Aplikacje WWW. Wykład #6

1. GraphQL

1.1 Wstęp

GraphQL jest językiem (technologią) zapytań i manipulacji danych, który jest alternatywą dla REST API.

Prace nad tą technologią rozpoczęły się w roku 2012 w firmie Facebook, a jednym z powodów była chęć stworzenia rozwiązania, które będzie charakteryzowało się mniejszą ilością danych niezbędną do przesłania w stosunku do REST API. Innym problemem, który chciano rozwiązać była niska elastyczność w kontekście danych możliwych do pobrania z danego andpointu REST, co prowadziło często do efektu eksplozji wystawionych endpointów lub dużej nadmiarowości danych zwracanych do klienta.

W roku 2015 ukazała się specyfikacja i przykładowa implementacja już jako projekt open source. Od roku 2018 powstała GraphQL Foundation, która znajduje się pod skrzydłami Linux Foundation.

> Projekt jest utrzymywany w formie repozytorium w serwisie Github: https://github.com/graphql

> Dokumentacja: https://graphql.org/

> Specyfikacja: https://spec.graphql.org/

1.2 REST API vs GraphQL

Przywołując główne powody powstania GraphQL, czyli redukcję ilości danych, które należy wymienić między aplikacją kliencką a serwerem, możemy wskazać, że jednymi z wad REST API są **overfetching** oraz **underfetching** danych.

Overfetching występuje wtedy, kiedy serwer zwraca więcej danych niż aktualnie klient potrzebuje, np. potrzeban jest tylko nazwa użytkownika, a endpoint zwraca instancję całej encji reprezentującej obiekt użytkownika systemu.
Underfetching z kolei to sytuacja, w której ilość danych z jednego endpointu jest niewystarczająca dla obsługi danego widoku, więc konieczne jest wykonanie wielu żądań do serwera.
Można oczywiście rozwiązać ten problem poprzez stworzenie dedykowanych endpointów dla każdego widoku po stronie frontendu, ale doprowadza to do efektu **eksplozji endpointów**, który doprowadza do sytuacji, w której utrzymanie takiej aplikacji staje się bardzo kosztowne, skomplikowane i potencjalnie mniej bezpieczne. Takie podejście wymaga również ciągłej współpracy zespołu frontend i backend we wdrażaniu zmian.

W przypadku GraphQL tworzy się zazwyczaj jeden endpoint, który dostarcza możliwość pobierania dowolnych danych, które zostały zdefiniowane w schemacie GraphQL (w powiązaniu z np. encjami samej aplikacji oczywiście). Dodatkowo język zapytań GraphQL pozwala na osadzanie filtrów oraz innych funkcji, podobnych do tych z klasycznego SQL-a, w zapytaniach po stronie klienta.

Jako przykład opisanych powyżej wad REST API oraz przewagi GraphQL w tym względzie niech posłuży poniższy przykład na podstawie implementacji z wykorzystaniem Django oraz Django Rest Framework.

Zadanie: pobranie imienia i nazwiska użytkownika oraz wszystkich powiązanych z nim tytułów postów wraz z nazwami topików.

Definicja modeli zostanie tutaj pominięta, a dotyczy modeli zdefiniowanych w aplikacji rozwijanej w ramach laboratorium.

Mamy więc modele: wbudowany model `django.contrib.auth.models.User`, `Category`, `Topic`, `Post`.

Zakładając, że nie posiadamy dedykowanego endpointu do obsługi takiego żądania, lista niezbędnych do wykorzystania endpointów może wyglądać tak:

```
# plik api_url.py
from django.urls import include, path
from . import api_views


urlpatterns = [
    # endpoint do odpytania o podstawowe dane użytkownika (imię, nazwisko, ale
pobiera też inne dane)
    path('users/<int:pk>/', api_views.get_user_data, name='get_user_data'),
    # endpoint pobierający wszystkie posty danego użytkownika
    path('users/<int:pk>/posts/', api_views.get_user_posts,
name='get_user_posts'),

    # pozostałe endpointy
    path('posts/', api_views.post_list, name='post_list'),
    path('posts/<int:pk>/', api_views.post_detail, name='post_detail'),
    path('posts/by_keyword/<keyword>/', api_views.post_list_by_keyword,
name='post_list_by_keyword'),
]
```

Teraz definicja widoków wymaganych do realizacji zadania.

```
# plik api_views.py

@api_view(['GET'])
def get_user_data(request, pk, format=None):
    """
    Pobiera dane użytkownika na podstawie jego ID.
    """
    try:
        user = User.objects.get(pk=pk)
    except User.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        serializer = UserSerializer(user)
        return Response(serializer.data)


@api_view(['GET'])
def get_user_posts(request, pk, format=None):
    """
```

```
    Pobiera posty stworzone przez użytkownika o podanym ID.
    """
    try:
        user = User.objects.get(pk=pk)
    except User.DoesNotExist:
        return Response(status=status.HTTP_404_NOT_FOUND)

    if request.method == 'GET':
        posts = Post.objects.filter(created_by=user)
        serializer = PostModelSerializer(posts, many=True)
        return Response(serializer.data)
```

I delikatna zmiana definicji serializatora dla modelu `Post`.

```
class PostModelSerializer(serializers.ModelSerializer):
    # pole topic nie będzie przechowywało tylko id topiku, ale cały zserializowany
obiekt
    # również przy tworzeniu/aktualizacji obiektu Post należy przekazywać JSON w
    # odpowiednim formacie
    topic = TopicSerializer()

    class Meta:
        model = Post
        fields = ['id', 'text', 'title', 'topic']
        read_only_fields = ['id', 'created_at', 'updated_at']
```

I teraz przykładowe żądania.

Aby pobrać imię i nazwisko dla użytkownika o id 1 wysyłamy żądanie na endpoint `http://127.0.0.1:8000/blog/users/1/` i otrzymujemy:

```
{
    "id": 1,
    "username": "jan.kowalski",
    "last_name": "Kowalski",
    "first_name": "Jan"
}
```

Aby dostarczyć wszystkie niezbędne dane z endpointów, które mamy, odpytujemy adres `http://127.0.0.1:8000/blog/users/1/posts/` dostajemy przykładowo poniższe dane:

```
[
    {
        "id": 2,
        "text": "Jakiś tekst posta.",
        "title": "test",
        "topic": {
```

```
            "id": 1,
            "name": "Test",
            "category": 1
        }
    },
    {
        "id": 1,
        "text": "nowy tekst",
        "title": "test",
        "topic": {
            "id": 1,
            "name": "Test",
            "category": 1
        }
    }
]
```

Biorąc pod uwagę, że potrzebowaliśmy tylko `User.first_name`, `User.last_name` oraz `Post.title` i `Post.topic.name` to nadmiarowość danych jest duża.

I teraz przykład pobrania tych samych danych z wykorzystaniem GraphQL.

Ograniczę się tutaj tylko do niewielkiej części konfiguracji GraphQL po stronie Django (moduł `graphene`), wskazując podobne punkty dostępu do danych jak w poprzednim przykładzie. Pozostałe elementy specyfikacji GraphQL zostaną przedstawione w dalszej części wykładu.

```
import graphene
from graphene_django import DjangoObjectType
from posts.models import Topic, Post
from django.contrib.auth.models import User


class UserType(DjangoObjectType):
    class Meta:
        model = User
        fields = ['id', 'username', 'first_name', 'last_name']


class TopicType(DjangoObjectType):
    class Meta:
        model = Topic
        fields = ['id', 'name', 'category', 'created', 'post_set']


class PostType(DjangoObjectType):
    class Meta:
        model = Post
        fields = ['id', 'title', 'text', 'topic', 'slug', 'created_at',
'updated_at', 'created_by']
```

```
class Query(graphene.ObjectType):

    # Posty danego topiku
    posts_by_user = graphene.List(PostType,
created_by=graphene.Int(required=True))


    def resolve_posts_by_user(self, info, id):
        return Post.objects.filter(created_by=id)


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

Następnie zapytanie do schemy o potrzebne nam dane.

```
{
  postsByUser (createdBy: 1) {
    id
    title
    topic {
      name
    }
    user: createdBy {
      firstName
      lastName
    }

  }
}
```

I dane, które zostały zwrócone. W powyższym zapytaniu zostały użyte aliasy, aby pole `createdBy` z modelu `Post` było zwracane jako `user`.

```
{
  "data": {
    "postsByUser": [
      {
        "id": "2",
        "title": "test",
        "topic": {
          "name": "Test"
        },
        "user": {
          "firstName": "",
          "lastName": ""
        }
      },
      {
        "id": "1",
```

```
          "title": "test",
          "topic": {
            "name": "Test"
          },
          "user": {
            "firstName": "",
            "lastName": ""
          }
        }
      ]
    }
}
```

W przypadku prezentowanego przykładu zapytanie zostało wykonane poprzez dostarczony frontend ze strony modułu graphene dla django.

Zapytanie można wysłać poprzez np. narzędzie curl jak poniżej.

```
curl -H "Content-Type: application/json" -X POST -d "{\"query\": \"query
{postsByUser(createdBy:1){id}}\"}" http://127.0.0.1:8000/graphql/

# output
{"data":{"postsByUser":[{"id":"2"},{"id":"1"}]}}
```

1.3 Schemat danych w GraphQL

GraphQL składa się z kilku kluczowych komponentów, które współpracują ze sobą, aby umożliwić efektywne zapytania i manipulacje danymi. Oto główne składniki:

- **Schema:**

  Definiuje strukturę danych, które są dostępne w API. Określa typy obiektów, ich pola oraz relacje między nimi.

- **Types:**

  Typy definiują strukturę danych. Mogą to być typy obiektowe (np. `Post`, `User`), typy skalara (np. `String`, `Int`) oraz typy wyliczeniowe (`enum`).

- **Queries:**

  Zapytania (queries) służą do pobierania danych. Użytkownicy mogą określić, jakie pola chcą otrzymać w odpowiedzi.

- **Mutations:**

  Mutacje (mutations) są używane do modyfikacji danych, takich jak tworzenie, aktualizowanie lub usuwanie obiektów.

- **Resolvers:**

Funkcje, które odpowiadają na zapytania i mutacje. Odpowiadają za pobieranie danych z bazy danych lub innego źródła.

- **Subscriptions:**

  Umożliwiają klientom subskrybowanie zmian w danych, co pozwala na otrzymywanie aktualizacji w czasie rzeczywistym.

- **Introspection:**

  Mechanizm, który pozwala na zapytania o strukturę API, co umożliwia klientom odkrywanie dostępnych typów i operacji.

Te komponenty współpracują, aby umożliwić elastyczne i wydajne interakcje z danymi w aplikacjach opartych na GraphQL.

GraphQL schema

> Dokumentacja: https://graphql.org/learn/schema/

Schema Definition Language (SDL) w GraphQL to specjalny język służący do definiowania struktury danych i typów dostępnych w API GraphQL.

Główne cechy SDL:

- **Definiowanie typów** - opisuje strukturę danych, które mogą być pobierane lub modyfikowane
- **Definiowanie pól** - każdy typ zawiera pola z określonymi typami danych
- **Definiowanie operacji** - `Query` (pobieranie danych), `Mutation` (modyfikacja danych), `Subscription` (subskrypcje) Przykład prostego schematu GraphQL SDL:

```
# zwykły komentarz

"""
Typ User, który reprezentuje użytkownika w systemie.
"""
type User {
  id: ID!
  firstName: String!
  lastName: String!
  email: String!
  age: Int
}

"""
Topic reprezentuje tematykę postów.
"""
type Topic {
  id: ID!
  name: String!
}

"""
```

```
  Post reprezentuje wpis na blogu lub forum.
  """
  type Post {
    id: ID!
    title: String!
    "Póki co zakładamy, że content to zwykły tekst."
    content: String!
    author: User!
    topic: Topic!
  }
```

Wyjaśnienie składni:

- `!` - pole wymagane (non-nullable)
- `[ ]` - tablica/lista
- `String, Int, ID, Boolean` - wbudowane typy skalarne

Możliwe jest również osadzanie dokumentacji w postaci komentarzy opisujących typy oraz pola.

> Wszystkie dostępne typy w GraphQL są opisane tutaj: https://spec.graphql.org/draft/#sec-Types

Typy skalarne

W GraphQL mamy zdefiniowane następujące **typy skalarne**:

- **Int:** liczba całkowita ze znakiem (32 bity).
- **Float:** liczba zmiennoprzecinkowa ze znakiem podwójnej precyzji.
- **String:** ciąg znaków w kodowaniu UTF-8.
- **Boolean:** wartość logiczna — true lub false.
- **ID:** unikalny identyfikator, często używany do ponownego pobrania obiektu lub jako klucz w pamięci podręcznej. Typ ID jest serializowany tak samo jak String, jednak oznaczenie pola jako ID sygnalizuje, że nie jest przeznaczone do czytania przez człowieka.

Typy wyliczeniowe (enum)

Typy **wyliczeniowe (enum)** w GraphQL pozwalają na definiowanie zestawu stałych wartości, które pole może przyjmować.

Przykład:

```
enum Role {
  ADMIN
  USER
  GUEST
}
```

W przypadku pozostałych typów odsyłam do dokumentacji.

Argumenty w GraphQL

Możliwe jest również definiowanie argumentów przy definicji typów danych (później rówiwnież przy zapytaniach).

Przykład za dokumentacją:

```
type Starship {
  id: ID!
  name: String!
  length(unit: LengthUnit = METER): Float
}
```

Powyżej mamy zdefiniowane pole `length`, które przyjmuje argument `unit` typu `LengthUnit` z wartością domyślną `METER`. Wszystkie argumenty są określone przez nazwę i typ.

Dyrektywy

Dyrektywy w GraphQL to specjalne instrukcje, które można dołączyć do pól lub fragmentów zapytań, aby zmienić sposób ich wykonywania. Dyrektywy pozwalają na dynamiczne dostosowywanie zapytań w zależności od określonych warunków.

> Specyfikacja dyrektyw: https://spec.graphql.org/draft/#sec-Type-System.Directives

Poniżej przykład wykorzystania dyrektywy `@deprecated`, która służy do oznaczania pól lub typów jako przestarzałych.:

```
type User {
  fullName: String
  name: String @deprecated(reason: "Use `fullName`.")
}
```

Inne dwie wbudowane dyrektywy to `@include` oraz `@skip`, które pozwalają na warunkowe uwzględnianie lub pomijanie pól w zapytaniach.

- **@include(if: Boolean)** - dołącz to pole do wyniku tylko wtedy, gdy argument jest prawdziwy.
- **@skip(if: Boolean)** - pomiń to pole w wyniku, jeśli argument jest prawdziwy.

GraphQL Queries - definicja i zapytania

W GraphQL dostępne są trzy główne typy operacji: **zapytania (queries)**, **mutacje (mutations)** oraz **subskrypcje (subscriptions)**.

Zapytania służą tylko do pobierania danych, bez ich modyfikacji.

Aby wykonać zapytanie o dane, należy najpierw zdefiniować operację `Query` w schemacie GraphQL, a następnie wysłać zapytanie do serwera GraphQL.

Przykład definicji operacji `Query` w schemacie GraphQL:

```
type Query {
  "Zapytanie z wymaganym argumentem"
  user(id: ID!): User
  users: [User!]!
  post(id: ID!): Post
  posts: [Post!]!
}
```

W najprostrzej formie zapytanie to po prostu wskazanie pól, które chcemy pobrać z określonego typu.

Zapytanie po stronie klienta może wyglądać tak:

```
{
  user(id: "1") {
    firstName
    lastName
    email
  }
}
```

Taka forma zapytania jest nazywana **operacją anonimową**, ponieważ nie posiada nazwy. Możliwe jest również nadanie nazwy zapytaniu, co jest przydatne w przypadku bardziej złożonych zapytań lub gdy chcemy mieć lepszą identyfikację zapytań w logach. Przykład z nazwą operacji:

```
type Query {
  GetUserById(id: ID!): User
}

# i zapytanie po stronie klienta
query GetUserById {
  user(id: "1") {
    firstName
    lastName
    email
  }
}
```

Podanie nazwy operacji jest wymagane dla operacji mutacji oraz subskrypcji.

Wszystkie te przykłady wymagają podania wartości argumentów w samym zapytaniu, co wymaga manipulacji tekstem zapytania po stronie klienta. Alternatywnie można zdefiniować zmienne zapytania, które pozwalają na przekazywanie wartości argumentów w sposób bardziej elastyczny. Przykład z wykorzystaniem zmiennych zapytania:

```
```

```
# zapytanie z wykorzystaniem zmiennych
query GetUserById($userId: ID!) {
  user(id: $userId) {
    firstName
    lastName
    email
  }
}

# i wartości zmiennych przekazywane oddzielnie
variables: {
  "userId": "1"
}
```

Zapytania GraphQL oferują jeszcze możliwość stosowania **fragmentów**, które pozwalają na ponowne wykorzystanie zestawów pól w różnych zapytaniach oraz metadanych, takich jak dyrektywy czy zapytanie o typ `__typename`. Odsyłam do dokumentacji w celu zapoznania się z tymi zagadnieniami.

GraphQL Mutations - definicja i zapytania

Mutacje to operacje w GraphQL służące do modyfikacji danych na serwerze, takie jak tworzenie, aktualizowanie lub usuwanie zasobów. Podobnie jak zapytania (queries), mutacje są definiowane w schemacie GraphQL i wykonywane przez klienta.

Przykład definicji operacji `Mutation` dla obiektu `User` w schemacie GraphQL:

```
# Definiowanie operacji Mutation (modyfikacja danych)
type Mutation {
  createUser(firstName: String!, lastName: String!, email: String!): User!
  updateUser(id: ID!, firstName: String, lastName: String): User
  deleteUser(id: ID!): Boolean!
}
```

Wywołanie powyższych mutacji po stronie klienta może wyglądać tak:

```
mutation CreateUser {
  createUser(firstName: "Jan", lastName: "Kowalski", email:
"jan.kowalski@example.com") {
    id
    firstName
    lastName
    email
  }
}

mutation UpdateUser {
  updateUser(id: "1", firstName: "Janusz") {
    id
```

```
    firstName
    lastName
    email
  }
}

mutation DeleteUser {
  deleteUser(id: "1")
}
```

Podobnie jak w przypadku zapytań, mutacje mogą również korzystać ze zmiennych, co pozwala na bardziej elastyczne przekazywanie danych. Przykład z wykorzystaniem zmiennych w mutacji:

```
mutation CreateUser($firstName: String!, $lastName: String!, $email: String!) {
  createUser(firstName: $firstName, lastName: $lastName, email: $email) {
    id
    firstName
    lastName
    email
  }
}
```

Jedną z różnic między zapytaniami a mutacjami jest to, że mutacje są wykonywane sekwencyjnie, co oznacza, że każda mutacja musi zakończyć się przed rozpoczęciem następnej. Zapewnia to spójność danych podczas modyfikacji.

GraphQL Subscriptions

Subskrypcje w GraphQL to mechanizm umożliwiający klientom otrzymywanie aktualizacji danych w czasie rzeczywistym. Dzięki subskrypcjom, klient może "subskrybować" określone zdarzenia lub zmiany danych na serwerze i otrzymywać powiadomienia, gdy te zmiany wystąpią.

Przykład definicji operacji `Subscription` w schemacie GraphQL:

```
type Subscription {
  userCreated: User!
  postAdded: Post!
}
```

Wywołanie subskrypcji po stronie klienta może wyglądać tak:

```
subscription OnUserCreated {
  userCreated {
    id
    firstName
    lastName
```

```
    email
  }
}
```

Faktycznie subskrypcje działają na zasadzie utrzymywania otwartego połączenia między klientem a serwerem, często za pomocą WebSocketów. Gdy na serwerze wystąpi zdarzenie, które klient subskrybuje (np. utworzenie nowego użytkownika), serwer wysyła odpowiednie dane do klienta/-ów.

Ten mechanizm jest szczególnie przydatny w aplikacjach wymagających aktualizacji w czasie rzeczywistym, takich jak czaty, powiadomienia czy aplikacje monitorujące.

Obsługa subskrypcji jest bardziej złożona niż zapytań i mutacji, ponieważ wymaga utrzymania stanu połączenia oraz zarządzania sesjami subskrypcji.

GraphQL Resolvers

Resolvers to funkcje w GraphQL, które odpowiadają na zapytania, mutacje i subskrypcje, pobierając lub modyfikując dane zgodnie z definicją schematu. Każde pole w schemacie GraphQL może mieć przypisany resolver, który określa, jak uzyskać wartość tego pola.

Resolver działa jako most między zapytaniem GraphQL a źródłem danych, takim jak baza danych, API lub inny system. Kiedy klient wysyła zapytanie, GraphQL wywołuje odpowiednie resolvery dla każdego pola w zapytaniu, aby zebrać wymagane dane. Jeśli pole nie ma przypisanego resolvera, GraphQL używa domyślnego zachowania, które zazwyczaj polega na bezpośrednim zwróceniu wartości z obiektu źródłowego. W przypadku typów skalarnych, takich jak `String` czy `Int`, domyślny resolver zwraca wartość bezpośrednio. W przypadku typów obiektowych, domyślny resolver zwraca obiekt, z którego można dalej pobierać pola, następnie wywołując kolejne resolvery dla tych pól aż do momentu osiągnięcia wartości skalarnych lub enumów.

Implementacja resolverów zależy od używanej biblioteki lub frameworka GraphQL.

W kolejnych częściach wykładu zostaną przedstawione przykłady implementacji resolverów w kontekście aplikacji Django z wykorzystaniem biblioteki `graphene-django`.

Introspekcja w GraphQL

Introspekcja w GraphQL to mechanizm, który pozwala klientom na zapytania o strukturę i możliwości API GraphQL. Dzięki introspekcji, klienci mogą odkrywać dostępne typy, pola, operacje (zapytania, mutacje, subskrypcje) oraz inne informacje o schemacie GraphQL bez konieczności posiadania zewnętrznej dokumentacji. Introspekcja jest szczególnie przydatna podczas tworzenia narzędzi do eksploracji API, takich jak GraphiQL czy Apollo Studio, które umożliwiają programistom interaktywne badanie i testowanie zapytań GraphQL.

W środowiskach produkcyjnych introspekcja może być wyłączona ze względów bezpieczeństwa, aby zapobiec ujawnianiu szczegółów implementacji API osobom nieuprawnionym.

Podsumowanie

Główne różnice między GraphQL a REST API to:

- Elastyczność zapytań: GraphQL pozwala klientom precyzyjnie określić, jakie dane chcą otrzymać, podczas gdy REST API ma sztywno zdefiniowane endpointy.
- Redukcja nadmiarowości danych: GraphQL minimalizuje overfetching i underfetching danych, co może prowadzić do bardziej efektywnej komunikacji między klientem a serwerem.
- Jeden endpoint: GraphQL zazwyczaj korzysta z jednego endpointu do obsługi wszystkich zapytań i mutacji, podczas gdy REST API wymaga wielu endpointów dla różnych zasobów.
- Typy i schemat: GraphQL posiada silnie typowany schemat, który definiuje strukturę danych i operacji, co ułatwia rozwój i utrzymanie API.

1.4 Wdrażanie GraphQL w aplikacji Django

Częściowa implementacja GraphQL w aplikacji Django została już przedsatwiona w części 1.2 tego wykładu. Bardziej szczegółowa implementacja bez pomijania szczegółów zostanie przedstawiona poniżej.

Instalacja wymaganych pakietów

Aby dodać obsługę GraphQL do aplikacji Django, należy zainstalować odpowiednie pakiety. Najpopularniejszą biblioteką do integracji GraphQL z Django jest `graphene-django`.

```
pip install graphene-django
```

Konfiguracja Django do obsługi GraphQL

W pliku `settings.py` należy dodać `graphene_django` do listy zainstalowanych aplikacji:

```
INSTALLED_APPS = [
    ...
    'graphene_django',
    ...
]
```

Następnie należy skonfigurować schemat GraphQL w pliku `settings.py` i przyjmujemy, że nasza aplikacja nazywa sie `blog`. W tym przypadku schemat będzie zdefiniowany w pliku `blog/schema.py`. Trzeba tutaj zaznaczyć, że to rozwiązanie nie jest zbyt elastyczne, gdyż wymaga albo reedycji tego schematu dla każdej nowej aplikacji, albo stworzenia jednego pliku `schema.py`, który będzie importował schematy z poszczególnych aplikacji.

```
GRAPHENE = {
    'SCHEMA': 'blog.schema.schema'  # Ścieżka do schematu GraphQL
}
```

Definiowanie schematu GraphQL

W pliku `blog/schema.py` definiujemy typy, zapytania oraz mutacje. Poniżej znajduje się przykładowa implementacja:

```
```

Definiowanie widoku GraphQL

W pliku `urls.py` głównego katalogu projektu Django należy dodać ścieżkę do widoku GraphQL:

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

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

# a jeżeli chcemy dodać obsługę CSRF
from django.views.decorators.csrf import csrf_exempt

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

Atrybut `graphiql=True` włącza interfejs graficzny GraphiQL, który umożliwia testowanie zapytań GraphQL bezpośrednio z okna przeglądarki internetowej.

Definicja schematu GraphQL dla aplikacji blog

W pliku `blog/schema.py` definiujemy typy, zapytania oraz mutacje. Poniżej znajduje się przykładowa implementacja:

```
import graphene
from graphene_django import DjangoObjectType
from posts.models import Category, Topic, Post
from django.contrib.auth.models import User


class CategoryType(DjangoObjectType):
    class Meta:
        model = Category
        fields = ['id', 'name']


class UserType(DjangoObjectType):
    class Meta:
```

```
        model = User
        fields = ['id', 'username', 'first_name', 'last_name']


class TopicType(DjangoObjectType):
    class Meta:
        model = Topic
        fields = ['id', 'name', 'category', 'created', 'post_set']


class PostType(DjangoObjectType):
    class Meta:
        model = Post
        fields = ['id', 'title', 'text', 'topic', 'slug', 'created_at',
'updated_at', 'created_by']


class Query(graphene.ObjectType):

    # Wszystkie tematy
    all_topics = graphene.List(TopicType)

    # Wszystkie posty
    all_posts = graphene.List(PostType)

    # Posty danego użytkownika
    posts_by_user = graphene.List(PostType,
created_by_id=graphene.Int(required=True))

    def resolve_all_topics(self, info):
        return Topic.objects.all()

    def resolve_all_posts(self, info):
        return Post.objects.all()

    def resolve_posts_by_user(self, info, created_by_id):
        return Post.objects.filter(created_by=created_by_id)


class CreatePost(graphene.Mutation):
    post = graphene.Field(PostType)
    success = graphene.Boolean()
    message = graphene.String()

    class Arguments:
        title = graphene.String(required=True)
        text = graphene.String(required=True)
        topic_id = graphene.Int(required=True)
        slug = graphene.String(required=True)
        created_by_id = graphene.Int(required=True)

    def mutate(self, info, title, text, topic_id, slug, created_by_id):
        try:
            post = Post.objects.create(
```

```
                title=title,
                text=text,
                topic_id=topic_id,
                slug=slug,
                created_by_id=created_by_id
            )
            return CreatePost(post=post, success=True, message="Post created
successfully")
        except Exception as e:
            return CreatePost(post=None, success=False, message=str(e))


class UpdatePost(graphene.Mutation):
    post = graphene.Field(PostType)
    success = graphene.Boolean()
    message = graphene.String()

    class Arguments:
        id = graphene.Int(required=True)
        title = graphene.String()
        text = graphene.String()
        topic_id = graphene.Int()
        slug = graphene.String()

    def mutate(self, info, id, title=None, text=None, topic_id=None, slug=None):
        try:
            post = Post.objects.get(id=id)
            if title:
                post.title = title
            if text:
                post.text = text
            if topic_id:
                post.topic_id = topic_id
            if slug:
                post.slug = slug
            post.save()
            return UpdatePost(post=post, success=True, message="Post updated
successfully")
        except Post.DoesNotExist:
            return UpdatePost(post=None, success=False, message="Post not found")
        except Exception as e:
            return UpdatePost(post=None, success=False, message=str(e))


class DeletePost(graphene.Mutation):
    success = graphene.Boolean()
    message = graphene.String()

    class Arguments:
        id = graphene.Int(required=True)

    def mutate(self, info, id):
        try:
            post = Post.objects.get(id=id)
```

```
            post.delete()
            return DeletePost(success=True, message="Post deleted successfully")
        except Post.DoesNotExist:
            return DeletePost(success=False, message="Post not found")
        except Exception as e:
            return DeletePost(success=False, message=str(e))


class Mutation(graphene.ObjectType):
    create_post = CreatePost.Field()
    update_post = UpdatePost.Field()
    delete_post = DeletePost.Field()


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

Wywołanie przykładowych zapytań i mutacji

Przykład zapytania tworzącego nowy `Post` z poziomu GraphiQL:

```
mutation {
  createPost(
    title: "Nowy post z GraphQL"
    text: "To jest zawartość mojego posta"
    topicId: 1
    slug: "moj-pierwszy-post"
    createdById: 1
  ) {
    post {
      id
      title
      text
      topic {
        id
        name
      }
      createdBy {
        id
        username
      }
      createdAt
    }
    success
    message
  }
}
```

I zwrócona wartość:

```
{
  "data": {
    "createPost": {
      "post": {
        "id": "3",
        "title": "Nowy post z GraphQL",
        "text": "To jest zawartość mojego posta",
        "topic": {
          "id": "1",
          "name": "Test"
        },
        "createdBy": {
          "id": "1",
          "username": "kropiak"
        },
        "createdAt": "2025-11-13T18:20:39.529703+00:00"
      },
      "success": true,
      "message": "Post created successfully"
    }
  }
}
```


Innym sposobem na deklarację mutacji jest wykorzystanie istniejących serializatorów Django Rest Framework jako bazy dla mutacji GraphQL. Poniżej przykład takiego podejścia.

```
from graphene_django.rest_framework.mutation import SerializerMutation
from posts.serializers import CategorySerializer


class CategoryMutation(SerializerMutation):
    class Meta:
        serializer_class = CategorySerializer
        model_operations = ['create', 'update']
        lookup_field = 'id'

# i jeszcze rejestracja mutacji w klasie Mutation
class Mutation(graphene.ObjectType):
    create_post = CreatePost.Field()
    update_post = UpdatePost.Field()
    delete_post = DeletePost.Field()
    update_category = CategoryMutation.Field()
```

Teraz wywołanie mutacji aktualizującej obiekt `Category` może wyglądać następująco.

```
mutation UpdateCategory {
        updateCategory (input: {
      id: 1
```

```
      name: "Nowa nazwa"
    }
    ) {
    id
    name
    errors {
            field
            messages
        }
  }
}
```

I zwrócona wartość:

```
{
  "data": {
    "updateCategory": {
      "id": 1,
      "name": "Nowa nazwa",
      "errors": null
    }
  }
}
```

Mimo dość licznych przykładów, temat GraphQL jest bardzo obszerny i wymaga dalszego zgłębiania, zwłaszcza w kontekście bardziej zaawansowanych funkcji, takich jak subskrypcje, optymalizacja zapytań (np. DataLoader), zabezpieczenia (autoryzacja i uwierzytelnianie) oraz integracja z różnymi źródłami danych.

W tym celu odsyłam czytelnika do oficjalnej dokumentacji GraphQL oraz bibliotek używanych w ekosystemie Django (tu `graphene-django`: https://docs.graphene-python.org/).