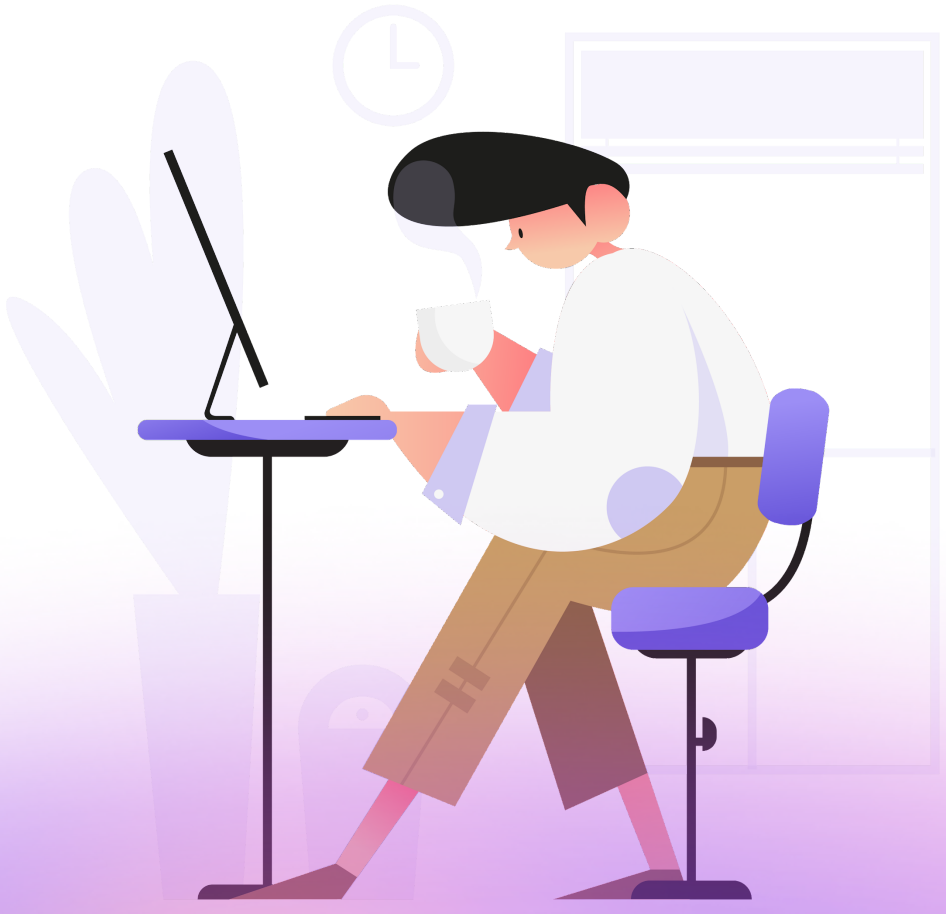
GeekBrains

Django REST Framework

GraphQL



На этом уроке

1. Узнаем, для чего используется GraphQL.
2. Научимся создавать GraphQL-схему
3. Научимся писать различные виды запросов к GraphQL.
4. Научимся создавать схему для связанных моделей данных.

Оглавление

GraphQL и REST

Введение

Мы рассмотрели возможности REST API и способы его построения с помощью DRF. На этом занятии мы узнаем альтернативный вариант построения API — GraphQL. Сначала рассмотрим общие понятия, а затем познакомимся с библиотекой Graphene-Django для использования GraphQL в Django-проектах.

Минусы REST-архитектуры

REST-архитектура обладает множеством плюсов, но у неё есть и минусы. Обычно они проявляются на этапе развития большого проекта. Рассмотрим основные из них.

Множество запросов к разным сущностям

Представим себе главную страницу интернет-магазина, на которой отображаются товары, категории товаров, горящие предложения, скидки, рекомендации пользователю, корзина. При использовании REST у нас будет свой ресурс (URL-адрес) для каждой сущности. Frontend-часть сайта будет делать множество запросов на одной странице, при этом для каждого запроса необходимо будет рассмотреть ситуацию, когда данные не загрузились.

Лишние поля в ответе

При создании REST API мы подготавливаем набор полей модели, которые будут доступны пользователю. Часто при развитии проекта на разных страницах нам нужны разные наборы полей. Например, где-то нужны все поля, где-то три из пяти, где-то всего одно. Получается, что мы должны либо усложнять логику на сервере и использовать разные `serializers` и `views`, либо на клиенте работать с данными, большинство полей которых нам не нужны.

Лишние связанные модели

Кроме простых типов данных бывают сущности, которые связаны с другими сущностями, например, внешним ключом. Таким образом, при получении одной модели за ней тянутся данные о других связанных. В некоторых случаях эти данные не нужны на клиенте либо нужны в другом виде. Например, возвращается словарь данных, а нам нужен только id.

GraphQL

Чтобы компенсировать минусы REST-архитектуры, возник другой подход, который заключается в создании специального гибкого языка запросов к серверу. С помощью него клиент может сам формировать запросы к серверу и получать необходимые данные.

Официальный сайт GraphQL содержит описание языка запросов к серверу. GraphQL не зависит от языка программирования и используемых технологий, поэтому может применяться в различных системах.

Пример GraphQL-запроса выглядит следующим образом:

```
{
 hero {
   name
   friends {
     name
     homeWorld {
       name
       climate
     }
     species {
       name
       lifespan
       origin {
         name
       }
     }
   }
 }
}
```

А описание схемы для этого запроса на языке JavaScript может выглядеть так:

```
type Query {
 hero: Character
}
type Character {
 name: String
 friends: [Character]
 homeWorld: Planet
 species: Species
}
type Planet {
 name: String
 climate: String
}
type Species {
 name: String
```

```
 lifespan: Int
 origin: Planet
}
```

Сервер отдаёт ответ на запрос в формате JSON.

Graphene-Python

Библиотека Graphene-Python реализует работу с GraphQL на языке Python. Она позволяет создать GraphQL-схему.

На официальном сайте библиотеки есть документация и примеры работы с этой библиотекой. Пример создания схемы выглядит следующим образом:

```
from graphene import ObjectType, String, Schema

class Query(ObjectType):
    # this defines a Field `hello` in our Schema with a single Argument `name`
    hello = String(name=String(default_value="stranger"))
    goodbye = String()

    # our Resolver method takes the GraphQL context (root, info) as well as
    # Argument (name) for the Field and returns data for the query Response
    def resolve_hello(root, info, name):
        return f'Hello {name}!'

    def resolve_goodbye(root, info):
        return 'See ya!'

schema = Schema(query=Query)
```

Рассмотрим основные части этого примера. Схема GraphQL состоит из различных типов. Сначала создаётся тип и в нём объявляются поля разных типов данных:

```
class Query(ObjectType):
    # this defines a Field `hello` in our Schema with a single Argument `name`
    hello = String(name=String(default_value="stranger"))
    goodbye = String()
```

Далее для каждого поля, если не задано значение по умолчанию, объявляется функция `resolver`. Эта функция показывает, как получить значение данного поля:

```
def resolve_hello(root, info, name):
        return f'Hello {name}!'
```

Внимание! *Имя функции `resolver`'а обязательно должно начинаться со слова `resolve`, а далее после подчёркивания содержать имя нужного поля. По этому соглашению `graphene` определяет, к какому полю относится та или иная функция.*

После того как создан один или несколько типов данных, они объединяются в одну схему данных:

```
schema = Schema(query=Query)
```

Далее мы пишем запросы на языке GraphQL, который обрабатывает схемы и возвращает результат. Конкретные примеры мы рассмотрим далее.

Graphene-Django

Для применения GraphQL в Django-проектах удобно использовать ещё одну библиотеку — Graphene-Django. Она позволяет на основе модели данных создать тип для GraphQL и таким образом быстро сформировать GraphQL-схему. Далее на примере нашего демонстрационного проекта мы подробно разберём основные возможности GraphQL и их реализацию в Graphene-Django.

Применение GraphQL и Graphene-Django

Создание GraphQL-схемы

Добавим гибкую GraphQL-схему в наш демонстрационный проект library.

Установка

Установим библиотеку Graphene-Django:

```
pip install graphene-django
```

Добавим приложение в `INSTALLED_APPS`:

```
INSTALLED_APPS = [
    ...
    "django.contrib.staticfiles", # Required for GraphiQL
    "graphene_django"
]
```

/library/settings.py

В urls.py проекта добавим адрес для GraphQL-запросов:

```
from django.urls import path
from graphene_django.views import GraphQLView

urlpatterns = [
    # ...
    path("graphql/", GraphQLView.as_view(graphiql=True)),
]
```

/library/urls.py

Параметр `graphiql` определяет, показывать или нет веб-интерфейс для удобного тестирования GraphQL-запросов. Мы указали `True`, так как будем им пользоваться для проверки работоспособности нашей схемы.

Внимание! *В отличие от множества URL-адресов для REST API, все запросы будут обрабатываться всего на одном адресе, который принято называть graphql.*

Далее в settings.py указываем путь до объекта с описанием схемы:

```
GRAPHENE = {
    "SCHEMA": "library.schema.schema"
}
```

/library/settings.py

В нашем случае мы создадим файл schema.py в папке с настройками проекта (папка library).

Создание схемы

Создадим файл schema.py и напишем в нём следующий код:

```
import graphene
```

```
class Query(graphene.ObjectType):
    hello = graphene.String(default_value="Hi!")

schema = graphene.Schema(query=Query)
```

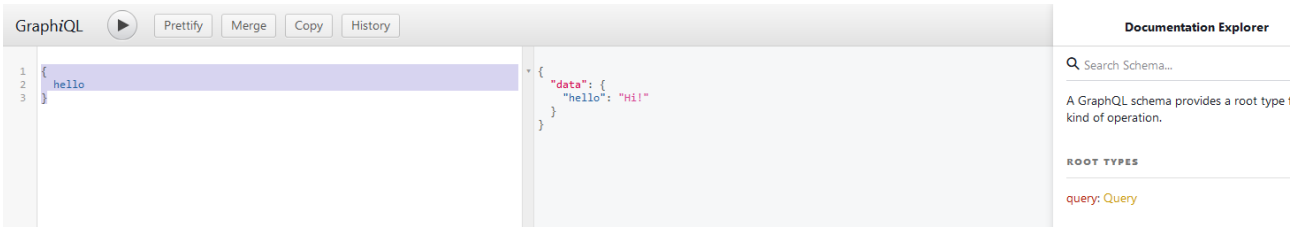
/library/schema.py

Для начала используем пример из официальной документации. Мы создали тип данных `Query` с полем `hello`, у которого `Hi` — значение по умолчанию. А далее создали схему данных `schema`.

Далее запустим сервер и перейдём по адресу http://127.0.0.1:8000/graphql/. Мы увидим удобный веб-интерфейс для формирования запросов. В левой части экрана введём следующий GraphQL-запрос:

```
{
  hello
}
```

После выполнения запроса мы увидим ответ в средней части:



А в правой части можно посмотреть доступные типы данных и их поля.

Создание типов данных на основании модели данных

Создание схемы данных

Теперь рассмотрим практический пример. Мы создадим для запросов к данным авторов и книг схему данных и разберём различные варианты.

Изменим код в schema.py так, чтобы он принял следующий вид:

```
import graphene
from graphene_django import DjangoObjectType
from mainapp.models import Book, Author


class BookType(DjangoObjectType):
```

```
    class Meta:
        model = Book
        fields = '__all__'


class Query(graphene.ObjectType):
    all_books = graphene.List(BookType)

    def resolve_all_books(root, info):
        return Book.objects.all()


schema = graphene.Schema(query=Query)
```

/library/schema.py

Рассмотрим этот код по частям. Сначала с помощью `DjangoObjectType` мы создали тип для описания книги:

```
class BookType(DjangoObjectType):
    class Meta:
        model = Book
        fields = '__all__'
```

Наследование от `DjangoObjectType` позволяет автоматически создать нужные типы полей для указанной модели и указать нужные поля.

Далее мы создали тип `Query`:

```
class Query(graphene.ObjectType):
    all_books = graphene.List(BookType)

    def resolve_all_books(root, info):
        return Book.objects.all()
```

В нём мы указали поле `all_books`, которое представляет собой список типов `BookType` и функцию `resolve-p` для получения всех книг.

Теперь в веб-интерфейсе введём следующий GraphQL-запрос:

```
{
  allBooks {
      id
      name
  }
}
```

`allBooks` — это указание поля, которое мы хотим получить. Обратите внимание, что это название пишется в `camelCase` нотации. `id` и `name` — это вложенные поля каждой книги. Мы можем указывать только те поля, которые хотим получить в ответе.

Пример ответа на этот запрос выглядит следующим образом:

```
{
  "data": {
      "allBooks": [
      {
      "id": "1",
      "name": "Алые паруса"
      },
      {
      "id": "2",
      "name": "Золотая цепь"
      },
      {
      "id": "3",
      "name": "Пиковая дама"
      },
      {
      "id": "4",
      "name": "Руслан и Людмила"
      }
      ]
  }
}
```

Обращение к полям связанной модели

Если мы попробуем обратиться к полю `author` модели `Book` и выполнить следующий запрос:

```
{
  allBooks {
      id
      name
      author
  }
}
```

то в ответе нам вернётся ошибка:

```
{
  "errors": [
```

```
      {
      "message": "Cannot query field \"author\" on type \"BookType\".",
      "locations": [
      {
      "line": 5,
      "column": 5
      }
      ]
      }
  ]
}
```

Это происходит потому, что мы не объявили тип данных для модели `Author`. Сделаем это:

```
...
class AuthorType(DjangoObjectType):
    class Meta:
        model = Author
        fields = '__all__'
...
```

/library/schema.py

Теперь после отправки запроса мы получим следующий ответ:

```
{
  "data": {
      "allBooks": [
      {
      "id": "1",
      "name": "Алые паруса",
      "author": {
      "id": "1"
      }
      },
      {
      "id": "2",
      "name": "Золотая цепь",
      "author": {
      "id": "1"
      }
      },
      {
      "id": "3",
      "name": "Пиковая дама",
      "author": {
      "id": "2"
      }
      },
```

```
      {
      "id": "4",
      "name": "Руслан и Людмила",
      "author": {
      "id": "2"
      }
      }
      ]
  }
}
```

Для автора мы также можем указывать нужные нам поля, например:

```
{
  allBooks {
      id
      name
      author {
      birthdayYear
      name
      }
  }
}
```

Основные виды GraphQL-запросов

Запрос к нескольким спискам

GraphQL позволяет получать несколько наборов данных одним запросом. Для демонстрации внесём в класс `Query` следующие изменения:

```
class Query(graphene.ObjectType):
    ...
    all_authors = graphene.List(AuthorType)

    def resolve_all_authors(root, info):
        return Author.objects.all()
    ...
```

/library/schema.py

Мы добавили поле `all_authors` и функцию `resolve-p` для этого поля.

Теперь мы можем получать данные сразу по книгам и по авторам, например:

```
{
  allBooks {
      name
  }
  allAuthors {
      name
  }
}
```

Ответ будет выглядеть следующим образом:

```
{
  "data": {
      "allBooks": [
      {
      "name": "Алые паруса"
      },
      {
      "name": "Золотая цепь"
      },
      {
      "name": "Пиковая дама"
      },
      {
      "name": "Руслан и Людмила"
      }
      ],
      "allAuthors": [
      {
      "name": "Грин"
      },
      {
      "name": "Пушкин"
      }
      ]
  }
}
```

Alias

Иногда нам может потребоваться получить одни и те же данные два раза. В таком случае мы можем использовать `alias` и указать имя для результата, например:

```
{
  allBooks {
      name
```

```
  }
  booksAgain: allBooks {
      name
  }
}
```

Ответ будет следующим:

```
{
  "data": {
      "allBooks": [
      {
      "name": "Алые паруса"
      },
      {
      "name": "Золотая цепь"
      },
```

Нам необходимо указать другое имя, так как ключи в словаре ответа должны быть уникальными.

related_name и рекурсивные запросы

Модель `Book` связана с `Author` по `ForeignKey`. Это позволяет как получать автора у книги напрямую — `book.author`, так и получать все книги автора по `related_name`. Если `related_name` явно не задан, то он будет называться `book_set`. GraphQL позволяет нам использовать `related_name` в своих запросах. Например:

```
{
  allAuthors {
      name
      bookSet {
      name
      }
  }
}
```

Мы хотим получить имя автора и все его книги по `related_name bookSet`. Ответ будет выглядеть так:

```
{
  "data": {
      "allAuthors": [
      {
      "name": "Грин",
      "bookSet": [
```

```
        {
                "name": "Алые паруса"
        },
        {
                "name": "Золотая цепь"
        }
        ]
        },
        {
        "name": "Пушкин",
        "bookSet": [
        {
                "name": "Пиковая дама"
        },
        {
                "name": "Руслан и Людмила"
        }
        ]
        }
        ]
  }
}
```

Также GraphQL позволяет делать рекурсивные запросы. Рассмотрим следующий пример:

```
{
  allAuthors {
       name
       bookSet {
       name
       author {
       name
       bookSet {
       name
       }
       }
       }
  }
}
```

В этом примере мы получаем всех авторов, затем книги каждого автора, затем автора для каждой книги, затем снова все книги этого автора.

Ответ будет иметь следующий вид:

```
{
  "data": {
       "allAuthors": [
```

```
{
"name": "Грин",
"bookSet": [
{
      "name": "Алые паруса",
      "author": {
      "name": "Грин",
      "bookSet": [
      {
            "name": "Алые паруса"
      },
      {
            "name": "Золотая цепь"
      }
      ]
      }
},
{
      "name": "Золотая цепь",
      "author": {
      "name": "Грин",
      "bookSet": [
      {
            "name": "Алые паруса"
      },
      {
            "name": "Золотая цепь"
      }
      ]
      }
}
]
},
{
"name": "Пушкин",
"bookSet": [
{
      "name": "Пиковая дама",
      "author": {
      "name": "Пушкин",
      "bookSet": [
      {
            "name": "Пиковая дама"
      },
      {
            "name": "Руслан и Людмила"
      }
      ]
      }
},
{
      "name": "Руслан и Людмила",
      "author": {
```

```
            "name": "Пушкин",
            "bookSet": [
            {
                  "name": "Пиковая дама"
            },
            {
                  "name": "Руслан и Людмила"
            }
            ]
            }
      }
      ]
      }
      ]
  }
}
```

Нужно быть аккуратными с такими запросами, так как они могут давать большую нагрузку на базу данных.

Запросы с параметрами

Получение одного объекта модели

GraphQL позволяет передавать в запросы параметры, а библиотека Graphene-Django — эти параметры обрабатывать. Рассмотрим пример получения одного автора. Внесём изменения в класс `Query`:

```
class Query(graphene.ObjectType):
    ...
    author_by_id = graphene.Field(AuthorType, id=graphene.Int(required=True))

    def resolve_author_by_id(self, info, id):
        try:
            return Author.objects.get(id=id)
        except Author.DoesNotExist:
            return None
    ...
```

/library/schema.py

В начале мы указали новое поле `author_by_id`:

```
author_by_id = graphene.Field(AuthorType, id=graphene.Int(required=True))
```

Это поле связано с `AuthorType` и имеет обязательный параметр `id` целого типа.

Далее создаём функцию `resolve-p`:

```
def resolve_author_by_id(self, info, id):
        try:
            return Author.objects.get(id=id)
        except Author.DoesNotExist:
            return None
```

В ней мы получаем автора из базы по id и обрабатываем ошибку, если автор с таким id не существует.

Запрос с параметром выглядит следующим образом:

```
{
  authorById(id: 1) {
      name
      id
      birthdayYear
  }
}
```

Параметры передаются в круглых скобках через двоеточие.

Ответ будет выглядеть так:

```
{
  "data": {
      "authorById": {
      "name": "Грин",
      "id": "1",
      "birthdayYear": 1880
      }
  }
}
```

Фильтрация данных

Параметры в запросе можно использовать и для фильтрации данных. Рассмотрим вариант получения книг по имени автора. Внесём изменения в класс `Query`:

```
class Query(graphene.ObjectType):
    ...
    books_by_author_name = graphene.List(BookType,
name=graphene.String(required=False))
```

```
    def resolve_books_by_author_name(self, info, name=None):
        books = Book.objects.all()
        if name:
            books = books.filter(author__name=name)
        return books
...
```

/library/schema.py

Мы создали новое поле `books_by_author_name` типа `List`, а в функции `resolve-p` написали код для получения книг по имени автора. Если ввести запрос с параметром:

```
{
  booksByAuthorName(name: "Грин") {
      name
  }
}
```

то в ответе мы получим нужные книги:

```
{
  "data": {
      "booksByAuthorName": [
      {
      "name": "Алые паруса"
      },
      {
      "name": "Золотая цепь"
      }
      ]
  }
}
```

Так как параметр может быть необязательным (`required=False`), то при запросе:

```
{
  booksByAuthorName {
      name
  }
}
```

мы получим список всех книг.

Изменение данных. Мутации

Кроме выборки данных GraphQL позволяет изменять данные. В терминах GraphQL это называется мутациями.

Рассмотрим пример изменения данных автора.

Для создания мутации нам нужен специальный объект. Добавим в schema.py следующий код:

```
import graphene
from graphene_django import DjangoObjectType
from mainapp.models import Book, Author

...

class AuthorMutation(graphene.Mutation):
    class Arguments:
        birthday_year = graphene.Int(required=True)
        id = graphene.ID()

    author = graphene.Field(AuthorType)

    @classmethod
    def mutate(cls, root, info, birthday_year, id):
        author = Author.objects.get(pk=id)
        author.birthday_year = birthday_year
        author.save()
        return AuthorMutation(author=author)

class Mutation(graphene.ObjectType):
    update_author = AuthorMutation.Field()

...

schema = graphene.Schema(query=Query, mutation=Mutation)
```

/library/schema.py

Рассмотрим этот код по частям. Сначала мы создали мутацию модели `Author`:

```
class AuthorMutation(graphene.Mutation):
    class Arguments:
        birthday_year = graphene.Int(required=True)
        id = graphene.ID()

    author = graphene.Field(AuthorType)

    @classmethod
    def mutate(cls, root, info, birthday_year, id):
        author = Author.objects.get(pk=id)
```

```
        author.birthday_year = birthday_year
        author.save()
        return AuthorMutation(author=author)
```

В классе `Arguments` мы указываем параметры мутации. Свойство `author` будет содержать итоговый объект после изменения. В методе `mutate` описана логика изменений. Мы берём автора по id и изменяем год его рождения. В самом конце мы возвращаем объект мутации с изменённым автором.

Далее мы создаём общий объект для нескольких мутаций и передаём его в схему по аналогии с объектом `Query`:

```
class Mutation(graphene.ObjectType):
    update_author = AuthorMutation.Field()

...

schema = graphene.Schema(query=Query, mutation=Mutation)
```

Теперь проверим, как работает мутация. Для этого вводим следующий запрос:

```
mutation updateAuthor {
      updateAuthor(birthdayYear: 1, id: 1) {
      author {
      id
      name
      birthdayYear
      }
      }
  }
```

После выполнения получаем ответ:

```
{
  "data": {
      "updateAuthor": {
      "author": {
      "id": "1",
      "name": "Грин",
      "birthdayYear": 1
      }
      }
  }
}
```

Объект был изменён, и нам вернулись его обновлённые данные.

Итоги

На этом занятии мы рассмотрели отличия GraphQL от REST и узнали, как добавить гибкую GraphQL-схему в Django-проект. В зависимости от задачи удобно применять GraphQL, REST или их сочетание.

Глоссарий

GraphQL — язык запросов к серверному API.

Graphene-Python — библиотека для работы с GraphQL на языке Python.

Graphene-Django — библиотека для работы с GraphQL в Django-проектах.

Mutation — описание изменения данных с помощью GraphQL.

Дополнительные материалы

1. Официальный сайт GraphQL.
2. Официальный сайт Graphene-Python.
3. Официальная документация Graphene-Django.
4. Статья о сравнении GraphQL и REST.

Практическое задание

Создание GraphQL-схемы.

В этой самостоятельной работе мы тренируем умения:

- использовать GraphQL;
- настраивать GraphQL-схему.

Смысл: использовать GraphQL для создания API.

Последовательность действий

1) С помощью GraphQL создать схему, которая позволит одновременно получать ToDo, проекты и пользователей, связанных с проектом.

2) * Подумать, какие ещё гибкие запросы могут быть полезны для этой системы, реализовать некоторые из них с помощью GraphQL.