Building API's in GraphQL

Liam Norman | DeveloperUG

HELLO! ✋

I am Liam Norman

- Software Engineer at Superbalist.com
- Organiser of CT PHP
- Blog at liamnorman.com
- **@liamjnorman**

What is GraphQL?

> GraphQL is a query language for your API, and a server-side runtime for executing queries by using a type system you define for your data

WHAT IS GRAPHQL?

- An API query language
- Developed by Facebook in 2012
- Not only to be used with React!
- Open sourced by facebook in 2015 - https://graphql.github.io/graphql-spec/

Our first Query!

- Using GraphiQL client

```
brew cask install graphiql

...

🍺  graphiql was successfully installed!
```

Example GraphQL Query

GraphQL Endpoint https://api.github.com/graphql

Method POST

Edit HTTP Headers

GraphiQL

Prettify

History

Docs

```
query {
  viewer{
    login,
    location
  }
}
```

```
{
  "data": {
    "viewer": {
      "login": "LiamNorman",
      "location": "Cape Town, South Africa"
    }
  }
}
```

Query Arguments

GraphQL Endpoint https://api.github.com/graphql

Method POST

Edit HTTP Headers

GraphiQL

Prettify

History

```
1 query userLogin {
2   user (login: "LiamNorman") {
3     name,
4     location,
5     isEmployee,
6   }
7
8 }
```

```
{
  "data": {
    "user": {
      "name": "Liam Norman",
      "location": "Cape Town, South Africa",
      "isEmployee": false
    }
  }
}
```

viewer

User

GitHub Campus Experts Program.

isDeveloperProgramMember: Boolean!

Whether or not this user is a GitHub Developer Program member.

isEmployee: Boolean!

Whether or not this user is a GitHub employee.

isHireable: Boolean!

Whether or not the user has marked themselves as for hire.

isSiteAdmin: Boolean!

Whether or not this user is a site administrator.

isViewer: Boolean!

Whether or not this user is the viewing user.

DESIGN PRINCIPLES

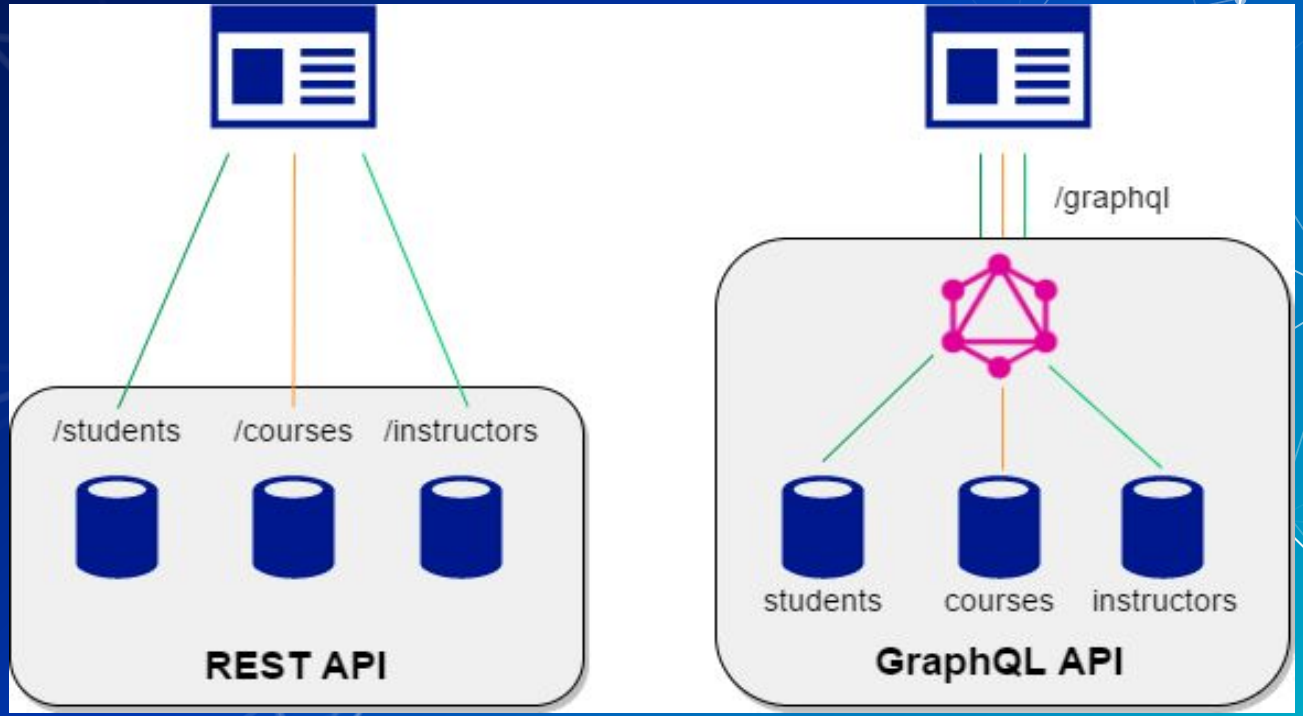
- Hierarchical
- Product Centric
- Strong Typing with Schemas
- Client-specified Queries
- Introspective

Why GraphQL?

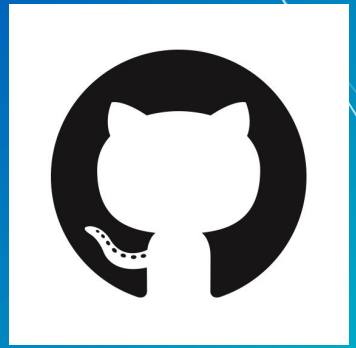
PROBLEMS GRAPHQL SOLVES

- Declarative Data Fetching
- No Overfetching
- Solves Underfetching
- Language and framework independent

HOW GRAPHQL WORKS



GraphQL Adoption



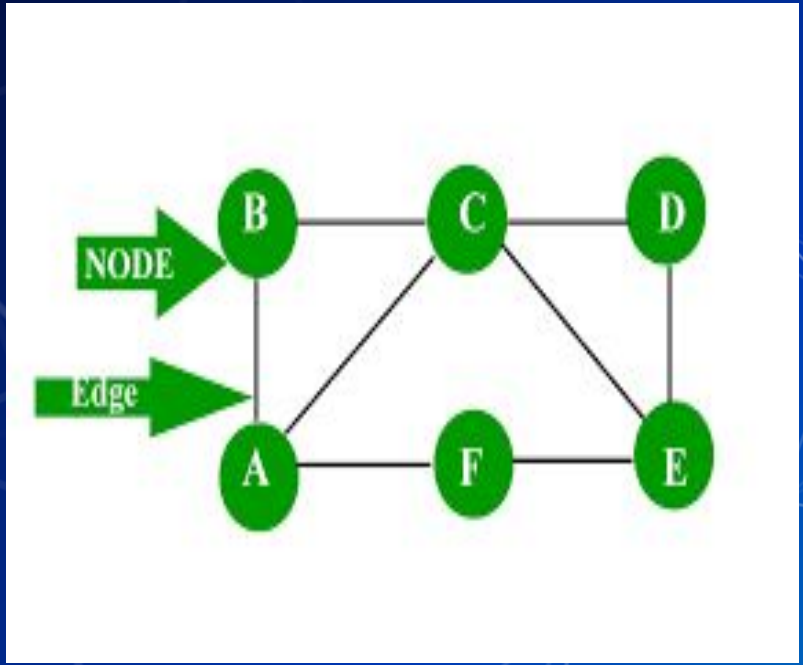
Thinking in Graphs

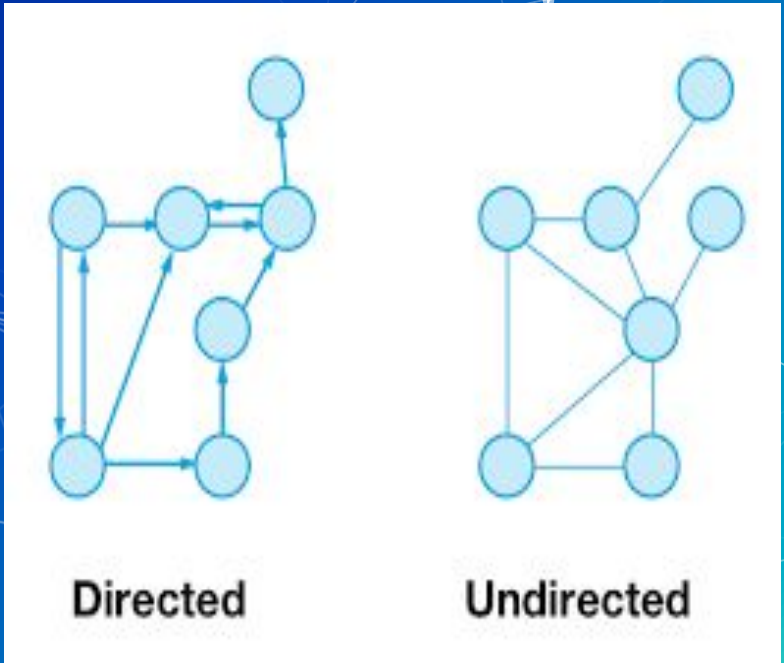
> With GraphQL, you model your business domain as a graph.

Graph Theory 101 (in short)

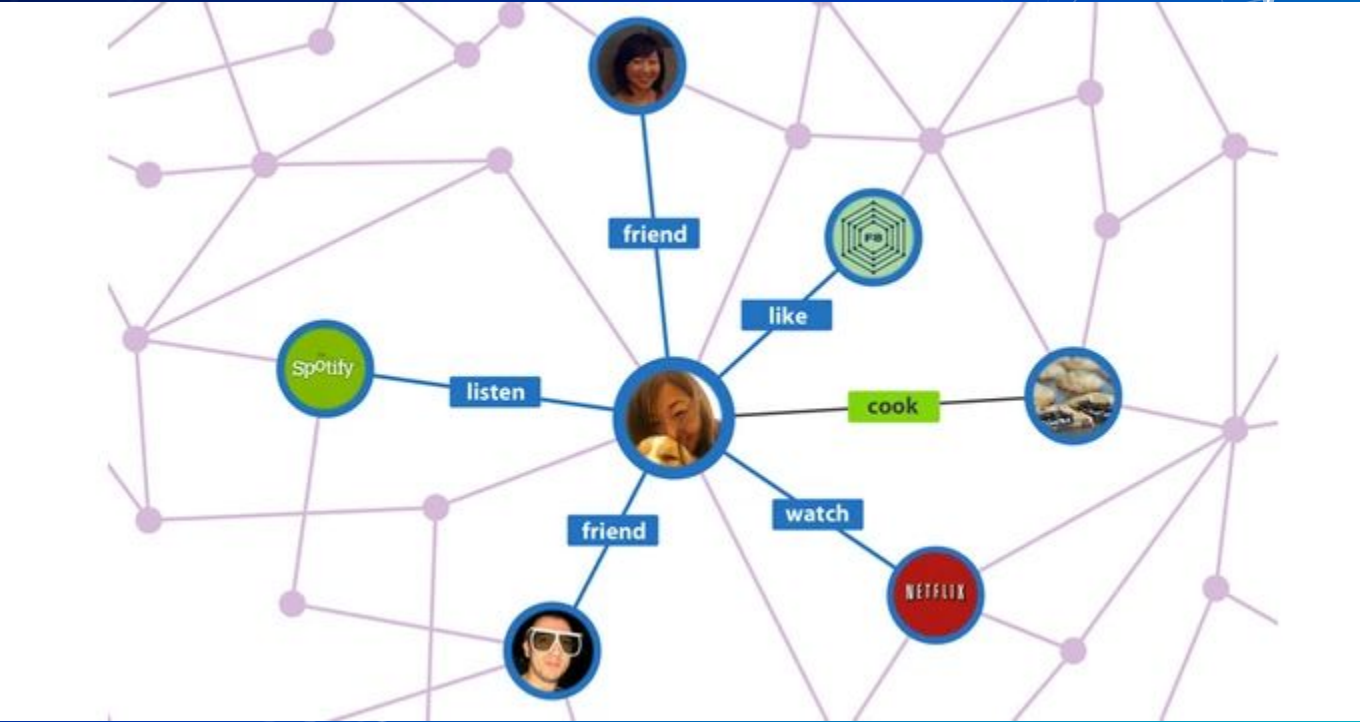
- **G = (V, E)**
- G = Graph , V = **vertices** or **nodes** and E equals **edges**
- Vertices = {1, 2, 3, 4}
- Edges = { {1, 2}, {1, 3} etc… }

Graph Theory 101 (in short)





Graph Theory 101 (in short)



GraphQL Schemas

GraphQL Schemas

- Schema First Design
- GraphQL schemas are defined by GraphQL SDL (schema definition language)
- GraphQL Schema documents are text documents that define the types available in your application

Defining Types

- We are going to look at a podcast application called **GraphPod**
- We are going to **design** a few types for our application
- All schema types have the **.graphql** extension
- First let's learn about GraphQL Schemas and the SDL

Types 101 - Scalar Types

- GraphQL has support for various scalar types
- **Int**
- **Float**
- **String**
- **Boolean**
- **ID** - represents unique identifier
- **!** represents required type

Types 101 - Object Types and Fields

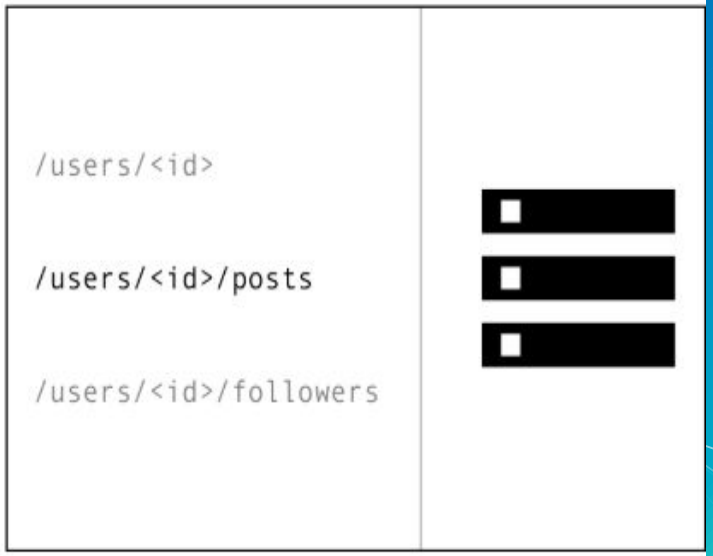
- The core unit of a GraphQL schema is the type. A type represents a custom object and these objects describe your application's features

```
type Hero {
    name: String!
    age: Int!
}
```

Types 101 - Queries and Mutations

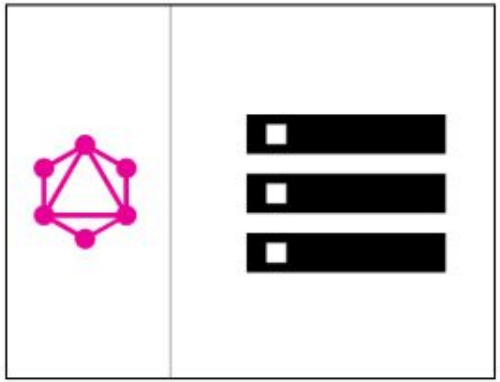
- There are two types that are special namely **query** and **mutation**, these are **root types**
- Query defines that we are going to be **fetching** data, think of it as similar to a **GET** HTTP request
- A **mutation** defines that we are going to be updating the data. Think of them as **POST/PUT/DELETE** HTTP requests


```
{
  "posts": [{
    "id": "ncwon3ce89hs"
    "title": "Learn GraphQL today",
    "content": "Lorem ipsum … ",
    "comments": [ … ],
  }]
}
```



```
query {
  User(id: "er3tg439frjw") {
    name
    posts {
      title
    }
    followers(last: 3) {
      name
    }
  }
}
```

HTTP POST



```
{
  "data": {
    "User": {
      "name": "Mary",
      "posts": [
        { title: "Learn GraphQL today" }
      ],
      "followers": [
        { name: "John" },
        { name: "Alice" },
        { name: "Sarah" },
      ]
    }
  }
}
```

Example GraphQL Mutation

```
mutation {
  createStation (
    description:"A myriad of Python related podcasts"
    url:"https://www.pythonpodcast.com/"
    name:"The Python Podcast"
    followers:100
    active:true
  ) {
    description
    url
    name
    followers
    active

  }
}
```

```
{
  "data": {
    "createStation": {
      "description": "A myriad of Python related podcasts",
      "url": "https://www.pythonpodcast.com/",
      "name": "The Python Podcast",
      "followers": 100,
      "active": true
    }
  }
}
```

Types 101 - Enumeration fields

- We can define enums using the **enum** type

```
enum HeroType {
  WARRIOR
  ARCHER
  SPELLCASTER
}

type Hero {
 heroType: HeroType!
}
```

Types 101 - Lists

- A **list** isn't a scalar but signifies an array of object types or scalars

```
type Hero {
 name: String!
 age: int!
 heroType: HeroType!
 lands: [Land]!
}
```

Types 101 - Interfaces

- A **interface** is an abstract type that includes a certain set of fields that a type must implement if it implements the interface
- Similar to how interfaces work in Java, PHP etc…
- Interfaces are useful when you are returning objects of **different** types
- Interfaces force **validation** and common design among data types

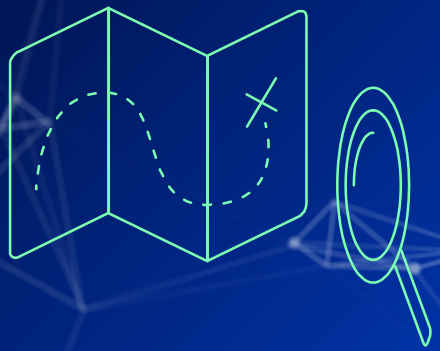
Types 101 - Interfaces

```
interface Weapon {
  id: ID!
  damage: Int!
  durability: Int!
}

type Axe implements Weapon {
  id: ID!
  damage: Int!
  durability: Int!
  name: String!
}
```

Types 101 - Unions

- A **union** is similar to a interface except they don't force to define common data types.

```
union UsedWeapon = Weapon | Axe

query {
 search (text: "axe") {
 __typename
  ... on Axe {
  damage
  name
}

  ... on Weapon {
  damage
  durability
}
}
```



Designing our Schema

Podcast Type

```
type Podcast {
    id: ID!
    name: String!
    description: String!
    url: String!
    postedBy: User!
}
```

User Type

```
type User {
    # note we are leveraging Django's user model
    id: ID!
    username: String!
    password: String!
    firstName: String!
    lastName: String!
}
```

Station Type

```
type Station {
    id: ID!
    name: String!
    description: String!
    url: String!
    active: Boolean
    followers: Int
}
```

Favourite Type

```
type Favourite {
    user: User
    podcast: Podcast
}
```

Building a GraphQL API

Our Toolkit

- Python 3.7
- **Graphene** 2.0 and Graphene-Django
- Django 2.1
- SQLite DB

Why Graphene?

- **Simple** and powerful
- **Extensible** (usage with Django, SQLAlchemy etc...)
- Used by **Yelp**, **Mozilla**, **dailymotion** etc...
- Graphene handles the GraphQL Server

Installation

```
source ~/venv/bin/activate
pip install django==2.1.4 graphene-django==2.2.0 django-filter==2.0.0 django-graphql-jwt==0.1.5
django-admin startproject graphpod
cd graphpod
python manage.py migrate
python manage.py runserver
```

Configuring Graphene with Django

```
INSTALLED_APPS = [
    'django.contrib.admin',
    ...
    'graphene_django',
]

DATABASES = {
    'default': {
        'ENGINE': 'django.db.backends.sqlite3',
        'NAME': os.path.join(BASE_DIR, 'db.sqlite3'),
    }
}


GRAPHENE = {
    'SCHEMA': 'graphpod.schema.schema'
}
```

Creating our apps

```
python manage.py startapp stations
python manage.py startapp podcasts
```

Crafting Stations - Model

```
from django.db import models

class Station(models.Model):
    description = models.TextField()
    url = models.URLField()
    name = models.TextField()
    followers = models.IntegerField(null=True)
    active = models.BooleanField(default=False)
```

Schema Organisation

- Each app will have a **schema.py** which correlates to what mutations and queries that application can serve to our GraphQL Server
- We first create a schema.py in /stations/ app
- We create a base schema.py as we defined in our settings which resolves all other queries and mutations.

Base Schema File (as in settings.py)

```python
import stations.schema
import graphene

class Query(stations.schema.Query, graphene.ObjectType):
    pass

class Mutation(stations.schema.Mutation, graphene.ObjectType):
   pass

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

Crafting Stations - Query Stations

```python
class StationType(DjangoObjectType):
    class Meta:
        model = Station

class Query(graphene.ObjectType):
    stations = graphene.List(StationType)

    def resolve_stations(self, info, **kwargs):
        return Station.objects.all()
```

GraphQL Resolvers Zoomed In

- A **resolver** defines (to the server) how we return schema objects when queried

```
class Query(graphene.ObjectType):
  # ...
  def resolve_stations(self, info, **kwargs):
        return Station.objects.all()
```

```
query {
  stations {
    id
    description
    url
    name
    followers
    active
  }
}
```


```
{
  "data": {
    "stations": [
      {
        "id": "2",
        "description": "A myriad of Python related podcasts",
        "url": "https://www.pythonpodcast.com/",
        "name": "The Python Podcast",
        "followers": 200,
        "active": true
      },
      {
        "id": "4",
        "description": "GraphQL related podcasts",
        "url": "https://graphqlradio.com/",
        "name": "GraphQL Radio",
        "followers": 1049,
        "active": true
      }
    ]
  }
}
```

Crafting Stations - Create Station

```
class CreateStation(graphene.Mutation):
    id = graphene.Int()
    url = graphene.String()
    name = graphene.String()
    description = graphene.String()
    followers = graphene.Int()
    active = graphene.Boolean()

    class Arguments:
        url = graphene.String()
        name = graphene.String()
        description = graphene.String()
        followers = graphene.Int()
        active = graphene.Boolean()
```

```python
    def mutate(self, info, url, name, description, followers, active):
        station = Station(
            url=url,
            name=name,
            description=description,
            followers=followers,
            active=active
        )

        station.save()

        return CreateStation(
            id=station.id,
            url=station.url,
            name=station.name,
            description=station.description,
            followers=station.followers,
            active=station.active
        )
```

```python
class Mutation(graphene.ObjectType):
    create_station = CreateStation.Field()
```

```
mutation {
  createStation (
    description:"GraphQL related podcasts"
    url:"https://graphqlradio.com/"
    name:"GraphQL Radio"
    followers:1049
    active:true
  ) {
    description
    url
    name
    followers
    active

  }
}
```

```
{
  "data": {
    "createStation": {
      "description": "GraphQL related podcasts",
      "url": "https://graphqlradio.com/",
      "name": "GraphQL Radio",
      "followers": 1049,
      "active": true
    }
  }
}
```

Crafting Stations - Delete Station

```
class DeleteStation(graphene.Mutation):
    id = graphene.Int()
    ok = graphene.Boolean()

    class Arguments:
        id = graphene.Int()
```

Crafting Stations - Delete Station

```
class DeleteStation(graphene.Mutation):
    id = graphene.Int()
    ok = graphene.Boolean()

    class Arguments:
        id = graphene.Int()
```

Crafting Stations - Delete Station

```python
def mutate(self, info, id):

        station = Station.objects.filter(id=id).first()

        if not station:
            raise Exception("No Station Found")

        station.delete()

        return DeleteStation(
            id=id,
            ok=True
        )
```

Crafting Stations - Delete Station

```
class Mutation(graphene.ObjectType):
    create_station = CreateStation.Field()
    update_station = UpdateStation.Field()
    delete_station = DeleteStation.Field()
```

```
mutation {
  deleteStation(id:5) {
    ok
    id
  }
}
```

```
{
  "data": {
    "deleteStation": {
      "ok": true,
      "id": 5
    }
  }
}
```

Crafting Stations - Update Station

```
class UpdateStation(graphene.Mutation):
    id = graphene.Int()
    url = graphene.String()
    name = graphene.String()
    description = graphene.String()
    followers = graphene.Int()
    active = graphene.Boolean()
    ok = graphene.Boolean()

    class Arguments:
        id = graphene.Int()
        url = graphene.String()
        name = graphene.String()
        description = graphene.String()
        followers = graphene.Int()
        active = graphene.Boolean()
```

Crafting Stations - Update Station

```python
def mutate(self, info, id, url, name, description, followers, active):

        station = Station.objects.filter(id=id).first()

        if not station:
            raise Exception("Station not Found!")

        station.url = url
        station.name = name
        station.description = description
        station.followers = followers
        station.active = active
        station.save()

        return UpdateStation(
            ok=True,
            id=station.id,
            url=station.url,
            name=station.name,
            description=station.description,
            followers=station.followers,
            active=station.active
        )
```

Crafting Stations - Update Station

```
class Mutation(graphene.ObjectType):
    create_station = CreateStation.Field()
    update_station = UpdateStation.Field()
```

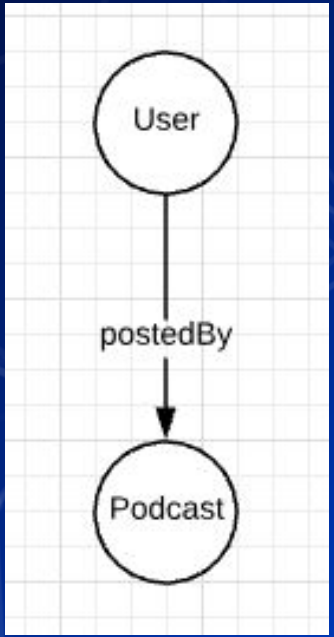
```
mutation {
  updateStation (
    id:2
    description:"A myriad of Python related podcasts"
    url:"https://www.pythonpodcast.com/"
    name:"The Python Podcast"
    followers:300
    active:true
  ) {
    ok
    description
    url
    name
    followers
    active
  }
}
```

```
{
  "data": {
    "updateStation": {
      "ok": true,
      "description": "A myriad of Python related podcasts",
      "url": "https://www.pythonpodcast.com/",
      "name": "The Python Podcast",
      "followers": 300,
      "active": true
    }
  }
}
```

CRUD Complete!

- Just like that we have a CRUD complete API
- We will repeat the same for podcasts
- Users is handled using **graphql-jwt**. This is used to leverage JSON web tokens with GraphQL and Django's user model
- Code can be found on https://github.com/LiamNorman/building-apis-in-graphql

One-to-one connections modelling




```
type Podcast {
    id: ID!
    name: String!
    description: String!
    url: String!
    postedBy: User!
}
```

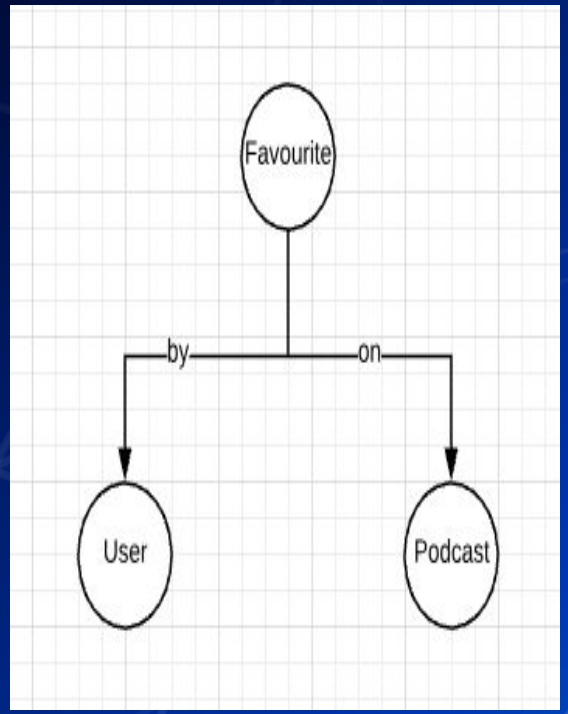
One-to-one connections modelling

```
in podcasts/schema.py:

from django.contrib.auth import get_user_model
class UserType(DjangoObjectType):
    class Meta:
        model = get_user_model()

posted_by = graphene.Field(UserType)
```

```
mutation {
  createPodcast (
    url:"https://mypodcast.example.com"
    description:"Example Podcast"
  ) {
    id
    url
    description
    postedBy {
      id
      username
    }
  }
}
```

```
{
  "data": {
    "createPodcast": {
      "id": 2,
      "url": "https://mypodcast.example.com",
      "description": "Example Podcast",
      "postedBy": {
        "id": "1",
        "username": "liam"
      }
    }
  }
}
```

One-to-many connections - Favouriting Podcasts




```
type Favourite {
    user: User
    podcast: Podcast
}
```

One-to-many connections - Favouriting Podcasts

```
# in podcasts/models.py
class Favourite(models.Model):
    user = models.ForeignKey(settings.AUTH_USER_MODEL, on_delete=models.CASCADE)
    podcast = models.ForeignKey('podcasts.podcast', related_name='favourites',
on_delete=models.CASCADE)

# in podcasts/schema.py

class FavouritePodcastType(DjangoObjectType):
    class Meta:
        model = Favourite
```

One-to-many connections - Favouriting Podcasts

```
class FavouritePodcast(graphene.Mutation):
    user = graphene.Field(UserType)
    podcast = graphene.Field(PodcastType)

    class Arguments:
        podcast_id = graphene.Int()
```

One-to-many connections - Favouriting Podcasts

```
def mutate(self, info, podcast_id):
    user = info.context.user
    if user.is_anonymous:
        raise Exception('You must be logged in to Favourite!')

    podcast = Podcast.objects.filter(id=podcast_id).first()
    if not podcast:
        raise Exception('Invalid Podcast!')

    Favourite.objects.create(
        user=user,
        podcast=podcast
    )

    return FavouritePodcast(user=user, podcast=podcast)
```

One-to-many connections - Favouriting Podcasts

```
class Mutation(graphene.ObjectType):
    favourite_podcast = FavouritePodcast.Field()
```

One-to-many connections - Querying Favourites

```python
class Query(graphene.ObjectType):
    favourites = graphene.List(FavouritePodcastType)

    def resolve_favourites(self, info, **kwargs):
        return Favourite.objects.all()
```

```
mutation {
  favouritePodcast (podcastId:1) {
    podcast{
      id
    }
  }
}
```

```
{
  "data": {
    "favouritePodcast": {
      "podcast": {
        "id": "1"
      }
    }
  }
}
```

User is fetched from auth, implicit

```
query {
  favourites {
    user {
      id
      username
    }
    podcast {
      id
      description
      url
    }
  }
}
```

```
{
  "data": {
    "favourites": [
      {
        "user": {
          "id": "1",
          "username": "liam"
        },
        "podcast": {
          "id": "1",
          "description": "Example Podcast",
          "url": "https://mypodcast.example.com"
        }
      }
    ]
  }
}
```

DEMO TIME

WHEN NOT TO USE GRAPHQL

- GraphQL is **not** a silver bullet
- Caching GraphQL queries **can** be difficult
- Useful in cases where you have **complex datasets** and evolving REST API's
- Minimizes **API** calls but can hit database more (without caching layer)

GITHUB REPO

- Found at https://github.com/LiamNorman/building-apis-in-graphql

LiamNorman / building-apis-in-graphql Private

Unwatch 1 | Star 0 | Fork 0

Code | Issues 0 | Pull requests 0 | Projects 0 | Wiki | Insights | Settings

Talk and codebase for Building APIs in GraphQL Presentation

Edit

Manage topics

3 commits | 1 branch | 0 releases | 1 contributor

Branch: master | New pull request | Create new file | Upload files | Find File | Clone or download

LiamNorman updated readme — Latest commit c5dc05f 2 minutes ago

slides	added codebase and slides	5 hours ago
src	added codebase and slides	5 hours ago
README.md	updated readme	2 minutes ago

README.md

building-apis-in-graphql

Talk and codebase for Building APIs in GraphQL Presentation

Dependencies

Ensure you have the following dependencies:

RESOURCES

- Learning GraphQL Book
- GraphQL Queries and Mutations
- Graphene Docs
- How to GraphQL
- Graphene Types
- GraphQL Python Guide

THANKS!

Any questions?

You can find me on twitter at

- @liamjnorman

CREDITS

- Presentation template by SlidesCarnival