gqlmod Documentation

Release 0.4.dev2

Jamie Bliss

Sep 15, 2019

CONTENTS:

gqlmod allows you to import GraphQL Query (`.gql`) files as modules and call the queries and mutations defined there. It will validate your queries at import time, to surface any problems as soon as possible.

gqlmod also defines mechanisms for handling different services (called providers) and different contexts with those services.

CHAPTER

ONE

USING GQLMOD

1.1 Summary

1. Install the `gqlmod` PyPI package, as well as any providers you need
2. Call *`gqlmod.enable_gql_import()`* as soon as possible (maybe in your `__main__.py` or top-level `__init__.py`)
3. Import your query file and start calling your queries.

1.2 Example

Listing 1: queries.gql

```
#~starwars~
query HeroForEpisode($ep: Episode!) {
  hero(episode: $ep) {
    name
    ... on Droid {
      primaryFunction
    }
    ... on Human {
      homePlanet
    }
  }
}
```

Listing 2: app.py

```
import gqlmod
gqlmod.enable_gql_import()

from queries import HeroForEpisode

resp = HeroForEpisode(ep='JEDI')
assert not resp.errors

print(resp.data)
```

1.3 Writing Query Files

Query files are simply text files full of named GraphQL queries and mutations.

One addition is the provider declaration:

```
#~starwars~
```

This tells the system what provider to connect to these queries, and therfore how to actually query the service, what schema to validate against, etc.

The name of the provider should be in the provider's docs.

1.4 Query functions

The generated functions have a specific form.

Query functions only take keyword arguments, matching the variables defined in the query. Optional and arguments with defaults may naturally be omitted.

The function returns a `graphql.ExecutionResult`. It has the following attributes:

- `data`: The result data
- `errors`: A list of errors that occurred, or an empty list if none occurred

Note that wether query functions are synchronous or asynchronous is up to the provider; see its documentation.

1.5 Using different provider contexts

All installed providers are available at startup, initialized with no arguments. For most services, this will allow you to execute queries as an anonymous user. However, most applications will want to authenticate to the service. You can use *`gqlmod.with_provider()`* to provide this data to the provider.

`gqlmod.with_provider()` is a context manager, and may be nested. That is, you can globally authenticate as your app, but also in specific parts authenticate as a user.

The specific arguments will vary by provider, but usually have this basic form:

```
with gqlmod.with_provider('spam-service', token=config['TOKEN']):
    resp = spam_queries.GetMenu(amount_of_spam=None)
```

1.6 Major Providers

Here is a list of some maintained providers:

- `starwars`: Builtin! A demo provider that works on static constant data.
- `cirrus-ci`: From gqlmod-cirrusci, connects to Cirrus CI
- `github`: From gqlmod-github, connects to the GitHub v4 API

You may be able to discover a provider at this places:

- The gqlmod topic on GitHub

- Searching gqlmod on PyPI

1.7 API Reference

gqlmod.**with_provider**(*name*, ***params*)
Uses a new instance of the provider (with the given parameters) for the duration of the context.

gqlmod.**enable_gql_import**()
Enables importing `.gql` files.

CHAPTER

TWO

WRITING A PROVIDER

Writing a provider is fairly staightforward.

1. Define a provider class
2. Add an entry point declaration

2.1 Provider Classes

A provider class is only required to be callable with a specific signature.

```
import graphql


class MyProvider:
    def __init__(self, token=None):
        self.token = token

    def __call__(self, query, variables):
        # Do stuff here

        return graphql.ExecutionResult(
            errors=[],
            data={'spam': 'eggs'}
        )
```

The arguments it takes are:

- `query`: (string) The query to give to the server
- `variables`: (dict) The variables for that query

The provider should return a `graphql.ExecutionResult` as shown above.

2.2 Entry point

In order to be discoverable by gqlmod, providers must define entrypoints. Specifically, in the `graphql_providers` group under the name you want `.gql` files to use. This can take a few different forms, depending on your project. A few examples:

Listing 1: setup.cfg

```
[options.entry_points]
graphql_providers =
    starwars = gqlmod_starwars:StarWarsProvider
```

Listing 2: setup.py

```
setup(
    # ...
    entry_points={
        'graphql_providers': [
            'starwars = gqlmod_starwars:StarWarsProvider'
        ]
    },
    # ...
)
```

Listing 3: pyproject.toml

```
# This is for poetry-based projects
[tool.poetry.plugins.graphql_providers]
"starwars" = "gqlmod_starwars:StarWarsProvider'"
```

2.3 Helpers

In order to help with common cases, gqlmod ships with several helpers

Note that many of them have additional requirements, which are encapsulated in extras.

2.3.1 urllib

Helpers for using urllib to build a provider. You probably want *UrllibJsonProvider*.

Requires the no extras.

class gqlmod.helpers.urllib.**UrllibJsonProvider**
A *UrllibProvider* that uses a JSON-based POST

class gqlmod.helpers.urllib.**UrllibProvider**
Help build an HTTP-based provider based on requests.

You should fill in *endpoint* and possibly override *modify_request()*.

endpoint = None
The URL to send requests to

modify_request(*req*)
Apply policies about the request, primarily authentication.

Accepts a `urllib.request.Request` object.

2.3.2 aiohttp

Helpers for using `aiohttp` to build a provider.

Requires the `aiohttp` extra.

class gqlmod.helpers.aiohttp.**AiohttpProvider**

Help build an HTTP-based provider based on aiohttp.

You should fill in *endpoint* and possibly override *modify_request_args()*.

endpoint = None

The URL to send requests to.

modify_request_args(*kwargs*)

Apply policies about the request, primarily authentication.

timeout = None

Timeout policy to use, if any.

use_json = False

Whether a JSON-based or form-like request should be used.

2.4 Extensions

In addition to the core querying interface, providers may influence the import process in a few different ways. These are all implemented as optional methods on the provider instance.

2.4.1 get_schema_str()

Providers may override the standard schema discovery mechanism by implementing `get_schema_str()`. This is useful for providers that don't have a primary service or don't allow anonymous access at all.

This method must be synchronous. An async variation is not supported.

Default behavior: Issue a GraphQL introspection query via the standard query path.

Parameters: None.

Returns: A `str` of the schema, in standard GraphQL schema language.

2.4.2 codegen_extra_kwargs()

Providers may add keyword arguments (variables) to the query call inside the generated module. These will be passed through the query pipeline back to the provider.

Default behavior: No additional variables are inserted.

Parameters:

- `graphql_ast` (positional, `graphql.language.OperationDefinitionNode`): The AST of the GraphQL query in question
- `schema` (positional, `graphql.type.GraphQLSchema`): The schema of the service

Returns: A `dict` of the names mapping to either simple values or `ast.AST` instances. (Note that the returned AST will be embedded into a right-hand expression context.)

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex

PYTHON MODULE INDEX

g

INDEX