Graphene Documentation

Release 1.0

Syrus Akbary

Jun 09, 2019

Contents

Contents:

CHAPTER 1

Getting started

1.1 What is GraphQL?

For an introduction to GraphQL and an overview of its concepts, please refer to the official introduction.

Let's build a basic GraphQL schema from scratch.

1.2 Requirements

- Python (2.7, 3.4, 3.5, 3.6, pypy)
- Graphene (2.0)

1.3 Project setup

```
pip install "graphene>=2.0"
```

1.4 Creating a basic Schema

A GraphQL schema describes your data model, and provides a GraphQL server with an associated set of resolve methods that know how to fetch data.

We are going to create a very simple schema, with a `Query` with only one field: `hello` and an input name. And when we query it, it should return `"Hello {argument}"`.

```
import graphene

class Query(graphene.ObjectType):
```

```
    hello = graphene.String(argument=graphene.String(default_value="stranger"))

    def resolve_hello(self, info, argument):
        return 'Hello ' + argument

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

1.5 Querying

Then we can start querying our schema:

```
result = schema.execute('{ hello }')
print(result.data['hello']) # "Hello stranger"

# or passing the argument in the query
result = schema.execute('{ hello (argument: "graph") }')
print(result.data['hello']) # "Hello graph"
```

Congrats! You got your first graphene schema working!

CHAPTER 2

Types Reference

2.1 Enums

An `Enum` is a special `GraphQL` type that represents a set of symbolic names (members) bound to unique, constant values.

2.1.1 Definition

You can create an `Enum` using classes:

```
import graphene

class Episode(graphene.Enum):
    NEWHOPE = 4
    EMPIRE = 5
    JEDI = 6
```

But also using instances of Enum:

```
Episode = graphene.Enum('Episode', [('NEWHOPE', 4), ('EMPIRE', 5), ('JEDI', 6)])
```

2.1.2 Value descriptions

It's possible to add a description to an enum value, for that the enum value needs to have the `description` property on it.

```
class Episode(graphene.Enum):
    NEWHOPE = 4
    EMPIRE = 5
    JEDI = 6
```

```
    @property
    def description(self):
        if self == Episode.NEWHOPE:
            return 'New Hope Episode'
        return 'Other episode'
```

2.1.3 Usage with Python Enums

In case the Enums are already defined it's possible to reuse them using the `Enum.from_enum` function.

```
graphene.Enum.from_enum(AlreadyExistingPyEnum)
```

`Enum.from_enum` supports a `description` and `deprecation_reason` lambdas as input so you can add description etc. to your enum without changing the original:

```
graphene.Enum.from_enum(
    AlreadyExistingPyEnum,
    description=lambda v: return 'foo' if v == AlreadyExistingPyEnum.Foo else 'bar')
```

2.1.4 Notes

`graphene.Enum` uses `enum.Enum` internally (or a backport if that's not available) and can be used in a similar way, with the exception of member getters.

In the Python `Enum` implementation you can access a member by initing the Enum.

```
from enum import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

assert Color(1) == Color.RED
```

However, in Graphene `Enum` you need to call get to have the same effect:

```
from graphene import Enum
class Color(Enum):
    RED = 1
    GREEN = 2
    BLUE = 3

assert Color.get(1) == Color.RED
```

2.2 Scalars

All Scalar types accept the following arguments. All are optional:

`name`: *string*

Override the name of the Field.

`description`: *string*

A description of the type to show in the GraphiQL browser.

`required`: *boolean*

If `True`, the server will enforce a value for this field. See NonNull. Default is `False`.

`deprecation_reason`: *string*

Provide a deprecation reason for the Field.

`default_value`: *any*

Provide a default value for the Field.

2.2.1 Base scalars

Graphene defines the following base Scalar Types:

`graphene.String`

Represents textual data, represented as UTF-8 character sequences. The String type is most often used by GraphQL to represent free-form human-readable text.

`graphene.Int`

Represents non-fractional signed whole numeric values. Int is a signed 32-bit integer per the GraphQL spec

`graphene.Float`

Represents signed double-precision fractional values as specified by IEEE 754.

`graphene.Boolean`

Represents *true* or *false*.

`graphene.ID`

Represents a unique identifier, often used to refetch an object or as key for a cache. The ID type appears in a JSON response as a String; however, it is not intended to be human-readable. When expected as an input type, any string (such as *"4"*) or integer (such as *4*) input value will be accepted as an ID.

Graphene also provides custom scalars for Dates, Times, and JSON:

`graphene.types.datetime.Date`

Represents a Date value as specified by iso8601.

`graphene.types.datetime.DateTime`

Represents a DateTime value as specified by iso8601.

`graphene.types.datetime.Time`

Represents a Time value as specified by iso8601.

`graphene.types.json.JSONString`

Represents a JSON string.

2.2.2 Custom scalars

You can create custom scalars for your schema. The following is an example for creating a DateTime scalar:

```
import datetime
from graphene.types import Scalar
from graphql.language import ast

class DateTime(Scalar):
    '''DateTime Scalar Description'''

    @staticmethod
    def serialize(dt):
        return dt.isoformat()

    @staticmethod
    def parse_literal(node):
        if isinstance(node, ast.StringValue):
            return datetime.datetime.strptime(
                node.value, "%Y-%m-%dT%H:%M:%S.%f")

    @staticmethod
    def parse_value(value):
        return datetime.datetime.strptime(value, "%Y-%m-%dT%H:%M:%S.%f")
```

2.2.3 Mounting Scalars

Scalars mounted in a `ObjectType`, `Interface` or `Mutation` act as `Fields`.

```
class Person(graphene.ObjectType):
    name = graphene.String()

# Is equivalent to:
class Person(graphene.ObjectType):
    name = graphene.Field(graphene.String)
```

Note: when using the `Field` constructor directly, pass the type and not an instance.

Types mounted in a `Field` act as `Arguments`.

```
graphene.Field(graphene.String, to=graphene.String())

# Is equivalent to:
graphene.Field(graphene.String, to=graphene.Argument(graphene.String))
```

2.3 Lists and Non-Null

Object types, scalars, and enums are the only kinds of types you can define in Graphene. But when you use the types in other parts of the schema, or in your query variable declarations, you can apply additional type modifiers that affect validation of those values.

2.3.1 NonNull

```
import graphene
```

```
class Character(graphene.ObjectType):
    name = graphene.NonNull(graphene.String)
```

Here, we're using a `String` type and marking it as Non-Null by wrapping it using the `NonNull` class. This means that our server always expects to return a non-null value for this field, and if it ends up getting a null value that will actually trigger a GraphQL execution error, letting the client know that something has gone wrong.

The previous `NonNull` code snippet is also equivalent to:

```
import graphene

class Character(graphene.ObjectType):
    name = graphene.String(required=True)
```

2.3.2 List

```
import graphene

class Character(graphene.ObjectType):
    appears_in = graphene.List(graphene.String)
```

Lists work in a similar way: We can use a type modifier to mark a type as a `List`, which indicates that this field will return a list of that type. It works the same for arguments, where the validation step will expect a list for that value.

2.3.3 NonNull Lists

By default items in a list will be considered nullable. To define a list without any nullable items the type needs to be marked as `NonNull`. For example:

```
import graphene

class Character(graphene.ObjectType):
    appears_in = graphene.List(graphene.NonNull(graphene.String))
```

The above results in the type definition:

```
type Character {
    appearsIn: [String!]
}
```

2.4 ObjectTypes

An ObjectType is the single, definitive source of information about your data. It contains the essential fields and behaviors of the data you're querying.

The basics:

- Each ObjectType is a Python class that inherits from `graphene.ObjectType`.
- Each attribute of the ObjectType represents a `Field`.

2.4.1 Quick example

This example model defines a Person, with a first and a last name:

```
import graphene

class Person(graphene.ObjectType):
    first_name = graphene.String()
    last_name = graphene.String()
    full_name = graphene.String()

    def resolve_full_name(root, info):
        return '{} {}'.format(root.first_name, root.last_name)
```

first_name and **last_name** are fields of the ObjectType. Each field is specified as a class attribute, and each attribute maps to a Field.

The above `Person` ObjectType has the following schema representation:

```
type Person {
  firstName: String
  lastName: String
  fullName: String
}
```

2.4.2 Resolvers

A resolver is a method that resolves certain fields within an `ObjectType`. If not specified otherwise, the resolver of a field is the `resolve_{field_name}` method on the `ObjectType`.

By default resolvers take the arguments `info` and `*args`.

NOTE: The resolvers on an `ObjectType` are always treated as `staticmethods`, so the first argument to the resolver method `self` (or `root`) need not be an actual instance of the `ObjectType`.

If an explicit resolver is not defined on the `ObjectType` then Graphene will attempt to use a property with the same name on the object or dict that is passed to the `ObjectType`.

```
import graphene

class Person(graphene.ObjectType):
    first_name = graphene.String()
    last_name = graphene.String()

class Query(graphene.ObjectType):
    me = graphene.Field(Person)
    best_friend = graphene.Field(Person)

    def resolve_me(_, info):
        # returns an object that represents a Person
        return get_human(name='Luke Skywalker')

    def resolve_best_friend(_, info):
        return {
            "first_name": "R2",
            "last_name": "D2",
        }
```

Resolvers with arguments

Any arguments that a field defines gets passed to the resolver function as kwargs. For example:

```
import graphene

class Query(graphene.ObjectType):
    human_by_name = graphene.Field(Human, name=graphene.String(required=True))

    def resolve_human_by_name(_, info, name):
        return get_human(name=name)
```

You can then execute the following query:

```
query {
    humanByName(name: "Luke Skywalker") {
        firstName
        lastName
    }
}
```

NOTE: if you define an argument for a field that is not required (and in a query execution it is not provided as an argument) it will not be passed to the resolver function at all. This is so that the developer can differenciate between a `undefined` value for an argument and an explicit `null` value.

For example, given this schema:

```
import graphene

class Query(graphene.ObjectType):
    hello = graphene.String(required=True, name=graphene.String())

    def resolve_hello(_, info, name):
        return name if name else 'World'
```

And this query:

```
query {
    hello
}
```

An error will be thrown:

```
TypeError: resolve_hello() missing 1 required positional argument: 'name'
```

You can fix this error in 2 ways. Either by combining all keyword arguments into a dict:

```
class Query(graphene.ObjectType):
    hello = graphene.String(required=True, name=graphene.String())

    def resolve_hello(_, info, **args):
        return args.get('name', 'World')
```

Or by setting a default value for the keyword argument:

```
class Query(graphene.ObjectType):
    hello = graphene.String(required=True, name=graphene.String())
```

```
    def resolve_hello(_, info, name='World'):
        return name
```

Resolvers outside the class

A field can use a custom resolver from outside the class:

```
import graphene

def resolve_full_name(person, info):
    return '{} {}'.format(person.first_name, person.last_name)

class Person(graphene.ObjectType):
    first_name = graphene.String()
    last_name = graphene.String()
    full_name = graphene.String(resolver=resolve_full_name)
```

2.4.3 Instances as data containers

Graphene `ObjectTypes` can act as containers too. So with the previous example you could do:

```
peter = Person(first_name='Peter', last_name='Griffin')

peter.first_name # prints "Peter"
peter.last_name # prints "Griffin"
```

2.4.4 Changing the name

By default the type name in the GraphQL schema will be the same as the class name that defines the `ObjectType`. This can be changed by setting the `name` property on the `Meta` class:

```
class MyGraphQlSong(graphene.ObjectType):
    class Meta:
        name = 'Song'
```

2.5 Interfaces

An *Interface* is an abstract type that defines a certain set of fields that a type must include to implement the interface.

For example, you can define an Interface `Character` that represents any character in the Star Wars trilogy:

```
import graphene

class Character(graphene.Interface):
    id = graphene.ID(required=True)
    name = graphene.String(required=True)
    friends = graphene.List(lambda: Character)
```

Any ObjectType that implements `Character` will have these exact fields, with these arguments and return types.

For example, here are some types that might implement `Character`:

```
class Human(graphene.ObjectType):
    class Meta:
        interfaces = (Character, )

    starships = graphene.List(Starship)
    home_planet = graphene.String()

class Droid(graphene.ObjectType):
    class Meta:
        interfaces = (Character, )

    primary_function = graphene.String()
```

Both of these types have all of the fields from the `Character` interface, but also bring in extra fields, `home_planet`, `starships` and `primary_function`, that are specific to that particular type of character.

The full GraphQL schema defition will look like this:

```
interface Character {
    id: ID!
    name: String!
    friends: [Character]
}

type Human implements Character {
    id: ID!
    name: String!
    friends: [Character]
    starships: [Starship]
    homePlanet: String
}

type Droid implements Character {
    id: ID!
    name: String!
    friends: [Character]
    primaryFunction: String
}
```

Interfaces are useful when you want to return an object or set of objects, which might be of several different types.

For example, you can define a field `hero` that resolves to any `Character`, depending on the episode, like this:

```
class Query(graphene.ObjectType):
    hero = graphene.Field(
        Character,
        required=True,
        episode=graphene.Int(required=True)
    )

    def resolve_hero(_, info, episode):
        # Luke is the hero of Episode V
        if episode == 5:
            return get_human(name='Luke Skywalker')
        return get_droid(name='R2-D2')

schema = graphene.Schema(query=Query, types=[Human, Droid])
```

This allows you to directly query for fields that exist on the Character interface as well as selecting specific fields on

any type that implements the interface using inline fragments.

For example, the following query:

```
query HeroForEpisode($episode: Int!) {
    hero(episode: $episode) {
        __typename
        name
        ... on Droid {
            primaryFunction
        }
        ... on Human {
            homePlanet
        }
    }
}
```

Will return the following data with variables `{ "episode": 4 }`:

```
{
    "data": {
        "hero": {
            "__typename": "Droid",
            "name": "R2-D2",
            "primaryFunction": "Astromech"
        }
    }
}
```

And different data with the variables `{ "episode": 5 }`:

```
{
    "data": {
        "hero": {
            "__typename": "Human",
            "name": "Luke Skywalker",
            "homePlanet": "Tatooine"
        }
    }
}
```

2.5.1 Resolving data objects to types

As you build out your schema in Graphene it's common for your resolvers to return objects that represent the data backing your GraphQL types rather than instances of the Graphene types (e.g. Django or SQLAlchemy models). This works well with `ObjectType` and `Scalar` fields, however when you start using Interfaces you might come across this error:

```
"Abstract type Character must resolve to an Object type at runtime for field Query.
↪hero ..."
```

This happens because Graphene doesn't have enough information to convert the data object into a Graphene type needed to resolve the `Interface`. To solve this you can define a `resolve_type` class method on the `Interface` which maps a data object to a Graphene type:

```
class Character(graphene.Interface):
    id = graphene.ID(required=True)
```

```
    name = graphene.String(required=True)

    @classmethod
    def resolve_type(cls, instance, info):
        if instance.type == 'DROID':
            return Droid
        return Human
```

2.6 Unions

Union types are very similar to interfaces, but they don't get to specify any common fields between the types.

The basics:

- Each Union is a Python class that inherits from `graphene.Union`.
- Unions don't have any fields on it, just links to the possible objecttypes.

2.6.1 Quick example

This example model defines several ObjectTypes with their own fields. `SearchResult` is the implementation of `Union` of this object types.

```
import graphene

class Human(graphene.ObjectType):
    name = graphene.String()
    born_in = graphene.String()

class Droid(graphene.ObjectType):
    name = graphene.String()
    primary_function = graphene.String()

class Starship(graphene.ObjectType):
    name = graphene.String()
    length = graphene.Int()

class SearchResult(graphene.Union):
    class Meta:
        types = (Human, Droid, Starship)
```

Wherever we return a SearchResult type in our schema, we might get a Human, a Droid, or a Starship. Note that members of a union type need to be concrete object types; you can't create a union type out of interfaces or other unions.

The above types have the following representation in a schema:

```
type Droid {
  name: String
  primaryFunction: String
}

type Human {
  name: String
  bornIn: String
```

```
}

type Ship {
  name: String
  length: Int
}

union SearchResult = Human | Droid | Starship
```

2.7 Schema

A Schema is created by supplying the root types of each type of operation, query and mutation (optional). A schema definition is then supplied to the validator and executor.

```
my_schema = Schema(
    query=MyRootQuery,
    mutation=MyRootMutation,
)
```

2.7.1 Types

There are some cases where the schema cannot access all of the types that we plan to have. For example, when a field returns an `Interface`, the schema doesn't know about any of the implementations.

In this case, we need to use the `types` argument when creating the Schema.

```
my_schema = Schema(
    query=MyRootQuery,
    types=[SomeExtraObjectType, ]
)
```

2.7.2 Querying

To query a schema, call the `execute` method on it.

```
my_schema.execute('{ lastName }')
```

2.7.3 Auto CamelCase field names

By default all field and argument names (that are not explicitly set with the `name` arg) will be converted from `snake_case` to `camelCase` (as the API is usually being consumed by a js/mobile client)

For example with the ObjectType

```
class Person(graphene.ObjectType):
    last_name = graphene.String()
    other_name = graphene.String(name='_other_Name')
```

the `last_name` field name is converted to `lastName`.

In case you don't want to apply this transformation, provide a `name` argument to the field constructor. `other_name` converts to `_other_Name` (without further transformations).

Your query should look like

```
{
    lastName
    _other_Name
}
```

To disable this behavior, set the `auto_camelcase` to `False` upon schema instantiation.

```
my_schema = Schema(
    query=MyRootQuery,
    auto_camelcase=False,
)
```

2.8 Mutations

A Mutation is a special ObjectType that also defines an Input.

2.8.1 Quick example

This example defines a Mutation:

```
import graphene

class CreatePerson(graphene.Mutation):
    class Arguments:
        name = graphene.String()

    ok = graphene.Boolean()
    person = graphene.Field(lambda: Person)

    def mutate(self, info, name):
        person = Person(name=name)
        ok = True
        return CreatePerson(person=person, ok=ok)
```

person and **ok** are the output fields of the Mutation when it is resolved.

Arguments attributes are the arguments that the Mutation `CreatePerson` needs for resolving, in this case **name** will be the only argument for the mutation.

mutate is the function that will be applied once the mutation is called.

So, we can finish our schema like this:

```
# ... the Mutation Class

class Person(graphene.ObjectType):
    name = graphene.String()
    age = graphene.Int()

class MyMutations(graphene.ObjectType):
    create_person = CreatePerson.Field()
```

```
# We must define a query for our schema
class Query(graphene.ObjectType):
    person = graphene.Field(Person)

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

2.8.2 Executing the Mutation

Then, if we query (`schema.execute(query_str)`) the following:

```
mutation myFirstMutation {
    createPerson(name:"Peter") {
        person {
            name
        }
        ok
    }
}
```

We should receive:

```
{
    "createPerson": {
        "person" : {
            "name": "Peter"
        },
        "ok": true
    }
}
```

2.8.3 InputFields and InputObjectTypes

InputFields are used in mutations to allow nested input data for mutations

To use an InputField you define an InputObjectType that specifies the structure of your input data

```
import graphene

class PersonInput(graphene.InputObjectType):
    name = graphene.String(required=True)
    age = graphene.Int(required=True)

class CreatePerson(graphene.Mutation):
    class Arguments:
        person_data = PersonInput(required=True)

    person = graphene.Field(Person)

    @staticmethod
    def mutate(root, info, person_data=None):
        person = Person(
            name=person_data.name,
            age=person_data.age
```

```
        )
        return CreatePerson(person=person)
```

Note that **name** and **age** are part of **person_data** now

Using the above mutation your new query would look like this:

```
mutation myFirstMutation {
    createPerson(personData: {name:"Peter", age: 24}) {
        person {
            name,
            age
        }
    }
}
```

InputObjectTypes can also be fields of InputObjectTypes allowing you to have as complex of input data as you need

```
import graphene

class LatLngInput(graphene.InputObjectType):
    lat = graphene.Float()
    lng = graphene.Float()

#A location has a latlng associated to it
class LocationInput(graphene.InputObjectType):
    name = graphene.String()
    latlng = graphene.InputField(LatLngInput)
```

2.8.4 Output type example

To return an existing ObjectType instead of a mutation-specific type, set the **Output** attribute to the desired ObjectType:

```
import graphene

class CreatePerson(graphene.Mutation):
    class Arguments:
        name = graphene.String()

    Output = Person

    def mutate(self, info, name):
        return Person(name=name)
```

Then, if we query (`schema.execute(query_str)`) the following:

```
mutation myFirstMutation {
    createPerson(name:"Peter") {
        name
        __typename
    }
}
```

We should receive:

```
{
    "createPerson": {
        "name": "Peter",
        "__typename": "Person"
    }
}
```

2.9 AbstractTypes

An AbstractType contains fields that can be shared among `graphene.ObjectType`, `graphene.Interface`, `graphene.InputObjectType` or other `graphene.AbstractType`.

The basics:

- Each AbstractType is a Python class that inherits from `graphene.AbstractType`.
- Each attribute of the AbstractType represents a field (a `graphene.Field` or `graphene.InputField` depending on where it is mounted)

2.9.1 Quick example

In this example UserFields is an `AbstractType` with a name. `User` and `UserInput` are two types that have their own fields plus the ones defined in `UserFields`.

```
import graphene

class UserFields(graphene.AbstractType):
    name = graphene.String()

class User(graphene.ObjectType, UserFields):
    pass

class UserInput(graphene.InputObjectType, UserFields):
    pass
```

```
type User {
  name: String
}

inputtype UserInput {
  name: String
}
```

CHAPTER 3

Execution

3.1 Executing a query

For executing a query a schema, you can directly call the `execute` method on it.

```
schema = graphene.Schema(...)
result = schema.execute('{ name }')
```

`result` represents the result of execution. `result.data` is the result of executing the query, `result.errors` is `None` if no errors occurred, and is a non-empty list if an error occurred.

3.1.1 Context

You can pass context to a query via `context`.

```
class Query(graphene.ObjectType):
    name = graphene.String()

    def resolve_name(root, info):
        return info.context.get('name')

schema = graphene.Schema(Query)
result = schema.execute('{ name }', context={'name': 'Syrus'})
```

3.1.2 Variables

You can pass variables to a query via `variables`.

```
class Query(graphene.ObjectType):
    user = graphene.Field(User, id=graphene.ID(required=True))

    def resolve_user(root, info, id):
```

```
        return get_user_by_id(id)

schema = graphene.Schema(Query)
result = schema.execute(
    '''
      query getUser($id: ID) {
        user(id: $id) {
          id
          firstName
          lastName
        }
      }
    ''',
    variables={'id': 12},
)
```

3.2 Middleware

You can use `middleware` to affect the evaluation of fields in your schema.

A middleware is any object or function that responds to `resolve(next_middleware, *args)`.

Inside that method, it should either:

- Send `resolve` to the next middleware to continue the evaluation; or
- Return a value to end the evaluation early.

3.2.1 Resolve arguments

Middlewares `resolve` is invoked with several arguments:

- `next` represents the execution chain. Call `next` to continue evaluation.
- `root` is the root value object passed throughout the query.
- `info` is the resolver info.
- `args` is the dict of arguments passed to the field.

3.2.2 Example

This middleware only continues evaluation if the `field_name` is not `'user'`

```
class AuthorizationMiddleware(object):
    def resolve(next, root, info, **args):
        if info.field_name == 'user':
            return None
        return next(root, info, **args)
```

And then execute it with:

```
result = schema.execute('THE QUERY', middleware=[AuthorizationMiddleware()])
```

3.2.3 Functional example

Middleware can also be defined as a function. Here we define a middleware that logs the time it takes to resolve each field

```
from time import time as timer

def timing_middleware(next, root, info, **args):
    start = timer()
    return_value = next(root, info, **args)
    duration = timer() - start
    logger.debug("{parent_type}.{field_name}: {duration} ms".format(
        parent_type=root._meta.name if root and hasattr(root, '_meta') else '',
        field_name=info.field_name,
        duration=round(duration * 1000, 2)
    ))
    return return_value
```

And then execute it with:

```
result = schema.execute('THE QUERY', middleware=[timing_middleware])
```

3.3 Dataloader

DataLoader is a generic utility to be used as part of your application's data fetching layer to provide a simplified and consistent API over various remote data sources such as databases or web services via batching and caching.

3.3.1 Batching

Batching is not an advanced feature, it's DataLoader's primary feature. Create loaders by providing a batch loading function.

```
from promise import Promise
from promise.dataloader import DataLoader

class UserLoader(DataLoader):
    def batch_load_fn(self, keys):
        # Here we return a promise that will result on the
        # corresponding user for each key in keys
        return Promise.resolve([get_user(id=key) for key in keys])
```

A batch loading function accepts a list of keys, and returns a `Promise` which resolves to a list of `values`.

Then load individual values from the loader. `DataLoader` will coalesce all individual loads which occur within a single frame of execution (executed once the wrapping promise is resolved) and then call your batch function with all requested keys.

```
user_loader = UserLoader()

user_loader.load(1).then(lambda user: user_loader.load(user.best_friend_id))

user_loader.load(2).then(lambda user: user_loader.load(user.best_friend_id))
```

A naive application may have issued *four* round-trips to a backend for the required information, but with `DataLoader` this application will make at most *two*.

Note that loaded values are one-to-one with the keys and must have the same order. This means that if you load all values from a single query, you must make sure that you then order the query result for the results to match the keys:

```
class UserLoader(DataLoader):
    def batch_load_fn(self, keys):
        users = {user.id: user for user in User.objects.filter(id__in=keys)}
        return Promise.resolve([users.get(user_id) for user_id in keys])
```

`DataLoader` allows you to decouple unrelated parts of your application without sacrificing the performance of batch data-loading. While the loader presents an API that loads individual values, all concurrent requests will be coalesced and presented to your batch loading function. This allows your application to safely distribute data fetching requirements throughout your application and maintain minimal outgoing data requests.

3.3.2 Using with Graphene

DataLoader pairs nicely well with Graphene/GraphQL. GraphQL fields are designed to be stand-alone functions. Without a caching or batching mechanism, it's easy for a naive GraphQL server to issue new database requests each time a field is resolved.

Consider the following GraphQL request:

```
{
  me {
    name
    bestFriend {
      name
    }
    friends(first: 5) {
      name
      bestFriend {
        name
      }
    }
  }
}
```

Naively, if `me`, `bestFriend` and `friends` each need to request the backend, there could be at most 13 database requests!

When using DataLoader, we could define the User type using our previous example with leaner code and at most 4 database requests, and possibly fewer if there are cache hits.

```
class User(graphene.ObjectType):
    name = graphene.String()
    best_friend = graphene.Field(lambda: User)
    friends = graphene.List(lambda: User)

    def resolve_best_friend(self, info):
        return user_loader.load(self.best_friend_id)

    def resolve_friends(self, info):
        return user_loader.load_many(self.friend_ids)
```

CHAPTER 4

Relay

Graphene has complete support for Relay and offers some utils to make integration from Python easy.

4.1 Nodes

A `Node` is an Interface provided by `graphene.relay` that contains a single field `id` (which is a `ID!`). Any object that inherits from it has to implement a `get_node` method for retrieving a `Node` by an *id*.

4.1.1 Quick example

Example usage (taken from the Starwars Relay example):

```
class Ship(graphene.ObjectType):
    '''A ship in the Star Wars saga'''
    class Meta:
        interfaces = (relay.Node, )

    name = graphene.String(description='The name of the ship.')

    @classmethod
    def get_node(cls, info, id):
        return get_ship(id)
```

The `id` returned by the `Ship` type when you query it will be a scalar which contains enough info for the server to know its type and its id.

For example, the instance `Ship(id=1)` will return `U2hpcDox` as the id when you query it (which is the base64 encoding of `Ship:1`), and which could be useful later if we want to query a node by its id.

4.1.2 Custom Nodes

You can use the predefined `relay.Node` or you can subclass it, defining custom ways of how a node id is encoded (using the `to_global_id` method in the class) or how we can retrieve a Node given a encoded id (with the `get_node_from_global_id` method).

Example of a custom node:

```
class CustomNode(Node):

    class Meta:
        name = 'Node'

    @staticmethod
    def to_global_id(type, id):
        return '{}:{}'.format(type, id)

    @staticmethod
    def get_node_from_global_id(info, global_id, only_type=None):
        type, id = global_id.split(':')
        if only_type:
            # We assure that the node type that we want to retrieve
            # is the same that was indicated in the field type
            assert type == only_type._meta.name, 'Received not compatible node.'

        if type == 'User':
            return get_user(id)
        elif type == 'Photo':
            return get_photo(id)
```

The `get_node_from_global_id` method will be called when `CustomNode.Field` is resolved.

4.1.3 Accessing node types

If we want to retrieve node instances from a `global_id` (scalar that identifies an instance by it's type name and id), we can simply do `Node.get_node_from_global_id(info, global_id)`.

In the case we want to restrict the instance retrieval to a specific type, we can do: `Node.get_node_from_global_id(info, global_id, only_type=Ship)`. This will raise an error if the `global_id` doesn't correspond to a Ship type.

4.1.4 Node Root field

As is required in the Relay specification, the server must implement a root field called `node` that returns a `Node` Interface.

For this reason, `graphene` provides the field `relay.Node.Field`, which links to any type in the Schema which implements `Node`. Example usage:

```
class Query(graphene.ObjectType):
    # Should be CustomNode.Field() if we want to use our custom Node
    node = relay.Node.Field()
```

4.2 Connection

A connection is a vitaminized version of a List that provides ways of slicing and paginating through it. The way you create Connection types in `graphene` is using `relay.Connection` and `relay.ConnectionField`.

4.2.1 Quick example

If we want to create a custom Connection on a given node, we have to subclass the `Connection` class.

In the following example, `extra` will be an extra field in the connection, and `other` an extra field in the Connection Edge.

```
class ShipConnection(Connection):
    extra = String()

    class Meta:
        node = Ship

    class Edge:
        other = String()
```

The `ShipConnection` connection class, will have automatically a `pageInfo` field, and a `edges` field (which is a list of `ShipConnection.Edge`). This `Edge` will have a `node` field linking to the specified node (in `ShipConnection.Meta`) and the field `other` that we defined in the class.

4.2.2 Connection Field

You can create connection fields in any Connection, in case any ObjectType that implements `Node` will have a default Connection.

```
class Faction(graphene.ObjectType):
    name = graphene.String()
    ships = relay.ConnectionField(ShipConnection)

    def resolve_ships(self, info):
        return []
```

4.3 Mutations

Most APIs don't just allow you to read data, they also allow you to write.

In GraphQL, this is done using mutations. Just like queries, Relay puts some additional requirements on mutations, but Graphene nicely manages that for you. All you need to do is make your mutation a subclass of `relay.ClientIDMutation`.

```
class IntroduceShip(relay.ClientIDMutation):

    class Input:
        ship_name = graphene.String(required=True)
        faction_id = graphene.String(required=True)

    ship = graphene.Field(Ship)
    faction = graphene.Field(Faction)
```

```
    @classmethod
    def mutate_and_get_payload(cls, root, info, **input):
        ship_name = input.ship_name
        faction_id = input.faction_id
        ship = create_ship(ship_name, faction_id)
        faction = get_faction(faction_id)
        return IntroduceShip(ship=ship, faction=faction)
```

4.3.1 Accepting Files

Mutations can also accept files, that's how it will work with different integrations:

```
class UploadFile(graphene.ClientIDMutation):
     class Input:
         pass
         # nothing needed for uploading file

     # your return fields
     success = graphene.String()

    @classmethod
    def mutate_and_get_payload(cls, root, info, **input):
        # When using it in Django, context will be the request
        files = info.context.FILES
        # Or, if used in Flask, context will be the flask global request
        # files = context.files

        # do something with files

        return UploadFile(success=True)
```

4.4 Useful links

- Getting started with Relay
- Relay Global Identification Specification
- Relay Cursor Connection Specification
- Relay input Object Mutation

CHAPTER 5

Testing in Graphene

Automated testing is an extremely useful bug-killing tool for the modern developer. You can use a collection of tests – a test suite – to solve, or avoid, a number of problems:

- When you're writing new code, you can use tests to validate your code works as expected.
- When you're refactoring or modifying old code, you can use tests to ensure your changes haven't affected your application's behavior unexpectedly.

Testing a GraphQL application is a complex task, because a GraphQL application is made of several layers of logic – schema definition, schema validation, permissions and field resolution.

With Graphene test-execution framework and assorted utilities, you can simulate GraphQL requests, execute mutations, inspect your application's output and generally verify your code is doing what it should be doing.

5.1 Testing tools

Graphene provides a small set of tools that come in handy when writing tests.

5.1.1 Test Client

The test client is a Python class that acts as a dummy GraphQL client, allowing you to test your views and interact with your Graphene-powered application programmatically.

Some of the things you can do with the test client are:

- Simulate Queries and Mutations and observe the response.
- Test that a given query request is rendered by a given Django template, with a template context that contains certain values.

5.1.2 Overview and a quick example

To use the test client, instantiate `graphene.test.Client` and retrieve GraphQL responses:

```
from graphene.test import Client

def test_hey():
    client = Client(my_schema)
    executed = client.execute('''{ hey }''')
    assert executed == {
        'data': {
            'hey': 'hello!'
        }
    }
```

5.1.3 Execute parameters

You can also add extra keyword arguments to the `execute` method, such as `context`, `root`, `variables`, ...:

```
from graphene.test import Client

def test_hey():
    client = Client(my_schema)
    executed = client.execute('''{ hey }''', context={'user': 'Peter'})
    assert executed == {
        'data': {
            'hey': 'hello Peter!'
        }
    }
```

5.1.4 Snapshot testing

As our APIs evolve, we need to know when our changes introduce any breaking changes that might break some of the clients of our GraphQL app.

However, writing tests and replicate the same response we expect from our GraphQL application can be tedious and repetitive task, and sometimes it's easier to skip this process.

Because of that, we recommend the usage of SnapshotTest.

SnapshotTest let us write all this tests in a breeze, as creates automatically the `snapshots` for us the first time the test is executed.

Here is a simple example on how our tests will look if we use `pytest`:

```
def test_hey(snapshot):
    client = Client(my_schema)
    # This will create a snapshot dir and a snapshot file
    # the first time the test is executed, with the response
    # of the execution.
    snapshot.assert_match(client.execute('''{ hey }'''))
```

If we are using `unittest`:

```
from snapshottest import TestCase
```

```
class APITestCase(TestCase):
    def test_api_me(self):
        """Testing the API for /me"""
        client = Client(my_schema)
        self.assertMatchSnapshot(client.execute('''{ hey }'''))
```

CHAPTER 6

Integrations

- Graphene-Django (source)
- Graphene-SQLAlchemy (source)
- Graphene-GAE (source)
- Graphene-Mongo (source)
- Starlette (source)
- FastAPI (source)