad. Linguagem de consulta estruturada), é uma linguagem de consulta usada em bancos de dados relacionais

[166] SQL (sigla trad. Linguagem de consulta estruturada), é uma linguagem de consulta usada em bancos de dados relacionais

4.4.2 Serviço para buscar por id

Figura 15 - Camada de serviço - Método buscar postagem por código

```
28    public Postagem buscar(int id) {
29        Optional<Postagem> obj = repo.findById(id);
30        return obj.orElseThrow(() -> new ObjetoNaoEncontrado(
31                "Objeto não encontrado! Id: " + id + ", Tipo: " + Postagem.class.getName()));
32    }
```

Fonte: Autores

A **figura 15** utiliza a classe *Optional*, que espera um tipo genérico como parâmetro, sua principal utilidade é encapsular o retorno, com ela podemos verificar se houve retorno ou não, através do método "orElseThrow()" podemos usar uma expressão Lambda e instanciar nossa classe de erro para quando não houver retorno. A lógica é: se houver postagem a retorne, senão propague uma exceção.

4.4.3 Serviço para salvar e deletar

Os métodos para salvar e deletar a postagem são bem simples, o método de salvar recebe como parâmetro uma classe já instanciada do tipo Postagem, e utilizando a instância de seu repositório faz a operação para salvar. O método de deletar recebe como parâmetro o identificador da postagem, verifica se ela existe através do método de buscar, e se não for propagado exceção, ele o deleta. Essa rotina é ilustrada em código na **figura 16.**

Figura 16: Camada de serviço - Método salvar e deletar postagem

```
34    public Postagem salvar(Postagem obj) {
35        obj.setId(null);
36        return repo.save(obj);
37    }
38    public void deletar(int id) {
39        buscar(id);//ou existe, ou irá gerar exception
40        repo.deleteById(id);
41    }
```

Fonte: Autores

4.4.4 Serviço para editar

O método de editar é preciso implementar um método auxiliar, isto pois os objetos são monitorados pela JPA, então para editar uma postagem é preciso buscar ela, inserir as modificações, e por fim atualizar ela. O *Spring Data* usa o método "save()", tanto para cadastrar quanto para editar, O *Spring Data* diferencia a ação do método pela chave primária da entidade, isto é, se o id for igual a nulo, o método irá cadastrar, se não for nulo, ele irá tentar editar. Então o método deverá receber um objeto do tipo Postagem, fazer a consulta para o JPA monitorar, passar as modificações para o objeto monitorado, e chamar o método "save()" de seu repositório. Este evento é ilustrado na **figura 17**.

Figura 17 - Camada de serviço - Métodos para editar postagem

```
39      public Postagem editar(Postagem obj) {
40          Postagem newObj = buscar(obj.getId());
41          modificar(newObj, obj);
42          return repo.save(newObj);
43
44      }
45      private void modificar(Postagem newObj, Postagem obj) {
46          newObj.setAutor(obj.getAutor());
47          newObj.setComentarios(obj.getComentarios());
48          newObj.setTexto(obj.getTexto());
49          newObj.setTitulo(obj.getTitulo());
50      }
51
```

Fonte: Autores

Com isso foi implementado toda a camada de serviço para classe de Postagem, seguindo a mesma lógica deverá ser implementado para as outras classes do sistema.

4.5 Camada de recurso

Terminado a camada de serviço, podemos iniciar a camada de "recurso", esta camada é onde são implementados os *endpoints*, essa é a porta de entrada para quem utiliza a *API*. Para iniciar deverá ser criado um pacote chamado "recurso". Dentro do pacote deve ser criado para cada classe, suas respectivas classes de recursos, contendo seus *endpoints*, este estudo irá exemplificar a classe de recurso de Postagem, o nome atribuído a classe é

"RecursoPostagem". ele irá estabelecer conexão com a camada de serviço e todos os métodos que criamos anteriormente.

4.5.1 Endpoint para cadastrar nova postagem

O primeiro *endpoint* escolhido para ser criado é o de cadastrar uma nova postagem, ele recebe como parâmetro uma postagem, utiliza a camada de serviço e faz a operação de salvar. No *Spring*, cada classe de recurso deverá conter a anotação "@RequestMapping" contendo o valor da *URI*, para diferenciar das outras classes de *endpoints*. Os métodos também são assinados com a mesma anotação, para o *Spring* identificar, através de *URI* ou Verbo *HTTP*, qual *endpoint* deverá ser chamado. Estes eventos são ilustrados na **figura 18.**

Figura 18 - Camada de recurso - *endpoint* salvar

```
18 @RestController
19 @RequestMapping(value="/postagem")
20 public class RecursoPostagem {
21
22     @Autowired
23     private ServicoPostagem servico;
24
25     @RequestMapping(method=RequestMethod.POST)
26     public ResponseEntity<Void> salvar(@RequestBody Postagem obj) {
27         obj = servico.salvar(obj);
28         URI uri = ServletUriComponentsBuilder.fromCurrentRequest()
29             .path("/{id}").buildAndExpand(obj.getId()).toUri();
30         return ResponseEntity.created(uri).build();
31     }
32
```

Fonte: Autores

Conforme podemos observar na **figura 18**, o método salvar possui a anotação que tem como funcionalidade definir o verbo *HTTP* daquele método, então toda vez que alguém fizer uma requisição *POST* na *URI* http://localhost:8080/postagem, o *Spring* irá chamar o método salvar.

Analisando ainda a **figura 18**, percebemos que ele retornar um "ResponseEntity", é uma classe Java que auxilia no retorno de corpo, cabeçalhos e *Status HTTP*. no método em

questão, caso consiga cadastrar ele retorna o *Status* 201 *created*, conforme na linha 30 do código.

4.5.2 Endpoint para buscar todos, buscar por id, editar e deletar postagem

Figura 19 - Camada de recurso - *endpoints*

```
33    @RequestMapping(method=RequestMethod.GET)
34    public ResponseEntity<List<Postagem>> buscarTodos() {
35        List<Postagem> list = servico.buscarTodos();
36        return ResponseEntity.ok().body(list);
37    }
38
39    @RequestMapping(value="/{id}", method=RequestMethod.GET)
40    public ResponseEntity<Postagem> buscar(@PathVariable Integer id) {
41        Postagem obj = servico.buscar(id);
42        return ResponseEntity.ok().body(obj);
43    }
44
45    @RequestMapping(value="/{id}", method=RequestMethod.PUT)
46    public ResponseEntity<Void> editar(@RequestBody Postagem obj, @PathVariable Integer id) {
47        obj.setId(id);
48        obj = servico.editar(obj);
49        return ResponseEntity.noContent().build();
50    }
51
52    @RequestMapping(value="/{id}", method=RequestMethod.DELETE)
53    public ResponseEntity<Void> deletar(@PathVariable Integer id) {
54        servico.deletar(id);
55        return ResponseEntity.noContent().build();
56    }
```

Fonte: Autores

A **figura 19**, possui muitas informações, de forma prática cada método possui sua regra para ser chamada, as que possuem "/{id}" na anotação, esperam um parâmetro na *URI*, para identificar seu valor, por exemplo: http://localhost:8080/postagem/**1** , são elencados três possíveis métodos para esta *URI*, buscar, editar e deletar, o que irá diferenciar é o verbo *HTTP* da requisição, pois cada um possui um verbo diferente. Então se uma requisição na mesma *URI* for com o verbo *DELETE*, ele irá entrar no *endpoint* de deletar. Para as outras classes deverão ser implementadas seguindo a mesmo lógica. As requisições e as funções de entrada são demonstradas na **Tabela 03**.

Tabela 03 - *Endpoints* de entrada

URI	***HTTP***	**Função**
http://localhost:8080/postagem	*GET*	buscarTodos()

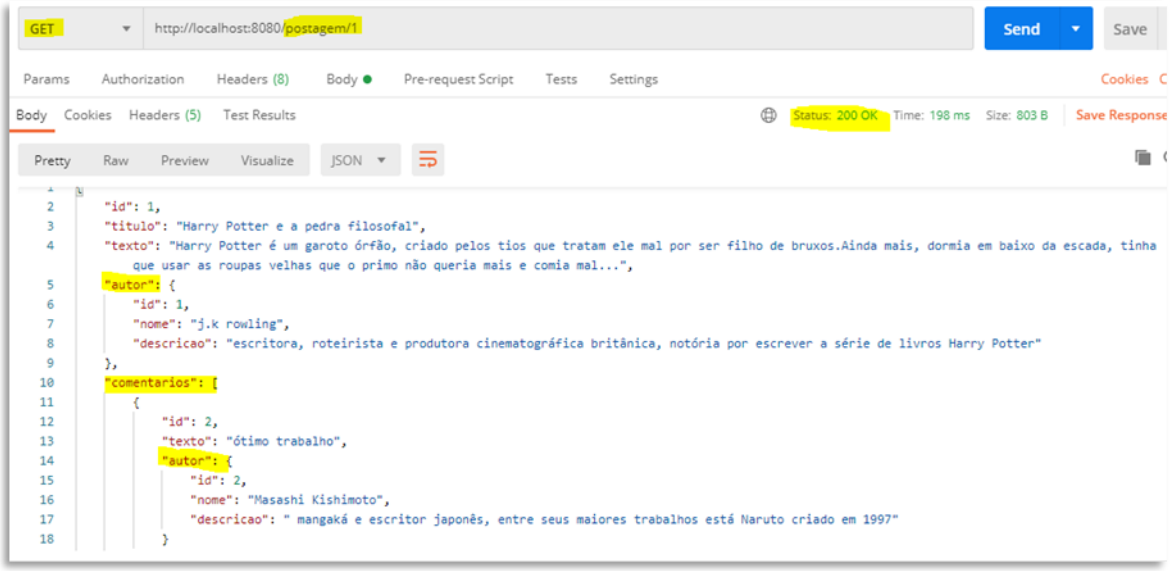
http://localhost:8080 /postagem	*POST*	salvar()
http://localhost:8080/postagem/1	*GET*	buscar()
http://localhost:8080/postagem/1	*PUT*	editar()
http://localhost:8080/postagem/1	*DELETE*	deletar()

Fonte: Autores

4.6 Considerações sobre a aplicação

O *back-end* da aplicação está terminado, basta consumir em alguma aplicação *front-end*, utilizando *React, Angular, Vue.JS* ou mesmo *Javascript* puramente. Foi omitido aqui, mas foi implementado uma classe para popular o banco de dados, ou seja, cadastrar alguns dados para podermos testar a aplicação. Então utilizando uma ferramenta de teste para cliente *HTTP* chamada *Postman*, é possível fazer todas as operações dos *endpoints*.

Figura 20 - Requisição *GET* com *Postman*



Fonte: Autores

Repare nas partes grifadas na **figura 20,** está ilustrando uma requisição do verbo ***GET***, na ***URI*** http://localhost:8080/**postagem/1.** Note que o código de retorno é **200 OK**, e que abaixo está o corpo do retorno no formato **JSON**. Analisando o corpo do retorno, vemos que dentro de postagem tem id, título, texto e um objeto do tipo autor. Também possui uma lista de comentários, em que cada comentário possui também seu objeto de autor.

5 CONSIDERAÇÕES FINAIS

Percebemos que sistemas construídos sobre os princípios de *API REST* oferecem mecanismos modernos para que um sistema possa funcionar em vários tipos de aparelhos ou sistemas operacionais. Isso é vantajoso porque há uma variedade de equipamentos, cada um funcionando com seu sistema operacional, na maioria dos casos não é viável construir um sistema para cada aparelho ou SO, pois gera custos demasiados. *API REST* vem sendo cada vez mais requisitadas pelas organizações, como meio de desacoplar os sistemas permitindo múltiplas integrações, ou seja, integração de dispositivos móveis e convencionais a um único sistema.

O ecossistema Spring é uma ferramenta poderosa para um desenvolvedor, é usada amplamente pela comunidade Java. Vale destacar sobretudo a **produtividade** proporcionada por ele, diminuindo o tempo e consequentemente os custos de desenvolvimento. O *back-end* aplicação que foi produzida está contida no github[167].

Este trabalho acadêmico faz parte da disciplina de Métodos e Técnicas de Pesquisa ministrada por Joranaide Alves Ramos.

REFERÊNCIAS

AUGUSTO, Cassio. **Web Services: O que é, como funciona e os protocolos SOAP e REST**. Ninja do Linux, 2019. Disponível em <http://ninjadolinux.com.br/web-services-soap-e-rest/#disqus_thread> Acesso em: 23. out. 2020.

FIELDING, Thomas. **Architectural Styles and the Design of Network-based Software Architectures.** Tese (Doutorado em Filosofia em ciência da informação e computação: Ensino Superior). 180p. University of california, irvine 2000. Disponível em: <https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>. Acesso em: 5 out. 2020.

GOUVEIA, Alexandre. **O que é uma API**. Universidade Positivo**,** 2016. Disponivel em: <https://www.up.edu.br/blogs/engenharia-da-computacao/2016/07/01/o-que-e-uma-api/>. Acesso em: 5 out. 2020.

MOZILLA. **Métodos de requisição HTTP.** MDN web docs, 2020. Disponível em <https://developer.mozilla.org/pt-BR/docs/Web/HTTP/Methods> Acesso em : 04. nov. 2020.

[167] O código completo da aplicação está disponível no GitHub, acesse <https://github.com/RomeuRocha/blog>

PEREIRA, Weberson. **API:** conceito, exemplos de uso e importância da integração para desenvolvedores. Take, 2019. Disponível em : <https://take.net/blog/devs/api-conceito-e-exemplos>. Acesso em: 5 out. 2020.

TILKOV, Stefan. **A Brief Introduction to REST.** Infoq, 2007. Disponível em: <https://www.infoq.com/articles/rest-introduction/>. Acesso em: 7 out. 2020.