

Fernando Baggi Tancini

**Extensão do *framework*
Graphene-Django, visando otimizar
o número de consultas ao banco de
dados**

RELATÓRIO DE PROJETO FINAL

**DEPARTAMENTO DE ENGENHARIA ELÉTRICA E
DEPARTAMENTO DE INFORMÁTICA**

**Programa de graduação em Engenharia de
Computação**

Rio de Janeiro
Novembro de 2021

Fernando Baggi Tancini

**Extensão do *framework* Graphene-Django,
visando otimizar o número de consultas ao
banco de dados**

Relatório de Projeto Final

Relatório de Projeto Final, apresentado ao programa de Engenharia de Computação da PUC-Rio como requisito parcial para a obtenção do título de Engenheiro de Computação.

Orientador: Prof. Marcos Kalinowski

Rio de Janeiro
Novembro de 2021

Resumo

Baggi Tancini, Fernando; kalinowski, Marcos. **Extensão do *framework* Graphene-Django, visando otimizar o número de consultas ao banco de dados**. Rio de Janeiro, 2021. 41p. Projeto de Graduação – Departamento de Engenharia Elétrica e Departamento de Informática, Pontifícia Universidade Católica do Rio de Janeiro.

Neste trabalho foi implementada uma extensão do *framework* Graphene-Django, que é usado para o desenvolvimento de APIs GraphQL baseadas em uma aplicações Django. Foram introduzidas funcionalidades que visam minimizar o número de consultas realizadas ao banco de dados pela aplicação sem tornar o desenvolvimento da API mais complexo ou extenso.

Palavras-chave

API; Framework; GraphQL; ORM.

Sumário

| | | |
|-----|--|----|
| 1 | Introdução | 4 |
| 1.1 | Contexto e Motivação | 4 |
| 1.2 | Objetivos | 5 |
| 1.3 | Estrutura da Monografia | 6 |
| 2 | Fundamentação Técnica | 7 |
| 2.1 | Introdução | 7 |
| 2.2 | Visão Geral do Django | 8 |
| 2.3 | Visão Geral do GraphQL e Graphene-Django | 11 |
| 2.4 | Considerações Finais | 13 |
| 3 | Extensão do Graphene-Django | 15 |
| 3.1 | Introdução | 15 |
| 3.2 | Requisitos da Extensão | 17 |
| 3.3 | Projeto da Extensão | 22 |
| 3.4 | Implementação | 25 |
| 3.5 | Prova de Conceito | 30 |
| 3.6 | Considerações Finais | 36 |
| 4 | Conclusão | 37 |
| 4.1 | Contribuições | 37 |
| 4.2 | Limitações | 37 |
| 4.3 | Trabalhos Futuros | 39 |
| | Referências bibliográficas | 40 |

1

Introdução

1.1

Contexto e Motivação

Ao longo de poucas décadas após a sua concepção, a internet vem se tornando cada vez mais essencial para a sociedade. Pode-se dizer que estamos vivendo a era da conectividade. Isso porque existe uma quantidade enorme e crescente de dados trafegando ao redor do globo a todo e qualquer instante. Toda essa comunicação serve a incontáveis propósitos, desde os mais banais, até aqueles sem os quais a sociedade como conhecemos não existiria.

Existe uma série de camadas (Forouzan, 2009) que torna possível a comunicação entre duas aplicações em computadores remotamente localizados. Após se tornar possível a troca de dados entre essas aplicações, surge a necessidade de se estabelecer padrões para essas comunicações, a fim de facilitar a interação entre as incontáveis aplicações que já existem e que ainda virão a existir. É nesse contexto que torna-se relevante o estabelecimento de bons protocolos de APIs.

Uma API, que em inglês refere-se a “*Application Programming Interface*”, consiste em uma interface para comunicação entre diferentes aplicações. Uma API pode ser projetada para ser requisitada por apenas determinadas aplicações previamente conhecidas, ou pode ser projetada para receber requisições de inúmeras aplicações desconhecidas. Entretanto, em ambos os casos é conveniente que a API siga algum padrão pré estabelecido, a fim de facilitar não só a sua compatibilidade com as outras aplicações como também sua própria implementação e manutenção.

Um padrão de API amplamente utilizado é o chamado REST (Postman, 2021). Contudo, ele não é o único colocado em prática atualmente. Há cerca de seis anos, a empresa Facebook publicou um protocolo chamado GraphQL. O qual, segundo a empresa, possui uma série de vantagens para suas APIs (Facebook, 2021a). Desde então, este protocolo vem sendo adotado cada vez mais por programadores ao redor do mundo (Greif e Benitte, 2021). Mais a frente no presente trabalho, o protocolo GraphQL é descrito de forma aprofundada.

O contexto que motivou este trabalho é o desenvolvimento de APIs GraphQL. Mais especificamente APIs implementadas em Python, a partir do *framework* Django. A escolha de Python como linguagem de programação é relevante por ela estar entre as linguagens com maior crescimento de popularidade (Srinath, 2017). Além disso, o Django também é bastante relevante pois é um dos dois *frameworks* para desenvolvimento *web* em Python mais utilizados (Stack Overflow, 2021). O outro *framework* também popular é o Flask, mas ele é considerado um *microframework* (Berga e Martinho, 2021) e não se propõe a ser um solução robusta.

O desenvolvimento de APIs GraphQL baseadas em Django pode ser alcançado através do *framework* Graphene-Django. Esse *framework* traz consigo funcionalidades que facilitam o desenvolvimento seguindo o padrão GraphQL. Contudo, analisando esse *framework*, foi possível encontrar algumas melhorias e novas funcionalidades que, caso incorporadas ao *framework*, tornam o desenvolvimento, manutenibilidade e desempenho da API muito superiores. A partir disso, este trabalho surge como uma oportunidade de trazer um impacto positivo no desenvolvimento de APIs à comunidade de desenvolvedores que utilizam esses grandes *frameworks open source*.

1.2 Objetivos

O objetivo deste trabalho consiste em realizar algumas mudanças na biblioteca *open source* Graphene-Django. Para entender a fundo as mudanças propostas, é necessário aprofundar alguns conceitos relacionados ao ORM do Django e ao processamento de uma requisição GraphQL, o que será feito no Capítulo 2 desta monografia. Contudo, nesta seção, será pontuado de forma superficial o objetivo deste projeto.

O protocolo GraphQL possui esse nome pois envolve a definição do que pode ser considerado um grafo. São definidos nós (objetos) e seus campos. Esses campos podem tanto ser valores como inteiros ou *strings*, quanto podem ser arestas que o ligam a outros nós do grafo. De forma simplificada, a requisição GraphQL consiste em indicar quais objetos deseja-se receber em resposta, assim como os campos desses objetos. Assim, um conceito chave do GraphQL é a imprevisibilidade do que será pedido nas requisições com exatidão quando se está desenvolvendo a API.

Outro ponto importante de se entender para compreender o propósito desse projeto é relacionado ao desempenho da API. Em geral, para se obter um bom desempenho e respostas rápidas da API, é desejável que se realize um número pequeno de consultas ao banco de dados. Por isso, o ideal é já saber

todos os dados que o cliente vai pedir à API em uma determinada requisição e, com base nisso, realizar poucas consultas ao banco de dados que já retornem todos os dados necessários.

Nesse contexto surge uma falta na implementação atual dessa biblioteca. Pois o *framework* tem ciência em relação à definição do grafo da API, à definição dos objetos do ORM e também ao que foi pedido na requisição do cliente. Ou seja, possui ao seu alcance tudo que é necessário para determinar e realizar essas consultas mínimas ao banco de dados. Com isso, o *framework* pouparia o programador desse dilema entre de arcar com um baixo desempenho ou ter que prever como serão os dados pedidos pelos clientes da API, o que é algo conceitualmente errado no GraphQL.

Portanto, o objetivo deste projeto consiste em evoluir a biblioteca Graphene-Django, a fim de tornar o desenvolvimento de APIs mais simples e sucinto, além de garantir um bom desempenho. Por isso, pode-se dizer que a utilização dessa extensão de *framework* é algo que contribui para a qualidade de software (Wagner, 2013) do projeto de API. Afinal, a “manutenibilidade” do software é um ponto que se beneficia muito quando a programação torna-se mais simples e sucinta. Além disso, a “eficiência de desempenho” é um ponto diretamente beneficiado pela redução no número de consultas realizadas ao banco de dados.

1.3

Estrutura da Monografia

Esta monografia possui, além do presente capítulo de introdução, mais três capítulos. Nesta seção está descrito de forma breve o que cada um desses capítulos abordam.

No Capítulo 2, Fundamentação Técnica, são abordados os conceitos teóricos e práticos de tecnologias que são necessários para o entendimento do projeto como um todo. Nele, são aprofundados alguns recursos dos *frameworks* Django, Graphene e Graphene-Django.

No Capítulo 3, Extensão do Graphene-Django, é onde estão consolidados os requisitos da extensão, assim como estão descritas as mudanças realizadas no código fonte do *framework*. Além disso, esse capítulo ainda contém a prova de conceito, que comprova que os objetivos do projeto de extensão foram alcançados com sucesso.

No Capítulo 4, Conclusão, estão presentes quais foram as contribuições realizadas pelo presente trabalho de extensão, assim como suas limitações. Por fim, estão descritos ainda nesse capítulo possibilidades de trabalhos futuros para dar continuidade ao presente projeto.

2

Fundamentação Técnica

2.1

Introdução

Um ORM, que em inglês refere-se a “*Object Relational Mapper*”, é uma espécie de interface entre o banco de dados relacional e uma aplicação programada em uma linguagem orientada a objetos. É muito conveniente poder realizar as operações no banco de dados a partir do próprio código de uma aplicação, sem ter de recorrer, por exemplo, à escrita direta de comandos em SQL para enviá-los ao banco de dados. Ter um ORM robusto é certamente um dos fatores que mantém o Django entre os *frameworks* de Python mais utilizados. É por meio dele que aplicações Django interagem com o banco de dados, para operações como definição de tabela, consulta e alteração de dados. Neste capítulo, na seção 2.2, é realizado o aprofundamento nos conhecimentos relacionados ao ORM do Django que são necessários para o entendimento do projeto.

Existem casos de API para os quais a adoção do protocolo GraphQL traz claros benefícios comparado ao protocolo REST. O principal deles é o caso de se criar uma API que será requisitada por diferentes aplicações das quais não é possível conhecer os requisitos. Para o Facebook, criador do protocolo, isso é relativamente comum. Um famoso exemplo é o caso da definição de um grafo baseado em um objeto Pessoa e um relacionamento de Pessoa com Pessoa chamado “amigos”. Nesse contexto, é possível deixar o cliente da requisição à API decidir qual é o retorno que se deseja. Portanto, um exemplo de pedido seria “me dê a pessoa que possui o ID tal e mais os amigos dessa pessoa”. Contudo, nada impede que algum outro cliente da API faça algum pedido que peça os amigos dos amigos dessa pessoa com esse tal ID. Ou seja, quem implementou a API não necessariamente precisa entender exatamente o que será requisitado de dados. Precisa apenas definir quais são os objetos e seus relacionamentos. Além, é claro, de definir os pontos de entrada. Nesse exemplo citado, um ponto de entrada seria uma Pessoa a partir de um ID.

Porém, os benefícios do uso desse protocolo não se limitam a esse caso de uso. É cada vez mais comum na produção de software a adoção de meto-

dologias que não possuem um planejamento inteiramente definido antes de se iniciar o desenvolvimento. Em vez disso, o processo de criação do software é planejado aos poucos, conforme as necessidades, análises e testes são observados com o projeto já em andamento. Esse é um dos pilares do desenvolvimento ágil (Kuhrmann et al., 2014). Nesse contexto, de desenvolvimento sem planejamento a longo prazo definido, é muito comum que o software sofra mudanças contínuas. Afinal, os próprios requisitos continuamente são renovados. Essas mudanças podem ser custosas caso a API não seja flexível e simples de ser incrementada e remodelada. Um bom *framework* de desenvolvimento de APIs GraphQL tem o potencial de tornar simples realizar alterações ao longo da API. Neste capítulo, na Seção 2.3, são aprofundados alguns conceitos relacionados a GraphQL necessários para o entendimento do projeto.

2.2

Visão Geral do Django

Existem duas características essenciais do Django para o desenvolvimento de uma API baseada em um banco de dados. São eles o ferramental para o recebimento e retorno de requisições HTTP através de diferentes *endpoints* e também o seu robusto ORM. Além disso, esse *framework* também oferece facilidades no desenvolvimento de uma interface de usuário, para manipulação dos dados por alguma equipe interna. Dentre essas características do Django, o que é mais relevante a este trabalho é o ORM presente nele. Pois, apesar de ser importante existir um *endpoint* numa API GraphQL, é necessário apenas de um para o seu funcionamento. É através do *endpoint /graphql* que, por padrão, são realizadas as comunicações com a API.

Um ponto chave de qualquer ORM é a definição dos objetos. No Django, são definidas subclasses de Model para se definir uma entidade do banco de dados, assim como seus campos. A partir disso, o *framework* maneja a definição do SQL necessário para a criação das respectivas tabelas necessárias a nível de banco de dados. Além disso, a partir desses modelos criados, é possível realizar operações no banco de dados para consulta, alteração ou deleção de dados relacionados a essas entidades. Essas operações são realizadas por meio do que é chamado de QuerySet.

Está exemplificado na Figura 2.1 a criação de duas entidades no Django. Note que ambas possuem seus respectivos campos (atributos em geral), assim como é presente relacionamentos entre as duas (atributos cujos valores são ForeignKey ou ManyToManyField). A partir de cada um desses campos de relacionamento são criados dois relacionamentos nomeados, cada um deles em um dos sentidos, sendo um deles o próprio nome do atributo e o outro o

argumento *related_name*. Ou seja, nesse caso, objetos do tipo Dog possuem *owner* e *friends*. Já objetos do tipo Person possuem os relacionamentos *owned_dog* e *dog_friends*.

```
1 class Person(Model):
2     class Meta:
3         verbose_name = "person"
4         verbose_name_plural = "people"
5
6     name = models.CharField("name", max_length=255)
7
8
9 class Dog(Model):
10     class Meta:
11         verbose_name = "dog"
12         verbose_name_plural = "dogs"
13
14     name = models.CharField("name", max_length=255)
15     owner = models.ForeignKey(Person, on_delete=models.SET_NULL, verbose_name="owner", related_name="owned_dogs")
16     friends = models.ManyToManyField(Person, verbose_name="friends", related_name="dog_friends")
17
```

Figura 2.1: Declaração de modelos Django

No exemplo da Figura 2.2, pode-se observar uma chamada ao banco de dados para consultar todos os objetos de Person no banco, assim como N chamadas para determinar os *owned_dogs* de cada um dos objetos Person, sendo N o número de entradas na tabela de Person. Porém, é possível reduzir esse número de requisições ao banco de dados. Na Figura 2.3, está presente uma forma alternativa de, em duas consultas ao banco, já se capturar todos os objetos Person e os respectivos objetos Dog relacionados.

```
17 people = Person.objects.all()
18 for person in people:
19     owned_dogs = person.owned_dogs.all()
20     print(person, owned_dogs)
21
```

Figura 2.2: Laço que gera a $O(N)$ consultas SQL

Esse recurso, chamado *prefetch*, é muito útil para se reduzir o número de chamadas realizadas ao banco de dados. O funcionamento dele é, de forma simplificada, realizar uma segunda chamada ao banco de dados para se obter também os objetos relacionados e, no Python, deixar os objetos relacionados já em *cache*. Depois disso, quando os objetos relacionados forem acessados, nenhuma nova consulta SQL será necessária. No contexto deste trabalho, pode-se realizar *prefetch* de todos dados necessários para o retorno da requisição GraphQL de uma vez e, assim, minimizar o número de consultas realizadas ao banco de dados (Django Contributors, 2021).

```
17 people = Person.objects.all().prefetch_related(Prefetch('owned_dogs'))
18 for person in people:
19     owned_dogs = person.owned_dogs.all()
20     print(person, owned_dogs)
21
```

Figura 2.3: Laço que gera $O(1)$ consultas SQL

Este é um recurso essencial para o entendimento deste trabalho. Um último ponto que é importante destacar é que é possível se realizar alterações nos QuerySets que serão utilizados na consulta SQL do *prefetch*. No exemplo da Figura 2.4 pode-se observar dois *prefetches* em cascata. Graças a esse recurso, o que poderia refletir em inúmeras chamadas ao banco de dados tornam-se apenas três chamadas, nesse exemplo de uso do recurso de *prefetch* em cascata.

```
17 dogs_for_prefetch = Dog.objects.all().prefetch_related(Prefetch("friends"))
18 people = Person.objects.all().prefetch_related(Prefetch('owned_dogs', queryset=dogs_for_prefetch))
19 for person in people:
20     owned_dogs = person.owned_dogs.all()
21     for dog in owned_dogs:
22         friends_of_the_dog = dog.friends.all()
23         print(person, dog, friends_of_the_dog)
24
```

Figura 2.4: Laço aninhado que gera $O(1)$ consultas SQL

Outra funcionalidade do Django que é importante ser explicada para o completo entendimento do trabalho de extensão é a função *annotate* de um QuerySet. Essa função possui como propósito criar uma nova coluna na tabela resultado da consulta SQL. Essa nova coluna, é claro, será também acessível a nível de Python. Vale ressaltar que esse tipo de operação em SQL não altera as tabelas em si, tendo efeito apenas na tabela de retorno da consulta SQL. Vejamos um exemplo simples na Figura 2.5. Neste exemplo, anotamos um novo campo chamado *repeated_name*.

```
30 people_with_repeated_name = Person.objects.all() \
31     .annotate(repeated_name=Concat(F('name'), Value(' '), F('name')))
32 for person in people_with_repeated_name:
33     print(f'Name: {person.name}')
34     print(f'Repeated name: {person.repeated_name}')
35
```

Figura 2.5: Uso do método *annotate* de QuerySet

Esse exemplo da Figura 2.5 é um caso que não oferece tantas vantagens comparado com a alternativa que seria não anotar a nível SQL e sim no Python fazer essa concatenação de nomes. Porém, existe a possibilidade de um campo anotado ser algo complexo e que exige cálculos envolvendo tabelas ao longo do banco inteiro. Realizando esse cálculo a nível de banco de dados, economiza-se o trabalho de pegar N dados do banco para se realizar algum cálculo como, por exemplo, uma média ou soma. Os casos de uso da funcionalidade *annotate* são diversos, porém basta esse simples exemplo de concatenação para o objetivo de compreender este trabalho.

2.3

Visão Geral do GraphQL e Graphene-Django

GraphQL é uma linguagem de consulta de API, que determina como serão as requisições e as respostas. Em uma comunicação com uma API, denomina-se cliente quem está realizando contato com a API, assim como denomina-se servidor a própria aplicação que serve a API. O protocolo estabelece uma espécie de linguagem pela qual o cliente expressa seu pedido ao servidor.

De forma simplificada, essa linguagem de consulta tem como objetivo descrever o que o cliente deseja receber na resposta do servidor. Na Figura 2.6, é possível observar um exemplo de pedido simples GraphQL. Esse exemplo ilustra um pedido para obter *name*, *email* e *posts* do usuário que está autenticado na API. Observa-se nesse pedido objetos e campos desses objetos que deseja-se obter na resposta. Existem dois tipos de campos, um cujo valor é um objeto e outro cujo valor não é um objeto. Quando o valor de um campo é um objeto, pode-se chamá-lo de uma conexão. Em uma visualização em grafo, pode-se considerar os objetos como nós, as conexões como arestas e demais campos como propriedades desses nós.

Um ponto importante de se entender sobre GraphQL é que existem dois possíveis “pontos de entrada” de uma requisição. Um deles é um objeto chamado Query, que está presente na Figura 2.6, e o outro é um objeto chamado Mutation. Ambos são objetos como quaisquer outros. Ou seja, nesse exemplo de requisição é possível notar que o objeto Query possui um campo chamado *user*. A diferença entre *query* e *mutation* é no campo da semântica. Uma *query* possui o propósito de consulta, enquanto uma *mutation* possui o propósito de alterar, criar ou deletar dados. Porém, a estrutura de definição, requisição e processamento é a mesma.

Outro ponto chave para a compreensão do GraphQL é o processo de “revolvimento de um campo”. Esse processo é definido pelo desenvolvedor da

```
1  query {  
2    user {  
3      name  
4      email  
5      posts {  
6        title  
7        body  
8        imageUrl  
9      }  
10   }  
11 }
```

Figura 2.6: Consulta GraphQL simples

API e consiste na definição de uma função para determinação do valor de um determinado campo de um objeto. Na Figura 2.7 é possível observar um exemplo de definição de campos dentro de um objeto. O objeto em questão é o `User`, visto na *query* de exemplo anterior. Pode-se observar a presença de 3 atributos de definição de campos e 3 métodos de resolução desses campos. Os métodos de resolução recebem como parâmetro *root*, que é o próprio objeto pai desse campo, e *info*, que é um agregado de informações referentes à requisição como um todo. Por exemplo, caso a API precisasse resolver uma lista de usuários e o campo *name* deles fosse solicitado, para cada usuário da lista, o método *resolve_name* seria chamado. Em cada execução dele, seria passado como parâmetro *root* um usuário de cada vez. Vale ressaltar que esse é um exemplo básico, sem recursos mais avançados do *framework* Graphene-Django que iriam tornar a definição desse objeto mais sucinta.

```
1  class UserType(ObjectType):  
2    name = graphene.String()  
3    def resolve_name(root, info):  
4        return root.name  
5  
6    email = graphene.String()  
7    def resolve_email(root, info):  
8        return root.email  
9  
10   posts = graphene.List(PostType)  
11   def resolve_posts(root, info):  
12       return root.posts.all()  
13
```

Figura 2.7: Definição de um ObjectType simples

Um outro ponto importante de uma API é a forma de receber dados do cliente. A forma principal de se receber dados é a passagem de parâmetro. Na Figura 2.8 encontram-se quatro casos de passagem de parâmetros para algum campo. É possível observar que não há diferença no GraphQL nos cenários de estar usando o objeto Query ou o objeto Mutation como nó de entrada do pedido. Afinal, ambos são objetos, cujas diferenças se dão no campo semântico e não sintático. Por fim, vale ressaltar que a passagem de parâmetro está relacionada a um campo, independente de quão profundo ele se encontre no pedido GraphQL e também independente de tipo do valor do campo, podendo ser objeto ou não.

```
1  query {  
2    post (id: 123) {  
3      title  
4      body  
5      datetime (timezone: "UTC")  
6    }  
7  }  
8  
9  mutation {  
10   editPost (id: 123, title: "New Title!", body: "New body!") {  
11     title  
12     body  
13     datetime (timezone: "UTC")  
14   }  
15 }  
16
```

Figura 2.8: Requisições GraphQL com passagem de parâmetros

Esses são os pontos principais para o entendimento de como funciona a definição de uma API GraphQL, a estrutura de uma requisição e a resolução de uma resposta. Essas são funcionalidades que já existem no nível do *framework* Graphene sozinho. Já o Graphene-Django adiciona algumas funcionalidades por cima para facilitar, por exemplo, a criação de um tipo de objeto GraphQL a partir de um modelo Django. Essa facilitação se dá pela classe `DjangoObjectType`. Na Figura 2.9 está um exemplo de um modelo Django e seu tipo de objeto GraphQL. Pode-se notar que não é necessário definir nem o tipo e nem o *resolver* do campo *name* do `DogType`. Afinal, isso pode ser descoberto pelo Graphene-Django a partir do modelo ao qual esse tipo se refere.

2.4

Considerações Finais

Pode-se considerar os recursos do Django apresentados neste capítulo, *annotate* e *prefetch_related*, como grandes aliados ao desempenho de uma aplicação que interage com um banco de dados relacional. O *annotate* por permitir

```
1 class Dog(Model):
2     class Meta:
3         verbose_name = 'dog'
4         verbose_name_plural = 'dogs'
5
6         name = CharField('name', max_length=255)
7
8
9 class DogType(DjangoObjectType):
10     class Meta:
11         model = Dog
12         fields = ['name']
13
```

Figura 2.9: Definição de um DjangoObjectType

que cálculos “computados” e/ou agregados sejam realizados diretamente no banco de dados, utilizando a interface do ORM do Django. O que é uma alternativa a carregar todos dados necessários ao cálculo para a aplicação, para então realizar as operações todas em Python. Já o *prefetch_related* é uma funcionalidade que pode contribuir substancialmente ao desempenho da aplicação por realizar uma quantidade reduzida de consultas SQL, quando comparada à alternativa de realizar poucas consultas SQL enxutas inicialmente, ao custo de se realizar uma série de novas consultas conforme fosse necessário acessar “objetos relacionados”.

Outro ponto importante de ser ressaltado ainda neste capítulo é em relação ao funcionamento de uma API GraphQL. Durante seu desenvolvimento, é importante ter a liberdade de não ter ciência prévia em relação aos dados que serão pedidos pelos clientes. Afinal, o principal objetivo desse protocolo é delegar o papel de escolha do que será retornado, em parte, para o cliente. Se os desenvolvedores da API tiverem que prever alguns padrões de pedidos dos clientes, a fim de realizar algum *annotate* ou *prefetch_related* para se obter um bom desempenho, torna-se evidente um conflito entre princípios do protocolo e sua utilização na prática.

Portanto, é notória a falta de suporte do *framework* Graphene-Django no que tange auxiliar os desenvolvedores de APIs a estarem ao mesmo tempo alinhados com os pilares do protocolo e ainda cheguem num desempenho desejável. É por isso que neste trabalho é descrito e realizado um incremento nesse *framework*. O qual possui como objetivo tornar simples o desenvolvimento de uma API que não precise prever os pedidos GraphQL para se obter um bom desempenho.

3

Extensão do Graphene-Django

3.1

Introdução

Pode-se dizer que obter um bom desempenho de resposta de uma API é algo desejável em praticamente qualquer contexto. Afinal, além de responder os clientes em um tempo menor, é possível gastar menos recursos computacionais no processamento da resposta. Esse gasto menor de recursos computacionais tem como consequência um gasto menor financeiro e, também, de recursos energéticos e ambientais. Portanto, não é à toa que um bom desempenho seja algo corriqueiramente procurado por desenvolvedores ao redor de todo o mundo.

Um ponto que possui grande impacto no desempenho de uma aplicação é a quantidade de consultas realizadas ao banco de dados. Essas consultas muitas vezes realizam comunicação com um servidor de banco de dados remotamente localizado. Portanto, além do gasto de recurso e tempo no estabelecimento de uma conexão com o banco de dados, existe uma latência nessa comunicação que deixa a operação ainda mais custosa. Ao projetar e desenvolver uma aplicação que servirá uma API GraphQL, é relativamente complicado otimizar o número de consultas realizadas ao banco de dados. Afinal, quando se está servindo um determinado objeto nessa API, é o cliente que decide quais campos e quais objetos relacionados a esse objeto que serão retornados.

Além disso, no processamento de um pedido GraphQL no *framework* em estudo neste trabalho, é realizado um processamento dos dados em cascata. Ou seja, supondo uma API que possui um objeto Post, relacionado a uma lista de objetos chamados Comment, que por sua vez está relacionado a um Profile (Figura 3.1), o funcionamento é o seguinte. Em um pedido como o da Figura 3.2, Primeiro é executado o *resolver* do campo post de Query. Em seguida, para se resolver o campo *comments* desse Post é executado o seu método *resolve_comments*, que retorna uma lista de comentários. Depois disso, para cada comentário, é necessário se resolver os campos *body* e *profile*. Ou seja, já é possível observar que, caso seja realizado uma consulta ao banco no *resolve_profile*, do objeto Comment, teremos N consultas, sendo N o número

de comentários nesse Post.

```
1  class Profile(ObjectType):
2      name = graphene.String()
3
4  > def resolve_name(root, info): ...
5
6
7  class Comment(ObjectType):
8      profile = graphene.Field(Profile)
9      body = graphene.String()
10
11 > def resolve_profile(root, info): ...
12
13
14 > def resolve_body(root, info): ...
15
16
17 class Post(ObjectType):
18     comments = graphene.List(Comment)
19
20 > def resolve_comments(root, info): ...
21
22
23 class Query(ObjectType):
24     post = graphene.Field(Post, id=graphene.ID())
25
26 > def resolve_post(root, info, id): ...
27
28
```

Figura 3.1: Schema GraphQL simples

```
1  query {
2      post (id: "123") {
3          comments {
4              body
5              profile {
6                  name
7              }
8          }
9      }
10 }
11
```

Figura 3.2: Requisição GraphQL contra *schema* da Figura 3.1

Contudo, quando o *framework* em questão é algum como Graphene-Django, que possui ciência tanto sobre o ORM da aplicação quando o Schema (definições dos objetos) GraphQL, é possível se otimizar essas requisições logo no início do processamento da resposta. É possível, por exemplo, realizar a operação de *prefetch_related* em forma de cascata até os objetos mais profundos da requisição GraphQL. Assim, a quantidade de consultas realizadas não seria $O(N)$, sendo N o número de comentários no Post em questão, e sim $O(1)$. Porém, esse *framework* não possui ainda essa funcionalidade e é neste trabalho que é realizada essa implementação.

Utilizando esta extensão criada, o desenvolvedor da API não precisa optar nem pelo caminho de aceitar o baixo desempenho e nem pelo caminho de prever os pedidos dos clientes para, em cada *resolver*, realizar operações como *prefetch_related* e *annotate*. Com essa extensão é possível realizar definições sucintas que resultam em um número otimizado de consultas ao banco de dados, mantendo os benefícios de uma API GraphQL.

3.2

Requisitos da Extensão

O principal objetivo deste projeto de extensão é que o *framework* torne o desenvolvimento e manutenção da API mais simples e eficiente. Para isso, é importante que seja responsabilidade do próprio *framework* montar parte das consultas SQL realizadas ao banco de dados, com base no que for pedido pelo cliente em cada requisição. Com isso, torna-se menos trabalhoso para o desenvolvedor a implementação de um *resolver* de um determinado campo. Pois, assim, o desenvolvedor pode focar seus esforços no que realmente é requisito específico daquele projeto e, certamente, não pode ser previsto ou implementado pelo próprio *framework*. Nesta seção serão descritos os dois requisitos deste projeto de extensão.

Facilitar o desenvolvimento de *resolvers* de campos numa API GraphQL é algo significativo por conta da isso ser um ponto central tanto do conceito teórico do GraphQL quanto da prática de sua implementação. Considerando uma aplicação Django com um banco de dados relacional, o trabalho de se adicionar uma API GraphQL a esta aplicação se dá em definir os objetos, seus campos e os *resolvers* desses campos. A parte de definição de objetos e campos é algo completamente relacionado às especificidades de cada projeto de API. Já a parte de definição de *resolvers* pode ser analisada em duas partes: uma que se refere aos requisitos específicos desse projeto e outra que se refere ao que já foi apresentado como uma espécie de “preparação” das consultas SQL que serão realizadas para se montar o retorno. Essa parte de “preparação” refere-se a, por exemplo, utilizar os recursos do Django para otimização do número de consultas SQL e é o ponto que possui maior possibilidade de ser totalmente automatizado pelo próprio *framework*. Portanto, este projeto de extensão refere-se, de forma simplificado, em tornar papel do *framework* esse trabalho denominado de “preparação” das consultas SQL.

Esse trabalho de preparação se dará após o retorno dos *resolvers* dos campos que serão desenvolvidos pelo desenvolvedor da API. Na Figura 3.3 está um exemplo de definição de modelos Django, objetos e campos GraphQL. Nessa API, supondo que algum cliente faça uma requisição como a da Fi-

gura 3.4, seria desejável que os os relacionamentos *residents* e *favorite_foods* fossem *prefetched* no banco de dados logo quando o objeto inicial da requisição, a casa, fosse buscado. Assim, é possível realizar $O(1)$ consultas ao banco, em contrapartida à alternativa de deixar os relacionamentos serem carregados da forma padrão preguiçosa (*lazy loading*, em Django), que resultaria em $O(N)$ consultas, sendo N o número de residentes dessa casa.

```

1  # ---- MODELS ----
2  class Food(Model):
3      name = CharField('name', max_length=255)
4
5
6  class Person(Model):
7      name = CharField('name', max_length=255)
8      home = ForeignKey('home', related_name='residents')
9      favorite_foods = ManyToManyField('favorite_foods', related_name='+')
10
11
12 class House(Model):
13     address = CharField('address', max_length=255)
14
15 # ---- GraphQL Objects ----
16 class FoodType(DjangoObjectType):
17     class Meta:
18         model = Food
19         fields = ['name']
20
21
22 class PersonType(DjangoObjectType):
23     class Meta:
24         model = Person
25         fields = ['name', 'favorite_foods']
26
27
28 class HouseType(DjangoObjectType):
29     class Meta:
30         model = House
31         fields = ['address', 'residents']
32
33
34 class Query(ObjectType):
35     house = graphene.Field(HouseType, id=graphene.ID())
36
37     def resolve_house(root, info, id):
38         logged_user = info.context.user
39         house = models.House.objects.filter(id=id, is_public=True)
40         if not is_house_visible(logged_user, house):
41             return None
42
43         models.House.objects.filter()
44         return house
45

```

Figura 3.3: Definições de uma API: modelos Django e *schema* GraphQL

Nesse exemplo da casa, uma alternativa para se realizar esses pré carregamentos de dados vindo do banco, sem considerar a extensão realizada pelo presente trabalho, seria o ilustrado na Figura 3.5. Porém, nota-se que essa alternativa vai contra o conceito central do GraphQL que é deixar a cargo do cliente decidir o que deseja receber como resposta. Afinal, em uma API robusta, realizar a “preparação” da consulta do banco de dados para todas as possibilidades de pedidos seria um trabalho, além de muito grande, ineficiente

já que as consultas SQL iriam buscar muito mais dados que o necessário, independente do que foi pedido. Portanto, o primeiro requisito do presente trabalho é que, no exemplo de implementação de uma API GraphQL ilustrado na Figura 3.3, o *framework* entenda o que está sendo pedido e intercepte o retorno do *resolver* do campo *house* (do objeto Query) para realizar exatamente as preparações necessárias.

```
1  query {  
2    house (id: "123") {  
3      residents {  
4        name  
5        favorite_foods {  
6          name  
7        }  
8      }  
9    }  
10 }  
11
```

Figura 3.4: Requisição GraphQL contra *schema* da Figura 3.3

```
37 def resolve_house(root, info, id):  
38     logged_user = info.context.user  
39     house = models.House.objects \  
40         .filter(id=id, is_public=True) \  
41         .prefetch_related('residents__favorite_foods')  
42     if not is_house_visible(logged_user, house):  
43         return None  
44  
45     models.House.objects.filter()  
46     return house  
47
```

Figura 3.5: Alteração no *schema* da Figura 3.3 para minimiza o número de consultas SQL no processamento da requisição da Figura 3.4 (quebrando princípios do GraphQL)

O segundo requisito desta extensão é que seja permitida a criação do que seria chamado de “campos anotados”. Os quais são campos, de um objeto GraphQL associado a um modelo Django, que não possuem *resolver*. Em vez disso, possuem métodos que indicam como que estes são anotados em um QuerySet do modelo em questão. Esse requisito pode ser ilustrado ainda no

mesmo exemplo do primeiro requisito. Suponha que o objeto `FoodType` possui um campo que necessita de algum cálculo, com base em outros dados do banco, como ilustrado na Figura 3.6. Suponha também que a requisição GraphQL ilustrada na Figura 3.4 passe a pedir, além do campo `name`, este campo `favorite_ratio` do objeto `FoodType`, que é o mais interno na requisição. Neste caso, mesmo com o primeiro requisito satisfeito, a quantidade de consultas seria proporcional a $P \times F$, sendo P o número de residentes da tal casa e F a quantidade de comidas favoritas de cada pessoa.

```
16 class FoodType(DjangoObjectType):
17     class Meta:
18         model = Food
19         fields = ('name',)
20
21     favorite_ratio = graphene.Float()
22
23     def resolve_favorite_ratio(root, info):
24         favorited_people = Person.objects \
25             .filter(favorite_foods=root) \
26             .distinct().count()
27         total_people = Person.objects.count()
28         return favorited_people / total_people
29
```

Figura 3.6: Definição de um campo que envolve um cálculo com base em dados do banco

Para superar esse problema, é possível desenhar dois caminhos:

(a) O desenvolvedor prevê as requisições GraphQL que serão feitas e realiza pré carregamento dos dados, ainda no *resolver* de entrada, que é o da casa a partir de um identificador. (Figura 3.7)

(b) O *framework* lida com isso e necessita, apenas, saber como que se anota tal campo, chamado `favorite_ratio`, num `QuerySet` do modelo `Food`. (Figura 3.8)

Pode-se afirmar que a abordagem “b” é melhor que a “a” por uma série de motivos. Primeiro por ser mais concisa e limpa. Segundo por não ir contra o princípio de GraphQL de ter de prever exatamente como serão as requisições dos clientes. Terceiro por um motivo que não está exemplificado nessa ilustração, que é o cenário de existirem muitos “pontos de entrada” (campos do objeto `Query` ou `Mutation`) que possam confluir em um objeto do tipo `FoodType`. Nesse cenário, o desenvolvedor teria obrigatoriamente que realizar essa operação do caminho “a” para todos esses pontos de entrada e, caso não o faça, teriam requisições GraphQL cujas respostas iriam ter o valor de `favorite_ratio` sempre como nulo, por não ter sido anotado.

```

16 class FoodType(DjangoObjectType):
17     class Meta:
18         model = Food
19         fields = ['name']
20
21         favorite_ratio = graphene.Float()
22
23     def resolve_favorite_ratio(root, info):
24         return getattr(root, 'favorite_ratio', None) # Safe getattr in case the annotation is not done
25
26 > class PersonType(DjangoObjectType):--
27
28
29
30
31
32 > class HouseType(DjangoObjectType):--
33
34
35
36
37
38 class Query(ObjectType):
39     house = graphene.Field(HouseType, id=graphene.ID())
40
41     def resolve_house(root, info, id):
42         food_queryset = Food.objects.all()
43         food_queryset = annotate_favorite_ratio(food_queryset)
44
45         person_queryset = Person.objects \
46             .prefetch_related(Prefetch('favorite_foods', queryset=food_queryset))
47
48         house = models.House.objects \
49             .filter(id=id, is_public=True) \
50             .prefetch_related(Prefetch('residents', queryset=person_queryset))
51
52         logged_user = info.context.user
53         if not is_house_visible(logged_user, house):
54             return None
55
56         models.House.objects.filter()
57         return house
58

```

Figura 3.7: Alteração no *schema* da Figura 3.3 para minimiza o número de consultas SQL no processamento de uma requisição similar à da Figura 3.4 com a adição do campo *favorite_ratio* no pedido (quebrando princípios do GraphQL)

```

16 class FoodType(DjangoObjectType):
17     class Meta:
18         model = Food
19         fields = ['name']
20
21         favorite_ratio = graphene.Float()
22
23     def annotate_favorite_ratio(queryset, info):
24         return annotate_favorite_ratio(queryset)
25
26 > class PersonType(DjangoObjectType):--
27
28
29
30
31
32 > class HouseType(DjangoObjectType):--
33
34
35
36
37
38 class Query(ObjectType):
39     house = graphene.Field(HouseType, id=graphene.ID())
40
41     def resolve_house(root, info, id):
42         logged_user = info.context.user
43         house = models.House.objects.filter(id=id, is_public=True)
44         if not is_house_visible(logged_user, house):
45             return None
46
47         models.House.objects.filter()
48         return house
49

```

Figura 3.8: Cenário de uso da extensão deste trabalho para se minimizar o número de consultas SQL (sem quebrar princípios do GraphQL)

Portanto tornam-se claros os benefícios ao desenvolvimento e manutenção do software de APIs GraphQL que utilizem a extensão produzida no presente trabalho. Os requisitos desta extensão, que estão descritos nesta seção, podem ser sintetizados como os seguintes:

1 - realização de *prefetch* dos objetos que são relacionados a nível de modelos Django (banco de dados), automaticamente com base na própria requisição que está sendo processada, para otimização do número de consultas ao banco de dados.

2 - permitir a definição de “campos anotados”, os quais o *framework* garante serem requisitados ao banco de dados logo no momento de consulta daquela determinada tabela no banco, sem aumentar o número de consultas.

3.3

Projeto da Extensão

Tanto o *framework* Django quanto o Graphene-Django utilizam um conceito chamado metaclasses, que é uma funcionalidade presente em parte das linguagens de programação orientadas a objeto (Cunningham, 2018). Uma metaclasses é uma espécie de classe cujas instâncias são classes e não objetos propriamente ditos. Da mesma forma que uma classe define características de suas instâncias, uma metaclasses define características de suas classes. Esse conceito é importante para o entendimento do funcionamento do *framework* estendido. No Python, a conversão já estabelecida pela própria linguagem é um método chamado `__init_subclass__`, que pode ser definido em qualquer classe. Então próximas classes que tiverem essa como superclasse terão esse método chamado a nível de inicialização da classe si, que ocorre durante a interpretação da definição da classe estendida. Ou seja, acontece apenas uma vez durante o ciclo de interpretação e execução de um *script* Python.

O *framework* Graphene, sobre o qual o Graphene-Django existe, estabelece um método chamado `__init_subclass__with_meta__`, que funciona como o `__init_subclass__` mas possui a facilidade de já entregar nos parâmetros dessa função os atributos da classe Meta definida dentro da subclasse sendo interpretada. Essa classe Meta já foi evidenciada em figuras no presente trabalho. Sua utilização é consideravelmente mais simples que uma explicação verbal. Na Figura 3.9 está ilustrado um exemplo de uso desse método, que é definido pelo BaseType, que é a classe base dos tipos no Graphene.

É a partir desse mecanismo que o Graphene-Django determina os tipos e *resolvers* de campos que estão no atributo *fields* da classe Meta de subclasses do DjangoObjectType. Portanto, o presente trabalho de extensão tem como objetivo utilizar esses conceitos para que atingir os requisitos descritos na Seção

```
1 class A(BaseType):
2     def __init_subclass_with_meta__(cls, **options):
3         text = options.get('text')
4         cls.print_my_text = lambda: print(text)
5
6     class B(A):
7         class Meta:
8             text = 'This is my text'
9
10    B.print_my_text() # -> prints 'This is my text'
11
```

Figura 3.9: Utilização do método `__init_subclass_with_meta__`, convencionado no tipo base do Graphene

3.2. O objetivo é criar funções que “preparem” o retorno de cada *resolver*, com base no que está sendo pedido em determinada requisição GraphQL. Para isso, a metaclasses de DjangoObjectType deve mapear quais são as preparações necessárias, para cada campo deste objeto separadamente. Assim, dependendo dos campos que forem pedidos pelo cliente, determinadas funções serão chamadas e outras não.

Essas funções de preparo tem como objetivo realizar, por exemplo, *prefetch_related* e *annotate*. O objetivo é também realizar essas preparações em cascata. Suponha dois modelos Django, A e B, que possuem um relacionamento entre si e que, no pedido GraphQL, esse relacionamento foi pedido. Nesse caso, quando o objeto A for preparado, é desejável que se realize o *prefetch* do modelo B logo na QuerySet do A. Nesse momento, é importante que o QuerySet de B que será usado no *prefetch* passe pelo processo de preparo também. Afinal, nesse processo de preparo de B, além de anotações eventualmente necessárias, também podem ser realizados alguns pré carregamentos de outros objetos relacionados a B que podem ter sido pedidos nessa requisição GraphQL. Ou seja, existe o que é chamado de “preparação em cascata”.

Além disso, é importante que o próprio *framework* dê início a esse processo de preparação em cascata, ou seja, que haja o que é chamado de “preparação automática”. Sem isso, seria necessário que o desenvolvedor preparasse todo QuerySet ou instância de modelo Django retornados em *resolvers*. Essa necessidade, contudo, seria muito prejudicial à agilidade do desenvolvimento, assim como a qualidade de software. Isso porque, em um cenário de esquecimento de preparo de um desses retornos, haveria casos de retorno da API em que, além de baixo desempenho, campos anotados seriam sempre retornados como nulo por não terem sido de fato anotados. Ou seja,

seria uma aplicação que depende de um trabalho repetitivo em diversos pontos do código para que não haja *bugs*. Por isso, a preparação automática é um ponto essencial para que esta extensão seja um impulsionador da qualidade de software de um projeto.

A regra para quando o *framework* deve interceptar o retorno de um *resolver* para prepará-lo é simples. Basta que sejam interceptados *resolvers* de campos cujo tipo dê em um `DjangoObjectType`, com exceção de quando trata-se de um campo presente no atributo *fields* da classe `Meta` de um `DjangoObjectType`. Afinal, quando trata-se de um caso desses, ele já será preparado em cascata. Na Figura 3.10 estão ilustradas todas as possibilidades de relacionamento entre `ObjectType` e `DjangoObjectType` e quais devem ter o *resolver* interceptado pelo *framework*. Os relacionamentos que devem ser interceptados estão representados como uma seta mais grossa e da cor verde. Mais detalhes sobre essa interceptação estão descritos na Seção 3.4.

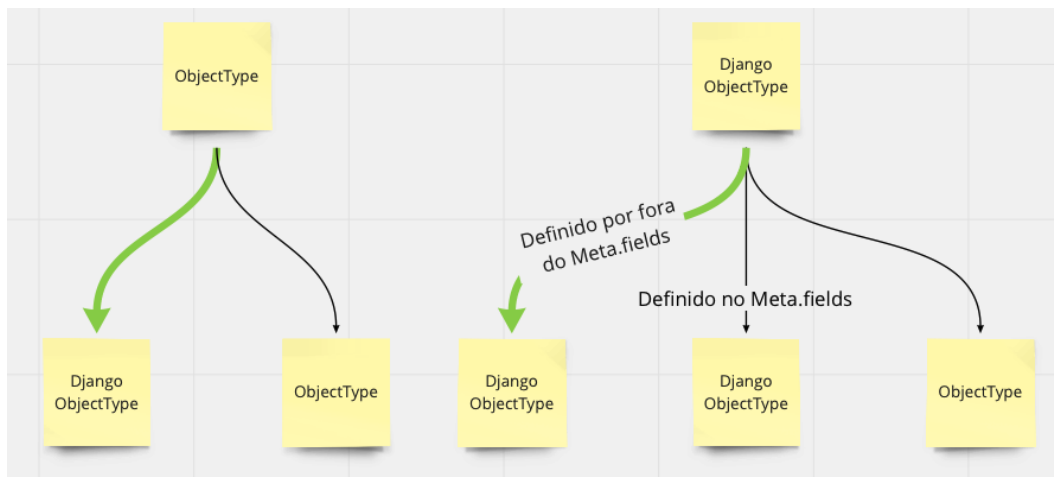


Figura 3.10: Ilustração dos casos de resolvimento de campos, indicando em quais deve haver o início de um preparo em cascata (seta verde e grossa: deve haver; seta preta e fina: não deve haver)

Com essa interceptação de *resolvers* para preparação, junto à preparação em cascata, é possível afirmarmos que todo retorno de um `DjangoObjectType` em uma requisição GraphQL vai envolver seu preparo. Assim, torna-se seguro dizer que, por exemplo, um campo anotado de um determinado tipo de objeto sempre será anotado quando for pedido pelo cliente. Além disso, pode-se dizer que o número de consultas para capturar os dados no banco de dados será otimizado.

3.4 Implementação

A implementação dessa extensão pode ser visualizada integralmente no link <https://github.com/FernandoTancini/Graphene-Django/pull/1>. Nesta seção estão descritos os dois pontos de implementação que são mais relevantes para se alcançar os objetivos da extensão como planejada. O primeiro ponto é a introdução da funcionalidade de preparação em cascata à classe `DjangoObjectType`. Já o segundo é relacionado à interceptação dos *resolvers* necessárias para se iniciar essas preparações em cascata, que também pode ser chamado de “preparo automático”.

Na Figura 3.11 é possível observar o único incremento realizado no método `__init_subclass_with_meta__` da classe `DjangoObjectType`. O objetivo desse pedaço de código é chamar os seus novos métodos internos de registro de campos Django e campos anotados. Esses métodos de registro encontram-se, respectivamente, na Figura 3.12 e Figura 3.13. Esse registro se refere à interpretação de quais são as operações necessárias, em um `QuerySet` do modelo Django ao qual esse `DjangoObjectType` se refere, para que os dados sejam buscados no banco de dados durante a própria execução do `QuerySet`. Durante esse registro, são definidas funções, que recebem e retornam um `QuerySet`. Essas funções são salvas na própria classe que está sendo interpretada, dentro do dicionário `__preparation_functions_by_field__`. Esse dicionário possui como chave o nome de um campo e como valor uma lista das funções de preparo que esse campo exige. Adicionar uma função a esse dicionário é uma tarefa realizada pelo método `__append_field_preparation_function__`, que encontra-se na Figura 3.14. Vale ressaltar que a escolha de se guardar uma lista de funções está relacionada apenas à extensibilidade futura, prevendo possíveis casos de próximas extensões em que isso torna-se necessário. Afinal, pelas funcionalidades de hoje, que são apenas campos Django e anotados, não existe a possibilidade de um campo ter mais de uma função de preparo.

```

@@ -255,6 +262,10 @@ def __init_subclass_with_meta__(
255 262         _meta.fields = django_fields
256 263         _meta.connection = connection
257 264
265 +         cls.__preparation_functions_by_field = {}
266 +         cls._register_annotation_fields()
267 +         cls._register_django_fields(model, django_fields)
268 +
269     super(DjangoObjectType, cls).__init_subclass_with_meta__(
270         _meta=_meta, interfaces=interfaces, **options
271     )

```

Figura 3.11: Código introduzido no método `__init_subclass_with_meta__` da classe `DjangoObjectType`

```

349 +
350 + @classmethod
351 + def _register_django_fields(cls, model, django_fields):
352 +     for field_name in django_fields.keys():
353 +         cls._register_django_field(model, field_name)
354 +
355 + @classmethod
356 + def _register_django_field(cls, model, field_name):
357 +     model_field = get_model_field(model, field_name)
358 +     related_model = model_field.related_model
359 +     if not related_model:
360 +         return
361 +
362 +     def prepare_function(queryset, selection, info):
363 +         graphene_type = get_graphene_type_of_field(cls, field_name)
364 +         if issubclass(graphene_type, Connection):
365 +             return queryset
366 +
367 +         related_qs = related_model.objects.all()
368 +         related_qs = graphene_type._prepare_in_cascade(related_qs, selection, info)
369 +         return queryset.prefetch_related(Prefetch(field_name, queryset=related_qs))
370 +
371 +     cls._append_field_preparation_function(field_name, prepare_function)
372 +

```

Figura 3.12: Métodos relacionados ao registro de campos Django na classe DjangoObjectType

```

372 +
373 + @classmethod
374 + def _register_annotation_fields(cls):
375 +     for field_name in dir(cls):
376 +         field_specification = getattr(cls, field_name)
377 +         if not isinstance(field_specification, (UnmountedType, MountedType)):
378 +             continue
379 +
380 +         annotate_function = getattr(cls, f"annotate_{field_name}", None)
381 +         if annotate_function:
382 +             cls._register_annotation_field(field_name, annotate_function)
383 +
384 + @classmethod
385 + def _register_annotation_field(cls, field_name, annotate_function):
386 +     def _annotate_function(root_qs, _selection, info):
387 +         return annotate_function(root_qs, info)
388 +
389 +     cls._append_field_preparation_function(field_name, _annotate_function)
390 +

```

Figura 3.13: Métodos relacionados ao registro de campos anotados na classe DjangoObjectType

```

340 +
341 + @classmethod
342 + def _append_field_preparation_function(cls, snake_case_name, preparation_function):
343 +     camel_case_name = to_camel_case(snake_case_name)
344 +
345 +     functions = cls._preparation_functions_by_field.get(camel_case_name, [])
346 +     functions.append(preparation_function)
347 +
348 +     cls._preparation_functions_by_field[camel_case_name] = functions
349 +

```

Figura 3.14: Método da classe DjangoObjectType cujo papel é adicionar entradas ao dicionário `_preparation_functions_by_field`

Com o dicionário `_preparation_functions_by_field` criado no momento de interpretação de uma subclasse de DjangoObjectType, torna-se possível

a implementação do método que realiza esse preparo como um todo com base no que foi pedido pelo cliente. Na Figura 3.15 é possível observar esse método, chamado *prepare*. Esse método recebe como parâmetro o valor a ser preparado e um objeto do tipo `GraphQLResolveInfo`, que reúne uma série de informações relacionadas à requisição como um todo, incluindo o pedido do cliente. A partir desses parâmetros, o método lida com o tipo de valor e chama um método interno chamado *__prepare_in_cascade* que tem como objetivo receber sempre um `QuerySet` e a seleção realizada pelo cliente (os campos pedidos), para então realizar todas preparações necessárias. Nesse ponto que as preparações começam a serem realizadas em cascata, pois possivelmente métodos de preparação de campos Django serão chamados e eles, por sua vez, realizam a preparação e o *prefetch* de um `QuerySet` do modelo relacionado, como pode ser observado na Figura 3.12.

```

308 + @classmethod
309 + def prepare(cls, value, info):
310 +     if isinstance(value, Model):
311 +         queryset = value._meta.model.objects.filter(pk=value.pk)
312 +     elif isinstance(value, QuerySet):
313 +         queryset = value
314 +     else:
315 +         return value
316 +
317 +     selection = info.field_nodes[0]
318 +     queryset = cls._prepare_in_cascade(queryset, selection, info)
319 +
320 +     if isinstance(value, Model):
321 +         return queryset.first()
322 +     else:
323 +         return queryset
324 +
325 + @classmethod
326 + def _prepare_in_cascade(cls, queryset, selection, info):
327 +     if not issubclass(queryset.model, cls._meta.model):
328 +         raise Exception(
329 +             f"{cls.__name__}._prepare_in_cascade() received a queryset from {queryset.model} model"
330 +         )
331 +
332 +     for sub_selection in selection.selection_set.selections:
333 +         functions = cls._preparation_functions_by_field.get(
334 +             sub_selection.name.value, []
335 +         )
336 +         for func in functions:
337 +             queryset = func(queryset, sub_selection, info)
338 +
339 +     return queryset
340 +

```

Figura 3.15: Métodos relacionados à preparação de um valor já resolvido, com base no que o cliente pediu e nos campos registrados nesse DjangoObjectType

A parte de lidar com o valor a ser preparado, que acontece no método *prepare*, é um ponto importante do sistema de preparação. Isso porque a preparação é algo que envolve consultas ao banco de dados e, por isso, deve acontecer por meio de um `QuerySet`. Porém, o retorno de um *resolver* só pode ser em formato de `QuerySet` quando o tipo daquele campo for uma lista objetos.

Quando o valor a ser resolvido é um único objeto, é natural que no *resolver* desse campo seja retornada apenas uma instância desse objeto. Contudo, deve-se transformar essa instância em um *QuerySet*, a fim de realizar novas consultas ao banco de dados para, após a preparação, transformar esse *QuerySet* em uma instância novamente. Isso ocorre no método *prepare* e possui algumas limitações como consequência, que serão melhor aprofundadas na Seção 4.2.

O segundo ponto que é importante para que o objetivo do projeto de extensão seja alcançado está relacionado ao “preparo automático”. Isso é alcançado pela extensão da classe *Schema*, da biblioteca *Graphene*. Uma instância dessa classe representa uma API *GraphQL* e é responsável pela execução de requisições *GraphQL*. Na Figura 3.16 encontra-se a extensão dessa classe, que foi chamada de *DjangoSchema*, com sua alteração realizada no seu método `__init__`, que é responsável pela inicialização de uma instância dessa classe. Ainda observando a Figura 3.16, é possível notar que o objetivo da extensão do método `__init__` consiste em aceitar um novo parâmetro chamado *automatic_preparation*, que por padrão tem *False* como valor. Caso esse parâmetro seja passado como *True*, é realizado um embrulho dos *resolvers*, pelo método `_wrap_resolvers_for_preparation`. A escolha de tornar esse comportamento opcional se deu ao fato de possíveis problemas de compatibilidade com outras funcionalidades dessa ou de outras bibliotecas, que está melhor descrito na Seção 4.2. Na Figura 3.17 encontra-se a implementação desse método responsável pelos embrulhos.

```
11 + class DjangoSchema(graphene.Schema):
12 +     def __init__(self, *args, **kwargs):
13 +         automatic_preparation = kwargs.pop("automatic_preparation", False)
14 +
15 +         super().__init__(*args, **kwargs)
16 +
17 +         if automatic_preparation:
18 +             self._wrap_resolvers_for_preparation()
19 +
```

Figura 3.16: Extensão da classe *Schema*, chamada *DjangoSchema*

```
20 +     def _wrap_resolvers_for_preparation(self):
21 +         types = self.graphql_schema.type_map.values()
22 +         object_types = [
23 +             _type for _type in types if isinstance(_type, GraphQLObjectType)
24 +         ]
25 +
26 +         for object_type in object_types:
27 +             root_graphene_type = getattr(object_type, "graphene_type", None)
28 +
29 +             for field_name, field in object_type.fields.items():
30 +                 field_type = get_underlying_type(field.type)
31 +                 field_graphene_type = getattr(field_type, "graphene_type", None)
32 +
33 +                 if self._should_wrap_resolver(
34 +                     root_graphene_type, field_name, field_graphene_type
35 +                 ):
36 +                     field.resolve = self._wrap_resolver(
37 +                         field.resolve, field_graphene_type.prepare
38 +                     )
39 +
```

Figura 3.17: Método da classe DjangoSchema cujo objetivo é embrulhar os métodos de resolvimento de campos que devem ter seus retornos automaticamente preparados

A implementação do método `_wrap_resolvers_for_preparation` consiste em, de forma simplificado, iterar sobre os tipos GraphQL que são objetos e, para cada um, iterar sobre seus campos para embrulhar aqueles que devem ser embrulhados. O método que determina se um campo desse tipo GraphQL deve ser embrulhado ou não é o `_should_wrap_resolver`, que encontra-se na Figura 3.18. Já o embrulho em si é realizado pelo método `_wrap_resolver`, que encontra-se na Figura 3.19.

O método `_should_wrap_resolver` identifica os campos que devem ter o `resolver` embrulhado com base na mesma lógica representada na Figura 3.10. A sua implementação baseia-se em sempre retornar `True`, caso não entre em um dos dois casos de *early returns* negativos. Esses dois casos de exceção são: (1) o campo não é do tipo `DjangoObjectType`; (2) a classe em análise é do tipo `DjangoObjectType` e o campo encontra-se no atributo `fields` da classe `Meta`, que no código é denominado como “campo Django”. Determinar se um campo é “campo Django” é uma tarefa realizada por um segundo método que também encontra-se na Figura 3.18, chamado `_is_django_field`. O qual tem o papel de apenas lidar com as diferenças de nomes em Python (*snake case*) e GraphQL (*camel case*) e verificar se o campo encontra-se no atributo `fields` da classe `Meta` dessa classe.

Por fim, a tarefa de embrulhar de fato os `resolvers` é responsabilidade do método `_wrap_resolver`. Na Figura 3.19, é possível observar que sua

```

40 +     @staticmethod
41 +     def _should_wrap_resolver(graphene_type, field_name, field_graphene_type):
42 +         if not (
43 +             field_graphene_type and isinstance(field_graphene_type, DjangoObjectType)
44 +         ):
45 +             return False
46 +
47 +         if isinstance(graphene_type, DjangoObjectType):
48 +             is_django_field = DjangoSchema._is_django_field(graphene_type, field_name)
49 +             if is_django_field:
50 +                 # It is already prepared of in the cascade preparation
51 +                 return False
52 +
53 +         return True
54 +
55 +     @staticmethod
56 +     def _is_django_field(django_object_type, camel_case_field):
57 +         camel_case_django_fields = [
58 +             to_camel_case(f_name) for f_name in django_object_type._meta.fields
59 +         ]
60 +         return camel_case_field in camel_case_django_fields
61 +

```

Figura 3.18: Método da classe DjangoSchema cujo objetivo é determinar caso um campo deve ter seu *resolver* embrulhado ou não

```

62 +     @staticmethod
63 +     def _wrap_resolver(resolve, prepare):
64 +         def wrapped_resolver(root, info, *args, **kwargs):
65 +             resolved_value = resolve(root, info, *args, **kwargs)
66 +             return prepare(resolved_value, info)
67 +
68 +         return wrapped_resolver

```

Figura 3.19: Método da classe DjangoSchema cujo objetivo é embrulhar um *resolver* em um novo método e retorná-lo

implementação é simples. Ela consiste em retornar um novo método que internamente chama o *resolver*. Porém, esse novo método prepara o retorno do *resolver* antes de retorná-lo. Poupano, assim, o desenvolvedor de realizar essa tarefa no final de todo *resolver* que fosse necessário.

Nesta seção estão os dois pontos principais da extensão, assim como a maioria do código sendo introduzido ao *framework* Graphene-Django. No link <https://github.com/FernandoTancini/Graphene-Django/pull/1> encontra-se a extensão completa.

3.5

Prova de Conceito

Para a prova de conceito, foi criado um repositório de um projeto fictício chamado *sample_app*, que encontra-se no link <https://github.com/>

FernandoTancini/tcc. Nesta seção estão destacados apenas os pontos desse projeto fictício que são importantes para o entendimento da prova de conceito da extensão do *framework* Graphene-Django realizada. Porém, todos detalhes de sua implementação podem ser visualizados nesse repositório.

Na Figura 3.20 estão as declarações de modelos Django desse projeto fictício. O banco de dados desse projeto de exemplo consiste em quatro principais modelos: Profile (perfil), Post (publicação), Comment (comentário) e CommentReaction (reação de comentário).

```
6 class Profile(models.Model):
7     class Meta:
8         verbose_name = 'profile'
9         verbose_name_plural = 'profiles'
10
11     name = models.CharField('name', max_length=255)
12
13 class Post(models.Model):
14     class Meta:
15         verbose_name = 'social interaction'
16         verbose_name_plural = 'social interactions'
17
18     profile = models.ForeignKey('base.Profile', deletion.CASCADE, related_name='posts')
19
20     date = models.DateField('date')
21     text = models.CharField('text', max_length=255)
22
23
24 class Comment(models.Model):
25     class Meta:
26         verbose_name = 'comment'
27         verbose_name_plural = 'comments'
28
29     post = models.ForeignKey('base.Post', deletion.CASCADE, related_name='comments')
30     profile = models.ForeignKey('base.Profile', deletion.CASCADE, related_name='comments')
31
32     date = models.DateField('date')
33     text = models.CharField('text', max_length=255)
34
35
36 class CommentReaction(models.Model):
37     class Meta:
38         verbose_name = 'comment reaction'
39         verbose_name_plural = 'comment reactions'
40
41     comment = models.ForeignKey('base.Comment', deletion.CASCADE, related_name='reactions')
42     profile = models.ForeignKey('base.Profile', deletion.CASCADE, related_name='reactions')
43
44     date = models.DateField('date')
45     kind = models.CharField('kind', max_length=255, choices=enums.ReactionKind.to_choices())
46
```

Figura 3.20: Definição dos modelos Django do projeto *sample_app*, que é utilizado na prova de conceito

Para se realizar a prova de conceito, nesse projeto também foi criada uma API GraphQL a partir da extensão do Graphene-Django realizada. A implementação dessa API envolve as definições de tipos GraphQL e, por último, a definição do objeto DjangoSchema. Essa prova de conceito consiste na

comparação da execução de uma mesma requisição GraphQL em dois cenários diferentes: (1) sem utilizar as novas funcionalidades introduzidas ao *framework* por esse extensão e (2) utilizando essas funcionalidades.

Portanto, existem dois cenários da definição do *schema* GraphQL, um sem funcionalidades novas, Figura 3.21, e outro com novas funcionalidades, Figura 3.22. É possível observar que as diferenças entre eles estão em dois pontos do código, relacionados ao campo *is_from_bully* e a inicialização da instância *schema*.

```

10 class ProfileType(DjangoObjectType):
11     class Meta:
12         model = Profile
13         fields = ['name']
14
15
16 class PostType(DjangoObjectType):
17     class Meta:
18         model = Post
19         fields = ['date', 'text', 'profile', 'comments']
20
21
22 class CommentType(DjangoObjectType):
23     class Meta:
24         model = Comment
25         fields = ['date', 'text', 'profile', 'reactions']
26
27
28 class CommentReactionType(DjangoObjectType):
29     class Meta:
30         model = CommentReaction
31         fields = ['date', 'kind', 'profile']
32
33     is_from_bully = graphene.Boolean()
34
35     def resolve_is_from_bully(root, info):
36         profile_qs = Profile.objects.filter(pk=root.profile_id)
37         is_from_bully = annotate_is_bully(profile_qs).first().is_bully
38
39         return is_from_bully
40
41     # def annotate_is_from_bully(root_queryset, info):
42     #     profile_qs = Profile.objects.filter(pk=OuterRef('profile'))
43     #     is_from_bully = annotate_is_bully(profile_qs).values('is_bully')
44     #     return root_queryset.annotate(is_from_bully=Subquery(is_from_bully))
45
46
47
48 class Query(graphene.ObjectType):
49     debug = graphene.Field(DjangoDebug, name="_debug")
50     feed_posts = graphene.List(
51         PostType,
52         offset=graphene.Int(required=True),
53         limit=graphene.Int(required=True))
54
55     def resolve_feed_posts(root, info, offset, limit):
56         return Post.objects.all()[offset:limit]
57
58
59 schema = Schema(query=Query)
60 # schema = DjangoSchema(query=Query, automatic_preparation=True)
61

```

Figura 3.21: Definição do *schema* do projeto *sample_app* sem utilizar as funcionalidades introduzidas pelo trabalho de extensão (que encontram-se comentadas)

```

10 class ProfileType(DjangoObjectType):
11     class Meta:
12         model = Profile
13         fields = ['name']
14
15
16 class PostType(DjangoObjectType):
17     class Meta:
18         model = Post
19         fields = ['date', 'text', 'profile', 'comments']
20
21
22 class CommentType(DjangoObjectType):
23     class Meta:
24         model = Comment
25         fields = ['date', 'text', 'profile', 'reactions']
26
27
28 class CommentReactionType(DjangoObjectType):
29     class Meta:
30         model = CommentReaction
31         fields = ['date', 'kind', 'profile']
32
33     is_from_bully = graphene.Boolean()
34
35     # def resolve_is_from_bully(root, info):
36     #     profile_qs = Profile.objects.filter(pk=root.profile_id)
37     #     is_from_bully = annotate_is_bully(profile_qs).first().is_bully
38
39     #     return is_from_bully
40
41     def annotate_is_from_bully(root_queryset, info):
42         profile_qs = Profile.objects.filter(pk=OuterRef('profile'))
43         is_from_bully = annotate_is_bully(profile_qs).values('is_bully')
44
45         return root_queryset.annotate(is_from_bully=Subquery(is_from_bully))
46
47
48 class Query(graphene.ObjectType):
49     debug = graphene.Field(DjangoDebug, name="_debug")
50     feed_posts = graphene.List(
51         PostType,
52         offset=graphene.Int(required=True),
53         limit=graphene.Int(required=True))
54
55     def resolve_feed_posts(root, info, offset, limit):
56         return Post.objects.all()[offset:limit]
57
58
59 # schema = Schema(query=Query)
60 schema = DjangoSchema(query=Query, automatic_preparation=True)
61

```

Figura 3.22: Definição do *schema* do projeto *sample_app* utilizando as funcionalidades introduzidas pelo trabalho de extensão

A diferença das duas abordagens com relação a classe do *schema* se dá pelo uso da nova classe chamada *DjangoSchema*, com preparação automática configurada. Vale notar que esse ponto sozinho já diminui bastante o número de consultas SQL. Porém, sem a troca do método *resolve_is_from_bully* pela sua forma de anotação, ainda haveria N consultas, sendo N o número de reações de comentário retornadas. Portanto, nesse cenário de utilização do *DjangoSchema*, é recomendado dar preferência a criar “campos anotados” em vez de criar *resolvers* que realizam consultas SQL.

A consulta GraphQL utilizada na prova de conceito encontra-se na Figura 3.23. Nota-se nessa consulta que, além campo *feedPosts*, é solicitado do objeto *Query* um campo chamado *__debug*. Esse campo, que tem como objetivo

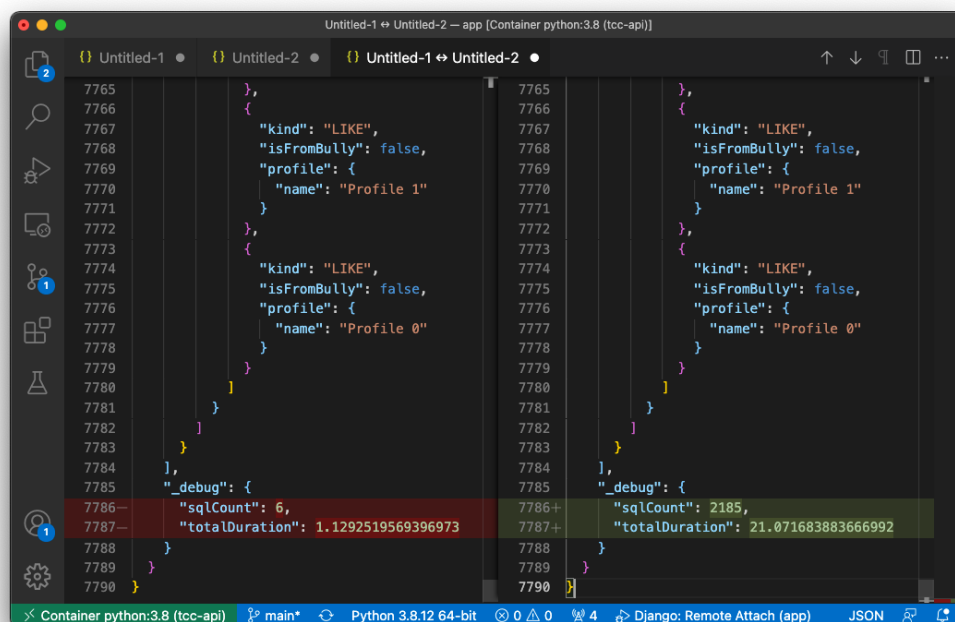
reunir algumas informações relacionadas à execução da solicitação GraphQL, já era uma funcionalidade do Graphene-Django anterior à extensão do presente trabalho. Porém, esse objeto possuía campos que não seriam tão diretamente úteis à prova de conceito. Portanto, para facilitar o presente trabalho, foram criados esses dois novos campos de utilidade para análise da execução de requisições GraphQL. Esses campos são: *sqlCount* (quantidade de consultas SQL) e *totalDuration* (duração da requisição GraphQL como um todo).

```
1  query {  
2    feedPosts (offset: 0, limit: 4) {  
3      date  
4      text  
5      profile {  
6        name  
7      }  
8      comments {  
9        date  
10       text  
11       profile {  
12         name  
13       }  
14       reactions {  
15         kind  
16         isFromBully  
17         profile {  
18           name  
19         }  
20       }  
21     }  
22   }  
23   _debug {  
24     sqlCount  
25     totalDuration  
26   }  
27 }  
28
```

Figura 3.23: Consulta GraphQL utilizada na prova de conceito

A resposta da API para a consulta GraphQL da Figura 3.23 nos cenários sem e com as funcionalidades novas, Figura 3.21 e Figura 3.22 respectivamente, são exatamente as mesmas, no que diz respeito aos dados retornados. Porém, a quantidade de consultas SQL realizadas e o tempo de resposta da API são extremamente diferentes. Na Figura 3.24 encontram-se dois arquivos, um para cada caso de retorno, sendo comprados pela IDE Visual Studio Code.

Ao realizar novamente cada uma das consultas, é possível observar leves alterações no tempo de execução, mas a quantidade de consultas SQL é sempre a mesma. Por isso, não é necessário para essa prova de conceito que seja realizados testes em massa para a coleta de uma média de tempo de execução em cada um dos casos. Afinal, esse tempo depende de uma série de fatores. Por exemplo, o poder de processamento do computador no qual a API está hospedada e a latência da conexão entre a aplicação da API e o banco de dados.



```
7765 },
7766 {
7767   "kind": "LIKE",
7768   "isFromBully": false,
7769   "profile": {
7770     "name": "Profile 1"
7771   }
7772 },
7773 {
7774   "kind": "LIKE",
7775   "isFromBully": false,
7776   "profile": {
7777     "name": "Profile 0"
7778   }
7779 },
7780 ],
7781 "_debug": {
7782   "sqlCount": 6,
7783   "totalDuration": 1.1292519569396973
7784 },
7785 },
7786 },
7787 },
7788 },
7789 },
7790 }
```

```
7765 },
7766 {
7767   "kind": "LIKE",
7768   "isFromBully": false,
7769   "profile": {
7770     "name": "Profile 1"
7771   }
7772 },
7773 {
7774   "kind": "LIKE",
7775   "isFromBully": false,
7776   "profile": {
7777     "name": "Profile 0"
7778   }
7779 },
7780 ],
7781 "_debug": {
7782   "sqlCount": 2185,
7783   "totalDuration": 21.071683883666992
7784 },
7785 },
7786 },
7787 },
7788 },
7789 },
7790 }
```

Figura 3.24: Comparação dos resultados produzidos pelos dois casos de *schema* testados, na esquerda o que utiliza as funcionalidades introduzidas pelo trabalho de extensão e na direita o que não utiliza

Portanto, não é relevante destacar algo como um percentual de diminuição no tempo de resposta da API para essa consulta. O que é extremamente relevante é destacar a diminuição da quantidade de consultas SQL realizadas, pois isso já é um ponto de melhora de desempenho por si só.

Na consulta GraphQL desta prova de conceito, a quantidade de consultas SQL realizadas ao utilizar as novas funcionalidades do *framework* Graphene-Django foi mais de 364 vezes menor quando comparado ao cenário de não utilização dessas funcionalidades. Vale lembrar que essa proporção não é uma regra, afinal, depende da quantidade de dados presentes no banco. No caso desta prova de conceito, a API retornou 4 publicações, cada uma como 16 comentários, que por sua vez possuem 16 reações cada um. Para se realizar esse mesmo teste, basta baixar esse repositório criado para a prova de conceito. O tutorial para o seu funcionamento encontra-se num arquivo chamado README.md, que encontra-se na raiz do repositório <https://github.com/FernandoTancini/tcc>. Esse tutorial, além de cobrir a parte de instalação e execução da API, cobre também a execução de um comando que cria no banco de dados a mesma quantidade de dados usados na prova de conceito.

3.6

Considerações Finais

Pode-se dizer que a expansão do *framework* teve sucesso em relação a alcançar seus objetivos iniciais. O número de consultas ao banco de dados foi minimizado sem que o processo de desenvolvimento e manutenção do projeto se tornasse mais complexo ou trabalhoso. Nesse sentido, é possível afirmar que a qualidade de software foi aumentada com sucesso, nos quesitos de “manutenibilidade” e “eficiência de desempenho” (Wagner, 2013).

Existem casos em que esse sistema de preparação dos QuerySets não funcionaria tão bem quanto no exemplo usado neste capítulo. Porém, existem inúmeras aplicações para as quais esta extensão iria servir muito positivamente ao desenvolvimento, manutenção e desempenho da API. Essas limitações estão melhor aprofundadas na Seção 4.2.

4

Conclusão

4.1

Contribuições

A principal contribuição deste trabalho é a extensão do *framework* Graphene-Django. Por se tratar de um *framework open source*, o código fonte desta extensão é também aberto e pode ser utilizado pela comunidade desenvolvedora futuramente. Para isso, basta que o projeto seja enviado ao Python Package Index (PIP), o que tornaria possível que os desenvolvedores baixassem e utilizassem essa extensão da mesma forma que fariam com o *framework* original. Além disso, existe ainda a possibilidade de submissão dessa extensão para o próprio repositório do *framework* em questão, através de um *pull request*. Porém, para realizar essa submissão, seria necessário superar algumas limitações que estão descritas na Seção 4.2, neste capítulo. De qualquer forma, as funcionalidades desenvolvidas pelo presente trabalho podem contribuir para o desenvolvimento de API GraphQL na medida em que tornam possível que a aplicação realize um número pequeno de consultas ao banco de dados ao mesmo tempo em que o desenvolvimento e manutenção se mantêm simplificados.

Outra contribuição realizada por esta monografia é prover fundamentos e explicações sobre as tecnologias envolvidas. Lendo este trabalho, é possível aprender sobre conceitos como, por exemplo, ORM, API, GraphQL e meta-classe. Além disso, também é possível aprender sobre o funcionamento dos *frameworks* Django, Graphene e Graphene-Django lendo esta monografia. Por isso, é possível dizer que este trabalho pode servir como um material de aprendizado sobre esses assuntos até para quem não possui necessariamente bagagem técnica ou teórica relacionadas a eles.

4.2

Limitações

Um ponto importante a ser levantado como uma limitação é o fato dessa nova funcionalidade incorporada à biblioteca Graphene-Django não ser compatível com todo e qualquer caso de uso. Existem dois casos de

incompatibilidade identificados. Esses pontos serão descritos nesta seção. Além disso, outra limitação do projeto é a falta de documentação sobre as novas funcionalidades introduzidas ao *framework*.

O primeiro caso de incompatibilidade identificado refere-se ao uso simultâneo das funcionalidades introduzidas pelo trabalho de extensão e de outras funcionalidades já existentes no submódulo *relay* do *framework* Graphene. Relay é um *framework* GraphQL do lado do cliente, que tem o propósito de facilitar e minimizar as chamadas GraphQL que o cliente realiza, principalmente em aplicações *front-end* grandes (Facebook, 2021b). Para que o servidor GraphQL seja compatível com as funcionalidades introduzidas por esse *framework* do lado do cliente, é necessário que a implementação do servidor siga um determinado protocolo. Contudo, o sistema criado neste trabalho de extensão não foi projetado considerando esse caso de uso. Por isso, é possível que o sistema de preparação das QuerySets não consiga minimizar o número de consultas SQL ou até mesmo gere falhas na aplicação no contexto do uso simultâneo dessas duas funcionalidades.

O outro caso de incompatibilidade é menos relevante. Ele refere-se ao retorno de *resolvers* que foram embrulhados pelo sistema de preparação automática. O sistema de preparação só atua no retorno do *resolver* caso ele seja do tipo QuerySet ou uma instância de Model. Porém, caso o retorno do *resolver* seja uma instância de Model que não esteja atrelada a uma entrada real no banco de dados, não é possível se realizar uma consulta a fim de realizar anotações e pré consultas que normalmente iria. Afinal, não haveria nada a ser consultado no banco de dados. Portanto, como o projeto de extensão não considerou esse caso durante sua implementação, torna-se um caso que potencialmente gera falhas na aplicação. Contudo, vale ressaltar que não é um caso recorrente em uma aplicação real, pois o objetivo de se usar um ORM consiste em realizar acessos ao banco de dados. Ou seja, utilizar o ORM para inicializar um objeto Python sem o intuito que ele seja atrelado ao banco de dados é algo normalmente não seria um propósito real.

Por conta das limitações descritas nesta seção, pode-se dizer que a extensão não está pronta para ser incorporada ao repositório original do *framework open source*. É por esse motivo que ainda não foi realizada a submissão desta extensão ao *framework* original por meio de um *pull request*, que seria aprovado ou rejeitado pelos seus mantenedores. Contudo, é claro, isso não impede o uso desta extensão por parte de desenvolvedores da comunidade para que tirem proveito das funcionalidades introduzidas.

4.3

Trabalhos Futuros

Os trabalhos futuros do presente trabalho de extensão são a resolução das incompatibilidades já observadas, assim como a documentação das funcionalidades introduzidas ao *framework*. Após isso, torna-se possível submeter a extensão à aprovação dos mantenedores do repositório oficial do *framework*, para que seja incorporada a ele. Até lá, é possível realizar a publicação desta extensão em uma biblioteca separada para que seja utilizada em projetos que podem se beneficiar com o seu uso.

Referências bibliográficas

- [Berga e Martinho, 2021] MARIANA BERGA; PEDRO MARTINHO. **Flask vs Django: Pirates Use Flask, The Navy Uses Django – Imaginary Cloud.** <https://www.imaginarycloud.com/blog/flask-vs-django/>, 2021. [Online; accessed 20-Novembro-2021].
- [Cunningham, 2018] CUNNINGHAM, H. C.. **Multiparadigm programming with python 3.** 2018.
- [Django Contributors, 2021] DJANGO CONTRIBUTORS. **prefetch_related, Django Documentation.** <https://docs.djangoproject.com/en/3.2/ref/models/queries/#prefetch-related>, 2021. [Online; accessed 20-Novembro-2021].
- [Facebook, 2021a] FACEBOOK. **GraphQL: A data query language.** <https://engineering.fb.com/2015/09/14/core-data/graphql-a-data-query-language/>, 2021. [Online; accessed 20-Novembro-2021].
- [Facebook, 2021b] FACEBOOK. **Relay Website.** <https://relay.dev>, 2021. [Online; accessed 20-Novembro-2021].
- [Forouzan, 2009] FOROUZAN, B. A.. **Comunicação de dados e redes de computadores.** AMGH Editora, 2009.
- [Greif e Benitte, 2021] SACHA GREIF; RAPHAËL BENITTE. **Data Layer, State of JS 2020.** <https://2020.stateofjs.com/en-US/technologies/datalayer/>, 2021. [Online; accessed 20-Novembro-2021].
- [Kuhrmann et al., 2014] KUHRMANN, MARCO; TELL, PAOLO; HEBIG, REGINA; KLUNDER, JIL ANN-CHRISTIN; MUNCH, JURGEN; LINSSSEN, OLIVER; PFAHL, DIETMAR; FELDERER, MICHAEL; PRAUSE, CHRISTIAN; MACDONELL, STEVE ; OTHERS. **What makes agile software development agile.** IEEE Transactions on Software Engineering, 2021.
- [Postman, 2021] POSTMAN. **API Technologies, 2021 State of API.** <https://www.postman.com/state-of-api/api-technologies/#api-technologies>, 2021. [Online; accessed 20-Novembro-2021].

- [Srinath, 2017] SRINATH, K.. **Python – the fastest growing programming language**. International Research Journal of Engineering and Technology (IRJET), 4(12):354–357, 2017.
- [Stack Overflow, 2021] STACK OVERFLOW. **Technology, 2020 Developer Survey**. <https://insights.stackoverflow.com/survey/2020#technology>, 2021. [Online; accessed 20-Novembro-2021].
- [Wagner, 2013] WAGNER, S.. **Software Product Quality Control**. Springer, Stuttgart, Germany, 2013.