PyQL Documentation

Release 3.0.0

Samuele Santi

Dec 17, 2020

Contents:

PyQL provides a high-level API for defining GraphQL schemas.

It uses graphql-core behind the scenes, which means generated schemas are fully compatible with that library.

What about Graphene?

Graphene serves a similar purpose, but its object-based API is not great for defining large schemas. Also, there are parts that are confusing / error prone (eg. mixing field attributes with field constructor parameters).

Also, we're trying to use Python facilities as much as possible for the schema definition (type annotations in particular), which makes the library friendlier to anyone already familiar with the Python language (no need to invent / learn new concepts).

This of course means the **minimum supported Python version is 3.5**, as type hints were first supported in that release.

CHAPTER 1

Getting started

1.1 Installation

Install from PyPi:

```
pip install pyql
```

1.2 Defining a basic schema

```
from pyql import Schema

schema = Schema()

@schema.query.field('hello')
def resolve_hello(root, info, argument: str = 'stranger') -> str:
    return 'Hello ' + argument
```

Or you can create your root Query object explicitly if you prefer doing so:

```
from pyql import Object, Schema

Query = Object('Query')

@Query.field('hello')
def resolve_hello(root, info, argument: str = 'stranger') -> str:
    return 'Hello ' + argument

schema = Schema(query=Query)
```

1.3 Querying

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

# Passing the argument as a variable
result = schema.execute("""
query hello($arg: String) {
    hello (argument: $arg)
}
""", variables={'arg': 'World'})
print(result.data['hello']) # "Hello World"
```

1.4 Useful links

- PyQL on PyPi
- PyQL source code on GitHub
- PyQL documentation

CHAPTER 2

Schema definition

One of the main goals of PyQL is providing a clean way of defining GraphQL schemas.

In contrast with Graphene, we attempt at providing a cleaner schema definition API, that also reduces ambiguity and verboseness. We attempt to use existing facilities in Python as much as possible (eg. type annotations).

2.1 Schema object

To define a schema, simply create an instance of `pywl.Schema`:

```
from pyql import Object, Schema

schema = Schema(
    query=Query,
    mutation=Mutation,
    subscription=Subscription)
```

You can pass your root query / mutation / subscription objects as arguments to the `Schema` constructor.

Warning: You **must** provide a root query object, and it must have at least one field. This requirement is enforced by `graphql-core`.

2.1.1 Passing extra types

If you are using *interfaces*, chances are you're only using your interface type in the schema definition (so the concrete types are not reachable from the root object).

If that's the case, you must pass them explicitly to the schema constructor:

```
from pyql import Object, Interface, Schema

Character = Interface('Character', ...)
Human = Object('Human', interfaces=[Character], ...)
Droid = Object('Droid', interfaces=[Character], ...)

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

Note: This is likely not going to be necessary in a future version, as we're planning to track concrete objects using a given interface, so we can resolve them automatically behind the scenes.

2.1.2 Compilation

To convert a `pyql.Schema` instance to a schema that's understood by `graphql-core`, you need to compile it:

```
compiled = schema.compile()
```

Now you can use it, eg:

```
from graphql import graphql

result = graphql(compiled, '{yourQuery}', ...)
```

2.1.3 Execution

For convenience, we provide a `schema.execute()` method to quickly run queries against the schema. This is especially useful during testing:

```
schema.execute("""
query foo($arg: String) {
    bar (arg: $arg) {
        baz, quux
    }
}
""", variables={'arg': 'VALUE'})
```

Behind the scenes, it will compile the schema and call `graphql()`.

Return value is an object with `errors` and `data` attributes.

2.2 Objects

Create an instance of `pyql.Object`:

```
Example = Object(
    'Example',
    description='An example object',
    fields={
        'my_str': str,
        'my_int': int,
```

(continues on next page)

(continued from previous page)

```
        'my_float': float,
        'my_bool': bool,
        'my_id': ID,
    })
```

Note: Field names will be converted automatically from `snake_case` to `camelCase` for you, so you can use the right naming convention in your Python / JavaScript code.

2.2.1 Field from resolver

You can define a field quickly by using the `Object.field` decorator. Field type and arguments will be picked up automatically by inspecting type annotations:

```
Example = Object('Example')

@Example.field('hello')
def resolve_hello(root, info, name: str = 'stranger') -> str:
    return 'Hello ' + name
```

Python types will be converted automatically to `GraphQL` types.

If you need to use custom types, simply annotate your resolver accordingly.

2.2.2 Resolver returning object

`Object` instances can be instantiated and treated as normal Python objects.

```
Example = Object('Example', {'foo': str, 'bar': str})

Query = Object('Query')

@Query.field('example')
def resolve_example(root, info) -> Example:
    return Example(foo='A', bar='B')

schema = Schema(query=Query)
```

2.2.3 Default resolver

The default resolver for a field will simply attempt to pick the same-named attribute from the root object.

This way you don't have to define something like this for every simple field you have on your objects:

```
@User.field('name')
def resolve_user_name(root, info) -> str:
    return root.name

@User.field('email')
def resolve_user_email(root, info) -> str:
    return root.email
```

(continues on next page)

(continued from previous page)

```
# ...
```

2.2.4 Namespace fields

Sometimes it's convenient to "namespace" objects. Problem is, field resolution will stop when an object resolver returns `None`, so you need to define your resolvers like this:

```
from pyql import ID, Object

User = Object('User', {'id': ID, 'name': str})

Users = Object('Users')

@Users.field('list')
def resolve_list_users(root, info) -> List[User]:
    pass

@Users.field('search')
def resolve_search_users(root, info, query: str) -> List[User]:
    pass

Query = Object('Query')

@Query.field('users')
def resolve_users(root, info) -> Users:
    # Needs to return something other than None, or the resolvers
    # for list / search will never be called
    return Users()
```

You can replace the `resolve_users` definition with:

```
Query.namespace_field('users', Users)
```

This allows you to run queries like:

```
{
    users {
        list {
            id
            name
        }
    }
}
```

2.2.5 Container types

Objects can be "instantiated" to create objects you can return from your resolvers:

```
MyObject = Object('MyObject', fields={'foo': str, 'bar': str})

@Query.field('example')
def resolve_example(root, info) -> MyObject:
    return MyObject(foo='FOO', bar='BAR')
```

This will also ensure types are understood correctly when using *interfaces*.

2.3 Scalar types

The following scalar types are currently defiened in the GraphQL spec, and supported by PyQL:

- `str` -> `String`
- `int` -> `Int`
- `float` -> `Float`
- `bool` -> `Boolean`
- The `ID` type, defined as `pyql.ID` (there is no equivalent in Python). Will accept strings (or ints) as field value.

2.3.1 Non-nulls

Resolver arguments without a default value will be considered `NonNull` automatically.

You can explicity wrap a type in `pyql.NonNull` for your output types (although it doesn't make too much sense to validate your output fields...).

2.3.2 Extra built-in scalar types

Some more scalar types, not defined by the GraphQL spec, are supported for convenience:

- `datetime.datetime`, in ISO 8601 format
- `datetime.date`, in ISO 8601 format
- `datetime.time`, in ISO 8601 format

2.3.3 Custom scalar types

- TODO: document how to create / register custom GraphQL scalar types
- TODO: also provide an API to register custom scalar types

2.4 Enums

You can use Enums as input / output types as you would with any scalar type.

Start by defining your Enum type:

```
from enum import Enum

class Color(Enum):
    RED = 'red'
    GREEN = 'green'
    BLUE = 'blue'
```

Then, simply use it to annotate your resolvers:

```
@Query.field('random_color')
def resolve_random_color(root, info) -> Color:
    return Color.RED
```

Or for an input type:

```
DESCRIPTIONS = {
    Color.RED: 'Cherry Red',
    Color.GREEN: 'Grass Green',
    Color.BLUE: 'Sky Blue',
}

@Query.field('describe_color')
def resolve_episode(root, info, color: Color) -> str:
    return DESCRIPTIONS[color]
```

2.4.1 Values vs names

Keep in mind that enum *values* will be used externally; member names are for internal use only.

So, for example, the first query will return:

```
{ "randomColor": "red" }
```

Likewise, the second and third query will accept `red` (the enum value) and not `"RED"` as input value.

Valid query:

```
{ describeColor(color: red) }
```

Invalid query:

```
{ describeColor(color: RED) }
```

2.5 Lists

You can use `typing.List` for defining list fields:

```
from typing import List

@Query.field('example_list')
def resolve_example_list(root, info) -> List[str]:
    return ['A', 'B', 'C']
```

In alternative, there's a `pyql.List` class you can use as well.

Note that you need to instantiate it, rather than subscripting:

```
from pyql import List

@Query.field('example_list')
def resolve_example_list(root, info) -> List(str):
    return ['A', 'B', 'C']
```

2.6 Interfaces

To define an interface, instantiate `pyql.Interface`:

```
from pyql import Interface

Character = Interface('Character', fields={
    'id': ID,
    'name': str,
})
```

To define an object using a given interface:

```
from pyql import Object

Human = Object('Human', interfaces=[Character], fields={
    'id': ID,
    'name': str,
    'home_planet': str,
})

Droid = Object('Droid', interfaces=[Character], fields={
    'id': ID,
    'name': str,
    'primary_function': str,
})
```

2.6.1 Automatic type resolution

TL;DR: if your resolver is returning instances of the correct object container, i.e. `Human(...)` or `Droid(...)` in the above example, the correct type will be figured out and everything will work just fine.

GraphQL core requires you to either pass a `resolve_type` function to your `Interface`, or provide a `is_type_of` function on the concrete `Object`.

For convenience, if no `is_type_of` is passed to an `Object`, we'll simply recognise as belonging to that object all instances of the "container" type for the object.

Or in better words:

```
MyObj = Object('MyObj', fields={'foo': str})

obj = MyObj(foo='VALUE')

# When compiling MyObj to a GraphQLObject, is_type_of
# will be defined as:
#
# def is_type_of(value, info):
#     return isinstance(value, MyObj.container_object)
```

2.7 Input objects

Input objects are used to pass structured objects as arguments to queries or mutations.

Create an instance of `pyql.InputObject`:

```
PostInput = InputObject('PostInput', fields={
    'title': NonNull(str),
    'body': str,
})
```

An example mutation making use of the input object:

```
Post = Object('Post', fields={
    'id': ID,
    'title': str,
    'body': str,
})

Mutation = Object('Mutation')

@Mutation.field('create_post')
def resolve_create_post(root, info, post: PostInput) -> Post:

    # The ``post`` argument is an instance of PostInput.
    # Attributes are accessible as expected.

    # Let's pretend we stored the data in our database and want to
    # return information about the newly created post:

    return Post(
        id='1',
        title=post.title,
        body=post.body)

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

A query against the schema might look like this:

```
query = """
mutation createPost($post: PostInput!) {
  createPost(post: $post) {
    id, title, body
  }
}
"""

variables = {'post': {'title': 'Hello', 'body': 'Hello world'}}

schema.execute(query, variables=variables)

assert result.errors is None
assert result.data == {
    'createPost': {
        'id': '1',
        'title': 'Hello',
        'body': 'Hello world',
    }
}
```

2.8 Documenting objects

Documentation loading from objects / resolvers is currently work in progress, but it's going to use Python docstrings as much as possible.

Argument documentation will also be obtained from parsing the resolver docstring.

TODO:

- unions
- documentation

CHAPTER 3

Indices and tables

- genindex
- modindex
- search