GRAPHQL AND DJANGO

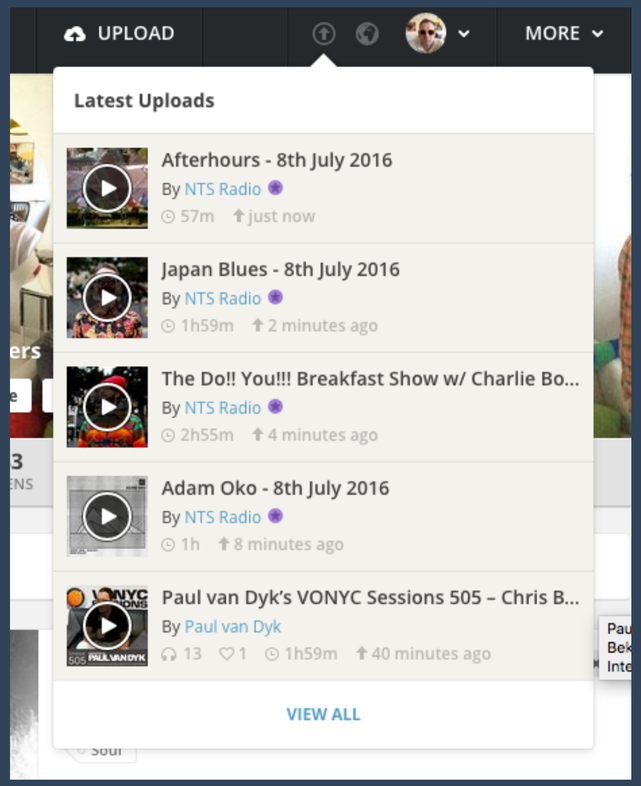
@MATCLAYTON


Mixcloud
Search
CATEGORIES
UPLOAD
SIGN UP
LOGIN
MORE
Re-think Radio
Listen to the best DJs and radio presenters in the world
BAZH
Join friends
OR SIGN UP WITH YOUR EMAIL

API'S SHOULD BE

1. DRY
2. DOCUMENTED
3. PREDICTABLE
4. PRODUCTIVE FOR DEVELOPERS TO WORK ON

SIMPLE EXAMPLE: NOTIFICATIONS



LETS BE GOOD, LETS USE REST

/notifications/

/notifications/1/

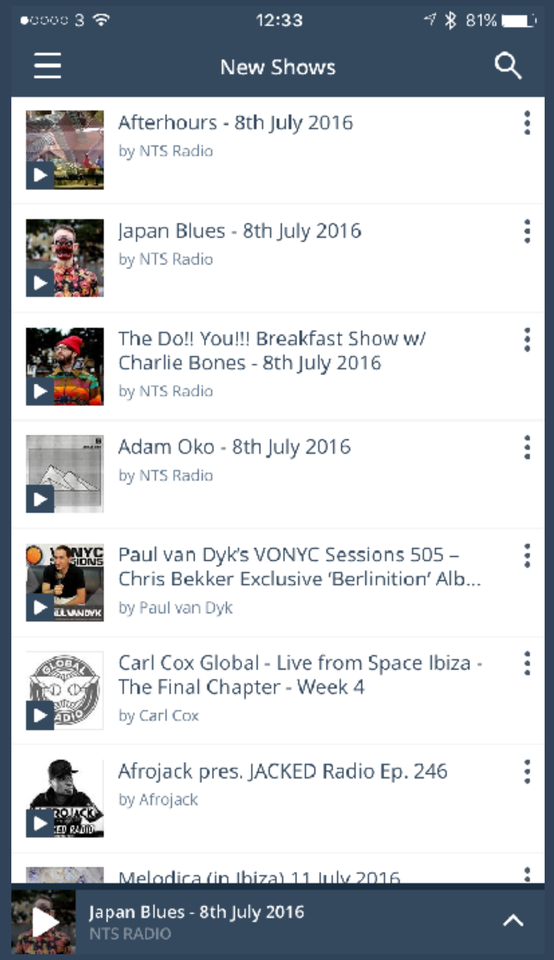
/notifications/2/

TOO MANY API CALLS SO WE BUNDLE IT ALL UP INTO

/notifications/combined/

THIS ISN'T REST, BUT ITS GOOD ENOUGH

MOBILE VERSION



EASY ENOUGH, LETS JUST ADD ANOTHER ENDPOINT

/notifications/mobile_combined/

NEW MOBILE VERSIONS RELEASED (E.G. LISTEN LATER)

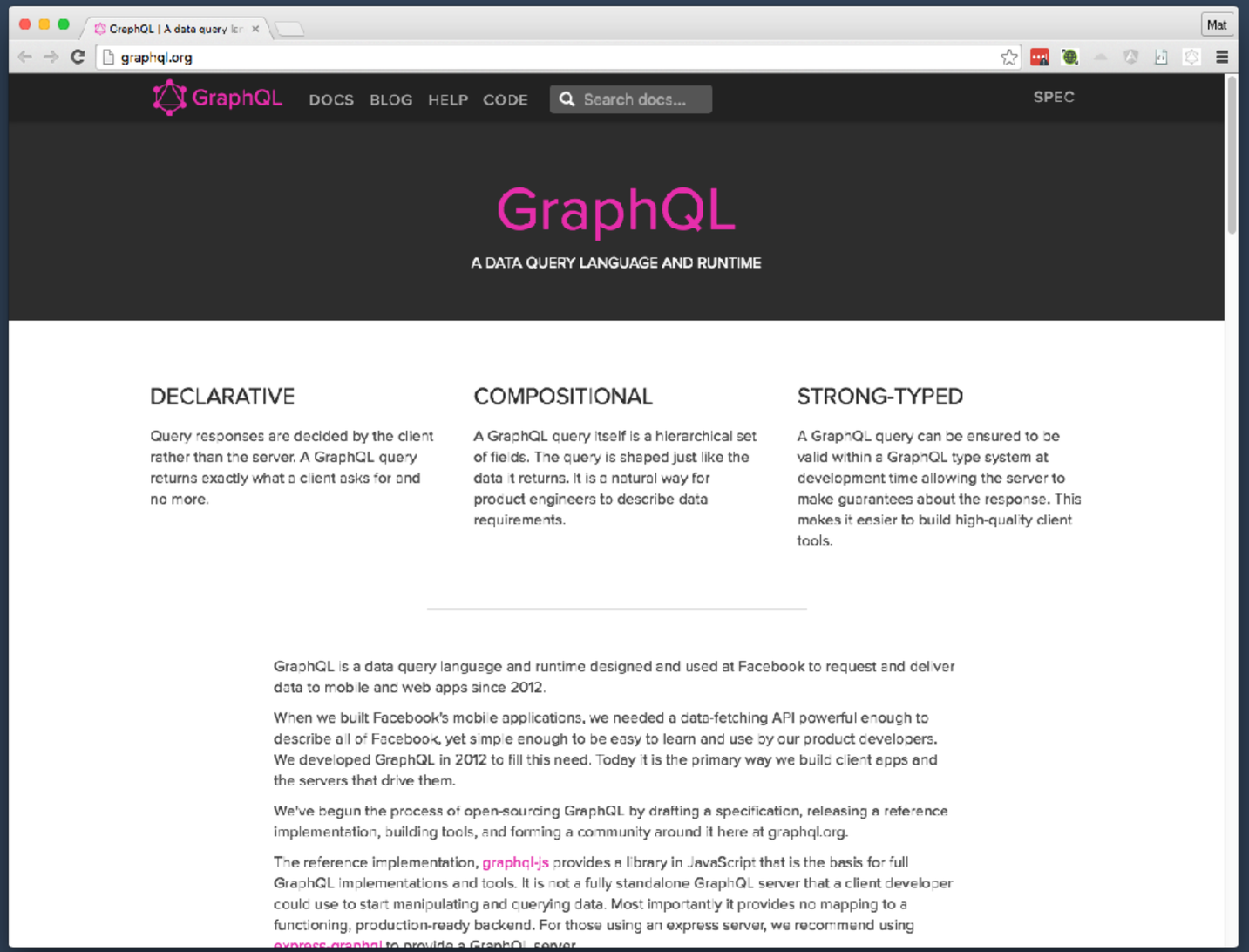
/notifications/mobile_combined/2/
/notifications/mobile_combined/3/
/notifications/mobile_combined/4/

OR WE CHOOSE TO OVERFETCH

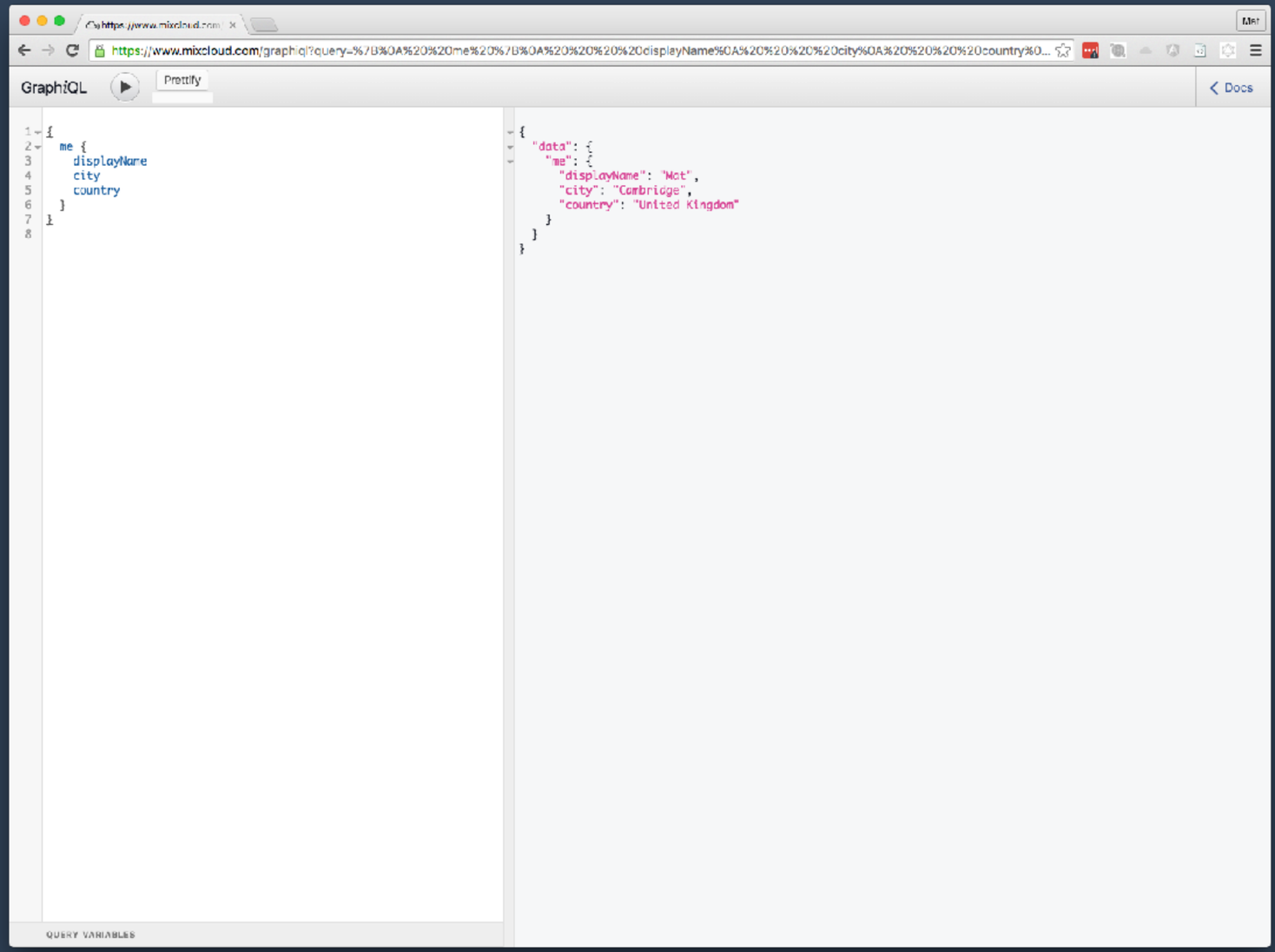
/notifications/mobile_combined/all_data/


DROWNING
IN PSEUDO
REST.

GRAPHQL


graphql.org
GraphQL
DOCS BLOG HELP CODE
Search docs...
SPEC
GraphQL
A DATA QUERY LANGUAGE AND RUNTIME
DECLARATIVE
Query responses are decided by the client rather than the server. A GraphQL query returns exactly what a client asks for and no more.
COMPOSITIONAL
A GraphQL query itself is a hierarchical set of fields. The query is shaped just like the data it returns. It is a natural way for product engineers to describe data requirements.
STRONG-TYPED
A GraphQL query can be ensured to be valid within a GraphQL type system at development time allowing the server to make guarantees about the response. This makes it easier to build high-quality client tools.
GraphQL is a data query language and runtime designed and used at Facebook to request and deliver data to mobile and web apps since 2012.
When we built Facebook's mobile applications, we needed a data-fetching API powerful enough to describe all of Facebook, yet simple enough to be easy to learn and use by our product developers. We developed GraphQL in 2012 to fill this need. Today it is the primary way we build client apps and the servers that drive them.
We've begun the process of open-sourcing GraphQL by drafting a specification, releasing a reference implementation, building tools, and forming a community around it here at graphql.org.
The reference implementation, graphql-js provides a library in JavaScript that is the basis for full GraphQL implementations and tools. It is not a fully standalone GraphQL server that a client developer could use to start manipulating and querying data. Most importantly it provides no mapping to a functioning, production-ready backend. For those using an express server, we recommend using

GRAPHIQL DEMO



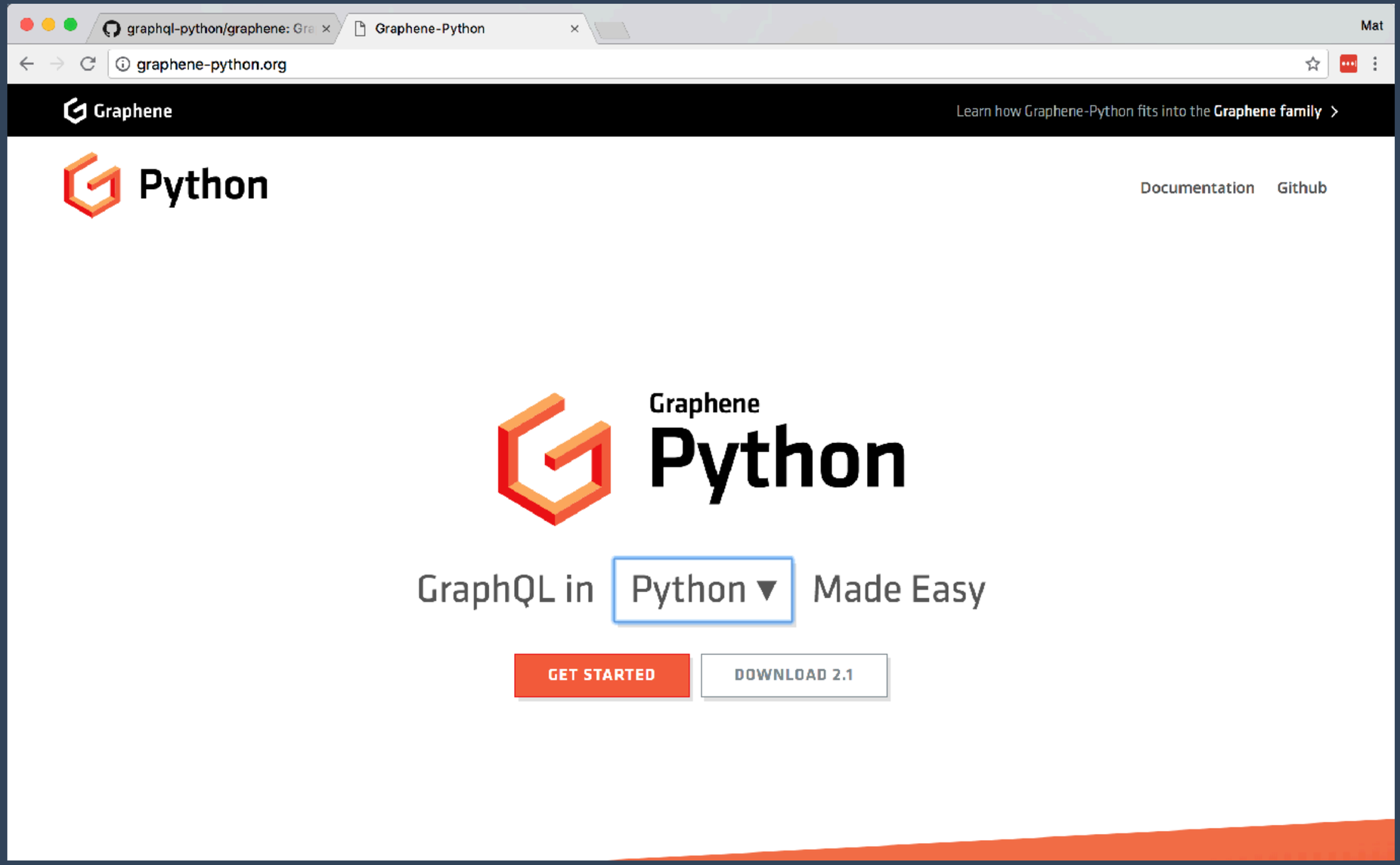
BENEFITS

1. HIERARCHICAL
2. CLIENT-SPECIFIED QUERIES
3. INBUILT BACKWARDS COMPATIBILITY
4. INDEPENDENT FROM HTTP (WEBSOCKETS?)
5. STRONGLY TYPED
6. INTROSPECTIVE (INCLUDING TOOLS + MOCKING)
7. DATA RE-USE

DOWNSIDES AGAINST REST

1. NO HTTP CACHING
   - 99% OF DATA IS BESPOKE TO THE VIEWER
   - INVALIDATION IS NEARLY IMPOSSIBLE
2. LONG QUERY STRINGS
   - POST
   - GZIP
   - MOST QUERIES STILL FIT IN A SINGLE PACKET
3. LACK OF MATURE SERVER IMPLEMENTATIONS

GRAPHENE-DJANGO



DJANGO: SETTINGS.PY

```
INSTALLED_APPS = (
    # ...
    'graphene_django',
)

GRAPHENE = {
    # Where your Graphene schema lives
    'SCHEMA': 'app.schema.schema'
}
```

DJANGO: URLS.PY

```
from django.conf.urls import url
from graphene_django.views import GraphQLView

urlpatterns = [
    # ...
    url(r'^graphql', GraphQLView.as_view(graphiql=True)),
]
```

HELLO WORLD: QUERY

```
import graphene


class Query(graphene.ObjectType):
    hello = graphene.String()

    def resolve_hello(self, info):
        return 'World'


schema = graphene.Schema(query=Query)

schema.execute('''
  query {
    hello
  }
''')
```

HELLO WORLD

QUERY

```
query {
    hello
}
```

RESPONSE

```
{
    “hello”:"World"
}
```

HELLO WORLD: LISTS

```
class UserType(DjangoObjectType):
    class Meta:
        model = User # Django Model


class Query(graphene.ObjectType):
    all_users = graphene.List(UserType)

    def resolve_all_users(self, info):
        return User.objects.all()


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

HELLO WORLD

QUERY

```
query {
  all_users {
    firstname
    lastname
  }
}
```

RESPONSE

```
{
  “all_users”: [
    {
      "firstname": "Mat",
      “lastname": "Clayton",
    },
    {
      "firstname": "Jon",
      “lastname": “Smith",
    },
  ]
}
```

HELLO WORLD: VARIABLES

```
class Query(graphene.ObjectType):
    all_users = graphene.List(UserType)
    firstname_users = graphene.List(UserType)

    def resolve_all_users(self, info):
        return User.objects.all()

    def resolve_firstname_users(self, info):
        firstname = info.context.get('firstname')
        return User.objects.filter(firstname=firstname)

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

HELLO WORLD

QUERY

```
query {
  firstname_users(firstname: “Mat”){
    firstname
    lastname
  }
}
```

RESPONSE

```
{
  “firstname_users”: [
    {
      "firstname": "Mat",
      “lastname": "Clayton",
    }
  ]
}
```

HELLO WORLD: MUTATIONS

```python
import graphene

class CreateUser(graphene.Mutation):
    class Arguments:
        firstname = graphene.String()

    ok = graphene.Boolean()
    user = graphene.Field(lambda: User)

    def mutate(self, info, firstname):
        user = User(firstame=firstname)
        return CreateUser(user=user, ok=True)
```

HELLO WORLD: MUTATIONS

MUTATION

```
mutation myFirstMutation {
    createUser(firstname:"Peter") {
        user {
            firstname
        }
        ok
    }
}
```

RESPONSE

```
{
    “createUser": {
        "user" : {
            "firstname": "Peter"
        },
        "ok": true
    }
}
```

TESTING

```
from graphene.test import Client

def test_hey():
    client = Client(schema)
    executed = client.execute('''query { hello }''')
    assert executed == {
        'data': {
            'hello': 'World'
        }
    }
```

SNAPSHOT TESTING

```
from snapshottest import TestCase

class APITestCase(TestCase):
    def test_api_me(self):
        client = Client(schema)
        self.assertMatchSnapshot(client.execute(
            '''query { hello }'''
        ))
```

REAL WORLD PROBLEMS

1. ARBITRARY QUERY ATTACKS
   - QUERIES CAN BE ARBITRARILY COMPLEX
   - ADD MIDDLEWARE TO DETECT AND ABORT QUERIES

2. N+1 DATABASE QUERIES
   - ADD A MIDDLEWARE TO DETECT N+1 PROBLEMS
   - USE DATALOADERS TO BATCH DATABASE QUERIES


ANY QUESTIONS?
WE ARE HIRING FOR DJANGO AND PYTHON
Mixcloud