gql 3

Release 3.0.0a6

graphql-python.org

Jun 09, 2021

CONTENTS

Warning: Please note that the following documentation describes the current version which is currently only available as a pre-release and needs to be installed with "*–pre*"

CHAPTER

ONE

CONTENTS

1.1 Introduction

GQL 3 is a GraphQL Client for Python 3.6+ which plays nicely with other graphql implementations compatible with the spec.

Under the hood, it uses GraphQL-core which is a Python port of GraphQL.js, the JavaScript reference implementation for GraphQL.

1.1.1 Installation

You can install GQL 3 and all the extra dependencies using pip:

```
pip install --pre gql[all]
```

Warning: Please note that the following documentation describes the current version which is currently only available as a pre-release and needs to be installed with "*–pre*"

After installation, you can start using GQL by importing from the top-level *gql* package.

Less dependencies

GQL supports multiple *transports* to communicate with the backend. Each transport can each necessitate specific dependencies. If you only need one transport, instead of using the "*all*" extra dependency as described above which installs everything, you might want to install only the dependency needed for your transport.

If for example you only need the *AIOHTTPTransport*, which needs the `aiohttp` dependency, then you can install GQL with:

```
pip install --pre gql[aiohttp]
```

The corresponding between extra dependencies required and the GQL transports is:

Extra dependency	Transports
aiohttp	*AIOHTTPTransport*
websockets	*WebsocketsTransport* *PhoenixChannelWebsocketsTransport*
requests	*RequestsHTTPTransport*

Note: It is also possible to install multiple extra dependencies if needed using commas: `gql[aiohttp, websockets]`

1.1.2 Reporting Issues and Contributing

Please visit the GitHub repository for gql if you're interested in the current development or want to report issues or send pull requests.

We welcome all kinds of contributions if the coding guidelines are respected. Please check the Contributing file to learn how to make a good pull request.

1.2 Usage

1.2.1 Basic usage

In order to execute a GraphQL request against a GraphQL API:

- create your gql *transport* in order to choose the destination url and the protocol used to communicate with it
- create a gql `Client` with the selected transport
- parse a query using `gql`
- execute the query on the client to get the result

```python
from gql import Client, gql
from gql.transport.aiohttp import AIOHTTPTransport

# Select your transport with a defined url endpoint
transport = AIOHTTPTransport(url="https://countries.trevorblades.com/")

# Create a GraphQL client using the defined transport
client = Client(transport=transport, fetch_schema_from_transport=True)

# Provide a GraphQL query
query = gql(
    """
    query getContinents {
      continents {
        code
        name
      }
    }
"""
)

# Execute the query on the transport
result = client.execute(query)
print(result)
```

Warning: Please note that this basic example won't work if you have an asyncio event loop running. In some python environments (as with Jupyter which uses IPython) an asyncio event loop is created for you. In that case you should use instead the *Async Usage example*.

1.2.2 Schema validation

If a GraphQL schema is provided, gql will validate the queries locally before sending them to the backend. If no schema is provided, gql will send the query to the backend without local validation.

You can either provide a schema yourself, or you can request gql to get the schema from the backend using introspection.

Using a provided schema

The schema can be provided as a String (which is usually stored in a .graphql file):

```
with open('path/to/schema.graphql') as f:
    schema_str = f.read()

client = Client(schema=schema_str)
```

OR can be created using python classes:

```
from .someSchema import SampleSchema
# SampleSchema is an instance of GraphQLSchema

client = Client(schema=SampleSchema)
```

See tests/starwars/schema.py for an example of such a schema.

Using introspection

In order to get the schema directly from the GraphQL Server API using the transport, you need to set the *fetch_schema_from_transport* argument of Client to True, and the client will fetch the schema directly after the first connection to the backend.

1.2.3 Subscriptions

Using the *websockets transport*, it is possible to execute GraphQL subscriptions:

```
from gql import gql, Client
from gql.transport.websockets import WebsocketsTransport

transport = WebsocketsTransport(url='wss://your_server/graphql')

client = Client(
    transport=transport,
    fetch_schema_from_transport=True,
)

query = gql('''
```

(continues on next page)

(continued from previous page)

```
    subscription yourSubscription {
        ...
    }
''')

for result in client.subscribe(query):
    print (result)
```

Note: The websockets transport can also execute queries or mutations, it is not restricted to subscriptions

1.2.4 Using variables

It is possible to provide variable values with your query by providing a Dict to the variable_values argument of the *execute* or the *subscribe* methods.

The variable values will be sent alongside the query in the transport message (there is no local substitution).

```
query = gql(
    """
    query getContinentName ($code: ID!) {
      continent (code: $code) {
        name
      }
    }
"""
)

params = {"code": "EU"}

# Get name of continent with code "EU"
result = client.execute(query, variable_values=params)
print(result)

params = {"code": "AF"}

# Get name of continent with code "AF"
result = client.execute(query, variable_values=params)
print(result)
```

1.2.5 HTTP Headers

If you want to add additional http headers for your connection, you can specify these in your transport:

```
transport = AIOHTTPTransport(url='YOUR_URL', headers={'Authorization': 'token'})
```

1.2.6 File uploads

GQL supports file uploads with the *aiohttp transport* using the GraphQL multipart request spec.

Single File

In order to upload a single file, you need to:

- set the file as a variable value in the mutation
- provide the opened file to the *variable_values* argument of *execute*
- set the *upload_files* argument to True

```
transport = AIOHTTPTransport(url='YOUR_URL')

client = Client(transport=transport)

query = gql('''
  mutation($file: Upload!) {
    singleUpload(file: $file) {
      id
    }
  }
''')

with open("YOUR_FILE_PATH", "rb") as f:

    params = {"file": f}

    result = client.execute(
        query, variable_values=params, upload_files=True
    )
```

File list

It is also possible to upload multiple files using a list.

```
transport = AIOHTTPTransport(url='YOUR_URL')

client = Client(transport=transport)

query = gql('''
  mutation($files: [Upload!]!) {
    multipleUpload(files: $files) {
      id
    }
  }
''')

f1 = open("YOUR_FILE_PATH_1", "rb")
f2 = open("YOUR_FILE_PATH_1", "rb")

params = {"files": [f1, f2]}

result = client.execute(
```

(continues on next page)

(continued from previous page)

```
    query, variable_values=params, upload_files=True
)

f1.close()
f2.close()
```

Streaming

If you use the above methods to send files, then the entire contents of the files must be loaded in memory before the files are sent. If the files are not too big and you have enough RAM, it is not a problem. On another hand if you want to avoid using too much memory, then it is better to read the files and send them in small chunks so that the entire file contents don’t have to be in memory at once.

We provide methods to do that for two different uses cases:

- Sending local files
- Streaming downloaded files from an external URL to the GraphQL API

Streaming local files

aiohttp allows to upload files using an asynchronous generator. See Streaming uploads on aiohttp docs.

In order to stream local files, instead of providing opened files to the *variables_values* argument of *execute*, you need to provide an async generator which will provide parts of the files.

You can use aiofiles to read the files in chunks and create this asynchronous generator.

Example:

```
transport = AIOHTTPTransport(url='YOUR_URL')

client = Client(transport=transport)

query = gql('''
  mutation($file: Upload!) {
    singleUpload(file: $file) {
      id
    }
  }
''')

async def file_sender(file_name):
    async with aiofiles.open(file_name, 'rb') as f:
        chunk = await f.read(64*1024)
            while chunk:
                yield chunk
                chunk = await f.read(64*1024)

params = {"file": file_sender(file_name='YOUR_FILE_PATH')}

result = client.execute(
            query, variable_values=params, upload_files=True
    )
```

Streaming downloaded files

If the file you want to upload to the GraphQL API is not present locally and needs to be downloaded from elsewhere, then it is possible to chain the download and the upload in order to limit the amout of memory used.

Because the *content* attribute of an aiohttp response is a *StreamReader* (it provides an async iterator protocol), you can chain the download and the upload together.

In order to do that, you need to:

- get the response from an aiohttp request and then get the StreamReader instance from *resp.content*
- provide the StreamReader instance to the *variable_values* argument of *execute*

Example:

```
# First request to download your file with aiohttp
async with aiohttp.ClientSession() as http_client:
    async with http_client.get('YOUR_DOWNLOAD_URL') as resp:

        # We now have a StreamReader instance in resp.content
        # and we provide it to the variable_values argument of execute

        transport = AIOHTTPTransport(url='YOUR_GRAPHQL_URL')

        client = Client(transport=transport)

        query = gql('''
          mutation($file: Upload!) {
            singleUpload(file: $file) {
              id
            }
          }
        ''')

        params = {"file": resp.content}

        result = client.execute(
            query, variable_values=params, upload_files=True
        )
```

1.3 Async vs Sync

On previous versions of GQL, the code was *sync* only , it means that when you ran *execute* on the Client, you could do nothing else in the current Thread and had to wait for an answer or a timeout from the backend to continue. The only http library was *requests*, allowing only sync usage.

From the version 3 of GQL, we support *sync* and *async* *transports* using asyncio.

With the *async transports*, there is now the possibility to execute GraphQL requests asynchronously, *allowing to execute multiple requests in parallel if needed*.

If you don't care or need async functionality, it is still possible, with *async transports*, to run the *execute* or *subscribe* methods directly from the Client (as described in the *Basic Usage* example) and GQL will execute the request in a synchronous manner by running an asyncio event loop itself.

This won't work though if you already have an asyncio event loop running. In that case you should use *Async Usage*

1.3.1 Async Usage

If you use an *async transport*, you can use GQL asynchronously using asyncio.

- put your code in an asyncio coroutine (method starting with `async def`)
- use `async with client as session:` to connect to the backend and provide a session instance
- use the `await` keyword to execute requests: `await session.execute(...)`
- then run your coroutine in an asyncio event loop by running `asyncio.run`

Example:

```
import asyncio

from gql import Client, gql
from gql.transport.aiohttp import AIOHTTPTransport


async def main():

    transport = AIOHTTPTransport(url="https://countries.trevorblades.com/graphql")

    # Using `async with` on the client will start a connection on the transport
    # and provide a `session` variable to execute queries on this connection
    async with Client(
        transport=transport, fetch_schema_from_transport=True,
    ) as session:

        # Execute single query
        query = gql(
            """
            query getContinents {
              continents {
                code
                name
              }
            }
        """
        )

        result = await session.execute(query)
        print(result)


asyncio.run(main())
```

IPython

Warning: On some Python environments, like *Jupyter* or *Spyder*, which are using *IPython*, an asyncio event loop is already created for you by the environment.

In this case, running the above code might generate the following error:

```
RuntimeError: asyncio.run() cannot be called from a running event loop
```

If that happens, depending on the environment, you should replace `asyncio.run(main())` by either:

```
await main()
```

OR:

```
loop = asyncio.get_running_loop()
loop.create_task(main())
```

1.4 Transports

GQL Transports are used to define how the connection is made with the backend. We have different transports for different underlying protocols (http, websockets, . . .)

1.4.1 Async Transports

Async transports are transports which are using an underlying async library. They allow us to *run GraphQL queries asynchronously*

AIOHTTPTransport

This transport uses the aiohttp library and allows you to send GraphQL queries using the HTTP protocol.

Note: GraphQL subscriptions are not supported on the HTTP transport. For subscriptions you should use the *websockets transport*.

```
import asyncio

from gql import Client, gql
from gql.transport.aiohttp import AIOHTTPTransport


async def main():

    transport = AIOHTTPTransport(url="https://countries.trevorblades.com/graphql")

    # Using `async with` on the client will start a connection on the transport
    # and provide a `session` variable to execute queries on this connection
    async with Client(
        transport=transport, fetch_schema_from_transport=True,
    ) as session:

        # Execute single query
        query = gql(
            """
            query getContinents {
              continents {
                code
```

(continues on next page)

(continued from previous page)

```
                name
            }
        }
    """
    )

    result = await session.execute(query)
    print(result)


asyncio.run(main())
```

Authentication

There are multiple ways to authenticate depending on the server configuration.

1. Using HTTP Headers

```
transport = AIOHTTPTransport(
    url='https://SERVER_URL:SERVER_PORT/graphql',
    headers={'Authorization': 'token'}
)
```

2. Using HTTP Cookies

You can manually set the cookies which will be sent with each connection:

```
transport = AIOHTTPTransport(url=url, cookies={"cookie1": "val1"})
```

Or you can use a cookie jar to save cookies set from the backend and reuse them later.

In some cases, the server will set some connection cookies after a successful login mutation and you can save these cookies in a cookie jar to reuse them in a following connection (See issue 197):

```
jar = aiohttp.CookieJar()
transport = AIOHTTPTransport(url=url, client_session_args={'cookie_jar': jar})
```

WebsocketsTransport

The websockets transport implements the Apollo websockets transport protocol.

This transport allows to do multiple queries, mutations and subscriptions on the same websocket connection.

```
import asyncio
import logging

from gql import Client, gql
from gql.transport.websockets import WebsocketsTransport

logging.basicConfig(level=logging.INFO)


async def main():
```

(continues on next page)

(continued from previous page)

```
    transport = WebsocketsTransport(url="wss://countries.trevorblades.com/graphql")

    # Using `async with` on the client will start a connection on the transport
    # and provide a `session` variable to execute queries on this connection
    async with Client(
        transport=transport, fetch_schema_from_transport=True,
    ) as session:

        # Execute single query
        query = gql(
            """
            query getContinents {
              continents {
                code
                name
              }
            }
        """
        )
        result = await session.execute(query)
        print(result)

        # Request subscription
        subscription = gql(
            """
            subscription {
                somethingChanged {
                    id
                }
            }
        """
        )
        async for result in session.subscribe(subscription):
            print(result)


asyncio.run(main())
```

Websockets SSL

If you need to connect to an ssl encrypted endpoint:

- use _wss_ instead of _ws_ in the url of the transport

```
sample_transport = WebsocketsTransport(
    url='wss://SERVER_URL:SERVER_PORT/graphql',
    headers={'Authorization': 'token'}
)
```

If you have a self-signed ssl certificate, you need to provide an ssl_context with the server public certificate:

```
import pathlib
import ssl

ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
```

(continues on next page)

(continued from previous page)

```
localhost_pem = pathlib.Path(__file__).with_name("YOUR_SERVER_PUBLIC_CERTIFICATE.pem")
ssl_context.load_verify_locations(localhost_pem)

sample_transport = WebsocketsTransport(
    url='wss://SERVER_URL:SERVER_PORT/graphql',
    ssl=ssl_context
)
```

If you have also need to have a client ssl certificate, add:

```
ssl_context.load_cert_chain(certfile='YOUR_CLIENT_CERTIFICATE.pem', keyfile='YOUR_
↪CLIENT_CERTIFICATE_KEY.key')
```

Websockets authentication

There are two ways to send authentication tokens with websockets depending on the server configuration.

1. Using HTTP Headers

```
sample_transport = WebsocketsTransport(
    url='wss://SERVER_URL:SERVER_PORT/graphql',
    headers={'Authorization': 'token'}
)
```

2. With a payload in the connection_init websocket message

```
sample_transport = WebsocketsTransport(
    url='wss://SERVER_URL:SERVER_PORT/graphql',
    init_payload={'Authorization': 'token'}
)
```

PhoenixChannelWebsocketsTransport

The PhoenixChannelWebsocketsTransport is an **EXPERIMENTAL** async transport which allows you to execute queries and subscriptions against an Absinthe backend using the Phoenix framework channels.

1.4.2 Sync Transports

Sync transports are transports which are using an underlying sync library. They cannot be used asynchronously.

RequestsHTTPTransport

The RequestsHTTPTransport is a sync transport using the requests library and allows you to send GraphQL queries using the HTTP protocol.

```
from gql import Client, gql
from gql.transport.requests import RequestsHTTPTransport

transport = RequestsHTTPTransport(
    url="https://countries.trevorblades.com/", verify=True, retries=3,
)
```

(continues on next page)

(continued from previous page)

```
client = Client(transport=transport, fetch_schema_from_transport=True)

query = gql(
    """
    query getContinents {
      continents {
        code
        name
      }
    }
"""
)

result = client.execute(query)
print(result)
```

1.5 Advanced

1.5.1 Async advanced usage

It is possible to send multiple GraphQL queries (query, mutation or subscription) in parallel, on the same websocket connection, using asyncio tasks.

In order to retry in case of connection failure, we can use the great backoff module.

```
# First define all your queries using a session argument:

async def execute_query1(session):
    result = await session.execute(query1)
    print(result)

async def execute_query2(session):
    result = await session.execute(query2)
    print(result)

async def execute_subscription1(session):
    async for result in session.subscribe(subscription1):
        print(result)

async def execute_subscription2(session):
    async for result in session.subscribe(subscription2):
        print(result)

# Then create a couroutine which will connect to your API and run all your queries as␣
↪tasks.
# We use a `backoff` decorator to reconnect using exponential backoff in case of␣
↪connection failure.

@backoff.on_exception(backoff.expo, Exception, max_time=300)
async def graphql_connection():

    transport = WebsocketsTransport(url="wss://YOUR_URL")
```

(continues on next page)

(continued from previous page)

```
    client = Client(transport=transport, fetch_schema_from_transport=True)

    async with client as session:
        task1 = asyncio.create_task(execute_query1(session))
        task2 = asyncio.create_task(execute_query2(session))
        task3 = asyncio.create_task(execute_subscription1(session))
        task4 = asyncio.create_task(execute_subscription2(session))

        await asyncio.gather(task1, task2, task3, task4)

asyncio.run(graphql_connection())
```

Subscriptions tasks can be stopped at any time by running

```
task.cancel()
```

1.5.2 Logging

GQL use the python logging module.

In order to debug a problem, you can enable logging to see the messages exchanged between the client and the server. To do that, set the loglevel at **INFO** at the beginning of your code:

```
import logging
logging.basicConfig(level=logging.INFO)
```

For even more logs, you can set the loglevel at **DEBUG**:

```
import logging
logging.basicConfig(level=logging.DEBUG)
```

Disabling logs

By default, the logs for the transports are quite verbose.

On the **INFO** level, all the messages between the frontend and the backend are logged which can be difficult to read especially when it fetches the schema from the transport.

It is possible to disable the logs only for a specific gql transport by setting a higher log level for this transport (**WARNING** for example) so that the other logs of your program are not affected.

For this, you should import the logger from the transport file and set the level on this logger.

For the RequestsHTTPTransport:

```
from gql.transport.requests import log as requests_logger
requests_logger.setLevel(logging.WARNING)
```

For the WebsocketsTransport:

```
from gql.transport.websockets import log as websockets_logger
websockets_logger.setLevel(logging.WARNING)
```

1.5.3 Execution on a local schema

It is also possible to execute queries against a local schema (so without a transport), even if it is not really useful except maybe for testing.

```
from gql import gql, Client

from .someSchema import SampleSchema

client = Client(schema=SampleSchema)

query = gql('''
    {
      hello
    }
''')

result = client.execute(query)
```

See tests/starwars/test_query.py for an example

1.5.4 Compose queries dynamically

Instead of providing the GraphQL queries as a Python String, it is also possible to create GraphQL queries dynamically. Using the *DSL module*, we can create a query using a Domain Specific Language which is created from the schema.

The following code:

```
ds = DSLSchema(StarWarsSchema)

query = dsl_gql(
    DSLQuery(
        ds.Query.hero.select(
            ds.Character.id,
            ds.Character.name,
            ds.Character.friends.select(ds.Character.name),
        )
    )
)
```

will generate a query equivalent to:

```
query = gql("""
    query {
      hero {
        id
        name
        friends {
          name
        }
      }
    }
""")
```

How to use

First generate the root using the `DSLSchema`:

```
ds = DSLSchema(client.schema)
```

Then use auto-generated attributes of the `ds` instance to get a root type (Query, Mutation or Subscription). This will generate a `DSLType` instance:

```
ds.Query
```

From this root type, you use auto-generated attributes to get a field. This will generate a `DSLField` instance:

```
ds.Query.hero
```

hero is a GraphQL object type and needs children fields. By default, there is no children fields selected. To select the fields that you want in your query, you use the `select` method.

To generate the children fields, we use the same method as above to auto-generate the fields from the `ds` instance (ie `ds.Character.name` is the field *name* of the type *Character*):

```
ds.Query.hero.select(ds.Character.name)
```

The select method return the same instance, so it is possible to chain the calls:

```
ds.Query.hero.select(ds.Character.name).select(ds.Character.id)
```

Or do it sequencially:

```
hero_query = ds.Query.hero

hero_query.select(ds.Character.name)
hero_query.select(ds.Character.id)
```

As you can select children fields of any object type, you can construct your complete query tree:

```
ds.Query.hero.select(
    ds.Character.id,
    ds.Character.name,
    ds.Character.friends.select(ds.Character.name),
)
```

Once your root query fields are defined, you can put them in an operation using `DSLQuery`, `DSLMutation` or `DSLSubscription`:

```
DSLQuery(
    ds.Query.hero.select(
        ds.Character.id,
        ds.Character.name,
        ds.Character.friends.select(ds.Character.name),
    )
)
```

Once your operations are defined, use the `dsl_gql` function to convert your operations into a document which will be able to get executed in the client or a session:

```
query = dsl_gql(
    DSLQuery(
        ds.Query.hero.select(
            ds.Character.id,
            ds.Character.name,
            ds.Character.friends.select(ds.Character.name),
        )
    )
)

result = client.execute(query)
```

Arguments

It is possible to add arguments to any field simply by calling it with the required arguments:

```
ds.Query.human(id="1000").select(ds.Human.name)
```

It can also be done using the *args* method:

```
ds.Query.human.args(id="1000").select(ds.Human.name)
```

Aliases

You can set an alias of a field using the *alias* method:

```
ds.Query.human.args(id=1000).alias("luke").select(ds.Character.name)
```

It is also possible to set the alias directly using keyword arguments of an operation:

```
DSLQuery(
    luke=ds.Query.human.args(id=1000).select(ds.Character.name)
)
```

Or using keyword arguments in the *select* method:

```
ds.Query.hero.select(
    my_name=ds.Character.name
)
```

Mutations

For the mutations, you need to start from root fields starting from `ds.Mutation` then you need to create the GraphQL operation using the class *DSLMutation*. Example:

```
query = dsl_gql(
    DSLMutation(
        ds.Mutation.createReview.args(
            episode=6, review={"stars": 5, "commentary": "This is a great movie!"}
        ).select(ds.Review.stars, ds.Review.commentary)
    )
)
```

Variable arguments

To provide variables instead of argument values directly for an operation, you have to:

- Instanciate a *DSLVariableDefinitions*:

```
var = DSLVariableDefinitions()
```

- From this instance you can generate *DSLVariable* instances and provide them as the value of the arguments:

```
ds.Mutation.createReview.args(review=var.review, episode=var.episode)
```

- Once the operation has been defined, you have to save the variable definitions used in it:

```
operation.variable_definitions = var
```

The following code:

```
var = DSLVariableDefinitions()
op = DSLMutation(
    ds.Mutation.createReview.args(review=var.review, episode=var.episode).select(
        ds.Review.stars, ds.Review.commentary
    )
)
op.variable_definitions = var
query = dsl_gql(op)
```

will generate a query equivalent to:

```
mutation ($review: ReviewInput, $episode: Episode) {
  createReview(review: $review, episode: $episode) {
    stars
    commentary
  }
}
```

Subscriptions

For the subscriptions, you need to start from root fields starting from `ds.Subscription` then you need to create the GraphQL operation using the class *DSLSubscription*. Example:

```
query = dsl_gql(
    DSLSubscription(
        ds.Subscription.reviewAdded(episode=6).select(ds.Review.stars, ds.Review.
↪commentary)
    )
)
```

Multiple fields in an operation

It is possible to create an operation with multiple fields:

```
DSLQuery(
    ds.Query.hero.select(ds.Character.name),
    hero_of_episode_5=ds.Query.hero(episode=5).select(ds.Character.name),
)
```

Operation name

You can set the operation name of an operation using a keyword argument to `dsl_gql`:

```
query = dsl_gql(
    GetHeroName=DSLQuery(ds.Query.hero.select(ds.Character.name))
)
```

will generate the request:

```
query GetHeroName {
    hero {
        name
    }
}
```

Multiple operations in a document

It is possible to create an Document with multiple operations:

```
query = dsl_gql(
    operation_name_1=DSLQuery( ... ),
    operation_name_2=DSLQuery( ... ),
    operation_name_3=DSLMutation( ... ),
)
```

Executable examples

Async example

```
import asyncio

from gql import Client
from gql.dsl import DSLQuery, DSLSchema, dsl_gql
from gql.transport.aiohttp import AIOHTTPTransport


async def main():

    transport = AIOHTTPTransport(url="https://countries.trevorblades.com/graphql")

    client = Client(transport=transport, fetch_schema_from_transport=True)
```

(continues on next page)

(continued from previous page)

```
    # Using `async with` on the client will start a connection on the transport
    # and provide a `session` variable to execute queries on this connection.
    # Because we requested to fetch the schema from the transport,
    # GQL will fetch the schema just after the establishment of the first session
    async with client as session:

        # Instanciate the root of the DSL Schema as ds
        ds = DSLSchema(client.schema)

        # Create the query using dynamically generated attributes from ds
        query = dsl_gql(
            DSLQuery(
                ds.Query.continents(filter={"code": {"eq": "EU"}}).select(
                    ds.Continent.code, ds.Continent.name
                )
            )
        )

        result = await session.execute(query)
        print(result)

        # This can also be written as:

        # I want to query the continents
        query_continents = ds.Query.continents

        # I want to get only the continents with code equal to "EU"
        query_continents(filter={"code": {"eq": "EU"}})

        # I want this query to return the code and name fields
        query_continents.select(ds.Continent.code)
        query_continents.select(ds.Continent.name)

        # I generate a document from my query to be able to execute it
        query = dsl_gql(DSLQuery(query_continents))

        # Execute the query
        result = await session.execute(query)
        print(result)


asyncio.run(main())
```

Sync example

```
from gql import Client
from gql.dsl import DSLQuery, DSLSchema, dsl_gql
from gql.transport.requests import RequestsHTTPTransport

transport = RequestsHTTPTransport(
    url="https://countries.trevorblades.com/", verify=True, retries=3,
)
```

(continues on next page)

(continued from previous page)

```
client = Client(transport=transport, fetch_schema_from_transport=True)

# Using `with` on the sync client will start a connection on the transport
# and provide a `session` variable to execute queries on this connection.
# Because we requested to fetch the schema from the transport,
# GQL will fetch the schema just after the establishment of the first session
with client as session:

    # We should have received the schema now that the session is established
    assert client.schema is not None

    # Instanciate the root of the DSL Schema as ds
    ds = DSLSchema(client.schema)

    # Create the query using dynamically generated attributes from ds
    query = dsl_gql(
        DSLQuery(ds.Query.continents.select(ds.Continent.code, ds.Continent.name))
    )

    result = session.execute(query)
    print(result)
```

1.6 gql-cli

GQL provides a python 3.6+ script, called *gql-cli* which allows you to execute GraphQL queries directly from the terminal.

This script supports http(s) or websockets protocols.

1.6.1 Usage

Send GraphQL queries from the command line using http(s) or websockets. If used interactively, write your query, then use Ctrl-D (EOF) to execute it.

```
usage: gql-cli [-h] [-V [VARIABLES [VARIABLES ...]]]
               [-H [HEADERS [HEADERS ...]]] [--version] [-d | -v]
               [-o OPERATION_NAME]
               server
```

Positional Arguments

server	the server url starting with http://, https://, ws:// or wss://

Named Arguments

-V, --variables	query variables in the form key:json_value
-H, --headers	http headers in the form key:value
--version	show program's version number and exit
-d, --debug	print lots of debugging statements (loglevel==DEBUG)
-v, --verbose	show low level messages (loglevel==INFO)
-o, --operation-name	set the operation_name value

1.6.2 Examples

Simple query using https

```
$ echo 'query { continent(code:"AF") { name } }' | gql-cli https://countries.
↪trevorblades.com
{"continent": {"name": "Africa"}}
```

Simple query using websockets

```
$ echo 'query { continent(code:"AF") { name } }' | gql-cli wss://countries.
↪trevorblades.com/graphql
{"continent": {"name": "Africa"}}
```

Query with variable

```
$ echo 'query getContinent($code:ID!) { continent(code:$code) { name } }' | gql-cli␣
↪https://countries.trevorblades.com --variables code:AF
{"continent": {"name": "Africa"}}
```

Interactive usage

Insert your query in the terminal, then press Ctrl-D to execute it.

```
$ gql-cli wss://countries.trevorblades.com/graphql --variables code:AF
```

Execute query saved in a file

Put the query in a file:

```
$ echo 'query {
  continent(code:"AF") {
    name
  }
}' > query.gql
```

Then execute query from the file:

```
$ cat query.gql | gql-cli wss://countries.trevorblades.com/graphql
{"continent": {"name": "Africa"}}
```

1.7 Reference

1.7.1 Top-Level Functions

The primary *gql* package includes everything you need to execute GraphQL requests, with the exception of the transports which are optional:

- the *gql* method to parse a GraphQL query
- the *Client* class as the entrypoint to execute requests and create sessions

class gql.**Client**(*schema: Optional[Union[str, graphql.type.schema.GraphQLSchema]] = None, introspection=None, type_def: Optional[str] = None, transport: Optional[Union[*gql.transport.transport.Transport*, *gql.transport.async_transport.AsyncTransport*]] = None, fetch_schema_from_transport: bool = False, execute_timeout: Optional[int] = 10*)

Bases: `object`

The Client class is the main entrypoint to execute GraphQL requests on a GQL transport.

It can take sync or async transports as argument and can either execute and subscribe to requests itself with the *execute* and *subscribe* methods OR can be used to get a sync or async session depending on the transport type.

To connect to an *async transport* and get an *async session*, use `async with client as session:`

To connect to a *sync transport* and get a *sync session*, use `with client as session:`

__init__(*schema: Optional[Union[str, graphql.type.schema.GraphQLSchema]] = None, introspection=None, type_def: Optional[str] = None, transport: Optional[Union[*gql.transport.transport.Transport*, *gql.transport.async_transport.AsyncTransport*]] = None, fetch_schema_from_transport: bool = False, execute_timeout: Optional[int] = 10*)

Initialize the client with the given parameters.

Parameters

- **schema** – an optional GraphQL Schema for local validation See *Schema validation*
- **transport** – The provided *transport*.
- **fetch_schema_from_transport** – Boolean to indicate that if we want to fetch the schema from the transport using an introspection query
- **execute_timeout** – The maximum time in seconds for the execution of a request before a TimeoutError is raised. Only used for async transports.

execute(*document: graphql.language.ast.DocumentNode, *args, **kwargs*) → Dict

Execute the provided document AST against the remote server using the transport provided during init.

This function **WILL BLOCK** until the result is received from the server.

Either the transport is sync and we execute the query synchronously directly OR the transport is async and we execute the query in the asyncio loop (blocking here until answer).

This method will:

- connect using the transport to get a session
- execute the GraphQL request on the transport session
- close the session and close the connection to the server

If you have multiple requests to send, it is better to get your own session and execute the requests in your session.

The extra arguments passed in the method will be passed to the transport execute method.

subscribe(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → Generator[Dict, None, None]

Execute a GraphQL subscription with a python generator.

We need an async transport for this functionality.

gql.**gql**(*request_string: str*) → graphql.language.ast.DocumentNode

Given a String containing a GraphQL request, parse it into a Document.

Parameters **request_string** (*str*) – the GraphQL request as a String

Returns a Document which can be later executed or subscribed by a *Client*, by an *async session* or by a *sync session*

Raises **GraphQLError** – if a syntax error is encountered.

1.7.2 Sub-Packages

gql.client

class gql.client.**AsyncClientSession**(*client:* gql.client.Client)

Bases: object

An instance of this class is created when using async with on a *client*.

It contains the async methods (execute, subscribe) to send queries on an async transport using the same session.

__init__(*client:* gql.client.Client)

Parameters **client** – the *client* used

async execute(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → Dict

Coroutine to execute the provided document AST asynchronously using the async transport.

The extra arguments are passed to the transport execute method.

async fetch_schema() → None

Fetch the GraphQL schema explicitely using introspection.

Don't use this function and instead set the fetch_schema_from_transport attribute to True

subscribe(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → AsyncGenerator[Dict, None]

Coroutine to subscribe asynchronously to the provided document AST asynchronously using the async transport.

The extra arguments are passed to the transport subscribe method.

property transport

class gql.client.**Client**(*schema: Optional[Union[str, graphql.type.schema.GraphQLSchema]] = None*, *introspection=None*, *type_def: Optional[str] = None*, *transport: Optional[Union[gql.transport.transport.Transport, gql.transport.async_transport.AsyncTransport]] = None*, *fetch_schema_from_transport: bool = False*, *execute_timeout: Optional[int] = 10*)

Bases: `object`

The Client class is the main entrypoint to execute GraphQL requests on a GQL transport.

It can take sync or async transports as argument and can either execute and subscribe to requests itself with the `execute` and `subscribe` methods OR can be used to get a sync or async session depending on the transport type.

To connect to an *async transport* and get an `async session`, use `async with client as session:`

To connect to a *sync transport* and get a `sync session`, use `with client as session:`

__init__(*schema: Optional[Union[str, graphql.type.schema.GraphQLSchema]] = None*, *introspection=None*, *type_def: Optional[str] = None*, *transport: Optional[Union[gql.transport.transport.Transport, gql.transport.async_transport.AsyncTransport]] = None*, *fetch_schema_from_transport: bool = False*, *execute_timeout: Optional[int] = 10*)

Initialize the client with the given parameters.

Parameters

- **schema** – an optional GraphQL Schema for local validation See *Schema validation*
- **transport** – The provided *transport*.
- **fetch_schema_from_transport** – Boolean to indicate that if we want to fetch the schema from the transport using an introspection query
- **execute_timeout** – The maximum time in seconds for the execution of a request before a TimeoutError is raised. Only used for async transports.

execute(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → Dict

Execute the provided document AST against the remote server using the transport provided during init.

This function **WILL BLOCK** until the result is received from the server.

Either the transport is sync and we execute the query synchronously directly OR the transport is async and we execute the query in the asyncio loop (blocking here until answer).

This method will:

- connect using the transport to get a session
- execute the GraphQL request on the transport session
- close the session and close the connection to the server

If you have multiple requests to send, it is better to get your own session and execute the requests in your session.

The extra arguments passed in the method will be passed to the transport execute method.

subscribe(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → Generator[Dict, None, None]

Execute a GraphQL subscription with a python generator.

We need an async transport for this functionality.

class gql.client.**SyncClientSession**(*client:* gql.client.Client)

Bases: object

An instance of this class is created when using with on the client.

It contains the sync method execute to send queries on a sync transport using the same session.

__init__(*client:* gql.client.Client)

Parameters client – the *client* used

execute(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → Dict

fetch_schema() → None

Fetch the GraphQL schema explicitely using introspection.

Don't use this function and instead set the fetch_schema_from_transport attribute to True

property transport

gql.transport

class gql.transport.transport.**Transport**

Bases: object

close()

Close the transport

This method doesn't have to be implemented unless the transport would benefit from it. This is currently used by the RequestsHTTPTransport transport to close the session's connection pool.

connect()

Establish a session with the transport.

abstract execute(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → graphql.execution.execute.ExecutionResult

Execute GraphQL query.

Execute the provided document AST for either a remote or local GraphQL Schema.

Parameters document – GraphQL query as AST Node or Document object.

Returns ExecutionResult

class gql.transport.local_schema.**LocalSchemaTransport**(*schema: graphql.type.schema.GraphQLSchema*)

Bases: *gql.transport.async_transport.AsyncTransport*

A transport for executing GraphQL queries against a local schema.

__init__(*schema: graphql.type.schema.GraphQLSchema*)

Initialize the transport with the given local schema.

Parameters schema – Local schema as GraphQLSchema object

async close()

No close needed on local transport

async connect()

No connection needed on local transport

async execute(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → graphql.execution.execute.ExecutionResult

Execute the provided document AST for on a local GraphQL Schema.

subscribe(*document: graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → AsyncGenerator[graphql.execution.execute.ExecutionResult, None]

Send a subscription and receive the results using an async generator

The results are sent as an ExecutionResult object

class gql.transport.requests.**RequestsHTTPTransport**(*url: str*, *headers: Optional[Dict[str, Any]] = None*, *cookies: Optional[Union[Dict[str, Any], requests.cookies.RequestsCookieJar]] = None*, *auth: Optional[requests.auth.AuthBase] = None*, *use_json: bool = True*, *timeout: Optional[int] = None*, *verify: bool = True*, *retries: int = 0*, *method: str = 'POST'*, ***kwargs: Any*)

Bases: *gql.transport.transport.Transport*

Sync Transport used to execute GraphQL queries on remote servers.

The transport uses the requests library to send HTTP POST requests.

__init__(*url: str*, *headers: Optional[Dict[str, Any]] = None*, *cookies: Optional[Union[Dict[str, Any], requests.cookies.RequestsCookieJar]] = None*, *auth: Optional[requests.auth.AuthBase] = None*, *use_json: bool = True*, *timeout: Optional[int] = None*, *verify: bool = True*, *retries: int = 0*, *method: str = 'POST'*, ***kwargs: Any*)

Initialize the transport with the given request parameters.

Parameters

- **url** – The GraphQL server URL.
- **headers** – Dictionary of HTTP Headers to send with the `Request` (Default: None).
- **cookies** – Dict or CookieJar object to send with the `Request` (Default: None).
- **auth** – Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth (Default: None).
- **use_json** – Send request body as JSON instead of form-urlencoded (Default: True).
- **timeout** – Specifies a default timeout for requests (Default: None).
- **verify** – Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use. (Default: True).
- **retries** – Pre-setup of the requests' Session for performing retries
- **method** – HTTP method used for requests. (Default: POST).
- **kwargs** – Optional arguments that `request` takes. These can be seen at the requests source code or the official docs

close()

Closing the transport by closing the inner session

connect()

Establish a session with the transport.

execute(*document: graphql.language.ast.DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *timeout: Optional[int] = None*) → graphql.execution.execute.ExecutionResult

Execute GraphQL query.

Execute the provided document AST against the configured remote server. This uses the requests library to perform a HTTP POST request to the remote server.

Parameters

- **document** – GraphQL query as AST Node object.
- **variable_values** – Dictionary of input parameters (Default: None).
- **operation_name** – Name of the operation that shall be executed. Only required in multi-operation documents (Default: None).
- **timeout** – Specifies a default timeout for requests (Default: None).

Returns The result of execution. *data* is the result of executing the query, *errors* is null if no errors occurred, and is a non-empty array if an error occurred.

class gql.transport.async_transport.**AsyncTransport**

Bases: object

abstract async close()

Coroutine used to Close an established connection

abstract async connect()

Coroutine used to create a connection to the specified address

abstract async execute(*document: graphql.language.ast.DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → graphql.execution.execute.ExecutionResult

Execute the provided document AST for either a remote or local GraphQL Schema.

abstract subscribe(*document: graphql.language.ast.DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → AsyncGenerator[graphql.execution.execute.ExecutionResult, None]

Send a query and receive the results using an async generator

The query can be a graphql query, mutation or subscription

The results are sent as an ExecutionResult object

gql.dsl

gql.dsl.**ast_from_value**(*value: Any*, *type_: Union[graphql.type.definition.GraphQLScalarType, graphql.type.definition.GraphQLEnumType, graphql.type.definition.GraphQLInputObjectType, graphql.type.definition.GraphQLWrappingType]*) → Optional[graphql.language.ast.ValueNode]

This is a partial copy paste of the ast_from_value function in graphql-core utilities/ast_from_value.py

Overwrite the if blocks that use recursion and add a new case to return a VariableNode when value is a DSLVariable

Produce a GraphQL Value AST given a Python object.

gql.dsl.**dsl_gql**(**operations: gql.dsl.DSLOperation*, ***operations_with_name: gql.dsl.DSLOperation*) → graphql.language.ast.DocumentNode

Given arguments instances of *DSLOperation* containing GraphQL operations, generate a Document which can be executed later in a gql client or a gql session.

Similar to the `gql.gql()` function but instead of parsing a python string to describe the request, we are using operations which have been generated dynamically using instances of `DSLField`, generated by instances of `DSLType` which themselves originated from a `DSLSchema` class.

Parameters

- **`*operations`** (`DSLOperation` (`DSLQuery`, `DSLMutation`, `DSLSubscription`)) – the GraphQL operations
- **`**operations_with_name`** (`DSLOperation` (`DSLQuery`, `DSLMutation`, `DSLSubscription`)) – the GraphQL operations with an operation name

Returns a Document which can be later executed or subscribed by a `Client`, by an `async session` or by a `sync session`

Raises **`TypeError`** – if an argument is not an instance of `DSLOperation`

class `gql.dsl.`**`DSLSchema`**(*schema: graphql.type.schema.GraphQLSchema*)

Bases: `object`

The DSLSchema is the root of the DSL code.

Attributes of the DSLSchema class are generated automatically with the *__getattr__* dunder method in order to generate instances of `DSLType`

`__init__`(*schema: graphql.type.schema.GraphQLSchema*)

Initialize the DSLSchema with the given schema.

Parameters **`schema`** (`GraphQLSchema`) – a GraphQL Schema provided locally or fetched using an introspection query. Usually *client.schema*

Raises **`TypeError`** – if the argument is not an instance of `GraphQLSchema`

class `gql.dsl.`**`DSLOperation`**(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Bases: `abc.ABC`

Interface for GraphQL operations.

Inherited by `DSLQuery`, `DSLMutation` and `DSLSubscription`

`operation_type: graphql.language.ast.OperationType`

`__init__`(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Given arguments of type `DSLField` containing GraphQL requests, generate an operation which can be converted to a Document using the `dsl_gql`.

The fields arguments should be fields of root GraphQL types (Query, Mutation or Subscription) and correspond to the operation_type of this operation.

Parameters

- **`*fields`** (`DSLField`) – root instances of the dynamically generated requests
- **`**fields_with_alias`** (`DSLField`) – root instances fields with alias as key

Raises

- **`TypeError`** – if an argument is not an instance of `DSLField`
- **`AssertionError`** – if an argument is not a field which correspond to the operation type

class `gql.dsl.`**`DSLQuery`**(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Bases: `gql.dsl.DSLOperation`

`operation_type: graphql.language.ast.OperationType = 'query'`

__init__(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Given arguments of type *DSLField* containing GraphQL requests, generate an operation which can be converted to a Document using the *dsl_gql*.

The fields arguments should be fields of root GraphQL types (Query, Mutation or Subscription) and correspond to the operation_type of this operation.

Parameters

- ***fields** (DSLField) – root instances of the dynamically generated requests
- ****fields_with_alias** (DSLField) – root instances fields with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLField*
- **AssertionError** – if an argument is not a field which correspond to the operation type

name: Optional[str]

variable_definitions: ***gql.dsl.DSLVariableDefinitions***

selection_set: graphql.language.ast.SelectionSetNode

class gql.dsl.**DSLMutation**(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Bases: *gql.dsl.DSLOperation*

operation_type: graphql.language.ast.OperationType = 'mutation'

__init__(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Given arguments of type *DSLField* containing GraphQL requests, generate an operation which can be converted to a Document using the *dsl_gql*.

The fields arguments should be fields of root GraphQL types (Query, Mutation or Subscription) and correspond to the operation_type of this operation.

Parameters

- ***fields** (DSLField) – root instances of the dynamically generated requests
- ****fields_with_alias** (DSLField) – root instances fields with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLField*
- **AssertionError** – if an argument is not a field which correspond to the operation type

name: Optional[str]

variable_definitions: ***gql.dsl.DSLVariableDefinitions***

selection_set: graphql.language.ast.SelectionSetNode

class gql.dsl.**DSLSubscription**(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Bases: *gql.dsl.DSLOperation*

operation_type: graphql.language.ast.OperationType = 'subscription'

__init__(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*)

Given arguments of type *DSLField* containing GraphQL requests, generate an operation which can be converted to a Document using the *dsl_gql*.

The fields arguments should be fields of root GraphQL types (Query, Mutation or Subscription) and correspond to the operation_type of this operation.

Parameters

- ***fields** (DSLField) – root instances of the dynamically generated requests
- ****fields_with_alias** (DSLField) – root instances fields with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLField*
- **AssertionError** – if an argument is not a field which correspond to the operation type

name: Optional[str]

variable_definitions: ***gql.dsl.DSLVariableDefinitions***

selection_set: graphql.language.ast.SelectionSetNode

class gql.dsl.**DSLVariable**(*name: str*)
Bases: object

The DSLVariable represents a single variable defined in a GraphQL operation

Instances of this class are generated for you automatically as attributes of the *DSLVariableDefinitions*

The type of the variable is set by the *DSLField* instance that receives it in the *args* method.

__init__(*name: str*)
Initialize self. See help(type(self)) for accurate signature.

to_ast_type(*type_: Union[graphql.type.definition.GraphQLWrappingType, graphql.type.definition.GraphQLNamedType]*) → graphql.language.ast.TypeNode

set_type(*type_: Union[graphql.type.definition.GraphQLWrappingType, graphql.type.definition.GraphQLNamedType]*) → *gql.dsl.DSLVariable*

class gql.dsl.**DSLVariableDefinitions**
Bases: object

The DSLVariableDefinitions represents variable definitions in a GraphQL operation

Instances of this class have to be created and set as the *variable_definitions* attribute of a DSLOperation instance

Attributes of the DSLVariableDefinitions class are generated automatically with the *__getattr__* dunder method in order to generate instances of *DSLVariable*, that can then be used as values in the *DSLField.args* method

__init__()
Initialize self. See help(type(self)) for accurate signature.

class gql.dsl.**DSLType**(*graphql_type: Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType]*)
Bases: object

The DSLType represents a GraphQL type for the DSL code.

It can be a root type (Query, Mutation or Subscription). Or it can be any other object type (Human in the StarWars schema). Or it can be an interface type (Character in the StarWars schema).

Instances of this class are generated for you automatically as attributes of the *DSLSchema*

Attributes of the DSLType class are generated automatically with the *__getattr__* dunder method in order to generate instances of *DSLField*

__init__(*graphql_type: Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType]*)
Initialize the DSLType with the GraphQL type.

> **Warning:** Don't instantiate this class yourself. Use attributes of the *DSLSchema* instead.

Parameters **graphql_type** – the GraphQL type definition from the schema

class gql.dsl.**DSLField**(*name: str*, *graphql_type: Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType]*, *graphql_field: graphql.type.definition.GraphQLField*)

Bases: `object`

The DSLField represents a GraphQL field for the DSL code.

Instances of this class are generated for you automatically as attributes of the *DSLType*

If this field contains children fields, then you need to select which ones you want in the request using the *select* method.

__init__(*name: str*, *graphql_type: Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType]*, *graphql_field: graphql.type.definition.GraphQLField*)

Initialize the DSLField.

> **Warning:** Don't instantiate this class yourself. Use attributes of the *DSLType* instead.

Parameters

- **name** – the name of the field
- **graphql_type** – the GraphQL type definition from the schema
- **graphql_field** – the GraphQL field definition from the schema

select(**fields: gql.dsl.DSLField*, ***fields_with_alias: gql.dsl.DSLField*) → *gql.dsl.DSLField*

Select the new children fields that we want to receive in the request.

If used multiple times, we will add the new children fields to the existing children fields.

Parameters

- ***fields** (`DSLField`) – new children fields
- ****fields_with_alias** (`DSLField`) – new children fields with alias as key

Returns itself

Raises **TypeError** – if any of the provided fields are not instances of the *DSLField* class.

alias(*alias: str*) → *gql.dsl.DSLField*

Set an alias

> **Note:** You can also pass the alias directly at the *select* method. `ds.Query.human.select(my_name=ds.Character.name)` is equivalent to: `ds.Query.human.select(ds.Character.name.alias("my_name"))`

Parameters **alias** (*str*) – the alias

Returns itself

args(***kwargs*) → *gql.dsl.DSLField*

Set the arguments of a field

The arguments are parsed to be stored in the AST of this field.

Note: You can also call the field directly with your arguments. `ds.Query.human(id=1000)` is equivalent to: `ds.Query.human.args(id=1000)`

Parameters **`**kwargs`** – the arguments (keyword=value)

Returns itself

Raises **`KeyError`** – if any of the provided arguments does not exist for this field.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

INDEX