GraphQL in Python and Django

Patrick Arminio @patrick91
Who am I

- Patrick Arminio
- Backend Engineer @ Verve
- Chairperson at Python Italia
- @patrick91 online
GraphQL?
WEB 1.0
REST APIs
While REST APIs are good, they have some shortcomings
Too many API calls (under-fetching)
http GET /user/1
{  
    "name": "Patrick",
    "friends": [
        "/users/2",
        "/users/3",
        "/users/4"
    ],
    "avatar": "/images/123"
}

http GET /user/2
http GET /user/3
http GET /user/4
http GET /user_with_friends/1
{
    "name": "Patrick",
    "friends": [
        { "name": "Fiorella" },
        { "name": "Marco" },
        { "name": "Marta" }
    ],
    "avatar": "/images/123"
}

http GET /user_with_friends/1
http GET /user_with_friends/1
http GET /user_with_friends_and_avatar/1
http GET /user_with_friends/1
http GET /user_with_friends_and_avatar/1
http GET /user_with_avatar/1
http GET /user_with_friends/1
http GET /user_with_friends_and_avatar/1
http GET /user_with_avatar/1
http GET /user_with_small_avatar/1
http GET /user_with_friends/1
http GET /user_with_friends_and_avatar/1
http GET /user_with_avatar/1
http GET /user_with_small_avatar/1
http GET /user_with_small_avatar_and_friends/1
http GET /user_with_friends/1
http GET /user_with_friends_and_avatar/1
http GET /user_with_avatar/1
http GET /user_with_small_avatar/1
http GET /user_with_small_avatar_and_friends/1
http GET /page-1
http GET /user_with_friends/1
http GET /user_with_friends_and_avatar/1
http GET /user_with_avatar/1
http GET /user_with_small_avatar/1
http GET /user_with_small_avatar_and_friends/1
http GET /page-1
http GET /page-2
 [...] At the time, we had over 1,000 different REST endpoints at Coursera (and now we have many more) [...]

Source: Coursera
https://dev-blog.apollodata.com/courseras-journey-to-graphql-a5ad3b77f39a
Too much data (over-fetching)
REST AND HYPERMEDIA LINKS ARE GREAT, BUT NOT ALWAYS THE BEST CHOICE WHEN BUILDING WEBSITES OR APPS.
Documentation
<table>
<thead>
<tr>
<th>Method</th>
<th>Route</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET</td>
<td>/api/ping.json</td>
<td>Raises an exception.</td>
</tr>
<tr>
<td>GET</td>
<td>/api/raise.json</td>
<td>Returns acme.</td>
</tr>
<tr>
<td>GET</td>
<td>/api/v1/().json</td>
<td>Returns acme.</td>
</tr>
<tr>
<td>GET</td>
<td>/api.json</td>
<td>Returns acme.</td>
</tr>
<tr>
<td>GET</td>
<td>/api/ring.json</td>
<td>Returns pong.</td>
</tr>
<tr>
<td>POST</td>
<td>/api/ring.json</td>
<td></td>
</tr>
<tr>
<td>PUT</td>
<td>/api/ring.json</td>
<td></td>
</tr>
<tr>
<td>GET</td>
<td>/api/decorated/ping.json</td>
<td>Returns pong.</td>
</tr>
<tr>
<td>POST</td>
<td>/api/spline.json</td>
<td>Creates a spline that can be reticulated.</td>
</tr>
<tr>
<td>GET</td>
<td>/api/data.json</td>
<td>Returns a plain text file.</td>
</tr>
<tr>
<td>POST</td>
<td>/api/avatar.json</td>
<td>Upload an image.</td>
</tr>
<tr>
<td>GET</td>
<td>/api/swagger_doc.json</td>
<td>Swagger compatible API description</td>
</tr>
<tr>
<td>GET</td>
<td>/api/swagger_doc/{name}.json</td>
<td>Swagger compatible API description for specific API!</td>
</tr>
</tbody>
</table>
Can we do better?
We could extend REST, but...
There won’t be a standard way
GraphQL! 🌟
GraphQL is a Query Language for APIs.

Source: https://graphql.org/
GraphQL is a specification
Single HTTP endpoint
http POST /graphql
{
  user(id: "1") {
    name
    friends {
      name
    }
    avatar
  }
}
}
{
  "user": {
    "name": "Patrick",
    "friends": [
      { "name": "Fiorella" },
      { "name": "Marco" },
      { "name": "Marta" }
    ],
    "avatar": "/images/123"
  }
}
GraphQL is typed
type Query {
    user(id: ID!): User
}

type User {
    name: String!
    friends: [Friend!]!
    avatar: String!
}

type Friend {
    name: String!
}
type Query {
  user(id: ID!): User
}

type User {
  name: String!
  friends: [Friend!]!
  avatar: String!
}

type Friend {
  name: String!
}
type Query {
  user(id: ID!): User
}

type User {
  name: String!
  friends: [Friend!]!
  avatar: String!
}

type Friend {
  name: String!
}
type Query {
  user(id: ID!): User
}

type User {
  name: String!
  friends: [Friend!]!
  avatar: String!
}

type Friend {
  name: String!
}
type Query {
  user(id: ID!): User
}

type User {
  name: String!
  friends: [Friend!]
  avatar: String!
}

type Friend {
  name: String!
}
type Query {
  user(id: ID!): User
}

type User {
  name: String!
  friends: [Friend!]!
  avatar: String!
}

type Friend {
  name: String!
}
Scalar Types

- Int
- Float
- String
- Boolean
- Any user defined scalars (IE. datetime)
Object Types are the objects defined in your GraphQL API. They are objects that have fields that can be of scalar types or other object types.
Type “modifiers”

- List
- Non-nulls
Why is this important?
Static checking
Documentation and introspection
Let’s see an example
Hit the Play Button to get a response here
Operations
3 Main operations

Query
Allows to request data from the server.

Subscription
Allows to subscribe to events, for example when a new user has been created.

Mutation
Allows you to modify/create data on the server. But it is not limited to data, can be used to run anything with side effects.
Query (shortcut)

```json
{
    user(id: "1") {
        name
    }
}
```
query QueryName($id: ID!) {
    user(id: $id) {
        name
    }
}
Mutation

```graphql
mutation MutationName($input: CreateUserInput!) {
  createUser(input: $input) {
    ok
  }
}
```
subscription SubscriptionName {
  onUserCreated {
    name
  }
}
Intermission
GraphQL in Python
2 libraries
Ariadne

https://github.com/mirumee/ariadne/

- Quite new
- Python 3.5+
- “Closer to GraphQL”
Graphene

https://graphene-python.org/

- Most popular
- Python 2.7+ and Python 3.5+
- Nice abstraction on top of GraphQL
- Support for Django and more frameworks
Let’s start with Ariadne
We need a schema

type Query {
    user(id: ID!): User
}

type User {
    name: String!
    friends: [Friend!]!
    avatar: String!
}


type Friend {
    name: String!
}
We need a schema

type Query {
  user(id: ID!): User
}

type User {
  name: String!
  friends: [Friend!]!
  avatar: String!
}

type Friend {
  name: String!
}
We need a schema

type Query {
  user(id: ID!): User
}

type User {
  name: String!
  friends: [Friend!]!
  avatar: String!
}

type Friend {
  name: String!
}
How do we link data to the fields?
Resolvers
Each field on each type is backed by a function called the `resolver` which is provided by the GraphQL server developer.
def resolve_user(_, info, id):
    return {
        "name": "Patrick",
        "friends": [
            {"name": "Fiorella"},
            {"name": "Marco"},
            {"name": "Marta"},
        ],
        "avatar": "/images/123",
    }
Attaching a resolver to a Type

resolvers = {
    "Query": {"user": resolve_user},
    "User": {"name": resolve_name},
}
Creating and running the server

```python
server = GraphQLMiddleware.make_simple_server(
    schema,
    resolvers
)

server.serve_forever()
```
Hit the Play Button to get a response here
Intermission
Graphene
The schema is defined in Python
Our schema

type Friend {
    name: String!
}

type User {
    name: String!
    friends: [Friend!]!
    avatar: String!
}

type Query {
    user(id: ID!): User
}
Defining types with Graphene - Friend

class FriendType(graphene.ObjectType):
    name = graphene.String(required=True)
Types and resolvers live together*

* resolvers can also be external functions
class UserType(graphene.ObjectType):
    name = graphene.String(required=True)
    friends = graphene.List(graphene.NonNull(FriendType))
    avatar = graphene.String(required=True)

    def resolve_friends(self, info):
        return [
            FriendType(name="Marta"),
            FriendType(name="Marco"),
            FriendType(name="Fiorella"),
        ]
class Query(graphene.ObjectType):
    user = graphene.Field(
        UserType,
        id=graphene.ID()
    )

    def resolve_user(self, info, id):
        return UserType(
            name="Patrick",
            avatar="/images/123"
        )
Finally, the schema

```
schema = graphene.Schema(query=Query)
```
Done!
Graphene has support for Django, meaning that:

- Has a built-in view
- It can create types from Django models
- It can create mutations from Forms and DRF Serializers
- Has support for Django filters
What about
Authentication
Authentication

When using GraphQL with HTTPs you have 3 options for authentication:

- Sessions
- HTTP Headers
- Field arguments
Sessions

Basically you rely on the browser sending cookies to your backend service, this works pretty well with Django. Good when you an API that works only with your frontend and when you don’t have a mobile application.
Headers

You can use headers when you have third party clients accessing your API or when you have a mobile app.

Usually it is used in combination with JWT tokens.
Field params

This might be a good solution when you only have a few fields that require authentication. It could work like this:

```json
{
    myBankStatement(token: "ABC123") {
        date
        amount
    }
}
```
Quite easy to create "malicious" queries
{ 
  thread(id: "some-id") { 
    messages(first: 99999) { 
      thread { 
        messages(first: 99999) { 
          thread { 
            messages(first: 99999) { 
              thread { 
                # ...repeat times 10000...
              }
            }
          }
        }
      }
    }
  }
}
Solution for “malicious” queries

To prevent bad queries to happen we can adopt various solutions:

- Timeouts
- Limits on nested fields
- Query cost
- Static queries
Timeouts

Check how long a query is taking, if it is taking more than 1 second you can kill it.

- Prevents huge queries from DOS-ing your server
- Prevents long waiting time
Limit on nested fields

You can parse the incoming GraphQL request and deny queries that are requesting for fields that are too nested. For example you can only allow for maxing 3 levels of nesting and no more.

Easy solution when you don’t need complex checks.
Query costs

This is useful if you have third party clients and when you also want to limit their API usage.

The idea is to give each field a cost and calculate the cost of the query based on the number of fields requested.

This works extremely well with paginated data (where you know how much data you’re asking for)
Query costs - example query

query {
  viewer {
    repositories(first: 50) {
      issues(first: 10) {
        title
      }
    }
  }
}
Query costs - calculating the cost

50 = 50 repositories
+
50 x 10 = 500 repository issues

= 550 total nodes
Static queries

Instead of allowing any query to be ran on your API you could allow only a predefined list of queries. You’d save those queries on a database and reference them by ID. So instead of doing a request passing the query to GraphQL you’d pass only the ID (and the variables if any).
http POST /graphql?id=123
Static queries

- Good to prevent unwanted queries
- Still allows to use all the advantages of GraphQL
- A bit cumbersome to deploy
- If you have third party you need a way for them to declare queries
- Potentially good for caching (see next slide)
http GET /graphql?id=123
Caching
Client Caching
Network Caching
Application Caching
Additional Things
Arguments and Inputs
{  
  search(text: "an") {  
    title  
  }  
}  
}
{
  createUser(input: { ... }) {
    user {
      name
    }
  }
}
Input Types
input CreateUserInput {
    name: String!
    age: Int
}

Enums
enum Conference {
    PYPARIS,
    PYCONX,
    PYCONUS
}
Interfaces
interface Character {
  id: ID!
  name: String!
}

type Human implements Character {
  id: ID!
  name: String!
  friends: [Character]
  starships: [Starship]
}
union SearchResult = Human | Droid

{
  search(text: "an") {
    ... on Human {
      name
      height
    }
    ... on Droid {
      name
      primaryFunction
    }
  }
}
Errors
{{
  user(id: "1") {
    name
    friends {
      name
    }
    avatar
  }
}}
And more
Frontend
Frontend developers benefit a lot from GraphQL, thanks to all the tooling available.
Relay

https://facebook.github.io/relay/

- Made by Facebook
- React Only
Apollo

https://www.apollographql.com/

- Supports many frameworks (React, Vue, etc)
- Big community
- Lots of tooling
Verve is Hiring

Want to work in an amazing company and use Python 3, GraphQL and Django?

https://verve.co/careers/
THANKS!

Patrick Arminio

@patrick91