Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

LUCAS CORREIA DA SILVA

ESTUDO COMPARATIVO DE DESIGN E DESENVOLVIMENTO DE APIS COM REST E GRAPHQL

Assis/SP

2020

Fundação Educacional do Município de Assis
Instituto Municipal de Ensino Superior de Assis
Campus "José Santilli Sobrinho"

LUCAS CORREIA DA SILVA

ESTUDO COMPARATIVO DE DESIGN E DESENVOLVIMENTO DE APIS COM REST E GRAPHQL

Trabalho de conclusão de Curso de Ciência da Computação do Instituto Municipal de Ensino Superior de Assis – IMESA e a Fundação Educacional do Município de Assis – FEMA, como requisito parcial à obtenção do Certificado de Conclusão.

Orientando: Lucas Correia da Silva
Orientador: Prof. MSc. Guilherme de Cleva Farto

Assis/SP

2020

FICHA CATALOGRÁFICA

S586e SILVA, Lucas Correia da
Estudo comparativo de design e desenvolvimento de apis com rest e graphql / Lucas Correia da Silva. – Assis, 2020.

43p.

Trabalho de conclusão do curso (Ciência da Computação). – Fundação Educacional do Município de Assis-FEMA

Orientador: Me. Guilherme de Cleva Farto

1.Rest 2.GraphQL 3.WevServices

CDD005.12

ESTUDO COMPARATIVO DE DESIGN E DESENVOLVIMENTO DE APIS COM REST E GRAPHQL

LUCAS CORREIA DA SILVA

Trabalho de Conclusão de Curso apresentado ao Instituto Municipal de Ensino Superior de Assis, como requisito do Curso de Graduação, avaliado pela seguinte comissão examinadora:

Orientador: Prof. MSc. Guilherme de Cleva Farto

Examinador: Prof. Esp. Célio Desiró

Assis/SP
2020

RESUMO

Com a necessidade de escalabilidade, agilidade e segurança nos projetos desenvolvidos para a *web*, foram sendo desenvolvidas ferramentas que suprissem essas necessidades, assim se tornou cada vez mais popular os *webservices* e alguns *designs* em arquiteturas. Hoje a mais popular no mercado é a REST que vem em grande parte das aplicações e revolucionam a visão de serviços, tornando as chamadas sintetizada e diretas. Com o passar do tempo surgiu outro *design* em arquitetura, o GraphQL, que hoje está passando a ser um dos principais concorrente do REST. Se atentando a está realidade o presente estudo tem o intuito de analisar não só qual *design* de arquitetura é mais ágil mas para quais finalidades cada uma atende como melhor opção de implementação.

Palavras Chave: *REST, GraphQL, WebServices*

ABSTRACT

With the need for scalability, agility and security in the projects developed for the web, tools were developed to meet these needs, so webservices and some architectural designs became increasingly popular. Today, the most popular on the market is REST, which comes in most applications and revolutionizes the service vision, making calls synthesized and direct. As time went by, another design in architecture, GraphQL, which today is becoming one of REST's main competitors. Given this reality, this study aims to analyze not only which architecture design is more agile but for which purposes each serves as the best implementation option.

Keywords: *REST, GraphQL, WebServices*

LISTA DE ILUSTRAÇÕES

LISTA DE TABELAS

SUMÁRIO

1. INTRODUÇÃO

A web em seus primórdios era apenas utilizada por universidades, assim uma se comunicava com a outra, compartilhando informações e se criou um compartilhamento limitado, com o passar dos anos e a evolução não do dos softwares mas sim dos hardwares, começou a se tornar possível a comunicação em massa e o inicio da popularização dos computadores de mesas, famosos desktops, começando a ter a preocupação dos desenvolvedores com seus usuários. A web nunca parou de crescer e sua utilização também não, um grande marco foi a miniaturização dos aparelhos e a sua popularização, como os famosos smartphones, que mantém o usuário cada vez mais tempo online, assim gerando um grande problema para os programadores, que é a forma mais eficaz de se compartilhar os dados, com menos informação e o mais preciso possível.

As WebServices são protocolos de comunicação web, mais popularmente HTTP e XML, eles têm o principio a comunicação de aplicações através da internet, promovendo a interoperabilidade entre informações que circula numa organização das diferentes aplicações. Com as WebServices e toda esta evolução é comum que se tenha criado varias linguagens para desenvolver web, ao ser notado este fato, houve a necessidade de se criar algo para solucionar a comunicação entre serviços de linguagens distintas, sendo assim se criou as APIs.

Com a popularização dos aparelhos celulares e outros equipamentos portáteis, tornou-se necessário que as aplicações fossem acessíveis a hardwares mais limitados, assim começou a preocupação em tornar os serviços cada vez mais sintetizados e diretos, consequentemente sendo mais rápidos possibilitando que tais apetrechos tivessem um bom desempenho. Nos anos 2000 Roy Fielding em sua tese de doutorado criou um design em arquitetura denominado Representationa State Transfer ou mais popular por sua sigla REST, esta que se utiliza dos métodos HTTP para se comunicar com o servidor, se utilizando de documentos em JSON, XML ou algum outro formato para fazer essa transição, nestes documentos contem informações, como se algo foi alterado ou um conjunto de links de hipertexto de conjuntos relacionados. Em resumo o REST prioriza informações simples, porem que sejam eficazes. Segundo Roy Fielding:

> *A REST é pretendida como uma imagem do design da aplicação se comportará: uma rede de websites (um estado virtual), onde o usuário progride com uma aplicação selecionando as ligações (transições do estado), tendo como resultado a página seguinte (que representa o estado seguinte da aplicação) que está sendo transferida ao usuário e apresentada para seu uso.*

Alguns anos depois o Facebook inc, em 2012 desenvolveu uma arquitetura de query language, chamada GraphQL e só obteve o uso publico em 2015, possibilitando desenvolver APIs que traga apenas os dados necessários para o usuário, reduzindo a sobrecarga nos serviços, aprimorando a experiência de tais, principalmente nos dispositivos moveis.

1.2 OBJETIVO GERAL

O objetivo geral deste trabalho é realizar uma pesquisa comparativa entre os designs de arquiteturas em WebServices, possibilitando elencar de tanto em REST quanto em GraphQL, seus pontos negativos, positivos, melhor cenário para a utilização e eficiência.

1.3 OBJETIVOS ESPECÍFICOS

Pretende-se com este trabalho o desenvolvimento de uma pesquisa exploratória que deixará mais claro a eficácia de cada design baseado em APIs.
Para a analise dos respectivos tópicos será utilizado a linguagem de programação Python, juntamente com seu framework, Django, que são ferramentas utilizadas para o desenvolvimento *web.*

. pesquisar sobre implementação web baseado em REST
. pesquisar sobre implementação web baseada em GraphQL
. pesquisar e analisar os métodos HTTP
. pesquisar e analisar a utilização de arquivos JSON
. especificar um estudo de caso:
    . situar-se sobre qual tipo de aplicações a serem desenvolvidas
    . definição dos casos de uso

. definir as regras de negócio
. realizar a implementação
. testes e validação
. resultados

1.4 JUSTIFICATIVAS

Nos últimos anos diversas ferramentas de desenvolvimento como frameworks e design de arquiteturas, que estão tornando o mercado cada vez mais variado e aberto a possibilidades inusitadas. Observando a eficiência do GraphQL, principalmente com o Facebook inc e a popularização do REST no desenvolvimento de diversas empresas, resolveu-se fazer uma comparação pois utilizam formas variadas de atender o enfoque do usuário e otimizar sua experiencias, principalmente no mercado mobile.

1.5 MOTIVAÇÃO

Este projeto de pesquisa tem o intuito de abordar uma tecnologia nova em ascensão no mundo da web, achei muito interessante por se tratar realmente de algo muito novo e que futuramente devido ao seu potencial previamente visto, tem o potencial de conquistar uma fatia significante do mercado.
Ambas as ferramentas são otimizadas de sua maneira, gerando dúvidas em quais pontos uma ou outra se sobressai.
Outro fator é o interesse em desenvolvimento web, por se tratar de algo novo e promissor tornou-se vistoso, considerando um tema adequado como tese.

1.6 PERSPECTIVAS DE CONTRIBUIÇÃO

Este projeto se tratou de uma tecnologia em ascensão e que não se encontra muitas informações ou outros projetos como este, se tornando uma grande ajuda para futuros interessados e curiosos. Ter enfoque no potencial de mercado agregado a cada

design de arquitetura estudada a partir de embasamento literário, tornando uma fonte de informações, já que esta monografia será disponibilizada em instituições de ensino.

1.7 METODOLOGIA DE PESQUISA

Alcançado as conclusões necessárias deste trabalho por meio de pesquisa teórica, se baseando em livros, monografias, artigos científicos, teses guias práticos e técnicos e fontes digitais confiáveis, materializando os fins, permitindo a implementação e comprovar as conclusões sobre a utilização dos *designs* de arquitetura em questão.
Primeiramente levantando informações sobre ambas as estruturas em estudo, possibilitando elencar suas características e a forma que são aplicadas em mercado produtivo, após a conclusão desta etapa foram implementadas duas APIs, testes e confirmação dos pontos levantados.

2. WEB SERVICES E APIS

Neste capítulo será descrito as características dos Web Services e das APIs, elencando suas características, criação, utilização e origem. Embasando o funcionamento das ferramentas estudadas, dando enfoque na diversidade de implementações, principalmente com as APIS.

2.1 WEB SERVICES

A partir da popularização da internet no final dos anos 90, foi avistada uma oportunidade de criar uma comunicação entre sistemas de informação, até então todos os dados da máquina ficavam em suas redes locais, com a necessidade de tráfego de dados foi assim criado os *Web Services*.

Com o tempo as possibilidades de desenvolvimento web se tornaram vastas, várias linguagens e *frameworks*, se criando a necessidade de haver um mecanismo que conseguisse integrar todos esses sistemas. Os *Web Services* têm por característica permitir que várias linguagens sejam executadas em diversas plataformas, possibilitando que transfiram e recebam informações uma das outras, se criando uma das tecnologias de computação distribuída, mais abrangente até então.

O W3C (2004) define *web services* da seguinte maneira:

> *Um Web Service é um sistema de software desenvolvido para permitir interações maquina- maquina através de uma rede. É uma interface descrita para ser consumida por maquinas (WSDL). Outros sistemas interagem com o Web Service através de mensagens SOAP, geralmente enviadas através de HTTP em conjunto com outros padrões relacionados à web (W3C, 2004).*

Cada aplicação ao consumir estes serviços recebe dados encapsulados, sendo assim uma aplicação não tem detalhes internos uma das outras, aprimorando os códigos, devido a facilidade do reaproveitamento do mesmo, tornando os sistemas cada vez mais sintetizados. Podemos tomar como exemplo o *PayPal*, diversas aplicações se utilizam de sua funcionalidade de efetuar pagamentos, porém estas aplicações não tem acesso aos dados cruciais do sistema.

Além destes benefícios, os *Web Services* possibilitam o desenvolvimento de aplicações em camadas, onde o desenvolvedor pode modificar ou alterar

completamente uma das camadas sem prejudicar as outras. Também, é utilizado para integrar sistemas legados a novos sistemas em desenvolvimento, proporcionando o reaproveitamento das regras de negócio e códigos já em produção.

2.2 APIS

As APIs (*Application Programming Interface*) funcionam de maneira similar ao processo de um restaurante. Por exemplo, o garçom proporciona uma lista de itens que estão disponíveis; os clientes fazem uma análise e solicitam o pedido, que é levado até a cozinha, onde este será preparado. Ao termino, o garçom retornará com o pedido pronto. Seguindo esta analogia, podemos identificar o ciclo de uma API.

Os clientes são como os usuários que realizam uma solicitação à aplicação web, representada pelo garçom, que pega a requisição feita e leva até o servidor, representado pela cozinha, onde será preparado a informação em forma de JSON ou XML.

Desta forma, ao utilizar-se dos arquivos JSON ou XML, permite que haja uma integração de várias linguagens de programação em um mesmo sistema, proporcionando uma flexibilidade durante o período de desenvolvimento de aplicações *web*.

De modo geral, as APIs são conjuntos de rotinas e padrões de programação definidos pelo desenvolvedor. Sua utilização vai desde os sistemas operacionais quanto softwares web, possibilitando que uma aplicação utilize funcionalidades de outras sem que se conheça a forma como foram desenvolvidas.

Um exemplo da utilização deste padrão, são os sistemas de pagamento, como a Pag Seguro ou *Pic Pay*, diversos sites se utilizam de seus serviços, porém sua execução é totalmente independente das outras funcionalidades.

Além disso, as APIs servem para o multiversionamento de aplicações, antes era necessário a criação de servidores múltiplos para cada plataforma, uma linha de programação para web, outra para os desktops e outra para os mobiles, com a utilização desta ferramenta é possível com uma única API manipular os dados e gerar as requisições em um servidor, para todas as aplicações.

Na figura 1 se exemplifica a conectividade obtida pela utilização de APIs, permitindo sua aplicação em ampla gama de operações, ficando em evidencia as utilidades possíveis deste estilo de desenvolvimento, algo importante com a atual necessidade de integração:

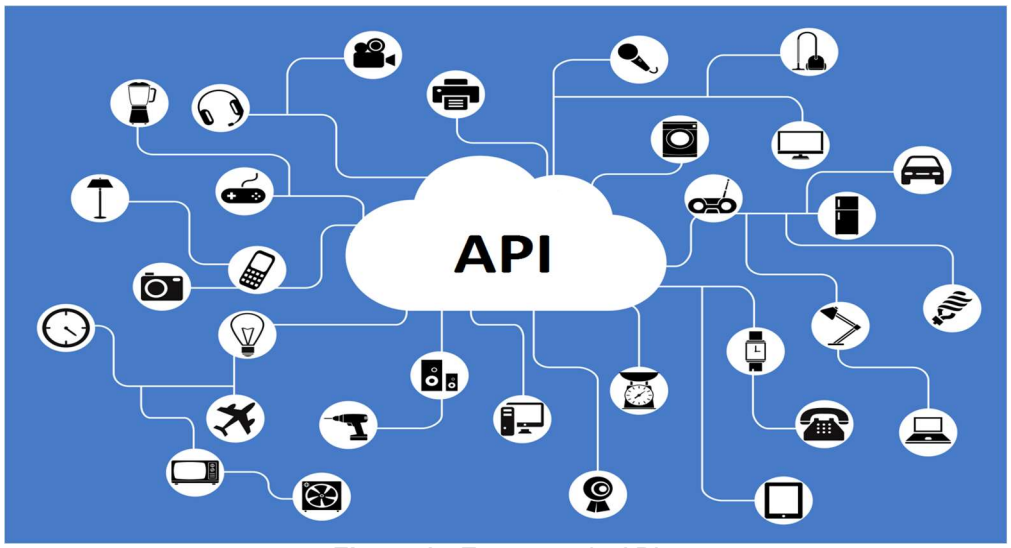

Figura 1 - Esquema de APIs.

Fonte:https://encrypted-tbn0.gstatic.com/images?q=tbn%3AANd9GcRoYxYuvfR895K7cQYNZEMXCDuHMnJfwvIbvg&usqp=CAU

Algumas empresas adotam durante o seu desenvolvimento o API *First*, que é arquitetar a camada de API antes do desenvolvimento do sistema, com isso mais de um sistema consegue ser desenvolvido em cima de uma mesma condição, garantindo assim competitividade no mercado e melhor custo beneficio, devido a quebra do tempo de desenvolvimento.

A utilização de APIs potencializa a escalabilidade da aplicação, caso necessário alguma modificação, basta adicionar uma nova API dentro de sua camada específica, permitindo que os programadores tenham mais liberdade em seus desenvolvimentos e facilitando a manutenção de seus sistemas. Essas modificações ocorrem por diversos motivos, como a mudança do enfoque do cliente ou mais especificamente uma regra de negócio mais eficiente.

Contudo, as APIs trazem benefícios imprescindíveis para o atual mercado tecnológico, possibilitando a integração entre linguagens de programação totalmente diferentes e tornando o desenvolvimento mais ágil. E também, proporcionando segurança e integridade aos dados de cada empresa.

3. REST E RESTFULL

REST é um design de arquitetura desenvolvido por Roy Fielding em sua tese de doutorado em 2000, um dos principais criadores do protocolo HTTP, esta se utiliza deste protocolo para gerenciar suas comunicações. Inicialmente o REST era considerado uma evolução da arquitetura HTTP, porém muitos desenvolvedores observaram o potencial deste design nos *Web Services*, tendo por objetivo integrar as aplicações através da Web, tornando uma alternativa à tecnologia utilizada até então, SOAP.
Este design de arquitetura tem um conjunto de princípios para considerar uma aplicação RESTfull, que utiliza e respeita todas as métricas de sua estrutura.
Essas métricas são:

- Ele usa o protocolo HTTP (verbos,*accept headers*, códigos de estado HTTP,*Content-Type*) de forma explícita e representativa para se comunicar. URIs são usados para expor a estrutura do serviço. Utiliza uma notação comum para transferência de dados como XML ou JSON.

- Não possui estado entre essas comunicações, ou seja, cada comunicação é independente e uniforme (padronizada) precisando passar toda informação necessária.

- Ele deve facilitar o cache de conteúdo no cliente.

- Deve ter uma definição clara do que faz parte do cliente e do servidor. O cliente não precisa saber como o servidor armazena dados, por exemplo. Assim cada implementação não depende da outra e se torna mais escalável.

- Permite o uso em camadas também facilitando a escalabilidade, confiabilidade e segurança.

- Frequentemente é criado com alguma forma de extensibilidade

Os atributos gerados por cada aplicação dentro de REST é denominado como recursos, sendo uma abstração de um tipo de informação, onde cada recurso possui seu identificador único, utilizado pelo sistema REST para diferenciar qual deve ser manipulado em um determinado momento. Caso, um sistema manipule 6 ou 7 recursos diferentes, a forma estrutural do REST identifica qual deve ser executado de acordo com seus identificadores, que devem ser informados em sua requisição.

A URI (*Uniform Resource Identifier*), é um padrão utilizado pela *web*, onde o REST se aproveita das URLS para poder solicitar seus recursos. Por exemplo, http://server.com.br/produtos, que poderia ser para exibir a tela de produtos. As URIs devem utilizar o domínio do sistema desenvolvido e assim denominando os recursos intuitivamente. Dessa forma, facilitando a utilização do usuário e também diminuindo a quantidade de documentações extensas.

As URIs e os protocolos HTTP são utilizadas em conjunto. Para que no momento de uma requisição, a API consiga interpretar o que deve ser feito com aquela chamada.

A seguir a tabela com os principais métodos HTTP:

GET	Obter os dados de um recurso.
POST	Criar um novo recurso.
PUT	Substituir os dados de um determinado recurso.
PATCH	Atualizar parcialmente um determinado recurso.
DELETE	Excluir um determinado recurso.
HEAD	Similar ao GET, mas utilizado apenas para se obter os cabeçalhos de resposta, sem os dados em si.
OPTIONS	Obter quais manipulações podem ser realizadas em um determinado recurso.

Tabela 1 - Métodos HTTP.

Os métodos mais utilizados são o GET, POST, PUT e DELETE, porém se houver a necessidade existem outros à serem utilizados. Através deste recurso o design de arquitetura se torna intuitivo e de fácil implementação, se aproveitando de recursos já existentes, potencializando suas funcionalidades, possibilitando a geração de requisições coerentes e concretas dentro de um sistema totalmente escalável.

De acordo com Roy Fielding:

> *No sistema em camadas, o mesmo é divido em camadas, onde cada camada conhece apenas a interface da camada superior. As camadas intermediarias podem ser utilizadas para melhorar a escalabilidade do sistema, permitindo o balanceamento de carga de serviços, através de múltiplas redes. (FIELDING,2000)*

O cache previne o desperdício de banda, armazenando dados enviados anteriormente ao usuário reprimindo o reenvio. Estas informações ficam gravadas no proxy HTTP, de tal forma que algumas páginas não precisam serem solicitas diretamente ao servidor, pois podem ficar mais próximas ao usuário, assim eliminando algumas ou totalmente a interação cliente e servidor, tornando a aplicação mais veloz, por evitar o tráfego ocorrido dentro do servidor.

4. GRAPHQL

O GraphQL é uma ferramenta recente no mercado, criada em 2012 e apenas utilizada pelo Facebook. Em 2015, foi disponibilizado as primeiras versões para uso, com o intuito de melhorar a experiência do usuário mobile.

> *Em 2012, o Facebook decidiu que era necessário refazer suas aplicações nativas mobile. O conjunto de tecnologias não era o suficiente para lidar com a quantidade de requisições, o que impactava na performance, e assim o aplicativo não atendia às exigências do usuário. (TEAM, 2016)*

O GraphQL é um interpretador de *Query Language*, possibilitando de maneira escalável modificar, remover e adicionar. O objetivo é trabalhar com as consultas dos clientes de maneira eficaz, facilitando o processamento e melhorando as respostas.

> *O GraphQL é uma linguagem de consulta de dados, além de um interpretador, criada pelo Facebook para trabalhar com APIs de forma alternativa. Seu objetivo é fornecer uma descrição completa e compreensível de dados disponíveis em interfaces de aplicação, permitindo que clientes façam consultas de dados que desejam trabalhar de forma precisa. (TEAM, 2016)*

Atualmente, disseminada no mercado, várias linguagens de programação aceitam implementações em GraphQL, como o Graphene, um framework em Python. E, também, o Spring que é um framework em Java. Encontrada em diversas plataformas de maneira volátil podendo ser trabalhada tanto com microsserviços quanto com solicitações no banco de dados, por mais que não seja o intuito.

Antes do surgimento do GraphQL, o mercado na nova ascensão dos *Web Services*, era dominado pelo design de arquitetura REST e suas aplicações RESTFULL. Porém, conta com alguns detalhes que torna seu desenvolvimento maçante e pouco escalável. Caso, seja necessário fazer uma solicitação de quantas pessoas curtiram a foto de uma celebridade x, isso em REST seria feito com um verbo HTTP, GET, podendo retornar um por vez, caso haja milhões de curtidas esta solicitação teria de ser feita essas mesmas milhões de vezes, o GraphQL por se tratar de uma *Query Language*, poderia resolver isso com uma única consulta, retornando assim a quantidade dos dados desejado, tendo maior escalabilidade e produtividade.

Durante uma solicitação o GraphQL através de uma Query gera uma *request* e seu *response* é dado através de um arquivo JSON, esta resposta geralmente é um espelho do que foi solicitado na Query, desta maneira ilustrada na figura 2:

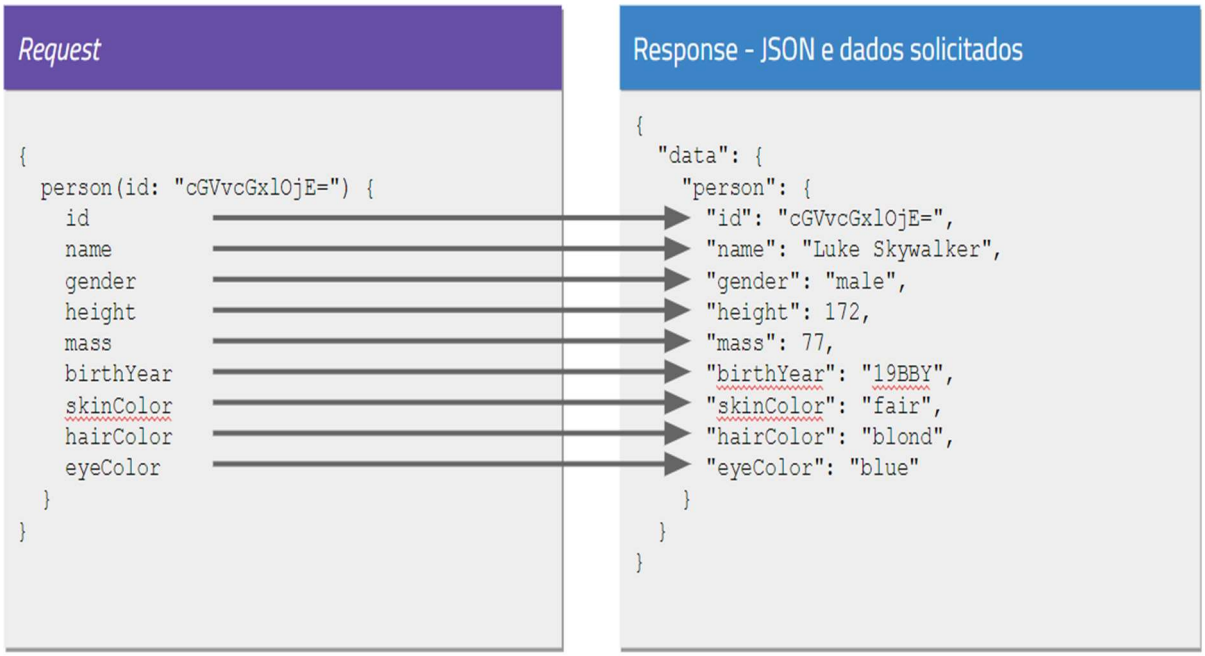

Figura 2 - Representação Request/Response formato JSON

Fonte: http://3k64nh47gxyj39ud4k2tc04b-wpengine.netdna-ssl.com/wp-content/uploads/2017/10/graphql04.png

Desta forma, o sistema irá obter apenas o que é necessário em cada *Request* gerada, evitando o envio de informações desnecessárias. Caso, não tenha todas as informações necessárias para realizar a *Request*, ela pode ser feita apenas com trechos. Por exemplo uma pesquisa, sabendo-se o tipo e parte do nome desejado, é possível realizar um *search*. Para exemplificar, na figura 3, podemos observar a estrutura utilizada em um *search* na estrutura de GraphQL:

Figura 3 - Search com um trecho de texto.

Fonte: https://3k64nh47gxyj39ud4k2tc04b-wpengine.netdna-ssl.com/wp-content/uploads/2017/10/slide32-1024x441.png

O GraphQL permite também retornar apenas fragmentos de alguma classe, por exemplo, em um cadastro, é preciso acessar somente o nome e a idade, através de uma Query, é solicitado os itens de desejo para o processamento.

Na figura 4 observamos a abstração de fragmento de dado em uma classe especifica, assim referenciando os campos desejados e sendo retornado apenas itens solicitados:

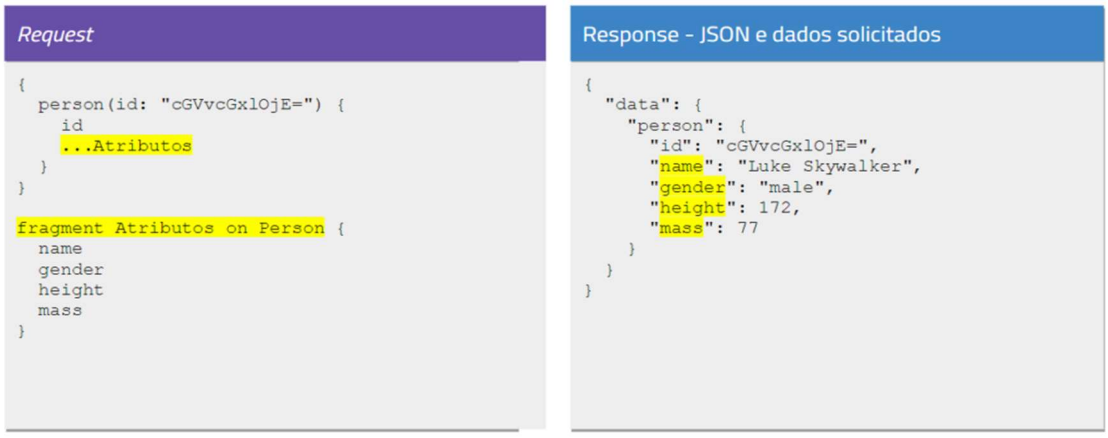

Figura 4 - Representação de resgate de trecho de uma classe.

Fonte: https://3k64nh47gxyj39ud4k2tc04b-wpengine.netdna-ssl.com/wp-content/uploads/2017/10/slide31-1024x440.png

GraphQL possui três características principais dentro de sua estrutura. A primeira característica seria o cliente especificar quais dados ele precisa, como em uma

pesquisa, essas informações são enviadas ao servidor, que retorna os dados mais relevantes. Sua segunda característica e facilitar a agregação de múltiplas fontes, o GraphQL proporciona uma manipulação dos dados de forma fluída e precisa, integrando múltiplos dispositivos com funcionalidades e necessidades distintas, como os smartphones, que necessitam de uma carga de dados mais sucinta e precisa para potencializar seu tempo de resposta. A terceira característica é utilizar um sistema de tipos para descrever seus dados, ao criar uma solicitação o GraphQL nomeia de qual classe ou tipo esse dado pertence, assim facilitando na hora de processar essa requisição, tornando assim a resposta rápida e precisa.

O GraphQL tem um esquema para cada tipo de dado, onde se encapsula seus atributos, ao gerar uma query, mais de um tipo de dado pode ser retornado. Para tornar mais visível este raciocínio, na figura 5 contem a estrutura de entrada de uma *query* referente a mais de um tipo de dados e sua forma de retorno:

Figura 5 – Exemplo de schema em GraphQL.

Fonte:https://lh3.googleusercontent.com/proxy/Edv3cbUEu2N-nN9zlBb9cyBR78blnPKgSA701zw8twaJxzekUpamkJRUWhh2zDYN_BUIvM-Urp4sKMZPMn4NJMrmQytJaef4lnSb9mVVGqBXORxiGX0f1f_HszY4PjGQWWzbHFBIow_4oN-aCnuMuS7RsbWF

Como podemos observar na figura 5, existe o *type People,* onde se encontra a representação das informações de cada classe a serem consultadas, neste caso *People* é o *type Query,* dentro deste campo se encontra as formas possíveis de resposta a cada solicitação, um que retorna um item da classe *People* com um

identificador único não nulo (*people(id: ID!): People*) e outro que retorna uma lista de itens da classe People (*peoples: [People]*).

Este processo é realizado pelos *resolvers*, uma funcionalidade da API que faz o tratamento da solicitação, podendo ser as *Querys* padrões ou uma *Mutation*, que é o nome dado a uma solicitação que altera algo já existente no sistema, responsável pelos deletes e *updates*, após identificar qual o método a ser tratado, faz a busca do dado solicitado e resultando em uma resposta, os *resolvers* conseguem buscar dados dentro do sistema ou nos bancos de dados disponíveis na aplicação.

Na figura 6 podemos observar a estrutura do ciclo de execução de uma requisição *GraphQL*, ao utilizar suas *Querys*, chamadas que permitem obter dados, ou as *Mutations,* estrutura responsável por qualquer tipo de modificação ou tratativa de dados:

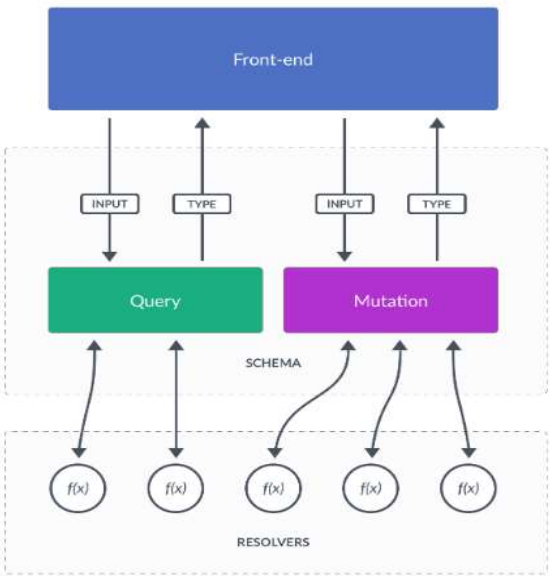

Figura 6 – Estrutura de uma solicitação GraphQL.
Fonte: https://miro.medium.com/max/2446/1*O8mWd4Hq4_3c9nM9VoRAIg.png

O GraphQL é utilizado para potencializar a comunicação entre o servidor pela sua forma direta e eficiente de buscar os dados, tendo em vista como o mercado aderiu ao REST, é difícil de uma hora para outro substituir um padrão tão robusto e popular. Sua utilização vem se popularizando através do Apollo, um framework construído pela equipe da Meteor, que desenvolveu ferramentas para trabalhar com o GraphQL. Por

exemplo, o graphql-tools, encontrado dentro do framework, que facilita a criação de schemas executáveis e o apollo-client, que tem o intuito de ser totalmente preparado para qualquer servidor ou framework UI.

A seguir uma tabela de algumas empresas que utilizam GraphQL no Brasil:

Nome	Website	Tecnologias Relacionadas	Cidade
Convenia	https://convenia.com.br/	Vue, Vuex, GraphQL, Apollo, NodeJS, PHP, MySQL, DynamoDB	São Paulo - SP
Decision6	https://decision6.com/	Vue, GraphQL, Apollo, Node, Restify, Go, PHP, PostgreSQL	Rio de Janeiro - RJ
Entria	https://entria.com.br/	React, React Native, GraphQL & Relay Modern	São Paulo - SP
Ignus	https://ignusdigital.recruitee.com//	React, React Native, GraphQL, Apollo, Node	Rio de Janeiro - RJ
In Loco	https://inloco.com.br/	React, React Native, GraphQL, Node	Recife - PE
LogusTech	https://logus.tech/	Vue, Vuex, GraphQL, Node, Apollo, Flask, MongoDB, MariaDB	Recife - PE
Quanto	https://contaquanto.com.br/	React Native, Redux, GraphQL, NodeJS, Relay, Jest, CockroachDB	São Paulo - SP

Tabela 2 – Empresas que utilizam GraphQL no Brasil.

O intuito do GraphQL é trazer as facilidades de uma *Query Language*, potencializando a geração das *Requests* e consequentemente tornando as *Responses* mais precisas, proporcionando um sistema mais robusto e que consiga se sair bem durante uma alta gama de solicitações.

5. DESENVOLVIMENTO DE APIS

Com o objetivo de identificar as funcionalidades dos designs de arquitetura, GraphQL e REST, foram desenvolvidas duas APIs, ambas na linguagem de programação Python. Uma desenvolvida com o Django Rest Framework, proporciona um desenvolvimento ágil de APIs REST e a outra sendo desenvolvida com Graphene, framework criado para desenvolvimento em GraphQL, em ambos os casos o projeto foi estruturado com o framework Django.

As ferramentas utilizadas foram o Postman e o Altair para testar as solicitações de ambas as APIs, testando suas funcionalidades de GET, POST, PUT e DELETE, observando o retorno de informações e a diferença de ambas.

5.1 DJANGO

O Django é um framework de código aberto escrito em Python, criado inicialmente por Simon Wilison e Adrian Hotovaty, este com o intuito de gerenciar sites jornalísticos americanos, o seu nome foi uma homenagem ao músico Django Reinhardt um guitarrista de jazz francês, considerado um dos maiores e mais influentes guitarristas de todos os tempos.

Devido a necessidade de eficiência e agilidade, pelos prazos das redações serem curtos, o Django foi criado em uma estrutura com a respectiva sigla MTV (*Model, Template, View*), permitindo a utilização do conceito DRY (*Don't Repeat Youtself*), este se baseia em facilitar a reutilização de código, a partir desta estruturação o projeto mesmo que visualmente esteja fragmentado, tem uma interdependência entre suas partes, os projetos ficam mais amigáveis aos olhos, potencializando a organização do projeto e o tornando mais legível. Na figura 7 ilustra a estrutura de um projeto em *Django,* sendo bem visível a forma de centralização de suas configurações principais e de como é organizado os arquivos de um projeto:

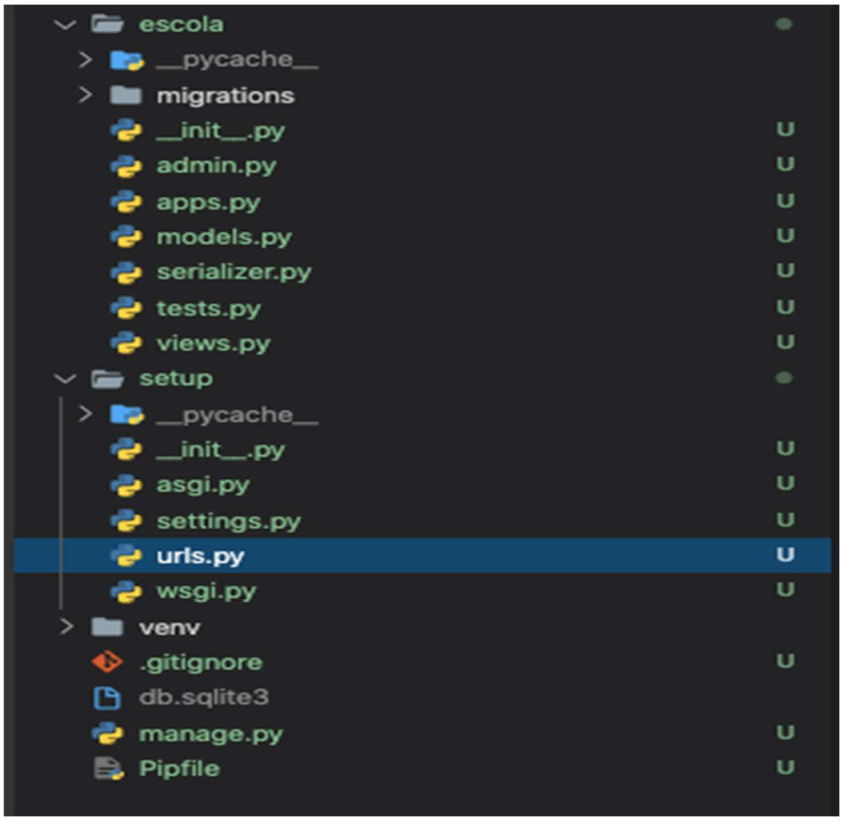

Figura 7: Estrutura projeto Django.

Por meio da imagem conseguimos observar como é estruturado um projeto Django, ele segue de uma aplicação central, onde se encontra os arquivos de configurações do projeto, e logo após de outros *apps*, todos dentro de um mesmo projeto, ambos com funcionalidades próprias, porém lincados e se intercomunicando em prol de uma única aplicação.

Django proporciona um sistema muito seguro, todo o seu desenvolvimento foi focado em prevenir os ataques mais comuns encontrados na web. E, também, com sua estruturação deixa o desenvolvimento mais flexível e permite a implementação de correções ou novas implementações de forma menos trabalhosa, já que a estrutura MVT, permite a criação de um único *app,* podendo fazer parte de qualquer código de acordo com a necessidade.

A figura 8 observamos a estrutura de inclusão dos *apps* existentes no projeto, ao serem incluídos em uma tupla, possibilita utilizar todas as suas funcionalidades e manipular os seus modelos:

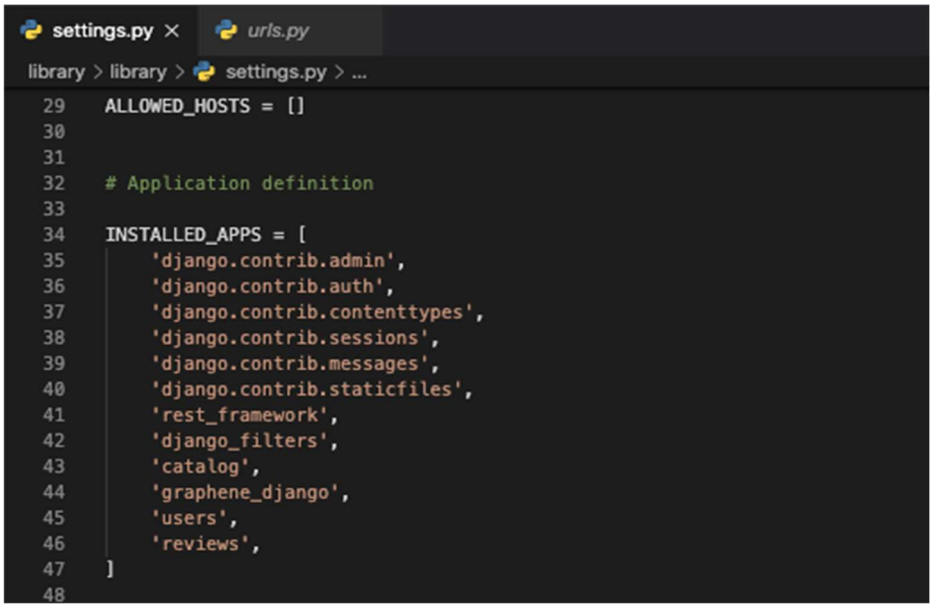

Figura 8: settings.py.

Neste trecho de código se encontra o local no arquivo settings.py, onde se engloba as configurações centrais do projeto, como as configurações do servidor, dos *apps*, entre outras funcionalidades que necessitem estarem em atividade. Esse trecho, apresenta a configuração de todos os *apps* que estão em atuação. Ao criar um novo *app,* é necessário incluí-lo nesta lista, para que suas funcionalidades sejam reconhecidas pela aplicação.

O Django trás consigo um ambiente de administração de dados, permitindo o desenvolvedor sem nenhuma linha de código, consiga gerenciar todos os dados disponíveis na implementação.

Na figura 9 fica evidenciado o layout da pagina administrativa inclusa no *framework* Django:

Figura 9: Pagina Admin.

5.2 DJANGO REST

O Django Rest inicia a aplicação da mesma forma de uma aplicação Django. A principal diferença é a utilização dos *serializers* e as *viewssets*, responsáveis pelo tráfego das urls e a possibilidade de se criar uma API REST.

Os *serializers* são necessários para fazer a serialização e a desserialização das instâncias disponíveis no desenvolvimento, no caso da imagem, temos ela feita sobre o modelo de Aluno, Curso e Matricula, em uma representação JSON ou XML. Existe uma validação dos dados para realizar o processo de desserialização trazendo de volta em tipos complexos. Utilizando a figura 10 observamos a estrutura emposta pelo framework para estruturar os seus *serializers*, passando os modelos de dados e os campos a serem abordados:

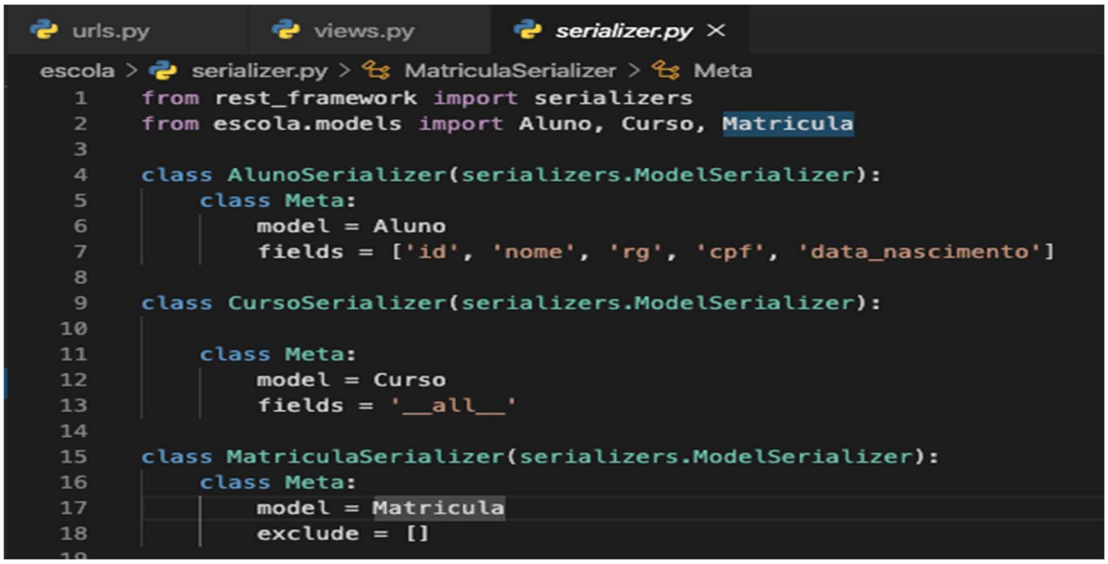

Figura 10: serializers.py.

Nas *viewsets* é definido as *querysets,* consultas das instâncias dos modelos, onde são mapeadas as entradas dos usuários primitivos, para uma instância do modelo. A partir dessas instâncias os *serializers* atuam, podendo fazer todos os seus processos. Esta *view* criada é passada como argumento para a seção onde se encontra as urls. Na figura 11 se impõem a estrutura de ordenação das classes responsáveis pela *viewsets* e os campos responsáveis pela inicialização e configuração de funções posteriormente utilizadas para a manipulação dos dados:

urls.py views.py

escola > views.py > AlunosViewSet

```python
from rest_framework import viewsets, generics
from escola.models import Aluno, Curso, Matricula
from escola.serializer import AlunoSerializer, CursoSerializer, MatriculaSerializer
from escola.serializer import ListaMatriculasAlunosSerializer, ListaAlunosMatriculadosCursoSerializer
from rest_framework.authentication import BasicAuthentication
from rest_framework.permissions import IsAuthenticated

class AlunosViewSet(viewsets.ModelViewSet):
    """Exibindo todos os alunos e alunas"""
    queryset = Aluno.objects.all()
    serializer_class = AlunoSerializer
    authentication_classes = [BasicAuthentication]
    permission_classes = [IsAuthenticated]

class CursosViewSet(viewsets.ModelViewSet):
    """Exibindo todos os cursos"""
    queryset = Curso.objects.all()
    serializer_class = CursoSerializer
    authentication_classes = [BasicAuthentication]
    permission_classes = [IsAuthenticated]


class MatriculaViewSet(viewsets.ModelViewSet):
    """linstando todas as matriculas"""
    queryset = Matricula.objects.all()
    serializer_class = MatriculaSerializer
    authentication_classes = [BasicAuthentication]
    permission_classes = [IsAuthenticated]

```

Figura 11: views.py.

Com os passos concluídos é definido as rotas, que são os caminhos onde a API utilizará pra realizar todos os métodos. Com uma das funcionalidades do *framework*, ao fazer o registo da rota é liberado o acesso para utilizá-lo e fazer executar as GETs, POSTs, PUTs e DELETEs. Assim na figura 12, analisamos a maneira de inicialização das Querys e definição de sua rota, na realização do seu registro e manipulação nas urls:

urls.py

setup > urls.py > ...

```python
from django.contrib import admin
from django.urls import path,include
from escola.views import AlunosViewSet, CursosViewSet, MatriculaViewSet
from escola.views import ListaMatriculasAluno, ListaAlunosMatriculos
from rest_framework import routers

router = routers.DefaultRouter()
router.register('alunos', AlunosViewSet, basename='Alunos')
router.register('cursos', CursosViewSet, basename='Cursos')
router.register('matriculas', MatriculaViewSet, basename='Matriculas')

urlpatterns = [
    path('admin/', admin.site.urls),
    path('', include(router.urls) ),
    path('aluno/<int:pk>/matriculas/', ListaMatriculasAluno.as_view() ),
    path('curso/<int:pk>/matriculas/', ListaAlunosMatriculos.as_view() )
]

```

Figura 12: urls.py.

Com todos esses passos executados durante o desenvolvimento, pode-se utilizar da API livremente. Ao instalar o *framework*, é liberado o acesso a uma área que proporciona a manipulação dos dados, neste caso fiz a utilização de duas funcionalidades chamadas IsAutenticated e BasicAutenticated, que permitem a implementação de segurança na aplicação. Ao adicionar essa funcionalidade em uma *viewset* específica, ela só será exibida se houver a autenticação do usuário.

O *layout* da pagina padrão contida no *framework,* se exemplifica na figura 13. Nesta pagina fica contida as urls de cada *app* disponível na aplicação e as formas de manipulação habilitada ao acesso do usuário:

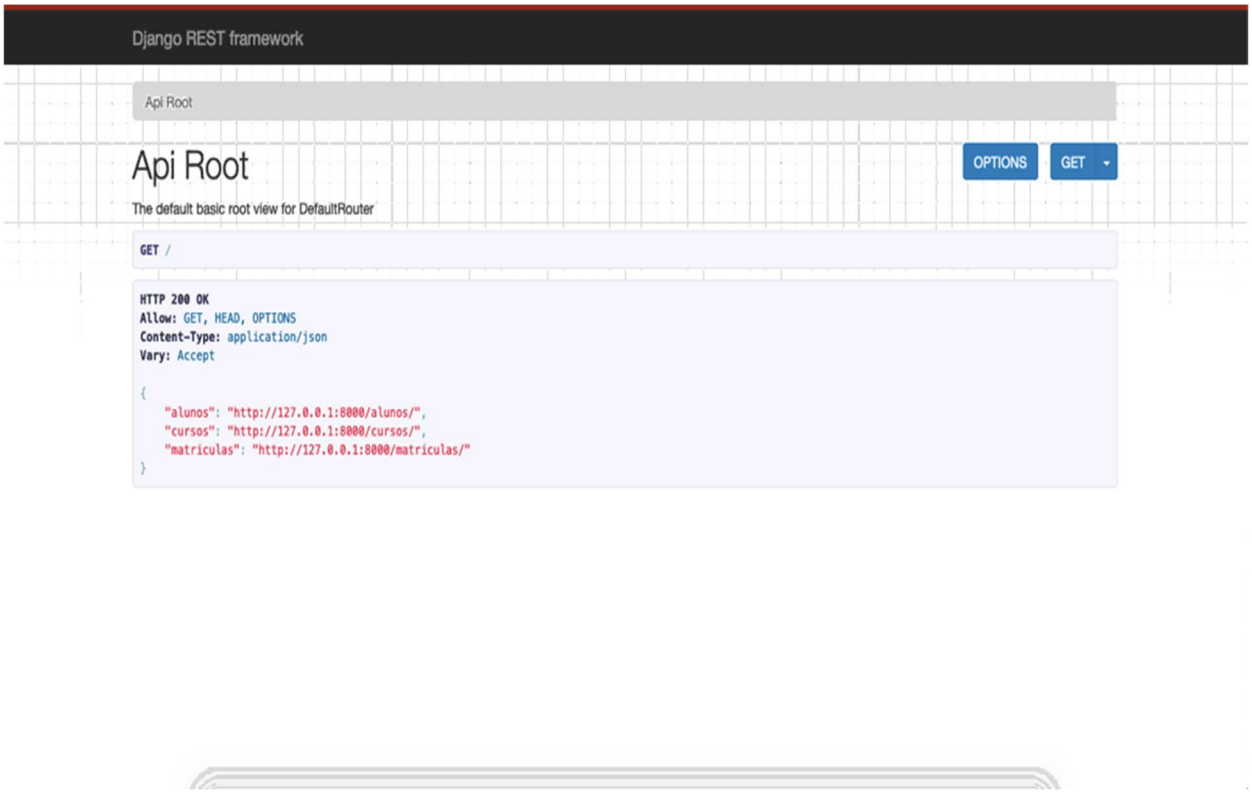

Figura 13: Home Page API REST.

Como descrito anteriormente, a API em REST faz o uso de arquivos em JSON ou XML para fazer o transporte de suas requisições. As imagens a seguir conterão alguns exemplos das formas dos verbos de manipulação de dados e suas respostas em JSON.

Na figura 14, com a utilização do *Postman*, ferramenta utilizado para analisar as requisições a partir do seu acesso em forma de urls, será descrito o modelo do verbo HTTP, GET, proporcionando a visualização de suas estrutura de acesso e sua resposta. No caso da figura de exemplificação, temos um arquivo no formato JSON como retorno:

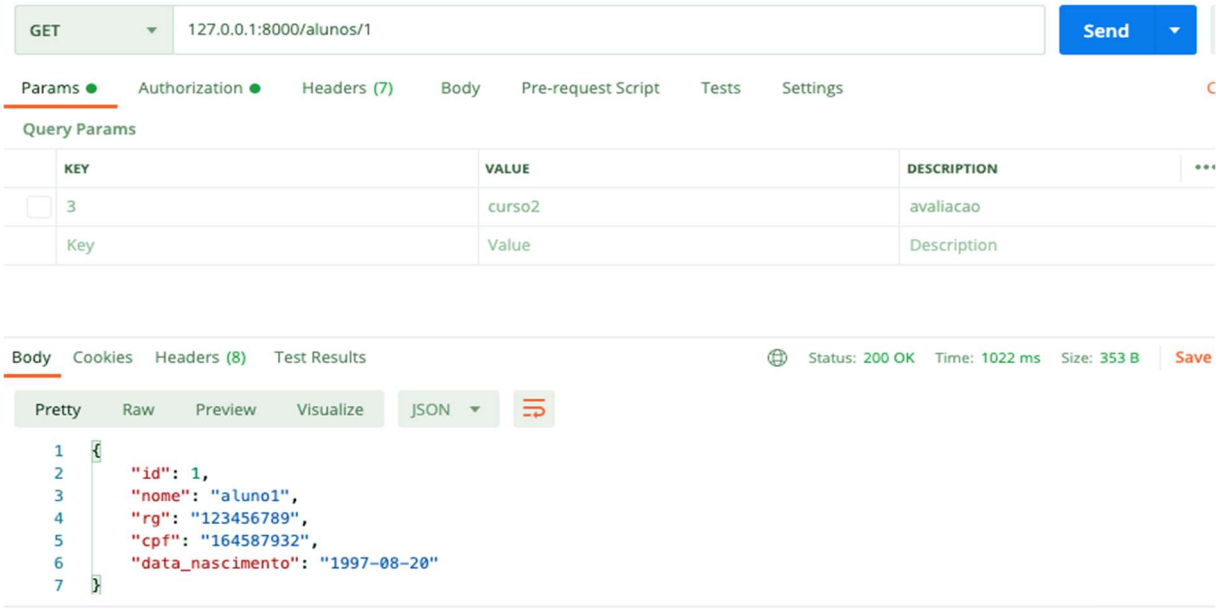

Figura 14: Método GET.

Nas figuras 15 e 16, utilizando a própria ferramenta fornecida pelo framework, foi feito uma atualização no formulário. Essa página, permite ao desenvolvedor inserir um código no formato JSON e envia-la a API, possibilitando a atualização dos dados:

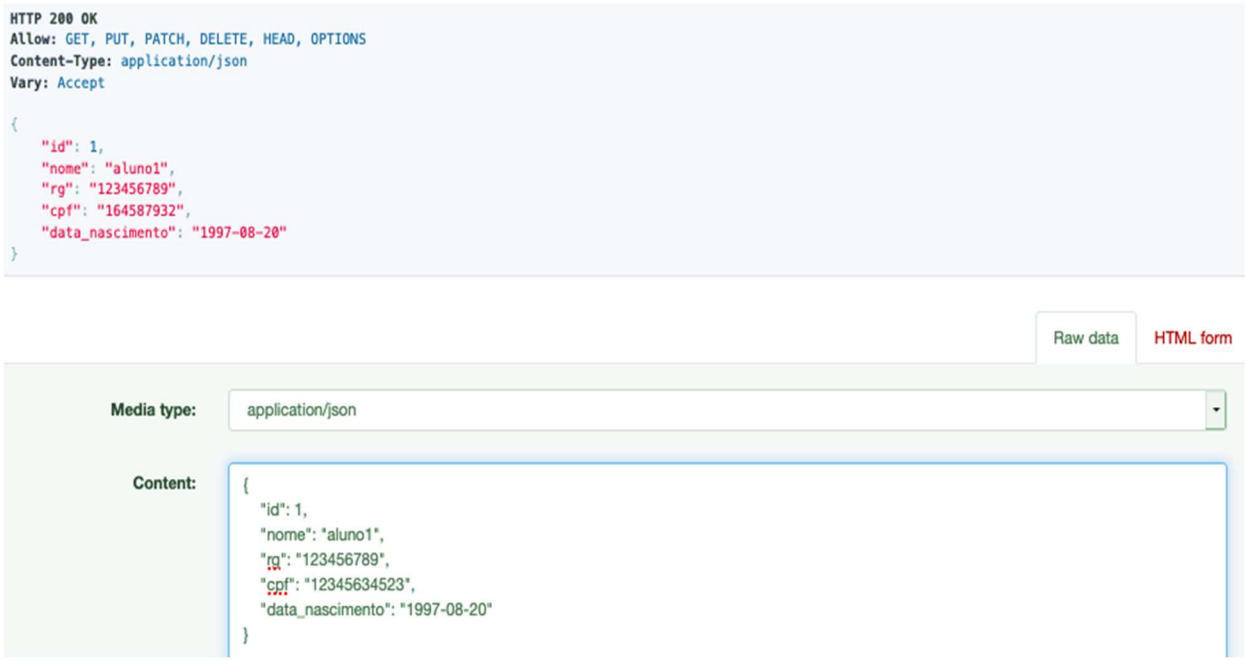

Figura 15: Execução método PUT.

Figura 16: Resultado após execução do método PUT.

Nas figuras 17, com auxilio da ferramenta *Postman,* executa-se um DELETE, forma de exclusão de um formulário já existente no banco de dados do sistema. Após ser executado o item correspondente a seu ip não poderá mais ser acessado, como exemplificado na figura 18, exemplificando o resultado do método DELETE:

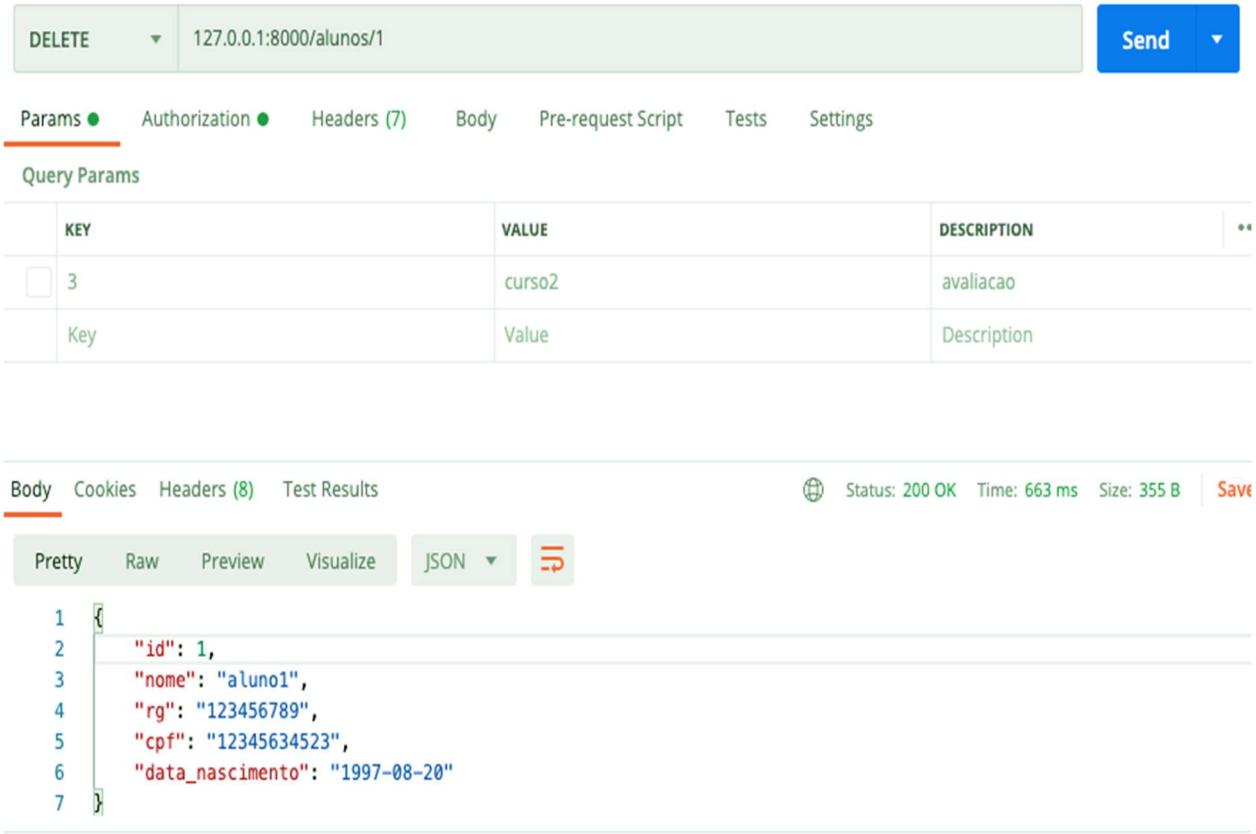

Figura 17: Método DELETE.

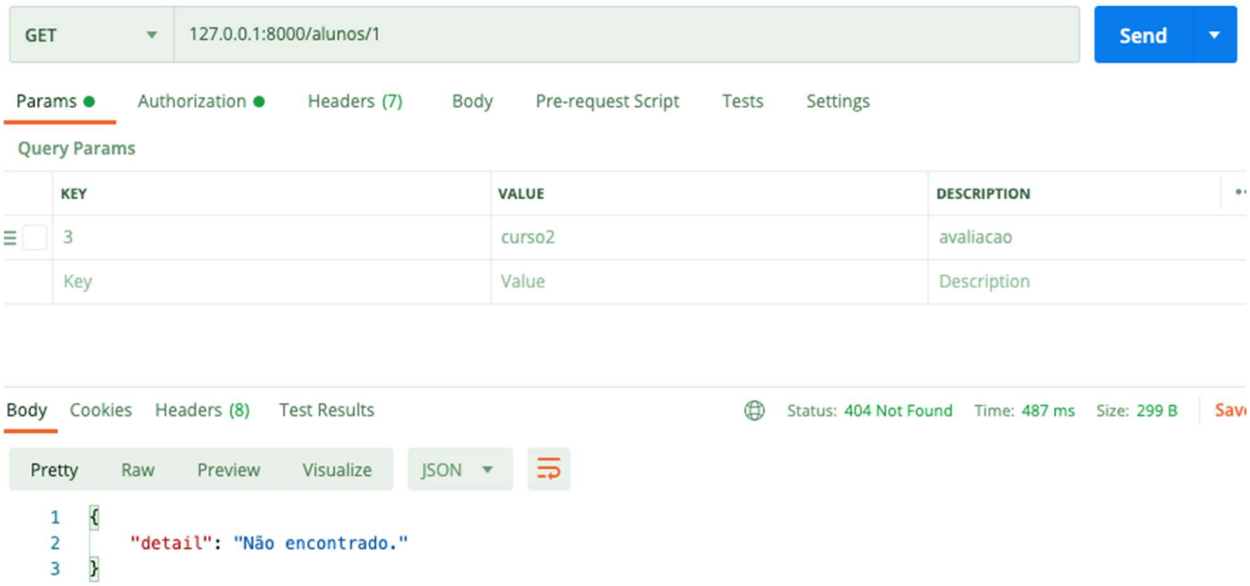

Figura 18: Após utilizar o método DELETE.

Na figura 19, exemplifica-se o campo responsável pela inserção dos dados, este disponibilizado pelo *framework,* onde se passa os parâmetros a serem adicionados, posteriormente clica-se no botão indicado com o nome POST, na parte inferior direita da página, realizando o método. A figura 20, ilustra a forma que se encontrará armazenado os dados após sua adição:

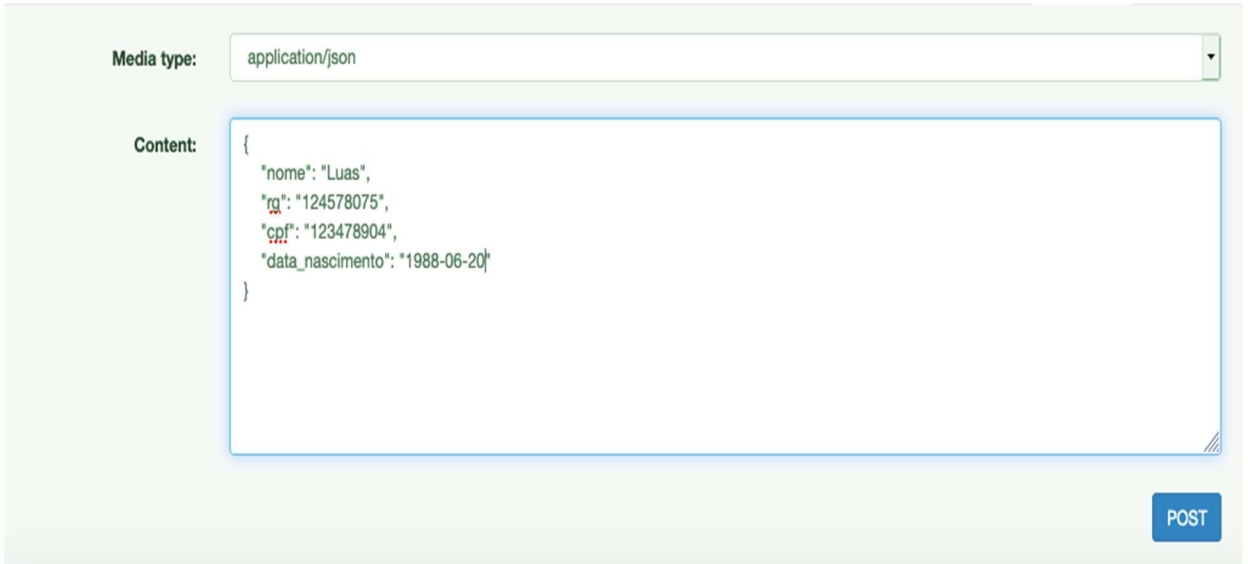

Figura 19: Método POST.

Figura 20: Resultado método POST.

A partir destas imagens podemos observar a forma como a API funciona, serializando as requisições e o trabalho dos modelos sob estas respostas no formato JSON. O REST trabalha com múltiplos *endpoints*, onde cada um retorna uma solicitação diferente. A API fica responsável por selecionar os itens desejados nos dados, pois em certas ocasiões não é necessário o modelo completo para conseguir retornar o objeto ao usuário. Desta maneira observamos como é estruturado os modelos, com suas chaves e valores.

5.2 Graphene

Este framework *open source,* foi desenvolvido por um grupo de desenvolvedores de todo o mundo, entre eles Syrus Akbary, Jhonathan Kim, Christoph Zwerschke, entre outros que são citados no site oficial (https://graphene-python.org/).

A principal diferença na estrutura de programação do Django Rest Framework para o Graphene é a criação dos *schemas*, onde se engloba funções para a execução das *mutations* e as *querys*, extremamente importante para que a utilização do GraphQl seja eficaz. No *schema.py* são definidas as *mutations*, classes que permitem a manipulação dos dados, responsável por qualquer modificação ou adição de dados no sistema. Dentro dessas classes se passa o *serializer*, que atua da mesma forma que nas aplicações REST. As *Querys* recebem classes com nomenclatura final node, onde se encontra os modelos de cada classe de dados, ao serem inclusas como

argumentos dentro de uma classe do tipo *DjangoObjectType*, tipo genérico exclusivo, para trabalhar com os modelos de dados do Django em GraphQl. As figuras 21 e 22 se observa um *schema*, visualizando sua forma de definição, estruturação das classe responsáveis pela inserção e modificação de dados:

settings.py | schema.py

library > catalog > schema.py > BooktImageMutation > mutate

```python
from graphene import relay, ObjectType, Mutation, Boolean, ID
from graphene_django import DjangoObjectType
from graphene_django.filter import DjangoFilterConnectionField
from .api.serializers import BookSerializer, AuthorSerializer
from graphene_django.rest_framework.mutation import SerializerMutation
from .filters import BookFilter
from .models import Book, Author, BookImage
from graphene_file_upload.scalars import Upload
import boto3
import uuid

#AWS S3 BUCKET
S3_BASE_URL = "s3.amazonaws.com"
BUCKET = 'libby-app'

class BookImageNode(DjangoObjectType):
    class Meta:
        model = BookImage

class BookNode(DjangoObjectType):
    class Meta:
        model = Book
        interfaces = (relay.Node, )

class AuthorNode(DjangoObjectType):
    class Meta:
        model = Author
        filter_fields = []
        interfaces = (relay.Node, )
```

Figura 21: Modelo de Schema GraphQl 1.

```python
class BookMutation(SerializerMutation):
    class Meta:
        serializer_class = BookSerializer

class BooktImageMutation(Mutation):
    class Arguments:
        file = Upload(required=True)
        id = ID(required=True)

    success = Boolean()

    def mutate(self, info, file, **data):
        photo_file = file
        book_id = data.get('id')

        if photo_file and book_id:
            s3 = boto3.client('s3')
            key = uuid.uuid4().hex[:6] + photo_file.name[photo_file.name.rfind('.'):]
            try:
                s3.upload_fileobj(photo_file, BUCKET, key)
                url = f"https://{BUCKET}.{S3_BASE_URL}/{key}"
                photo = BookImage(url=url, book_id=book_id)
                photo.save()
            except Exception as err:
                print('Oops! Tivemos um problema em fazer o upload de sua imagem: %s' % err)
                return BooktImageMutation(success=False)

        else:
            print('Falta do id do livro ou url da imagem')
            return BooktImageMutation(success = False)
```

Figura 22: Modelo de Schema GraphQl 2.

Em *url.py* é definida uma rota responsável por toda a movimentação dos dados, no caso da API em GraphQl é utilizado um painel que faz parte dos pacotes de ferramentas do framework, onde se insere as *querys* e é executado obtendo um JSON referente a solicitação executada. Na figura 23 é ilustrada a declaração da url padrão, servindo de padrão de acesso para a API em GrapqQL possa fazer o processamento dos dados*:*

```python
from django.contrib import admin
from django.urls import path, include
from django.views.decorators.csrf import csrf_exempt
#from graphene_django.views import GraphQLView
from graphene_file_upload.django import FileUploadGraphQLView

urlpatterns = [
    path('admin/', admin.site.urls),
    path('api/', include('catalog.api.urls')),
    path('graphql/', csrf_exempt(FileUploadGraphQLView.as_view(graphiql=True)))
]
```

Figura 23: urls.py.

Na figura 24 é empregado a estrutura de acesso para obter a execução de uma *query*, onde se faz um processo semelhante ao GET, feito em REST. Ao ser executado oferece uma resposta no formato de JSON, devido a desserialização dos dados. A principal diferença entre o método GET e a execução da *query* de resgate de dados, é a possibilidade de trazer informações de mais de um modelo de dados e a seleção dos campos desejados, sendo retornado apenas o necessário:

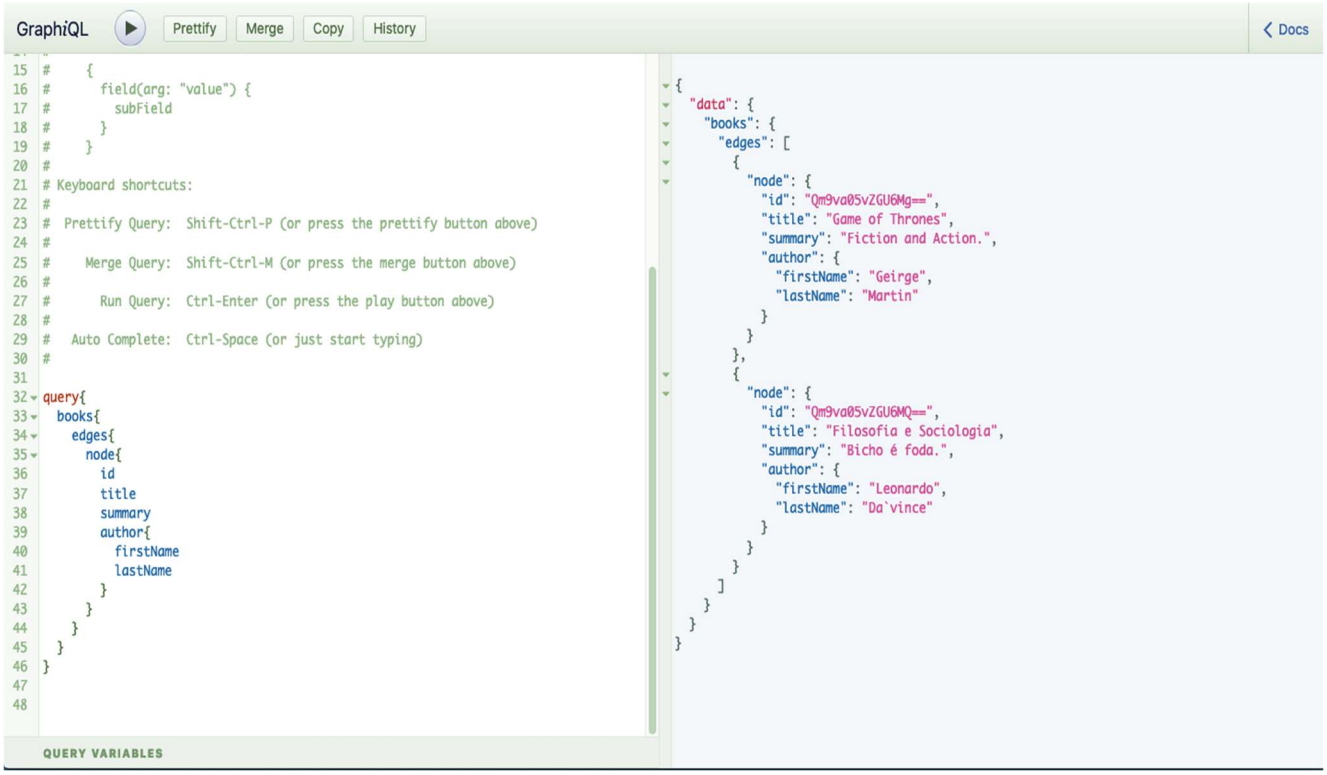

Figura 24: Método GET em dois modelos de dados.

Na figura 25 observamos a inclusão de um livro, e a baixo o que se selecionou para a visualização, pode-se atentar que o *summary* foi adicionado. Porém, como ele não foi solicitado, não foi retornado nenhum valor. Supondo, em uma pesquisa, onde inicialmente mostre apenas os títulos dos livros, ao utilizar esta funcionalidade, disponível em GraphQL, seria apresentado apenas os atributos necessários, que estivessem cadastrados:

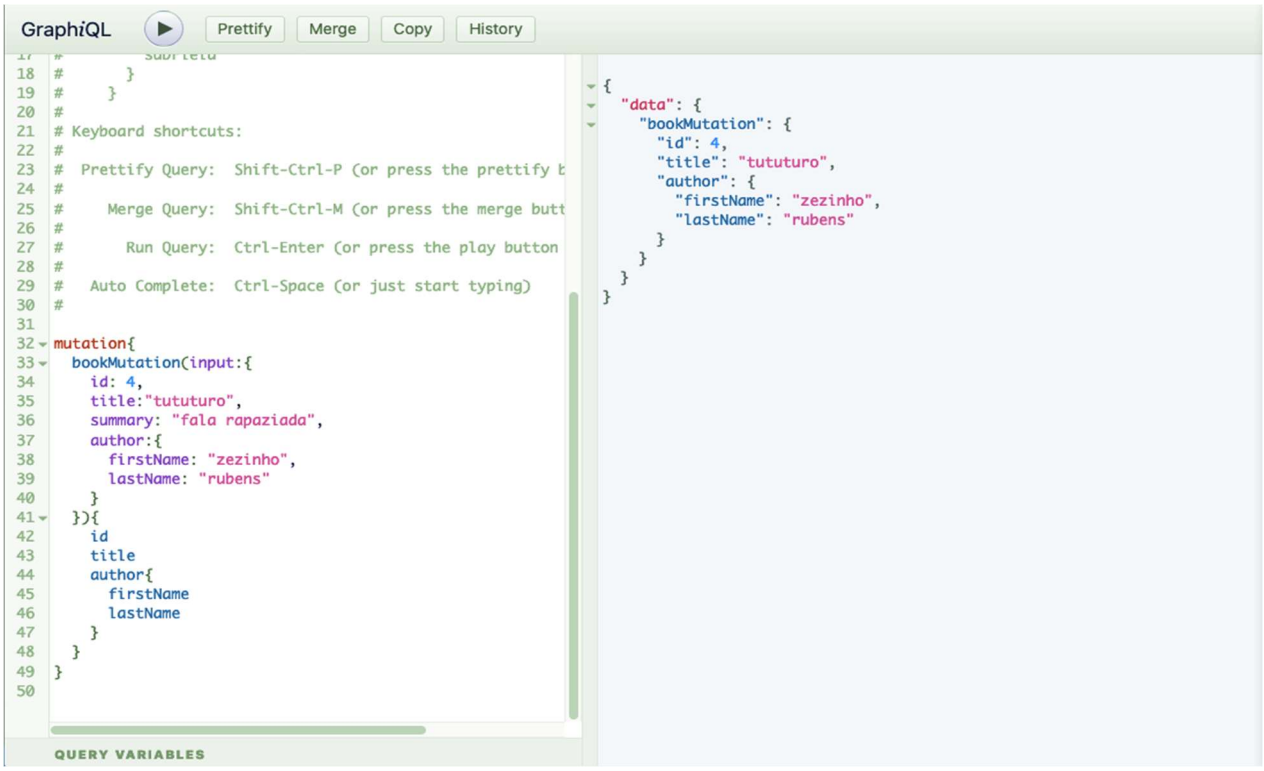

Figura 25: Mutations POST.

Na figura 26 ilustra a *mutation* executando uma exclusão, o item ao ser excluído passa um parâmetro *boolean*, o tornando *True,* assim o sistema intende que ele deve ser excluído e elimina os dados existentes no identificador inserido:

Figura 26: Mutation DELETE.

6. CONCLUSÃO

Neste capítulo serão descritas as considerações finais de tudo o desenvolvimento do projeto, cujo objetivo proposto foi atingido através de técnicas de *frameworks* apresentados neste. Durante a execução do projeto foram encontrados alguns desafios, sendo estes superados por meio de uma ampla revisão bibliográfica.

Foi possível concluir, que ambos os designs de arquitetura tem seus lados positivos e negativos. Porém, em uma larga escala, pelo GraphQL utilizar apenas um *endpoint* durante o seu tempo de execução, obtendo maior eficácia. Durante o desenvolvimento a estrutura acaba sendo mais intuitiva e direta, a ferramenta possibilita que na própria *Query* seja explícito quais campos precisaram ser exibidos durante o seu retorno.

Neste estudo é apresentado a preocupação dos frameworks em proporcionar aos desenvolvedores um design de arquitetura que melhor atenda suas necessidades. O Graphene possibilita a transformação de um projeto REST para GraphQL, com algumas modificações, porém não são todas as ferramentas que tem essa preocupação em disponibilizar ao desenvolvedor a melhor opção que lhe atenda de maneira fácil e ágil de implementar.

Para alcançar estas conclusões foi desenvolvido duas APIs, em tempo de desenvolvimento, devido aos frameworks utilizados ambos possuírem escalabilidade e agilidade, porém em tempo de execução em larga escala as tecnologias que utilizam GraphQL possuem maior eficiência. Estas também permitem um reaproveitamento de código mais preciso e intuitivo, proporcionando ao desenvolvedor um controle preciso de suas requisições no banco de dados, já que as *Querys,* a partir de sua estrutura, traz e envia somente o necessário para a execução das operações.

Atualmente com o crescimento do mercado e o número de acessos simultâneos se tornando cada vez maior, o design de arquitetura GraphQL tem um potencial de tomar parte do mercado. Redes sociais, aplicativos bancários e *E-commerces,* poderiam potencializar suas aplicações partindo deste conceito de um único *endpoint*, pelo fato de obter apenas as informações necessárias.

REFERÊNCIAS

CONSTANTINO, Lucas. GraphQL hoje usando Apollo em aplicações que utilizam APIs REST. Disponível em: https://imasters.com.br/apis-microsservicos/graphql-hoje-usando-apollo-em-aplicacoes-que-utilizam-apis-rest. Acesso em: 18 de mar. de 2020.

DELFINO, Daniel. O Futuro da APIs nos negócios. Disponível em: https://vertigo.com.br/o-futuro-das-apis-nos-negocios/. Acesso em: 13 de mar. de 2020.

Documentação Django Rest Framework. Django REST framework. Disponivel em: https://www.django-rest-framework.org/. Acesso em: 05 de Agos. De 2020.

FERNANDES, André. O que é API? Entenda de uma maneira simples. Disponivel em: https://vertigo.com.br/o-que-e-api-entenda-de-uma-maneira-simples/. Acesso em: 13 de mar. de 2020.

JÚNIOR, Isac. GraphQL: API intuitiva e flexível para descrever requisitos. Disponível em: https://www.zup.com.br/blog/graphql-api-para-descrever-requisitos. Acesso em: 18 de mar. de 2020.

LIMA, J. C. R. **Web Services (SOAP x REST).** 2012. 41 f. Monografia (Tecnologia em Processamento de Dados) – Faculdade de Tecnologia de São Paulo, São Paulo, 2012.

PAIVA, Esequiel; MATIUSSO, Mario. Disponível em: http://matiusso.dx.am/Arquivos/Aplicacoes%20RESTful.pdf. Acesso em: 1 de nov. de 2019.

RIBAS, Weder; NODA, Edgar; MARQUES, Daniela. Disponível em: http://hto.ifsp.edu.br/portal/images/thumbnails/images/IFSP/Cursos/Coord_ADS/Arquivos/TCCs/2018/TCC_WederAlvesRibas_HT1320386.pdf. Acesso em: 1 de nov. de 2019.

Roy Thomas Fielding. Architectural Styles and the Design of Networkbased Software Architectures. PhD thesis, UNIVERSITY OF CALIFORNIA, 2000.

TEAM, Facebook. GraphQQL Documentation disponivel em: https://graphql.org/learn/schema/. Acesso em: 11 de mar. de 2020.

VARANDA, Daniel. A Ascensão di GraphQL na Era das APIs. Disponível em: https://sensedia.com/api/a-ascensao-do-graphql-na-era-das-apis/. Acesso em: 11 de mar. de 2020.

W3C. *HTML 4.01 Specification*. 1999a .Disponível em http://www.w3.org/TR/html4/cover.html. Acesso em Novembro 2011.