gql 3

Release 3.6.0b2

graphql-python.org

Apr 14, 2024

CONTENTS

1 Contents **1**

1.1 Introduction 1
1.2 Usage 2
1.3 Async vs Sync 15
1.4 Transports 17
1.5 Advanced 30
1.6 gql-cli 47
1.7 Reference 50

2 Indices and tables **95**

Python Module Index **97**

Index **99**

CHAPTER

ONE

CONTENTS

1.1 Introduction

GQL 3 is a GraphQL Client for Python 3.7+ 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 "gql[all]"
```

To have the latest pre-releases versions of gql, you can use:

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

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 necessitate specific dependencies. If you only need one transport you might want to install only the dependency needed for your transport, instead of using the "*all*" extra dependency as described above, which installs everything.

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

```
pip install gql[aiohttp]
```

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

Extra dependencies	Classes
aiohttp	*AIOHTTPTransport*
websockets	*WebsocketsTransport* *PhoenixChannelWebsocketsTransport* *AppSyncWebsocketsTransport*
requests	*RequestsHTTPTransport*
httpx	*HTTPTXTransport* *HTTPXAsyncTransport*
botocore	*AppSyncIAMAuthentication*

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

Installation with conda

It is also possible to install gql using conda.

To install gql with all extra dependencies:

```
conda install gql-with-all
```

To install gql with less dependencies, you might want to instead install a combinaison of the following packages: `gql-with-aiohttp`, `gql-with-websockets`, `gql-with-requests`, `gql-with-botocore`

If you want to have the latest pre-releases version of gql and graphql-core, you can install them with conda using:

```
conda install -c conda-forge -c conda-forge/label/graphql_core_alpha -c conda-forge/
↪label/gql_beta gql-with-all
```

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

```
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)
```

Note: You can download a schema from a server by using *gql-cli*

```
$ gql-cli https://SERVER_URL/graphql --print-schema --schema-download
input_value_deprecation:true > schema.graphql
```

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('''
    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!) {
```

(continues on next page)

(continued from previous page)

```
      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'})
```

After the connection, the latest response headers can be found in `transport.response_headers`

1.2.6 File uploads

GQL supports file uploads with the *aiohttp transport*, the *requests transport*, the *httpx transport*, and the *httpx async 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')
# Or transport = RequestsHTTPTransport(url='YOUR_URL')
# Or transport = HTTPXTransport(url='YOUR_URL')
# Or transport = HTTPXAsyncTransport(url='YOUR_URL')

client = Client(transport=transport)

query = gql('''
  mutation($file: Upload!) {
```

(continues on next page)

(continued from previous page)

```
    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
    )
```

Setting the content-type

If you need to set a specific Content-Type attribute to a file, you can set the `content_type` attribute of the file like this:

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

    # Setting the content-type to a pdf file for example
    f.content_type = "application/pdf"

    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')
# Or transport = RequestsHTTPTransport(url='YOUR_URL')
# Or transport = HTTPXTransport(url='YOUR_URL')
# Or transport = HTTPXAsyncTransport(url='YOUR_URL')

client = Client(transport=transport)

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

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

(continues on next page)

(continued from previous page)

```
f2 = open("YOUR_FILE_PATH_2", "rb")

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

result = client.execute(
    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

Note: Streaming is only supported with the *aiohttp transport*

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 *variable_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:
```

(continues on next page)

(continued from previous page)

```
        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
        )
```

1.2.7 Custom scalars and enums

Custom scalars

Scalar types represent primitive values at the leaves of a query.

GraphQL provides a number of built-in scalars (Int, Float, String, Boolean and ID), but a GraphQL backend can add additional custom scalars to its schema to better express values in their data model.

For example, a schema can define the Datetime scalar to represent an ISO-8601 encoded date.

The schema will then only contain:

```
scalar Datetime
```

When custom scalars are sent to the backend (as inputs) or from the backend (as outputs), their values need to be serialized to be composed of only built-in scalars, then at the destination the serialized values will be parsed again to be able to represent the scalar in its local internal representation.

Because this serialization/unserialization is dependent on the language used at both sides, it is not described in the schema and needs to be defined independently at both sides (client, backend).

A custom scalar value can have two different representations during its transport:

- as a serialized value (usually as json):
  - in the results sent by the backend
  - in the variables sent by the client alongside the query
- as "literal" inside the query itself sent by the client

To define a custom scalar, you need 3 methods:

- a `serialize` method used:
  - by the backend to serialize a custom scalar output in the result
  - by the client to serialize a custom scalar input in the variables
- a `parse_value` method used:
  - by the backend to unserialize custom scalars inputs in the variables sent by the client
  - by the client to unserialize custom scalars outputs from the results
- a `parse_literal` method used:
  - by the backend to unserialize custom scalars inputs inside the query itself

To define a custom scalar object, graphql-core provides the `GraphQLScalarType` class which contains the implementation of the above methods.

Example for Datetime:

```
from datetime import datetime
from typing import Any, Dict, Optional

from graphql import GraphQLScalarType, ValueNode
from graphql.utilities import value_from_ast_untyped


def serialize_datetime(value: Any) -> str:
```

(continues on next page)

(continued from previous page)

```
    return value.isoformat()


def parse_datetime_value(value: Any) -> datetime:
    return datetime.fromisoformat(value)


def parse_datetime_literal(
    value_node: ValueNode, variables: Optional[Dict[str, Any]] = None
) -> datetime:
    ast_value = value_from_ast_untyped(value_node, variables)
    return parse_datetime_value(ast_value)


DatetimeScalar = GraphQLScalarType(
    name="Datetime",
    serialize=serialize_datetime,
    parse_value=parse_datetime_value,
    parse_literal=parse_datetime_literal,
)
```

If you get your schema from a "schema.graphql" file or from introspection, then the generated schema in the gql Client will contain default `GraphQLScalarType` instances where the serialize and parse_value methods simply return the serialized value without modification.

In that case, if you want gql to parse custom scalars to a more useful Python representation, or to serialize custom scalars variables from a Python representation, then you can use the *update_schema_scalars* or *update_schema_scalar* methods to modify the definition of a scalar in your schema so that gql could do the parsing/serialization.

```
from gql.utilities import update_schema_scalar

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

client = Client(schema=schema_str, ...)

update_schema_scalar(client.schema, "Datetime", DatetimeScalar)

# or update_schema_scalars(client.schema, [DatetimeScalar])
```

Enums

GraphQL Enum types are a special kind of scalar that is restricted to a particular set of allowed values.

For example, the schema may have a Color enum and contain:

```
enum Color {
    RED
    GREEN
    BLUE
}
```

Graphql-core provides the `GraphQLEnumType` class to define an enum in the schema (See graphql-core schema building docs).

This class defines how the enum is serialized and parsed.

If you get your schema from a "schema.graphql" file or from introspection, then the generated schema in the gql Client will contain default `GraphQLEnumType` instances which should serialize/parse enums to/from its String representation (the `RED` enum will be serialized to `'RED'`).

You may want to parse enums to convert them to Python Enum types. In that case, you can use the *update_schema_enum* to modify the default `GraphQLEnumType` to use your defined Enum.

Example:

```
from enum import Enum
from gql.utilities import update_schema_enum

class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

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

client = Client(schema=schema_str, ...)

update_schema_enum(client.schema, 'Color', Color)
```

Serializing Inputs

To provide custom scalars and/or enums in inputs with gql, you can:

- serialize the inputs manually
- let gql serialize the inputs using the custom scalars and enums defined in the schema

Manually

You can serialize inputs yourself:

- in the query itself
- in variables

This has the advantage that you don't need a schema...

In the query

- custom scalar:

```
query = gql(
    """{
    shiftDays(time: "2021-11-12T11:58:13.461161", days: 5)
}"""
)
```

- enum:

```
query = gql("{opposite(color: RED)}")
```

In a variable

- custom scalar:

```
query = gql("query shift5days($time: Datetime) {shiftDays(time: $time, days: 5)}")

variable_values = {
    "time": "2021-11-12T11:58:13.461161",
}

result = client.execute(query, variable_values=variable_values)
```

- enum:

```
query = gql(
    """
    query GetOppositeColor($color: Color) {
        opposite(color:$color)
    }"""
)

variable_values = {
    "color": 'RED',
}

result = client.execute(query, variable_values=variable_values)
```

Automatically

If you have custom scalar and/or enums defined in your schema (See: *Custom scalars* and *Enums*), then you can request gql to serialize your variables automatically.

- use `Client(..., serialize_variables=True)` to request serializing variables for all queries
- use `execute(..., serialize_variables=True)` or `subscribe(..., serialize_variables=True)` if you want gql to serialize the variables only for a single query.

Examples:

- custom scalars:

```
from gql.utilities import update_schema_scalars

from .myscalars import DatetimeScalar

async with Client(transport=transport, fetch_schema_from_transport=True) as session:

    # We update the schema we got from introspection with our custom scalar type
    update_schema_scalars(session.client.schema, [DatetimeScalar])

    # In the query, the custom scalar in the input is set to a variable
    query = gql("query shift5days($time: Datetime) {shiftDays(time: $time, days: 5)}")

    # the argument for time is a datetime instance
    variable_values = {"time": datetime.now()}

    # we execute the query with serialize_variables set to True
    result = await session.execute(
        query, variable_values=variable_values, serialize_variables=True
    )
```

- enums:

```
from gql.utilities import update_schema_enum

from .myenums import Color

async with Client(transport=transport, fetch_schema_from_transport=True) as session:

    # We update the schema we got from introspection with our custom enum
    update_schema_enum(session.client.schema, 'Color', Color)

    # In the query, the enum in the input is set to a variable
    query = gql(
        """
        query GetOppositeColor($color: Color) {
            opposite(color:$color)
        }"""
    )

    # the argument for time is an instance of our Enum type
    variable_values = {
        "color": Color.RED,
    }

    # we execute the query with serialize_variables set to True
    result = client.execute(
        query, variable_values=variable_values, serialize_variables=True
    )
```

Parsing output

By default, gql returns the serialized result from the backend without parsing (except json unserialization to Python default types).

if you want to convert the result of custom scalars to custom objects, you can request gql to parse the results.

- use `Client(..., parse_results=True)` to request parsing for all queries
- use `execute(..., parse_result=True)` or `subscribe(..., parse_result=True)` if you want gql to parse only the result of a single query.

Same example as above, with result parsing enabled:

```
from gql.utilities import update_schema_scalars

async with Client(transport=transport, fetch_schema_from_transport=True) as session:

    update_schema_scalars(session.client.schema, [DatetimeScalar])

    query = gql("query shift5days($time: Datetime) {shiftDays(time: $time, days: 5)}")

    variable_values = {"time": datetime.now()}

    result = await session.execute(
        query,
        variable_values=variable_values,
        serialize_variables=True,
        parse_result=True,
    )

    # now result["time"] type is a datetime instead of string
```

1.2.8 Extensions

When you execute (or subscribe) GraphQL requests, the server will send responses which may have 3 fields:

- data: the serialized response from the backend
- errors: a list of potential errors
- extensions: an optional field for additional data

If there are errors in the response, then the `execute` or `subscribe` methods will raise a `TransportQueryError`.

If no errors are present, then only the data from the response is returned by default.

```
result = client.execute(query)
# result is here the content of the data field
```

If you need to receive the extensions data too, then you can run the `execute` or `subscribe` methods with `get_execution_result=True`.

In that case, the full execution result is returned and you can have access to the extensions field

```
result = client.execute(query, get_execution_result=True)
# result is here an ExecutionResult instance
```

(continues on next page)

(continued from previous page)

```
# result.data is the content of the data field
# result.extensions is the content of the extensions field
```

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(
```

(continues on next page)

(continued from previous page)

```
            """
            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.

Reference: *gql.transport.aiohttp.AIOHTTPTransport*

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

```python
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)
```

(continues on next page)

(continued from previous page)

```
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})
```

HTTPXAsyncTransport

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

Reference: *gql.transport.httpx.HTTPXAsyncTransport*

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.httpx import HTTPXAsyncTransport


async def main():

    transport = HTTPXAsyncTransport(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
```

(continues on next page)

(continued from previous page)

```
    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())
```

Authentication

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

1. Using HTTP Headers

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

WebsocketsTransport

The websockets transport supports both:

- the Apollo websockets transport protocol.
- the GraphQL-ws websockets transport protocol

It will propose both subprotocols to the backend and detect the supported protocol from the response http headers returned by the backend.

Note: For some backends (graphql-ws before version 5.6.1 without backwards compatibility), it may be necessary to specify only one subprotocol to the backend. It can be done by using `subprotocols=[WebsocketsTransport.`

GRAPHQLWS_SUBPROTOCOL] or subprotocols=[WebsocketsTransport.APOLLO_SUBPROTOCOL] in the transport arguments.

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

Reference: *gql.transport.websockets.WebsocketsTransport*

```
import asyncio
import logging

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

logging.basicConfig(level=logging.INFO)


async def main():

    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)
```

(continues on next page)

(continued from previous page)

```
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

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

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

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

2. With a payload in the connection_init websocket message

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

Keep-Alives

Apollo protocol

With the Apollo protocol, the backend can optionally send unidirectional keep-alive ("ka") messages (only from the server to the client).

It is possible to configure the transport to close if we don't receive a "ka" message within a specified time using the `keep_alive_timeout` parameter.

Here is an example with 60 seconds:

```
transport = WebsocketsTransport(
    url='wss://SERVER_URL:SERVER_PORT/graphql',
    keep_alive_timeout=60,
)
```

One disadvantage of the Apollo protocol is that because the keep-alives are only sent from the server to the client, it can be difficult to detect the loss of a connection quickly from the server side.

GraphQL-ws protocol

With the GraphQL-ws protocol, it is possible to send bidirectional ping/pong messages. Pings can be sent either from the client or the server and the other party should answer with a pong.

As with the Apollo protocol, it is possible to configure the transport to close if we don't receive any message from the backend within the specified time using the `keep_alive_timeout` parameter.

But there is also the possibility for the client to send pings at a regular interval and verify that the backend sends a pong within a specified delay. This can be done using the `ping_interval` and `pong_timeout` parameters.

Here is an example with a ping sent every 60 seconds, expecting a pong within 10 seconds:

```
transport = WebsocketsTransport(
    url='wss://SERVER_URL:SERVER_PORT/graphql',
    ping_interval=60,
    pong_timeout=10,
)
```

Underlying websockets protocol

In addition to the keep-alives described above for the apollo and graphql-ws protocols, there are also ping frames sent by the underlying websocket connection itself for both of them.

These pings are enabled by default (every 20 seconds) and could be modified or disabled by passing extra arguments to the `connect` call of the websockets client using the `connect_args` argument of the transport.

```
# Disabling websocket protocol level pings
transport = WebsocketsTransport(
    url='wss://SERVER_URL:SERVER_PORT/graphql',
    connect_args={"ping_interval": None},
)
```

See the websockets keepalive documentation for details.

PhoenixChannelWebsocketsTransport

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

Reference: *gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport*

```
import asyncio

from gql import Client, gql
from gql.transport.phoenix_channel_websockets import PhoenixChannelWebsocketsTransport


async def main():

    transport = PhoenixChannelWebsocketsTransport(
        channel_name="YOUR_CHANNEL", url="wss://YOUR_URL/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) as session:

        # Execute single query
        query = gql(
            """
            query yourQuery {
                ...
            }
        """
        )

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


asyncio.run(main())
```

AppSyncWebsocketsTransport

AWS AppSync allows you to execute GraphQL subscriptions on its realtime GraphQL endpoint.

See Building a real-time websocket client for an explanation.

GQL provides the `AppSyncWebsocketsTransport` transport which implements this for you to allow you to execute subscriptions.

Note: It is only possible to execute subscriptions with this transport. For queries or mutations, See *AppSync GraphQL Queries and mutations*

How to use it:

- choose one *authentication method* (API key, IAM, Cognito user pools or OIDC)

- instantiate a `AppSyncWebsocketsTransport` with your GraphQL endpoint as url and your auth method

---

Note: It is also possible to instantiate the transport without an auth argument. In that case, gql will use by default the *IAM auth* which will try to authenticate with environment variables or from your aws credentials file.

Note: All the examples in this documentation are based on the sample app created by following this AWS blog post

Full example with API key authentication from environment variables:

```
import asyncio
import os
import sys
from urllib.parse import urlparse

from gql import Client, gql
from gql.transport.appsync_auth import AppSyncApiKeyAuthentication
from gql.transport.appsync_websockets import AppSyncWebsocketsTransport

# Uncomment the following lines to enable debug output
# import logging
# logging.basicConfig(level=logging.DEBUG)


async def main():

    # Should look like:
    # https://XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com/graphql
    url = os.environ.get("AWS_GRAPHQL_API_ENDPOINT")
    api_key = os.environ.get("AWS_GRAPHQL_API_KEY")

    if url is None or api_key is None:
        print("Missing environment variables")
        sys.exit()

    # Extract host from url
    host = str(urlparse(url).netloc)

    print(f"Host: {host}")

    auth = AppSyncApiKeyAuthentication(host=host, api_key=api_key)

    transport = AppSyncWebsocketsTransport(url=url, auth=auth)

    async with Client(transport=transport) as session:

        subscription = gql(
            """
subscription onCreateMessage {
  onCreateMessage {
    message
  }
```

(continues on next page)

(continued from previous page)

```
}
"""
        )

        print("Waiting for messages...")

        async for result in session.subscribe(subscription):
            print(result)


asyncio.run(main())
```

Reference: *gql.transport.appsync_websockets.AppSyncWebsocketsTransport*

Authentication methods

API key

Use the `AppSyncApiKeyAuthentication` class to provide your API key:

```
auth = AppSyncApiKeyAuthentication(
    host="XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com",
    api_key="YOUR_API_KEY",
)

transport = AppSyncWebsocketsTransport(
    url="https://XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com/graphql",
    auth=auth,
)
```

Reference: *gql.transport.appsync_auth.AppSyncApiKeyAuthentication*

IAM

For the IAM authentication, you can simply create your transport without an auth argument.

The region name will be autodetected from the url or from your AWS configuration (`.aws/config`) or the environment variable:

- AWS_DEFAULT_REGION

The credentials will be detected from your AWS configuration file (`.aws/credentials`) or from the environment variables:

- AWS_ACCESS_KEY_ID
- AWS_SECRET_ACCESS_KEY
- AWS_SESSION_TOKEN (optional)

```
transport = AppSyncWebsocketsTransport(
    url="https://XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com/graphql",
)
```

OR You can also provide the credentials manually by creating the `AppSyncIAMAuthentication` class yourself:

```
from botocore.credentials import Credentials

credentials = Credentials(
    access_key = os.environ.get("AWS_ACCESS_KEY_ID"),
    secret_key= os.environ.get("AWS_SECRET_ACCESS_KEY"),
    token=os.environ.get("AWS_SESSION_TOKEN", None),   # Optional
)

auth = AppSyncIAMAuthentication(
    host="XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com",
    credentials=credentials,
    region_name="your region"
)

transport = AppSyncWebsocketsTransport(
    url="https://XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com/graphql",
    auth=auth,
)
```

Reference: *gql.transport.appsync_auth.AppSyncIAMAuthentication*

Json Web Tokens (jwt)

AWS provides json web tokens (jwt) for the authentication methods:

- Amazon Cognito user pools
- OpenID Connect (OIDC)

For these authentication methods, you can use the `AppSyncJWTAuthentication` class:

```
auth = AppSyncJWTAuthentication(
    host="XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com",
    jwt="YOUR_JWT_STRING",
)

transport = AppSyncWebsocketsTransport(
    url="https://XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com/graphql",
    auth=auth,
)
```

Reference: *gql.transport.appsync_auth.AppSyncJWTAuthentication*

AppSync GraphQL Queries and mutations

Queries and mutations are not allowed on the realtime websockets endpoint. But you can use the *AIOHTTPTransport* to create a normal http session and reuse the authentication classes to create the headers for you.

Full example with API key authentication from environment variables:

```
import asyncio
import os
import sys
from urllib.parse import urlparse

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

# Uncomment the following lines to enable debug output
# import logging
# logging.basicConfig(level=logging.DEBUG)


async def main():

    # Should look like:
    # https://XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com/graphql
    url = os.environ.get("AWS_GRAPHQL_API_ENDPOINT")
    api_key = os.environ.get("AWS_GRAPHQL_API_KEY")

    if url is None or api_key is None:
        print("Missing environment variables")
        sys.exit()

    # Extract host from url
    host = str(urlparse(url).netloc)

    auth = AppSyncApiKeyAuthentication(host=host, api_key=api_key)

    transport = AIOHTTPTransport(url=url, auth=auth)

    async with Client(
        transport=transport,
        fetch_schema_from_transport=False,
    ) as session:

        query = gql(
            """
mutation createMessage($message: String!) {
  createMessage(input: {message: $message}) {
    id
    message
    createdAt
  }
}"""
```

(continues on next page)

(continued from previous page)

```
        )

        variable_values = {"message": "Hello world!"}

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


asyncio.run(main())
```

From the command line

Using *gql-cli*, it is possible to execute GraphQL queries and subscriptions from the command line on an AppSync endpoint.

- For queries and mutations, use the `--transport appsync_http` argument:

```
# Put the request in a file
$ echo 'mutation createMessage($message: String!) {
  createMessage(input: {message: $message}) {
    id
    message
    createdAt
  }
}' > mutation.graphql

# Execute the request using gql-cli with --transport appsync_http
$ cat mutation.graphql | gql-cli $AWS_GRAPHQL_API_ENDPOINT --transport appsync_http␣
↪-V message:"Hello world!"
```

- For subscriptions, use the `--transport appsync_websockets` argument:

```
echo "subscription{onCreateMessage{message}}" | gql-cli $AWS_GRAPHQL_API_ENDPOINT --
↪transport appsync_websockets
```

- You can also get the full GraphQL schema from the backend from introspection:

```
$ gql-cli $AWS_GRAPHQL_API_ENDPOINT --transport appsync_http --print-schema >␣
↪schema.graphql
```

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.

Reference: *gql.transport.requests.RequestsHTTPTransport*

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

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

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

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

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

HTTPXTransport

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

Reference: *gql.transport.httpx.HTTPXTransport*

```
from gql import Client, gql
from gql.transport.httpx import HTTPXTransport

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

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

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

(continues on next page)

(continued from previous page)

```
    }
"""
)

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")

    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))
```

(continues on next page)

(continued from previous page)

```
        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 Async permanent session

Sometimes you want to have a single permanent reconnecting async session to a GraphQL backend, and that can be difficult to manage manually with the `async with client as session` syntax.

It is now possible to have a single reconnecting session using the *connect_async* method of Client with a `reconnecting=True` argument.

```
# Create a session from the client which will reconnect automatically.
# This session can be kept in a class for example to provide a way
# to execute GraphQL queries from many different places
session = await client.connect_async(reconnecting=True)

# You can run execute or subscribe method on this session
result = await session.execute(query)

# When you want the connection to close (for cleanup),
# you call close_async
await client.close_async()
```

When you use `reconnecting=True`, gql will watch the exceptions generated during the execute and subscribe calls and, if it detects a TransportClosed exception (indicating that the link to the underlying transport is broken), it will try to reconnect to the backend again.

Retries

Connection retries

With `reconnecting=True`, gql will use the backoff module to repeatedly try to connect with exponential backoff and jitter with a maximum delay of 60 seconds by default.

You can change the default reconnecting profile by providing your own backoff decorator to the `retry_connect` argument.

```
# Here wait maximum 5 minutes between connection retries
retry_connect = backoff.on_exception(
    backoff.expo,  # wait generator (here: exponential backoff)
    Exception,     # which exceptions should cause a retry (here: everything)
    max_value=300, # max wait time in seconds
)
session = await client.connect_async(
```

(continues on next page)

(continued from previous page)

```
    reconnecting=True,
    retry_connect=retry_connect,
)
```

Execution retries

With `reconnecting=True`, by default we will also retry up to 5 times when an exception happens during an execute call (to manage a possible loss in the connection to the transport).

There is no retry in case of a `TransportQueryError` exception as it indicates that the connection to the backend is working correctly.

You can change the default execute retry profile by providing your own backoff decorator to the `retry_execute` argument.

```
# Here Only 3 tries for execute calls
retry_execute = backoff.on_exception(
    backoff.expo,
    Exception,
    max_tries=3,
)
session = await client.connect_async(
    reconnecting=True,
    retry_execute=retry_execute,
)
```

If you don't want any retry on the execute calls, you can disable the retries with `retry_execute=False`

Note: If you want to retry even with `TransportQueryError` exceptions, then you need to make your own backoff decorator on your own method:

```
@backoff.on_exception(backoff.expo,
                      Exception,
                      max_tries=3)
async def execute_with_retry(session, query):
    return await session.execute(query)
```

Subscription retries

There is no `retry_subscribe` as it is not feasible with async generators. If you want retries for your subscriptions, then you can do it yourself with backoff decorators on your methods.

```
@backoff.on_exception(backoff.expo,
                      Exception,
                      max_tries=3,
                      giveup=lambda e: isinstance(e, TransportQueryError))
async def execute_subscription1(session):
    async for result in session.subscribe(subscription1):
        print(result)
```

FastAPI example

```
# First install fastapi and uvicorn:
#
# pip install fastapi uvicorn
#
# then run:
#
# uvicorn fastapi_async:app --reload

import logging

from fastapi import FastAPI, HTTPException
from fastapi.responses import HTMLResponse
from gql import Client, gql
from gql.transport.aiohttp import AIOHTTPTransport

logging.basicConfig(level=logging.DEBUG)
log = logging.getLogger(__name__)

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

client = Client(transport=transport)

query = gql(
    """
query getContinentInfo($code: ID!) {
  continent(code:$code) {
    name
    code
    countries  {
      name
      capital
    }
  }
}
"""
)

app = FastAPI()


@app.on_event("startup")
async def startup_event():
    print("Connecting to GraphQL backend")

    await client.connect_async(reconnecting=True)
    print("End of startup")


@app.on_event("shutdown")
async def shutdown_event():
    print("Shutting down GraphQL permanent connection...")
```

(continues on next page)

(continued from previous page)

```
    await client.close_async()
    print("Shutting down GraphQL permanent connection... done")


continent_codes = [
    "AF",
    "AN",
    "AS",
    "EU",
    "NA",
    "OC",
    "SA",
]


@app.get("/", response_class=HTMLResponse)
def get_root():

    continent_links = ", ".join(
        [f'<a href="continent/{code}">{code}</a>' for code in continent_codes]
    )

    return f"""
    <html>
        <head>
            <title>Continents</title>
        </head>
        <body>
            Continents: {continent_links}
        </body>
    </html>
"""


@app.get("/continent/{continent_code}")
async def get_continent(continent_code):

    if continent_code not in continent_codes:
        raise HTTPException(status_code=404, detail="Continent not found")

    try:
        result = await client.session.execute(
            query, variable_values={"code": continent_code}
        )
    except Exception as e:
        log.debug(f"get_continent Error: {e}")
        raise HTTPException(status_code=503, detail="GraphQL backend unavailable")

    return result
```

Console example

```
import asyncio
import logging

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

logging.basicConfig(level=logging.INFO)

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


class GraphQLContinentClient:
    def __init__(self):
        self._client = Client(
            transport=AIOHTTPTransport(url="https://countries.trevorblades.com/")
        )
        self._session = None

        self.get_continent_name_query = gql(GET_CONTINENT_NAME)

    async def connect(self):
        self._session = await self._client.connect_async(reconnecting=True)

    async def close(self):
        await self._client.close_async()

    async def get_continent_name(self, code):
        params = {"code": code}

        answer = await self._session.execute(
            self.get_continent_name_query, variable_values=params
        )

        return answer.get("continent").get("name")


async def main():
    continent_client = GraphQLContinentClient()

    continent_codes = ["AF", "AN", "AS", "EU", "NA", "OC", "SA"]

    await continent_client.connect()

    while True:
```

(continues on next page)

(continued from previous page)

```
        answer = await ainput("\nPlease enter a continent code or 'exit':")
        answer = answer.strip()

        if answer == "exit":
            break
        elif answer in continent_codes:

            try:
                continent_name = await continent_client.get_continent_name(answer)
                print(f"The continent name is {continent_name}\n")
            except Exception as exc:
                print(f"Received exception {exc} while trying to get continent name")

        else:
            print(f"Please enter a valid continent code from {continent_codes}")

    await continent_client.close()


asyncio.run(main())
```

1.5.3 Logging

GQL uses 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)
```

1.5.4 Error Handing

Local errors

If gql detects locally that something does not correspond to the GraphQL specification, then gql may raise a **GraphQLError** from graphql-core.

This may happen for example:

- if your query is not valid
- if your query does not correspond to your schema
- if the result received from the backend does not correspond to the schema if `parse_results` is set to True

Transport errors

If an error happens with the transport, then gql may raise a *TransportError*

Here are the possible Transport Errors:

- *TransportProtocolError*: Should never happen if the backend is a correctly configured GraphQL server. It means that the answer received from the server does not correspond to the transport protocol.
- *TransportServerError*: There was an error communicating with the server. If this error is received, then the connection with the server will be closed. This may happen if the server returned a 404 http header for example. The http error code is available in the exception `code` attribute.
- *TransportQueryError*: There was a specific error returned from the server for your query. The message you receive in this error has been created by the backend, not gql! In that case, the connection to the server is still available and you are free to try to send other queries using the same connection. The message of the exception contains the first error returned by the backend. All the errors messages are available in the exception `errors` attribute.

  If the error message begins with `Error while fetching schema:`, it means that gql was not able to get the schema from the backend. If you don't need the schema, you can try to create the client with `fetch_schema_from_transport=False`
- *TransportClosed*: This exception is generated when the client is trying to use the transport while the transport was previously closed.
- *TransportAlreadyConnected*: Exception generated when the client is trying to connect to the transport while the transport is already connected.

HTTP

For HTTP transports, we should get a json response which contain `data` or `errors` fields. If that is not the case, then the returned error depends whether the http return code is below 400 or not.

- **json response:**
  - **with data or errors keys:**
    - no errors key -> no exception
    - errors key -> raise **TransportQueryError**
  - **no data or errors keys:**
    - http code < 400: raise **TransportProtocolError**
    - http code >= 400: raise **TransportServerError**
- **not a json response:**
  - http code < 400: raise **TransportProtocolError**
  - http code >= 400: raise **TransportServerError**

1.5.5 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

1.5.6 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(
```

(continues on next page)

(continued from previous page)

```
        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)
```

Note: If your argument name is a Python keyword (for, in, from, ...), you will receive a SyntaxError (See issue #308). To fix this, you can provide the arguments by unpacking a dictionary.

For example, instead of using `from=5`, you can use `**{"from":5}`

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:

- Instantiate 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
  }
}
```

Variable arguments with a default value

If you want to provide a **default value** for your variable, you can use the `default` method on a variable.

The following code:

```
var = DSLVariableDefinitions()
op = DSLMutation(
    ds.Mutation.createReview.args(
        review=var.review.default({"stars": 5, "commentary": "Wow!"}),
        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 = {stars: 5, commentary: "Wow!"}, $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( ... ),
)
```

Fragments

To define a Fragment, you have to:

- Instantiate a *DSLFragment* with a name:

```
name_and_appearances = DSLFragment("NameAndAppearances")
```

- Provide the GraphQL type of the fragment with the *on* method:

```
name_and_appearances.on(ds.Character)
```

- Add children fields using the *select* method:

```
name_and_appearances.select(ds.Character.name, ds.Character.appearsIn)
```

Once your fragment is defined, to use it you should:

- select it as a field somewhere in your query:

```
query_with_fragment = DSLQuery(ds.Query.hero.select(name_and_appearances))
```

- add it as an argument of *dsl_gql* with your query:

```
query = dsl_gql(name_and_appearances, query_with_fragment)
```

The above example will generate the following request:

```
fragment NameAndAppearances on Character {
    name
    appearsIn
}

{
    hero {
        ...NameAndAppearances
    }
}
```

Inline Fragments

To define an Inline Fragment, you have to:

- Instantiate a *DSLInlineFragment*:

```
human_fragment = DSLInlineFragment()
```

- Provide the GraphQL type of the fragment with the *on* method:

```
human_fragment.on(ds.Human)
```

- Add children fields using the *select* method:

```
human_fragment.select(ds.Human.homePlanet)
```

Once your inline fragment is defined, to use it you should:

- select it as a field somewhere in your query:

```
query_with_inline_fragment = ds.Query.hero.args(episode=6).select(
    ds.Character.name,
    human_fragment
)
```

The above example will generate the following request:

```
hero(episode: JEDI) {
    name
    ... on Human {
      homePlanet
    }
}
```

Note: because the *on* and *select* methods return `self`, this can be written in a concise manner:

```
query_with_inline_fragment = ds.Query.hero.args(episode=6).select(
    ds.Character.name,
    DSLInlineFragment().on(ds.Human).select(ds.Human.homePlanet)
)
```

Meta-fields

To define meta-fields (`__typename`, `__schema` and `__type`), you can use the *DSLMetaField* class:

```
query = ds.Query.hero.select(
    ds.Character.name,
    DSLMetaField("__typename")
)
```

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:

        # Instantiate 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,
)

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

    # Instantiate 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 3.7+ 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 ...]] [-H [HEADERS ...]] [--version]
               [-d | -v] [-o OPERATION_NAME] [--print-schema]
               [--schema-download [SCHEMA_DOWNLOAD ...]]
```

(continues on next page)

(continued from previous page)

```
               [--execute-timeout EXECUTE_TIMEOUT]
               [--transport {auto,aiohttp,phoenix,websockets,appsync_http,appsync_
↪websockets}]
               [--api-key API_KEY | --jwt JWT]
               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

--print-schema get the schema from instrospection and print it

Default: False

--schema-download **select the introspection query arguments to download the schema.**

Only useful if –print-schema is used. By default, it will:

- request field descriptions
- not request deprecated input fields

Possible options:

- descriptions:false for a compact schema without comments
- input_value_deprecation:true to download deprecated input fields
- specified_by_url:true
- schema_description:true
- directive_is_repeatable:true

--execute-timeout set the execute_timeout argument of the Client (default: 10)

Default: 10

--transport Possible choices: auto, aiohttp, phoenix, websockets, appsync_http, appsync_websockets

select the transport. 'auto' by default: aiohttp or websockets depending on url scheme

Default: "auto"

AWS AppSync options

By default, for an AppSync backend, the IAM authentication is chosen.

If you want API key or JWT authentication, you can provide one of the following arguments:

--api-key	Provide an API key for authentication
--jwt	Provide an JSON Web token for authentication

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"}}
```

Print the GraphQL schema in a file

```
$ gql-cli https://countries.trevorblades.com/graphql --print-schema > schema.graphql
```

Note: By default, deprecated input fields are not requested from the backend. You can add `--schema-download input_value_deprecation:true` to request them.

Note: You can add `--schema-download descriptions:false` to request a compact schema without comments.

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, GraphQLSchema]] = None*, *introspection: Optional[IntrospectionQuery] = None*, *transport: Optional[Union[*Transport*,* AsyncTransport*]] = None*, *fetch_schema_from_transport: bool = False*, *introspection_args: Optional[Dict] = None*, *execute_timeout: Optional[Union[int, float]] = 10*, *serialize_variables: bool = False*, *parse_results: bool = False*, *batch_interval: float = 0*, *batch_max: 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, GraphQLSchema]] = None*, *introspection: Optional[IntrospectionQuery] = None*, *transport: Optional[Union[*Transport*,* AsyncTransport*]] = None*, *fetch_schema_from_transport: bool = False*, *introspection_args: Optional[Dict] = None*, *execute_timeout: Optional[Union[int, float]] = 10*, *serialize_variables: bool = False*, *parse_results: bool = False*, *batch_interval: float = 0*, *batch_max: 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.
- **introspection_args** – arguments passed to the get_introspection_query method of graphql-core.
- **execute_timeout** – The maximum time in seconds for the execution of a request before a TimeoutError is raised. Only used for async transports. Passing None results in waiting forever for a response.
- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. Default: False.
- **parse_results** – Whether gql will try to parse the serialized output sent by the backend. Can be used to deserialize custom scalars or enums.
- **batch_interval** – Time to wait in seconds for batching requests together. Batching is disabled (by default) if 0.
- **batch_max** – Maximum number of requests in a single batch.

property batching_enabled

async close_async()

Close the async transport and stop the optional reconnecting task.

close_sync()

Close the sync session and the sync transport.

If batching is enabled, this will block until the remaining queries in the batching queue have been processed.

async connect_async(*reconnecting=False*, ***kwargs*)

Connect asynchronously with the underlying async transport to produce a session.

That session will be a permanent auto-reconnecting session if `reconnecting=True`.

If you call this method, you should call the *close_async* method for cleanup.

Parameters

- **reconnecting** – if True, create a permanent reconnecting session
- ****kwargs** – additional arguments for the *ReconnectingAsyncClientSession init method*.

connect_sync()

Connect synchronously with the underlying sync transport to produce a session.

If you call this method, you should call the *close_sync* method for cleanup.

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[False] = False*, ***kwargs*) → Dict[str, Any]

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[True]*, ***kwargs*) → ExecutionResult

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: bool*, ***kwargs*) → Union[Dict[str, Any], ExecutionResult]

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.

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: Literal[False]*, ***kwargs*) → List[Dict[str, Any]]

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: Literal[True]*, ***kwargs*) → List[ExecutionResult]

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: bool*, ***kwargs*) → Union[List[Dict[str, Any]], List[ExecutionResult]]

Execute multiple GraphQL requests in a batch 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 requests on the transport session
- close the session and close the connection to the server

If you want to perform multiple executions, it is better to use the context manager to keep a session active.

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

subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[False] = False*, ***kwargs*) → Generator[Dict[str, Any], None, None]

subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[True]*, ***kwargs*) → Generator[ExecutionResult, None, None]

subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: bool*, ***kwargs*) → Union[Generator[Dict[str, Any], None, None], Generator[ExecutionResult, None, None]]

Execute a GraphQL subscription with a python generator.

We need an async transport for this functionality.

class gql.**GraphQLRequest**(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*)

Bases: `object`

GraphQL Request to be executed.

__init__(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → None

document: DocumentNode

GraphQL query as AST Node object.

operation_name: Optional[str] = None

Name of the operation that shall be executed. Only required in multi-operation documents (Default: None).

serialize_variable_values(*schema: GraphQLSchema*) → *GraphQLRequest*

variable_values: Optional[Dict[str, Any]] = None

Dictionary of input parameters (Default: None).

gql.**gql**(*request_string: str | Source*) → DocumentNode

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

Parameters
request_string (*str | Source*) – 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:* 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:* Client)

Parameters
client – the *client* used

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[False] = False*, ***kwargs*) → Dict[str, Any]

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[True]*, ***kwargs*) → ExecutionResult

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: bool*, ***kwargs*) → Union[Dict[str, Any], ExecutionResult]

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

Raises a TransportQueryError if an error has been returned in the ExecutionResult.

Parameters

- **document** – GraphQL query as AST Node object.
- **variable_values** – Dictionary of input parameters.
- **operation_name** – Name of the operation that shall be executed.
- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. By default use the serialize_variables argument of the client.
- **parse_result** – Whether gql will deserialize the result. By default use the parse_results argument of the client.
- **get_execution_result** – return the full ExecutionResult instance instead of only the "data" field. Necessary if you want to get the "extensions" field.

The extra arguments are passed to the transport execute method.

async fetch_schema() → None

Fetch the GraphQL schema explicitly using introspection.

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

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[False] = False*, ***kwargs*) → AsyncGenerator[Dict[str, Any], None]

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[True]*, ***kwargs*) → AsyncGenerator[ExecutionResult, None]

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: bool*, ***kwargs*) → Union[AsyncGenerator[Dict[str, Any], None], AsyncGenerator[ExecutionResult, None]]

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

Raises a TransportQueryError if an error has been returned in the ExecutionResult.

Parameters

- **document** – GraphQL query as AST Node object.
- **variable_values** – Dictionary of input parameters.
- **operation_name** – Name of the operation that shall be executed.
- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. By default use the serialize_variables argument of the client.
- **parse_result** – Whether gql will deserialize the result. By default use the parse_results argument of the client.
- **get_execution_result** – yield the full ExecutionResult instance instead of only the "data" field. Necessary if you want to get the "extensions" field.

The extra arguments are passed to the transport subscribe method.

property transport

class gql.client.**Client**(*schema: Optional[Union[str, GraphQLSchema]] = None*, *introspection: Optional[IntrospectionQuery] = None*, *transport: Optional[Union[*Transport*,* AsyncTransport*]] = None*, *fetch_schema_from_transport: bool = False*, *introspection_args: Optional[Dict] = None*, *execute_timeout: Optional[Union[int, float]] = 10*, *serialize_variables: bool = False*, *parse_results: bool = False*, *batch_interval: float = 0*, *batch_max: 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, GraphQLSchema]] = None*, *introspection: Optional[IntrospectionQuery] = None*, *transport: Optional[Union[*Transport*,* AsyncTransport*]] = None*, *fetch_schema_from_transport: bool = False*, *introspection_args: Optional[Dict] = None*, *execute_timeout: Optional[Union[int, float]] = 10*, *serialize_variables: bool = False*, *parse_results: bool = False*, *batch_interval: float = 0*, *batch_max: 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.
- **introspection_args** – arguments passed to the get_introspection_query method of graphql-core.
- **execute_timeout** – The maximum time in seconds for the execution of a request before a TimeoutError is raised. Only used for async transports. Passing None results in waiting forever for a response.

- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. Default: False.
- **parse_results** – Whether gql will try to parse the serialized output sent by the backend. Can be used to deserialize custom scalars or enums.
- **batch_interval** – Time to wait in seconds for batching requests together. Batching is disabled (by default) if 0.
- **batch_max** – Maximum number of requests in a single batch.

property batching_enabled

async close_async()

Close the async transport and stop the optional reconnecting task.

close_sync()

Close the sync session and the sync transport.

If batching is enabled, this will block until the remaining queries in the batching queue have been processed.

async connect_async(*reconnecting=False*, ***kwargs*)

Connect asynchronously with the underlying async transport to produce a session.

That session will be a permanent auto-reconnecting session if `reconnecting=True`.

If you call this method, you should call the *close_async* method for cleanup.

Parameters

- **reconnecting** – if True, create a permanent reconnecting session
- ****kwargs** – additional arguments for the *ReconnectingAsyncClientSession init method*.

connect_sync()

Connect synchronously with the underlying sync transport to produce a session.

If you call this method, you should call the *close_sync* method for cleanup.

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[False] = False*, ***kwargs*) → Dict[str, Any]

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[True]*, ***kwargs*) → ExecutionResult

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: bool*, ***kwargs*) → Union[Dict[str, Any], ExecutionResult]

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.

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: Literal[False]*, ***kwargs*) → List[Dict[str, Any]]

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: Literal[True]*, ***kwargs*) → List[ExecutionResult]

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: bool*, ***kwargs*) → Union[List[Dict[str, Any]], List[ExecutionResult]]

Execute multiple GraphQL requests in a batch 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 requests on the transport session
- close the session and close the connection to the server

If you want to perform multiple executions, it is better to use the context manager to keep a session active.

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

subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[False] = False*, ***kwargs*) → Generator[Dict[str, Any], None, None]

subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: Literal[True]*, ***kwargs*) → Generator[ExecutionResult, None, None]

subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, ***, *get_execution_result: bool*, ***kwargs*) → Union[Generator[Dict[str, Any], None, None], Generator[ExecutionResult, None, None]]

Execute a GraphQL subscription with a python generator.

We need an async transport for this functionality.

class gql.client.**ReconnectingAsyncClientSession**(*client: Client*, *retry_connect: Union[bool, Callable[[_CallableT], _CallableT]] = True*, *retry_execute: Union[bool, Callable[[_CallableT], _CallableT]] = True*)

Bases: *AsyncClientSession*

An instance of this class is created when using the *connect_async* method of the *Client* class with `reconnecting=True`.

It is used to provide a single session which will reconnect automatically if the connection fails.

__init__(*client: Client*, *retry_connect: Union[bool, Callable[[_CallableT], _CallableT]] = True*, *retry_execute: Union[bool, Callable[[_CallableT], _CallableT]] = True*)

Parameters

- **client** – the *client* used.
- **retry_connect** – Either a Boolean to activate/deactivate the retries for the connection to the transport OR a backoff decorator to provide specific retries parameters for the connections.
- **retry_execute** – Either a Boolean to activate/deactivate the retries for the execute method OR a backoff decorator to provide specific retries parameters for this method.

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: bool = False*, ***kwargs*) → Union[Dict[str, Any], ExecutionResult]

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

Raises a TransportQueryError if an error has been returned in the ExecutionResult.

Parameters

- **document** – GraphQL query as AST Node object.
- **variable_values** – Dictionary of input parameters.
- **operation_name** – Name of the operation that shall be executed.
- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. By default use the serialize_variables argument of the client.
- **parse_result** – Whether gql will deserialize the result. By default use the parse_results argument of the client.
- **get_execution_result** – return the full ExecutionResult instance instead of only the "data" field. Necessary if you want to get the "extensions" field.

The extra arguments are passed to the transport execute method.

async fetch_schema() → None

Fetch the GraphQL schema explicitly using introspection.

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

async start_connecting_task()

Start the task responsible to restart the connection of the transport when requested by an event.

async stop_connecting_task()

Stop the connecting task.

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: bool = False*, ***kwargs*) → Union[AsyncGenerator[Dict[str, Any], None], AsyncGenerator[ExecutionResult, None]]

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

Raises a TransportQueryError if an error has been returned in
the ExecutionResult.

Parameters

- **document** – GraphQL query as AST Node object.
- **variable_values** – Dictionary of input parameters.
- **operation_name** – Name of the operation that shall be executed.
- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. By default use the serialize_variables argument of the client.
- **parse_result** – Whether gql will deserialize the result. By default use the parse_results argument of the client.
- **get_execution_result** – yield the full ExecutionResult instance instead of only the "data" field. Necessary if you want to get the "extensions" field.

The extra arguments are passed to the transport subscribe method.

property transport

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

Parameters
client – the *client* used

close()

Close the transport and cleanup the batching thread if batching is enabled.

Will wait until all the remaining requests in the batch processing queue have been executed.

connect()

Connect the transport and initialize the batch threading loop if batching is enabled.

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *, *get_execution_result: Literal[False] = False*, ***kwargs*) → Dict[str, Any]

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *, *get_execution_result: Literal[True]*, ***kwargs*) → ExecutionResult

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *, *get_execution_result: bool*, ***kwargs*) → Union[Dict[str, Any], ExecutionResult]

Execute the provided document AST synchronously using the sync transport.

Raises a TransportQueryError if an error has been returned in
the ExecutionResult.

Parameters

- **document** – GraphQL query as AST Node object.
- **variable_values** – Dictionary of input parameters.
- **operation_name** – Name of the operation that shall be executed.
- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. By default use the serialize_variables argument of the client.
- **parse_result** – Whether gql will deserialize the result. By default use the parse_results argument of the client.
- **get_execution_result** – return the full ExecutionResult instance instead of only the "data" field. Necessary if you want to get the "extensions" field.

The extra arguments are passed to the transport execute method.

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: Literal[False]*, ***kwargs*) → List[Dict[str, Any]]

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: Literal[True]*, ***kwargs*) → List[ExecutionResult]

execute_batch(*requests: List[GraphQLRequest]*, ***, *serialize_variables: Optional[bool] = None*, *parse_result: Optional[bool] = None*, *get_execution_result: bool*, ***kwargs*) → Union[List[Dict[str, Any]], List[ExecutionResult]]

Execute multiple GraphQL requests in a batch, using the sync transport. This method sends the requests to the server all at once.

Raises a TransportQueryError if an error has been returned in any
ExecutionResult.

Parameters

- **requests** – List of requests that will be executed.
- **serialize_variables** – whether the variable values should be serialized. Used for custom scalars and/or enums. By default use the serialize_variables argument of the client.
- **parse_result** – Whether gql will deserialize the result. By default use the parse_results argument of the client.
- **get_execution_result** – return the full ExecutionResult instance instead of only the "data" field. Necessary if you want to get the "extensions" field.

The extra arguments are passed to the transport execute method.

fetch_schema() → None

Fetch the GraphQL schema explicitly 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: ABC

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: DocumentNode*, **args*, ***kwargs*) → 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

execute_batch(*reqs: List[*GraphQLRequest*]*, **args*, ***kwargs*) → List[ExecutionResult]

Execute multiple GraphQL requests in a batch.

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

Parameters
reqs – GraphQL requests as a list of GraphQLRequest objects.

Returns
a list of ExecutionResult objects

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

Bases: ABC

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: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → ExecutionResult

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

abstract subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → AsyncGenerator[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

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

Bases: *AsyncTransport*

A transport for executing GraphQL queries against a local schema.

__init__(*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: DocumentNode*, **args*, ***kwargs*) → ExecutionResult

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

async subscribe(*document: DocumentNode*, **args*, ***kwargs*) → AsyncGenerator[ExecutionResult, None]

Send a subscription and receive the results using an async generator

The results are sent as an ExecutionResult object

gql.transport.aiohttp

class gql.transport.aiohttp.**AIOHTTPTransport**(*url: str, headers: ~typing.Optional[~typing.Union[~typing.Mapping[~typing.Union[str, ~multidict._multidict.istr], str], ~multidict._multidict.CIMultiDict, ~multidict._multidict.CIMultiDictProxy]] = None, cookies: ~typing.Optional[~typing.Union[~typing.Mapping[str, ~typing.Union[str, BaseCookie[str], Morsel[Any]]], ~typing.Iterable[~typing.Tuple[str, ~typing.Union[str, BaseCookie[str], Morsel[Any]]]], BaseCookie[str]]] = None, auth: ~typing.Optional[~typing.Union[~aiohttp.helpers.BasicAuth, AppSyncAuthentication]] = None, ssl: ~typing.Union[~ssl.SSLContext, bool, ~aiohttp.client_reqrep.Fingerprint] = False, timeout: ~typing.Optional[int] = None, ssl_close_timeout: ~typing.Optional[~typing.Union[int, float]] = 10, json_serialize: ~typing.Callable = <function dumps>, json_deserialize: ~typing.Callable = <function loads>, client_session_args: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None*)

Bases: *AsyncTransport*

Async Transport to execute GraphQL queries on remote servers with an HTTP connection.

This transport use the aiohttp library with asyncio.

file_classes: Tuple[Type[Any], ...] = (<class 'io.IOBase'>, <class 'aiohttp.streams.StreamReader'>, typing.AsyncGenerator)

__init__(*url: str, headers: ~typing.Optional[~typing.Union[~typing.Mapping[~typing.Union[str, ~multidict._multidict.istr], str], ~multidict._multidict.CIMultiDict, ~multidict._multidict.CIMultiDictProxy]] = None, cookies: ~typing.Optional[~typing.Union[~typing.Mapping[str, ~typing.Union[str, BaseCookie[str], Morsel[Any]]], ~typing.Iterable[~typing.Tuple[str, ~typing.Union[str, BaseCookie[str], Morsel[Any]]]], BaseCookie[str]]] = None, auth: ~typing.Optional[~typing.Union[~aiohttp.helpers.BasicAuth, AppSyncAuthentication]] = None, ssl: ~typing.Union[~ssl.SSLContext, bool, ~aiohttp.client_reqrep.Fingerprint] = False, timeout: ~typing.Optional[int] = None, ssl_close_timeout: ~typing.Optional[~typing.Union[int, float]] = 10, json_serialize: ~typing.Callable = <function dumps>, json_deserialize: ~typing.Callable = <function loads>, client_session_args: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None*) → None

Initialize the transport with the given aiohttp parameters.

Parameters

- **url** – The GraphQL server URL. Example: 'https://server.com:PORT/path'.
- **headers** – Dict of HTTP Headers.
- **cookies** – Dict of HTTP cookies.
- **auth** – BasicAuth object to enable Basic HTTP auth if needed Or Appsync Authentication class
- **ssl** – ssl_context of the connection. Use ssl=False to disable encryption
- **ssl_close_timeout** – Timeout in seconds to wait for the ssl connection to close properly
- **json_serialize** – Json serializer callable. By default json.dumps() function
- **json_deserialize** – Json deserializer callable. By default json.loads() function
- **client_session_args** – Dict of extra args passed to aiohttp.ClientSession

async connect() → None

Coroutine which will create an aiohttp ClientSession() as self.session.

Don't call this coroutine directly on the transport, instead use `async with` on the client and this coroutine will be executed to create the session.

Should be cleaned with a call to the close coroutine.

static create_aiohttp_closed_event(*session*) → Event

Work around aiohttp issue that doesn't properly close transports on exit.

See https://github.com/aio-libs/aiohttp/issues/1925#issuecomment-639080209

Returns:
An event that will be set once all transports have been properly closed.

async close() → None

Coroutine which will close the aiohttp session.

Don't call this coroutine directly on the transport, instead use `async with` on the client and this coroutine will be executed when you exit the async context manager.

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *extra_args: Optional[Dict[str, Any]] = None*, *upload_files: bool = False*) → ExecutionResult

Execute the provided document AST against the configured remote server using the current session. This uses the aiohttp library to perform a HTTP POST request asynchronously to the remote server.

Don't call this coroutine directly on the transport, instead use `execute` on a client or a session.

Parameters

- **document** – the parsed GraphQL request
- **variable_values** – An optional Dict of variable values
- **operation_name** – An optional Operation name for the request
- **extra_args** – additional arguments to send to the aiohttp post method
- **upload_files** – Set to True if you want to put files in the variable values

Returns
an ExecutionResult object.

gql.transport.appsync_auth

class gql.transport.appsync_auth.**AppSyncAuthentication**

Bases: ABC

AWS authentication abstract base class

All AWS authentication class should have a `get_headers` method which defines the headers used in the authentication process.

get_auth_url(*url: str*) → str

Returns
a url with base64 encoded headers used to establish a websocket connection to the appsync-realtime-api.

abstract get_headers(*data: Optional[str] = None*, *headers: Optional[Dict[str, Any]] = None*) → Dict[str, Any]

class gql.transport.appsync_auth.**AppSyncApiKeyAuthentication**(*host: str*, *api_key: str*)

Bases: *AppSyncAuthentication*

AWS authentication class using an API key

__init__(*host: str*, *api_key: str*) → None

Parameters

- **host** – the host, something like: XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com
- **api_key** – the API key

get_headers(*data: Optional[str] = None*, *headers: Optional[Dict[str, Any]] = None*) → Dict[str, Any]

get_auth_url(*url: str*) → str

Returns
a url with base64 encoded headers used to establish a websocket connection to the appsync-realtime-api.

class gql.transport.appsync_auth.**AppSyncJWTAuthentication**(*host: str*, *jwt: str*)

Bases: *AppSyncAuthentication*

AWS authentication class using a JWT access token.

It can be used either for:

- Amazon Cognito user pools
- OpenID Connect (OIDC)

`__init__`(*host: str*, *jwt: str*) → None

Parameters

- **host** – the host, something like: XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com
- **jwt** – the JWT Access Token

`get_headers`(*data: Optional[str] = None*, *headers: Optional[Dict[str, Any]] = None*) → Dict[str, Any]

`get_auth_url`(*url: str*) → str

Returns
a url with base64 encoded headers used to establish a websocket connection to the appsync-realtime-api.

`class gql.transport.appsync_auth.AppSyncIAMAuthentication`(*host: str*, *region_name: Optional[str] = None*, *signer: Optional[botocore.auth.BaseSigner] = None*, *request_creator: Optional[Callable[[Dict[str, Any]], botocore.awsrequest.AWSRequest]] = None*, *credentials: Optional[botocore.credentials.Credentials] = None*, *session: Optional[botocore.session.Session] = None*)

Bases: *AppSyncAuthentication*

AWS authentication class using IAM.

Note: There is no need for you to use this class directly, you could instead intantiate the `gql.transport.appsync.AppSyncWebsocketsTransport` without an auth argument.

During initialization, this class will use botocore to attempt to find your IAM credentials, either from environment variables or from your AWS credentials file.

`__init__`(*host: str*, *region_name: Optional[str] = None*, *signer: Optional[botocore.auth.BaseSigner] = None*, *request_creator: Optional[Callable[[Dict[str, Any]], botocore.awsrequest.AWSRequest]] = None*, *credentials: Optional[botocore.credentials.Credentials] = None*, *session: Optional[botocore.session.Session] = None*) → None

Initialize itself, saving the found credentials used to sign the headers later.

if no credentials are found, then a NoCredentialsError is raised.

`get_headers`(*data: Optional[str] = None*, *headers: Optional[Dict[str, Any]] = None*) → Dict[str, Any]

`get_auth_url`(*url: str*) → str

Returns
a url with base64 encoded headers used to establish a websocket connection to the appsync-realtime-api.

gql.transport.appsync_websockets

class gql.transport.appsync_websockets.**AppSyncWebsocketsTransport**(*url: str*, *auth: Optional[AppSyncAuthentication] = None*, *session: Optional[botocore.session.Session] = None*, *ssl: Union[SSLContext, bool] = False*, *connect_timeout: int = 10*, *close_timeout: int = 10*, *ack_timeout: int = 10*, *keep_alive_timeout: Optional[Union[int, float]] = None*, *connect_args: Dict[str, Any] = {}*)

Bases: *WebsocketsTransportBase*

Async Transport used to execute GraphQL subscription on AWS appsync realtime endpoint.

This transport uses asyncio and the websockets library in order to send requests on a websocket connection.

__init__(*url: str*, *auth: Optional[AppSyncAuthentication] = None*, *session: Optional[botocore.session.Session] = None*, *ssl: Union[SSLContext, bool] = False*, *connect_timeout: int = 10*, *close_timeout: int = 10*, *ack_timeout: int = 10*, *keep_alive_timeout: Optional[Union[int, float]] = None*, *connect_args: Dict[str, Any] = {}*) → None

Initialize the transport with the given parameters.

Parameters

- **url** – The GraphQL endpoint URL. Example: https://XXXXXXXXXXXXXXXXXXXXXXXXXX.appsync-api.REGION.amazonaws.com/graphql
- **auth** – Optional AWS authentication class which will provide the necessary headers to be correctly authenticated. If this argument is not provided, then we will try to authenticate using IAM.
- **ssl** – ssl_context of the connection.
- **connect_timeout** – Timeout in seconds for the establishment of the websocket connection. If None is provided this will wait forever.
- **close_timeout** – Timeout in seconds for the close. If None is provided this will wait forever.
- **ack_timeout** – Timeout in seconds to wait for the connection_ack message from the server. If None is provided this will wait forever.
- **keep_alive_timeout** – Optional Timeout in seconds to receive a sign of liveness from the server.
- **connect_args** – Other parameters forwarded to websockets.connect

auth: Optional[*AppSyncAuthentication*]

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *send_stop: Optional[bool] = True*) → AsyncGenerator[ExecutionResult, None]

Send a subscription query and receive the results using a python async generator.

Only subscriptions are supported, queries and mutations are forbidden.

The results are sent as an ExecutionResult object.

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → ExecutionResult

This method is not available.

Only subscriptions are supported on the AWS realtime endpoint.

Raise
AssertionError

async close() → None

Coroutine used to Close an established connection

async connect() → None

Coroutine which will:

- connect to the websocket address
- send the init message
- wait for the connection acknowledge from the server
- create an asyncio task which will be used to receive and parse the websocket answers

Should be cleaned with a call to the close coroutine

async wait_closed() → None

payloads: Dict[str, Any]

payloads is a dict which will contain the payloads received for example with the graphql-ws protocol: 'ping', 'pong', 'connection_ack'

gql.transport.exceptions

exception gql.transport.exceptions.**TransportError**

Bases: Exception

Base class for all the Transport exceptions

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception gql.transport.exceptions.**TransportProtocolError**

Bases: *TransportError*

Transport protocol error.

The answer received from the server does not correspond to the transport protocol.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception gql.transport.exceptions.**TransportServerError**(*message: str*, *code: Optional[int] = None*)

Bases: *TransportError*

The server returned a global error.

This exception will close the transport connection.

__init__(*message: str*, *code: Optional[int] = None*)

code: Optional[int]

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception gql.transport.exceptions.**TransportQueryError**(*msg: str*, *query_id: Optional[int] = None*, *errors: Optional[List[Any]] = None*, *data: Optional[Any] = None*, *extensions: Optional[Any] = None*)

Bases: *TransportError*

The server returned an error for a specific query.

This exception should not close the transport connection.

__init__(*msg: str*, *query_id: Optional[int] = None*, *errors: Optional[List[Any]] = None*, *data: Optional[Any] = None*, *extensions: Optional[Any] = None*)

query_id: Optional[int]

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

errors: Optional[List[Any]]

data: Optional[Any]

extensions: Optional[Any]

exception gql.transport.exceptions.**TransportClosed**

Bases: *TransportError*

Transport is already closed.

This exception is generated when the client is trying to use the transport while the transport was previously closed.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

exception gql.transport.exceptions.**TransportAlreadyConnected**

Bases: *TransportError*

Transport is already connected.

Exception generated when the client is trying to connect to the transport while the transport is already connected.

__init__(**args*, ***kwargs*)

args

with_traceback()

Exception.with_traceback(tb) – set self.__traceback__ to tb and return self.

gql.transport.phoenix_channel_websockets

class gql.transport.phoenix_channel_websockets.**Subscription**(*query_id: int*)

Bases: object

Records listener_id and unsubscribe query_id for a subscription.

__init__(*query_id: int*) → None

class gql.transport.phoenix_channel_websockets.**PhoenixChannelWebsocketsTransport**(*channel_name: str = '__absinthe__:control'*, *heartbeat_interval: float = 30*, **args*, ***kwargs*)

Bases: *WebsocketsTransportBase*

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

__init__(*channel_name: str = '__absinthe__:control'*, *heartbeat_interval: float = 30*, **args*, ***kwargs*) → None

Initialize the transport with the given parameters.

Parameters

- **channel_name** – Channel on the server this transport will join. The default for Absinthe servers is "__absinthe__:control"
- **heartbeat_interval** – Interval in second between each heartbeat messages sent by the client

async close() → None

Coroutine used to Close an established connection

async connect() → None

Coroutine which will:

- connect to the websocket address
- send the init message

- wait for the connection acknowledge from the server
- create an asyncio task which will be used to receive and parse the websocket answers

Should be cleaned with a call to the close coroutine

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → ExecutionResult

Execute the provided document AST against the configured remote server using the current session.

Send a query but close the async generator as soon as we have the first answer.

The result is sent as an ExecutionResult object.

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *send_stop: Optional[bool] = True*) → AsyncGenerator[ExecutionResult, None]

Send a query and receive the results using a python async generator.

The query can be a graphql query, mutation or subscription.

The results are sent as an ExecutionResult object.

async wait_closed() → None

payloads: Dict[str, Any]

payloads is a dict which will contain the payloads received for example with the graphql-ws protocol: 'ping', 'pong', 'connection_ack'

gql.transport.requests

class gql.transport.requests.**RequestsHTTPTransport**(*url: str*, *headers: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None*, *cookies: ~typing.Optional[~typing.Union[~typing.Dict[str, ~typing.Any], ~requests.cookies.RequestsCookieJar]] = None*, *auth: ~typing.Optional[~requests.auth.AuthBase] = None*, *use_json: bool = True*, *timeout: ~typing.Optional[int] = None*, *verify: ~typing.Union[bool, str] = True*, *retries: int = 0*, *method: str = 'POST'*, *retry_backoff_factor: float = 0.1*, *retry_status_forcelist: ~typing.Collection[int] = (429, 500, 502, 503, 504)*, *json_serialize: ~typing.Callable = <function dumps>*, *json_deserialize: ~typing.Callable = <function loads>*, ***kwargs: ~typing.Any*)

Bases: *Transport*

Sync Transport used to execute GraphQL queries on remote servers.

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

file_classes: Tuple[Type[Any], ...] = (<class 'io.IOBase'>,)

__init__(*url: str, headers: ~typing.Optional[~typing.Dict[str, ~typing.Any]] = None, cookies: ~typing.Optional[~typing.Union[~typing.Dict[str, ~typing.Any], ~requests.cookies.RequestsCookieJar]] = None, auth: ~typing.Optional[~requests.auth.AuthBase] = None, use_json: bool = True, timeout: ~typing.Optional[int] = None, verify: ~typing.Union[bool, str] = True, retries: int = 0, method: str = 'POST', retry_backoff_factor: float = 0.1, retry_status_forcelist: ~typing.Collection[int] = (429, 500, 502, 503, 504), json_serialize: ~typing.Callable = <function dumps>, json_deserialize: ~typing.Callable = <function loads>, **kwargs: ~typing.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).
- **retry_backoff_factor** – A backoff factor to apply between attempts after the second try. urllib3 will sleep for: {backoff factor} * (2 ** ({number of previous retries}))
- **retry_status_forcelist** – A set of integer HTTP status codes that we should force a retry on. A retry is initiated if the request method is in allowed_methods and the response status code is in status_forcelist. (Default: [429, 500, 502, 503, 504])
- **json_serialize** – Json serializer callable. By default json.dumps() function
- **json_deserialize** – Json deserializer callable. By default json.loads() function
- **kwargs** – Optional arguments that request takes. These can be seen at the requests source code or the official docs

connect()

Establish a session with the transport.

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *timeout: Optional[int] = None*, *extra_args: Optional[Dict[str, Any]] = None*, *upload_files: bool = False*) → 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).
- **extra_args** – additional arguments to send to the requests post method
- **upload_files** – Set to True if you want to put files in the variable values

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.

execute_batch(*reqs: List[GraphQLRequest]*, *timeout: Optional[int] = None*, *extra_args: Optional[Dict[str, Any]] = None*) → List[ExecutionResult]

Execute multiple GraphQL requests in a batch.

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

Parameters

- **reqs** – GraphQL requests as a list of GraphQLRequest objects.
- **timeout** – Specifies a default timeout for requests (Default: None).
- **extra_args** – additional arguments to send to the requests post method

Returns
A list of results of execution. For every result *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.

close()

Closing the transport by closing the inner session

gql.transport.httpx

class gql.transport.httpx.**HTTPXTransport**(*url: ~typing.Union[str, ~httpx.URL], json_serialize: ~typing.Callable = <function dumps>, json_deserialize: ~typing.Callable = <function loads>, **kwargs*)

Bases: *Transport*, _HTTPXTransport

Sync Transport used to execute GraphQL queries on remote servers.

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

client: Optional[Client] = None

connect()

Establish a session with the transport.

execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *extra_args: Optional[Dict[str, Any]] = None*, *upload_files: bool = False*) → ExecutionResult

Execute GraphQL query.

Execute the provided document AST against the configured remote server. This uses the httpx 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).
- **extra_args** – additional arguments to send to the httpx post method
- **upload_files** – Set to True if you want to put files in the variable values

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.

close()
Closing the transport by closing the inner session

__init__(*url: ~typing.Union[str, ~httpx.URL], json_serialize: ~typing.Callable = <function dumps>, json_deserialize: ~typing.Callable = <function loads>, **kwargs*)
Initialize the transport with the given httpx parameters.

Parameters
- **url** – The GraphQL server URL. Example: 'https://server.com:PORT/path'.
- **json_serialize** – Json serializer callable. By default json.dumps() function.
- **json_deserialize** – Json deserializer callable. By default json.loads() function.
- **kwargs** – Extra args passed to the *httpx* client.

execute_batch(*reqs: List[GraphQLRequest]*, **args*, ***kwargs*) → List[ExecutionResult]
Execute multiple GraphQL requests in a batch.

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

Parameters
reqs – GraphQL requests as a list of GraphQLRequest objects.

Returns
a list of ExecutionResult objects

file_classes: Tuple[Type[Any], ...] = (<class 'io.IOBase'>,)

response_headers: Optional[Headers] = None

class gql.transport.httpx.**HTTPXAsyncTransport**(*url: ~typing.Union[str, ~httpx.URL], json_serialize: ~typing.Callable = <function dumps>, json_deserialize: ~typing.Callable = <function loads>, **kwargs*)

Bases: *AsyncTransport*, _HTTPXTransport

Async Transport used to execute GraphQL queries on remote servers.

The transport uses the httpx library with anyio.

client: Optional[AsyncClient] = None

async connect()
Coroutine used to create a connection to the specified address

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *extra_args: Optional[Dict[str, Any]] = None*, *upload_files: bool = False*) → ExecutionResult

Execute GraphQL query.

Execute the provided document AST against the configured remote server. This uses the httpx library to perform a HTTP POST request asynchronously 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).
- **extra_args** – additional arguments to send to the httpx post method
- **upload_files** – Set to True if you want to put files in the variable values

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.

`async close()`

Closing the transport by closing the inner session

`__init__`(*url: ~typing.Union[str, ~httpx.URL], json_serialize: ~typing.Callable = <function dumps>, json_deserialize: ~typing.Callable = <function loads>, **kwargs*)

Initialize the transport with the given httpx parameters.

Parameters

- **url** – The GraphQL server URL. Example: 'https://server.com:PORT/path'.
- **json_serialize** – Json serializer callable. By default json.dumps() function.
- **json_deserialize** – Json deserializer callable. By default json.loads() function.
- **kwargs** – Extra args passed to the *httpx* client.

`file_classes: Tuple[Type[Any], ...] = (<class 'io.IOBase'>,)`

`response_headers: Optional[Headers] = None`

gql.transport.websockets

`class gql.transport.websockets.WebsocketsTransport`(*url: str*, *headers: Optional[Union[Headers, Mapping[str, str], Iterable[Tuple[str, str]], SupportsKeysAndGetItem]] = None*, *ssl: Union[SSLContext, bool] = False*, *init_payload: Dict[str, Any] = {}*, *connect_timeout: Optional[Union[int, float]] = 10*, *close_timeout: Optional[Union[int, float]] = 10*, *ack_timeout: Optional[Union[int, float]] = 10*, *keep_alive_timeout: Optional[Union[int, float]] = None*, *ping_interval: Optional[Union[int, float]] = None*, *pong_timeout: Optional[Union[int, float]] = None*, *answer_pings: bool = True*, *connect_args: Dict[str, Any] = {}*, *subprotocols: Optional[List[Subprotocol]] = None*)

Bases: *WebsocketsTransportBase*

Async Transport used to execute GraphQL queries on remote servers with websocket connection.

This transport uses asyncio and the websockets library in order to send requests on a websocket connection.

APOLLO_SUBPROTOCOL = 'graphql-ws'

GRAPHQLWS_SUBPROTOCOL = 'graphql-transport-ws'

__init__(*url: str*, *headers: Optional[Union[Headers, Mapping[str, str], Iterable[Tuple[str, str]], SupportsKeysAndGetItem]] = None*, *ssl: Union[SSLContext, bool] = False*, *init_payload: Dict[str, Any] = {}*, *connect_timeout: Optional[Union[int, float]] = 10*, *close_timeout: Optional[Union[int, float]] = 10*, *ack_timeout: Optional[Union[int, float]] = 10*, *keep_alive_timeout: Optional[Union[int, float]] = None*, *ping_interval: Optional[Union[int, float]] = None*, *pong_timeout: Optional[Union[int, float]] = None*, *answer_pings: bool = True*, *connect_args: Dict[str, Any] = {}*, *subprotocols: Optional[List[Subprotocol]] = None*) → None

Initialize the transport with the given parameters.

Parameters

- **url** – The GraphQL server URL. Example: 'wss://server.com:PORT/graphql'.
- **headers** – Dict of HTTP Headers.
- **ssl** – ssl_context of the connection. Use ssl=False to disable encryption
- **init_payload** – Dict of the payload sent in the connection_init message.
- **connect_timeout** – Timeout in seconds for the establishment of the websocket connection. If None is provided this will wait forever.
- **close_timeout** – Timeout in seconds for the close. If None is provided this will wait forever.
- **ack_timeout** – Timeout in seconds to wait for the connection_ack message from the server. If None is provided this will wait forever.
- **keep_alive_timeout** – Optional Timeout in seconds to receive a sign of liveness from the server.
- **ping_interval** – Delay in seconds between pings sent by the client to the backend for the graphql-ws protocol. None (by default) means that we don't send pings. Note: there are also pings sent by the underlying websockets protocol. See the *keepalive documentation* for more information about this.
- **pong_timeout** – Delay in seconds to receive a pong from the backend after we sent a ping (only for the graphql-ws protocol). By default equal to half of the ping_interval.
- **answer_pings** – Whether the client answers the pings from the backend (for the graphql-ws protocol). By default: True
- **connect_args** – Other parameters forwarded to websockets.connect
- **subprotocols** – list of subprotocols sent to the backend in the 'subprotocols' http header. By default: both apollo and graphql-ws subprotocols.

ping_received: asyncio.Event

ping_received is an asyncio Event which will fire each time a ping is received with the graphql-ws protocol

pong_received: asyncio.Event

pong_received is an asyncio Event which will fire each time a pong is received with the graphql-ws protocol

async send_ping(*payload: Optional[Any] = None*) → None

Send a ping message for the graphql-ws protocol

async send_pong(*payload: Optional[Any] = None*) → None

Send a pong message for the graphql-ws protocol

async close() → None

Coroutine used to Close an established connection

async connect() → None

Coroutine which will:

- connect to the websocket address
- send the init message
- wait for the connection acknowledge from the server
- create an asyncio task which will be used to receive and parse the websocket answers

Should be cleaned with a call to the close coroutine

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → ExecutionResult

Execute the provided document AST against the configured remote server using the current session.

Send a query but close the async generator as soon as we have the first answer.

The result is sent as an ExecutionResult object.

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *send_stop: Optional[bool] = True*) → AsyncGenerator[ExecutionResult, None]

Send a query and receive the results using a python async generator.

The query can be a graphql query, mutation or subscription.

The results are sent as an ExecutionResult object.

async wait_closed() → None

payloads: Dict[str, Any]

payloads is a dict which will contain the payloads received for example with the graphql-ws protocol: 'ping', 'pong', 'connection_ack'

gql.transport.websockets_base

class gql.transport.websockets_base.**ListenerQueue**(*query_id: int*, *send_stop: bool*)

Bases: object

Special queue used for each query waiting for server answers

If the server is stopped while the listener is still waiting, Then we send an exception to the queue and this exception will be raised to the consumer once all the previous messages have been consumed from the queue

__init__(*query_id: int*, *send_stop: bool*) → None

async get() → Tuple[str, Optional[ExecutionResult]]

async put(*item: Tuple[str, Optional[ExecutionResult]]*) → None

async set_exception(*exception: Exception*) → None

class gql.transport.websockets_base.**WebsocketsTransportBase**(*url: str*, *headers: Optional[Union[Headers, Mapping[str, str], Iterable[Tuple[str, str]], SupportsKeysAndGetItem]] = None*, *ssl: Union[SSLContext, bool] = False*, *init_payload: Dict[str, Any] = {}*, *connect_timeout: Optional[Union[int, float]] = 10*, *close_timeout: Optional[Union[int, float]] = 10*, *ack_timeout: Optional[Union[int, float]] = 10*, *keep_alive_timeout: Optional[Union[int, float]] = None*, *connect_args: Dict[str, Any] = {}*)

Bases: AsyncTransport

abstract *Async Transport* used to implement different websockets protocols.

This transport uses asyncio and the websockets library in order to send requests on a websocket connection.

__init__(*url: str*, *headers: Optional[Union[Headers, Mapping[str, str], Iterable[Tuple[str, str]], SupportsKeysAndGetItem]] = None*, *ssl: Union[SSLContext, bool] = False*, *init_payload: Dict[str, Any] = {}*, *connect_timeout: Optional[Union[int, float]] = 10*, *close_timeout: Optional[Union[int, float]] = 10*, *ack_timeout: Optional[Union[int, float]] = 10*, *keep_alive_timeout: Optional[Union[int, float]] = None*, *connect_args: Dict[str, Any] = {}*) → None

Initialize the transport with the given parameters.

Parameters

- **url** – The GraphQL server URL. Example: 'wss://server.com:PORT/graphql'.
- **headers** – Dict of HTTP Headers.
- **ssl** – ssl_context of the connection. Use ssl=False to disable encryption
- **init_payload** – Dict of the payload sent in the connection_init message.
- **connect_timeout** – Timeout in seconds for the establishment of the websocket connection. If None is provided this will wait forever.
- **close_timeout** – Timeout in seconds for the close. If None is provided this will wait forever.
- **ack_timeout** – Timeout in seconds to wait for the connection_ack message from the server. If None is provided this will wait forever.
- **keep_alive_timeout** – Optional Timeout in seconds to receive a sign of liveness from the server.
- **connect_args** – Other parameters forwarded to websockets.connect

payloads: Dict[str, Any]

payloads is a dict which will contain the payloads received for example with the graphql-ws protocol: 'ping', 'pong', 'connection_ack'

async subscribe(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*, *send_stop: Optional[bool] = True*) → AsyncGenerator[ExecutionResult, None]

Send a query and receive the results using a python async generator.

The query can be a graphql query, mutation or subscription.

The results are sent as an ExecutionResult object.

async execute(*document: DocumentNode*, *variable_values: Optional[Dict[str, Any]] = None*, *operation_name: Optional[str] = None*) → ExecutionResult

Execute the provided document AST against the configured remote server using the current session.

Send a query but close the async generator as soon as we have the first answer.

The result is sent as an ExecutionResult object.

async connect() → None

Coroutine which will:

- connect to the websocket address
- send the init message
- wait for the connection acknowledge from the server
- create an asyncio task which will be used to receive and parse the websocket answers

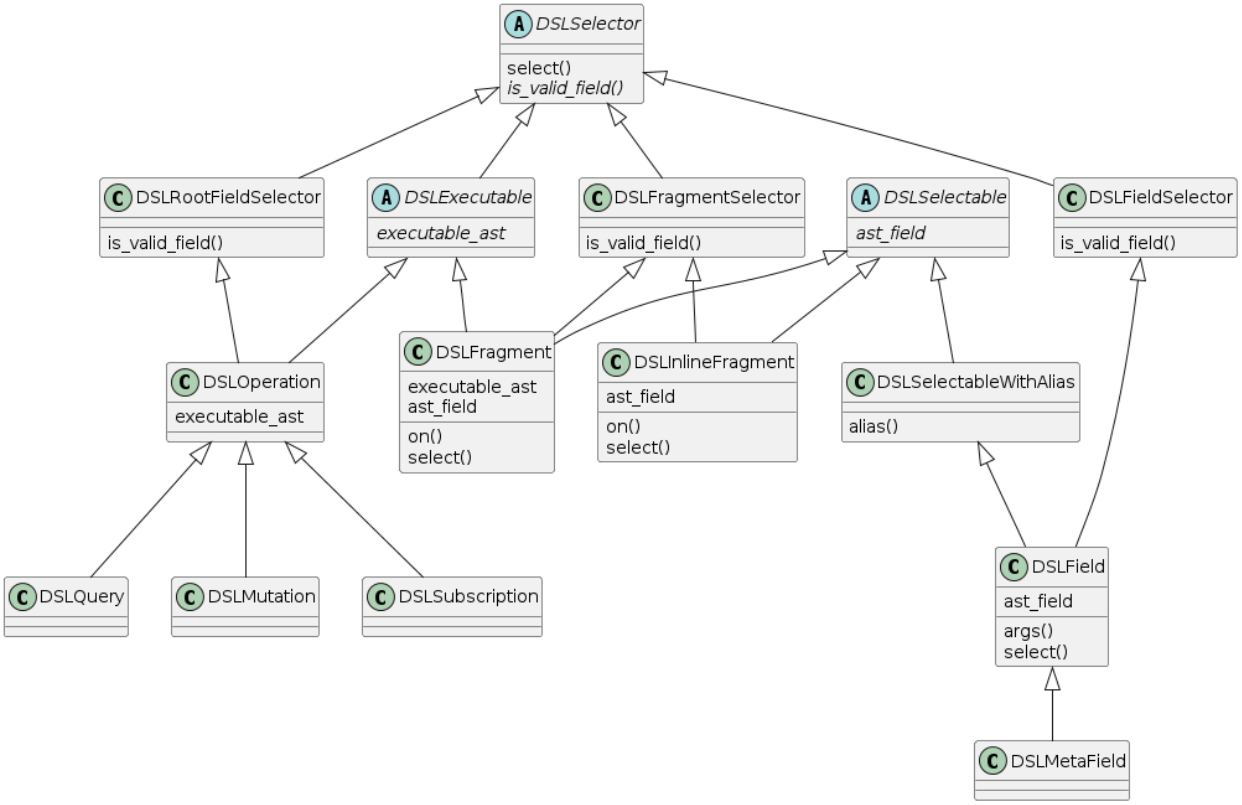
Should be cleaned with a call to the close coroutine

async close() → None

Coroutine used to Close an established connection

async wait_closed() → None

gql.dsl



gql.dsl.**ast_from_serialized_value_untyped**(*serialized: Any*) → Optional[ValueNode]

Given a serialized value, try our best to produce an AST.

Anything ressembling an array (instance of Mapping) will be converted to an ObjectFieldNode.

Anything ressembling a list (instance of Iterable - except str) will be converted to a ListNode.

In some cases, a custom scalar can be serialized differently in the query than in the variables. In that case, this function will not work.

gql.dsl.**ast_from_value**(*value: Any*, *type_: Union[GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull[Union[GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList]]]*) → Optional[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.

Raises a GraphQLError instead of returning None if we receive an Undefined of if we receive a Null value for a Non-Null type.

gql.dsl.**dsl_gql**(**operations:* DSLExecutable, ***operations_with_name:* DSLExecutable) → DocumentNode

Given arguments instances of `DSLExecutable` containing GraphQL operations or fragments, 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** (DSLQuery, DSLMutation, DSLSubscription, DSLFragment) – the GraphQL operations and fragments
- ****operations_with_name** (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 *DSLExecutable*
- **AttributeError** – if a type has not been provided in a *DSLFragment*

class gql.dsl.**DSLSchema**(*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: 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.**DSLSelector**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: ABC

DSLSelector is an abstract class which defines the *select* method to select children fields in the query.

Inherited by *DSLRootFieldSelector*, *DSLFieldSelector* *DSLFragmentSelector*

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

selection_set: SelectionSetNode

abstract is_valid_field(*field:* DSLSelectable) → bool

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

class gql.dsl.**DSLExecutable**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: *DSLSelector*

Interface for the root elements which can be executed in the *dsl_gql* function

Inherited by *DSLOperation* and *DSLFragment*

selection_set: SelectionSetNode

abstract property executable_ast

Generates the ast for *dsl_gql*.

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

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

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

Parameters

- ***fields** (DSLSelectable) – root fields or fragments
- ****fields_with_alias** (DSLSelectable) – root fields or fragments with alias as key

Raises

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

name: Optional[str]

variable_definitions: *DSLVariableDefinitions*

abstract is_valid_field(*field:* DSLSelectable) → bool

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

class gql.dsl.**DSLRootFieldSelector**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: *DSLSelector*

Class used to define the *is_valid_field* method for root fields for the *select* method.

Inherited by *DSLOperation*

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid for a root field.

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

the `__typename` field can only be added to Query or Mutation. the `__schema` and `__type` field can only be added to Query.

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

selection_set: SelectionSetNode

class gql.dsl.**DSLOperation**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: *DSLExecutable*, *DSLRootFieldSelector*

Interface for GraphQL operations.

Inherited by *DSLQuery*, *DSLMutation* and *DSLSubscription*

operation_type: OperationType

property executable_ast: OperationDefinitionNode

Generates the ast for *dsl_gql*.

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

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

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

Parameters

- ***fields** (DSLSelectable) – root fields or fragments
- ****fields_with_alias** (DSLSelectable) – root fields or fragments with alias as key

Raises

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

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid for a root field.

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

the `__typename` field can only be added to Query or Mutation. the `__schema` and `__type` field can only be added to Query.

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

variable_definitions: *DSLVariableDefinitions*

name: Optional[str]

selection_set: SelectionSetNode

class gql.dsl.**DSLQuery**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: *DSLOperation*

operation_type: OperationType = 'query'

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

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

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

Parameters

- ***fields** (DSLSelectable) – root fields or fragments
- ****fields_with_alias** (DSLSelectable) – root fields or fragments with alias as key

Raises

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

property executable_ast: OperationDefinitionNode

Generates the ast for *dsl_gql*.

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid for a root field.

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

the `__typename` field can only be added to Query or Mutation. the `__schema` and `__type` field can only be added to Query.

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

variable_definitions: *DSLVariableDefinitions*

name: Optional[str]

selection_set: SelectionSetNode

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

Bases: *DSLOperation*

operation_type: OperationType = 'mutation'

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

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

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

Parameters

- ***fields** (DSLSelectable) – root fields or fragments
- ****fields_with_alias** (DSLSelectable) – root fields or fragments with alias as key

Raises

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

property executable_ast: OperationDefinitionNode

Generates the ast for *dsl_gql*.

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid for a root field.

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

the `__typename` field can only be added to Query or Mutation. the `__schema` and `__type` field can only be added to Query.

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

variable_definitions: *DSLVariableDefinitions*

name: Optional[str]

selection_set: SelectionSetNode

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

Bases: *DSLOperation*

operation_type: OperationType = 'subscription'

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

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

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

Parameters

- ***fields** (DSLSelectable) – root fields or fragments
- ****fields_with_alias** (DSLSelectable) – root fields or fragments with alias as key

Raises

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

property executable_ast: OperationDefinitionNode

Generates the ast for *dsl_gql*.

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid for a root field.

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

the `__typename` field can only be added to Query or Mutation. the `__schema` and `__type` field can only be added to Query.

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

variable_definitions: *DSLVariableDefinitions*

name: Optional[str]

selection_set: 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*)

to_ast_type(*type_: Union[GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull[Union[GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList]]]*) → TypeNode

set_type(*type_: Union[GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList, GraphQLNonNull[Union[GraphQLScalarType, GraphQLEnumType, GraphQLInputObjectType, GraphQLList]]]*) → *DSLVariable*

default(*default_value: Any*) → *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 *args* method.

__init__()

class gql.dsl.**DSLType**(*graphql_type: Union[GraphQLObjectType, GraphQLInterfaceType]*, *dsl_schema:* DSLSchema)

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[GraphQLObjectType, GraphQLInterfaceType]*, *dsl_schema:* DSLSchema)

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
- **dsl_schema** – reference to the DSLSchema which created this type

class gql.dsl.**DSLSelectable**

Bases: ABC

DSLSelectable is an abstract class which indicates that the subclasses can be used as arguments of the *select* method.

Inherited by *DSLField*, *DSLFragment* *DSLInlineFragment*

ast_field: Union[FieldNode, InlineFragmentNode, FragmentSpreadNode]

class gql.dsl.**DSLFragmentSelector**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: *DSLSelector*

Class used to define the *is_valid_field* method for fragments for the *select* method.

Inherited by *DSLFragment*, *DSLInlineFragment*

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid.

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments
- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

selection_set: SelectionSetNode

class gql.dsl.**DSLFieldSelector**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: *DSLSelector*

Class used to define the *is_valid_field* method for fields for the *select* method.

Inherited by *DSLField*,

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid.

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Select the fields which should be added.

Parameters

- ***fields** (DSLSelectable) – fields or fragments

- ****fields_with_alias** (DSLSelectable) – fields or fragments with alias as key

Raises

- **TypeError** – if an argument is not an instance of *DSLSelectable*
- **GraphQLError** – if an argument is not a valid field

selection_set: SelectionSetNode

class gql.dsl.**DSLSelectableWithAlias**

Bases: *DSLSelectable*

DSLSelectableWithAlias is an abstract class which indicates that the subclasses can be selected with an alias.

ast_field: FieldNode

alias(*alias: str*) → *DSLSelectableWithAlias*

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

class gql.dsl.**DSLField**(*name: str*, *parent_type: Union[GraphQLObjectType, GraphQLInterfaceType]*, *field: GraphQLField*, *dsl_type: Optional[*DSLType*] = None*)

Bases: *DSLSelectableWithAlias*, *DSLFieldSelector*

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*, *parent_type: Union[GraphQLObjectType, GraphQLInterfaceType]*, *field: GraphQLField*, *dsl_type: Optional[*DSLType*] = None*)

Initialize the DSLField.

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

Parameters

- **name** – the name of the field
- **parent_type** – the GraphQL type definition from the schema of the parent type of the field
- **field** – the GraphQL field definition from the schema
- **dsl_type** – reference of the DSLType instance which created this field

field: GraphQLField

ast_field: FieldNode

args(***kwargs*) → *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.

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias) → *DSLField*

Calling *select* method with corrected typing hints

alias(*alias: str*) → *DSLSelectableWithAlias*

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

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid.

selection_set: SelectionSetNode

class gql.dsl.**DSLMetaField**(*name: str*)

Bases: *DSLField*

DSLMetaField represents a GraphQL meta-field for the DSL code.

meta-fields are reserved field in the GraphQL type system prefixed with "__" two underscores and used for introspection.

meta_type = <GraphQLObjectType 'meta_field'>

__init__(*name: str*)

Initialize the meta-field.

Parameters
name – the name between __typename, __schema or __type

alias(*alias: str*) → *DSLSelectableWithAlias*

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*) → *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.

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid.

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias) → *DSLField*

Calling *select* method with corrected typing hints

ast_field: FieldNode

field: GraphQLField

selection_set: SelectionSetNode

class gql.dsl.**DSLInlineFragment**(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Bases: *DSLSelectable*, *DSLFragmentSelector*

DSLInlineFragment represents an inline fragment for the DSL code.

__init__(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias)

Initialize the DSLInlineFragment.

Parameters

- ***fields** (DSLSelectable (DSLField, DSLFragment or DSLInlineFragment)) – new children fields
- ****fields_with_alias** (DSLField) – new children fields with alias as key

ast_field: InlineFragmentNode

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias) → *DSLInlineFragment*

Calling *select* method with corrected typing hints

on(*type_condition:* DSLType) → *DSLInlineFragment*

Provides the GraphQL type of this inline fragment.

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid.

selection_set: SelectionSetNode

class gql.dsl.**DSLFragment**(*name: str*)

Bases: *DSLSelectable*, *DSLFragmentSelector*, *DSLExecutable*

DSLFragment represents a named GraphQL fragment for the DSL code.

is_valid_field(*field:* DSLSelectable) → bool

Check that a field is valid.

selection_set: SelectionSetNode

variable_definitions: *DSLVariableDefinitions*

__init__(*name: str*)

Initialize the DSLFragment.

Parameters

name (*str*) – the name of the fragment

name: str

property ast_field: FragmentSpreadNode

ast_field property will generate a FragmentSpreadNode with the provided name.

Note: We need to ignore the type because of issue #4125 of mypy.

select(**fields:* DSLSelectable, ***fields_with_alias:* DSLSelectableWithAlias) → *DSLFragment*

Calling *select* method with corrected typing hints

on(*type_condition:* DSLType) → *DSLFragment*

Provides the GraphQL type of this fragment.

Parameters

type_condition (DSLType) – the provided type

property executable_ast: FragmentDefinitionNode

Generates the ast for *dsl_gql*.

Raises

AttributeError – if a type has not been provided

gql.utilities

gql.utilities.**build_client_schema**(*introspection: IntrospectionQuery*) → GraphQLSchema

This is an alternative to the graphql-core function `build_client_schema` but with default include and skip directives added to the schema to fix issue #278

> **Warning:** This function will be removed once the issue graphql-js#3419 has been fixed and ported to graphql-core so don't use it outside gql.

gql.utilities.**get_introspection_query_ast**(*descriptions: bool = True*, *specified_by_url: bool = False*, *directive_is_repeatable: bool = False*, *schema_description: bool = False*, *type_recursion_level: int = 7*) → DocumentNode

Get a query for introspection as a document using the DSL module.

Equivalent to the get_introspection_query function from graphql-core but using the DSL module and allowing to select the recursion level.

Optionally, you can exclude descriptions, include specification URLs, include repeatability of directives, and specify whether to include the schema description as well.

gql.utilities.**node_tree**(*obj: Node*, ***, *ignore_loc: bool = True*, *ignore_block: bool = True*, *ignored_keys: Optional[List] = None*)

Method which returns a tree of Node elements as a String.

Useful to debug deep DocumentNode instances created by gql or dsl_gql.

WARNING: the output of this method is not guaranteed and may change without notice.

gql.utilities.**parse_result**(*schema: GraphQLSchema*, *document: DocumentNode*, *result: Optional[Dict[str, Any]]*, *operation_name: Optional[str] = None*) → Optional[Dict[str, Any]]

Unserialize a result received from a GraphQL backend.

Parameters

- **schema** – the GraphQL schema
- **document** – the document representing the query sent to the backend
- **result** – the serialized result received from the backend
- **operation_name** – the optional operation name

Returns

a parsed result with scalars and enums parsed depending on their definition in the schema.

Given a schema, a query and a serialized result, provide a new result with parsed values.

If the result contains only built-in GraphQL scalars (String, Int, Float, ...) then the parsed result should be unchanged.

If the result contains custom scalars or enums, then those values will be parsed with the parse_value method of the custom scalar or enum definition in the schema.

gql.utilities.**serialize_value**(*type_: GraphQLType*, *value: Any*) → Any

Given a GraphQL type and a Python value, return the serialized value.

This method will serialize the value recursively, entering into lists and dicts.

Can be used to serialize Enums and/or Custom Scalars in variable values.

Parameters

- **type** – the GraphQL type
- **value** – the provided value

gql.utilities.**serialize_variable_values**(*schema: GraphQLSchema*, *document: DocumentNode*, *variable_values: Dict[str, Any]*, *operation_name: Optional[str] = None*) → Dict[str, Any]

Given a GraphQL document and a schema, serialize the Dictionary of variable values.

Useful to serialize Enums and/or Custom Scalars in variable values.

Parameters

- **schema** – the GraphQL schema
- **document** – the document representing the query sent to the backend
- **variable_values** – the dictionnary of variable values which needs to be serialized.
- **operation_name** – the optional operation_name for the query.

gql.utilities.**update_schema_enum**(*schema: GraphQLSchema*, *name: str*, *values: Union[Dict[str, Any], Type[Enum]]*, *use_enum_values: bool = False*)

Update in the schema the GraphQLEnumType corresponding to the given name.

Example:

```
from enum import Enum

class Color(Enum):
    RED = 0
    GREEN = 1
    BLUE = 2

update_schema_enum(schema, 'Color', Color)
```

Parameters

- **schema** – a GraphQL Schema already containing the GraphQLEnumType type.
- **name** – the name of the enum in the GraphQL schema
- **values** – Either a Python Enum or a dict of values. The keys of the provided values should correspond to the keys of the existing enum in the schema.
- **use_enum_values** – By default, we configure the GraphQLEnumType to serialize to enum instances (ie: .parse_value() returns Color.RED). If use_enum_values is set to True, then .parse_value() returns 0. use_enum_values=True is the defaut behaviour when passing an Enum to a GraphQLEnumType.

gql.utilities.**update_schema_scalar**(*schema: GraphQLSchema*, *name: str*, *scalar: GraphQLScalarType*)

Update the scalar in a schema with the scalar provided.

Parameters

- **schema** – the GraphQL schema
- **name** – the name of the custom scalar type in the schema
- **scalar** – a provided scalar type

This can be used to update the default Custom Scalar implementation when the schema has been provided from a text file or from introspection.

gql.utilities.**update_schema_scalars**(*schema: GraphQLSchema*, *scalars: List[GraphQLScalarType]*)

Update the scalars in a schema with the scalars provided.

Parameters

- **schema** – the GraphQL schema
- **scalars** – a list of provided scalar types

This can be used to update the default Custom Scalar implementation when the schema has been provided from a text file or from introspection.

If the name of the provided scalar is different than the name of the custom scalar, then you should use the *update_schema_scalar* method instead.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

PYTHON MODULE INDEX

g

gql, 50
gql.client, 53
gql.dsl, 79
gql.transport.aiohttp, 62
gql.transport.appsync_auth, 64
gql.transport.appsync_websockets, 66
gql.transport.exceptions, 67
gql.transport.httpx, 72
gql.transport.phoenix_channel_websockets, 69
gql.transport.requests, 70
gql.transport.websockets, 74
gql.transport.websockets_base, 76
gql.utilities, 92

INDEX

Symbols

__init__() (*gql.Client method*), 50
__init__() (*gql.GraphQLRequest method*), 53
__init__() (*gql.client.AsyncClientSession method*), 53
__init__() (*gql.client.Client method*), 55
__init__() (*gql.client.ReconnectingAsyncClientSession method*), 58
__init__() (*gql.client.SyncClientSession method*), 59
__init__() (*gql.dsl.DSLExecutable method*), 81
__init__() (*gql.dsl.DSLField method*), 88
__init__() (*gql.dsl.DSLFieldSelector method*), 87
__init__() (*gql.dsl.DSLFragment method*), 91
__init__() (*gql.dsl.DSLFragmentSelector method*), 87
__init__() (*gql.dsl.DSLInlineFragment method*), 90
__init__() (*gql.dsl.DSLMetaField method*), 89
__init__() (*gql.dsl.DSLMutation method*), 84
__init__() (*gql.dsl.DSLOperation method*), 82
__init__() (*gql.dsl.DSLQuery method*), 83
__init__() (*gql.dsl.DSLRootFieldSelector method*), 82
__init__() (*gql.dsl.DSLSchema method*), 80
__init__() (*gql.dsl.DSLSelector method*), 80
__init__() (*gql.dsl.DSLSubscription method*), 85
__init__() (*gql.dsl.DSLType method*), 86
__init__() (*gql.dsl.DSLVariable method*), 86
__init__() (*gql.dsl.DSLVariableDefinitions method*), 86
__init__() (*gql.transport.aiohttp.AIOHTTPTransport method*), 62
__init__() (*gql.transport.appsync_auth.AppSyncApiKeyAuthentication method*), 64
__init__() (*gql.transport.appsync_auth.AppSyncIAMAuthentication method*), 65
__init__() (*gql.transport.appsync_auth.AppSyncJWTAuthentication method*), 65
__init__() (*gql.transport.appsync_websockets.AppSyncWebsocketsTransport method*), 66
__init__() (*gql.transport.exceptions.TransportAlreadyConnected method*), 69
__init__() (*gql.transport.exceptions.TransportClosed method*), 68
__init__() (*gql.transport.exceptions.TransportError method*), 67
__init__() (*gql.transport.exceptions.TransportProtocolError method*), 67
__init__() (*gql.transport.exceptions.TransportQueryError method*), 68
__init__() (*gql.transport.exceptions.TransportServerError method*), 68
__init__() (*gql.transport.httpx.HTTPXAsyncTransport method*), 74
__init__() (*gql.transport.httpx.HTTPXTransport method*), 73
__init__() (*gql.transport.local_schema.LocalSchemaTransport method*), 61
__init__() (*gql.transport.phoenix_channel_websockets.PhoenixChannelW method*), 69
__init__() (*gql.transport.phoenix_channel_websockets.Subscription method*), 69
__init__() (*gql.transport.requests.RequestsHTTPTransport method*), 70
__init__() (*gql.transport.websockets.WebsocketsTransport method*), 75
__init__() (*gql.transport.websockets_base.ListenerQueue method*), 76
__init__() (*gql.transport.websockets_base.WebsocketsTransportBase method*), 77

A

AIOHTTPTransport (*class in gql.transport.aiohttp*), 62
alias() (*gql.dsl.DSLField method*), 89
alias() (*gql.dsl.DSLMetaField method*), 90
alias() (*gql.dsl.DSLSelectableWithAlias method*), 88
APOLLO_SUBPROTOCOL (*gql.transport.websockets.WebsocketsTransport attribute*), 75
AppSyncApiKeyAuthentication (*class in gql.transport.appsync_auth*), 64
AppSyncAuthentication (*class in gql.transport.appsync_auth*), 64
AppSyncIAMAuthentication (*class in gql.transport.appsync_auth*), 65
AppSyncJWTAuthentication (*class in gql.transport.appsync_auth*), 64
AppSyncWebsocketsTransport (*class in gql.transport.appsync_websockets*), 66

args (*gql.transport.exceptions.TransportAlreadyConnected attribute*), 69
args (*gql.transport.exceptions.TransportClosed attribute*), 68
args (*gql.transport.exceptions.TransportError attribute*), 67
args (*gql.transport.exceptions.TransportProtocolError attribute*), 67
args (*gql.transport.exceptions.TransportQueryError attribute*), 68
args (*gql.transport.exceptions.TransportServerError attribute*), 68
args() (*gql.dsl.DSLField method*), 89
args() (*gql.dsl.DSLMetaField method*), 90
ast_field (*gql.dsl.DSLField attribute*), 89
ast_field (*gql.dsl.DSLFragment property*), 91
ast_field (*gql.dsl.DSLInlineFragment attribute*), 91
ast_field (*gql.dsl.DSLMetaField attribute*), 90
ast_field (*gql.dsl.DSLSelectable attribute*), 87
ast_field (*gql.dsl.DSLSelectableWithAlias attribute*), 88
ast_from_serialized_value_untyped() (*in module gql.dsl*), 79
ast_from_value() (*in module gql.dsl*), 79
AsyncClientSession (*class in gql.client*), 53
AsyncTransport (*class in gql.transport.async_transport*), 61
auth (*gql.transport.appsync_websockets.AppSyncWebsocketsTransport attribute*), 66

B

batching_enabled (*gql.Client property*), 51
batching_enabled (*gql.client.Client property*), 56
build_client_schema() (*in module gql.utilities*), 92

C

Client (*class in gql*), 50
Client (*class in gql.client*), 55
client (*gql.transport.httpx.HTTPXAsyncTransport attribute*), 73
client (*gql.transport.httpx.HTTPXTransport attribute*), 72
close() (*gql.client.SyncClientSession method*), 59
close() (*gql.transport.aiohttp.AIOHTTPTransport method*), 63
close() (*gql.transport.appsync_websockets.AppSyncWebsocketsTransport method*), 67
close() (*gql.transport.async_transport.AsyncTransport method*), 61
close() (*gql.transport.httpx.HTTPXAsyncTransport method*), 74
close() (*gql.transport.httpx.HTTPXTransport method*), 73
close() (*gql.transport.local_schema.LocalSchemaTransport method*), 62
close() (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebs method*), 69
close() (*gql.transport.requests.RequestsHTTPTransport method*), 72
close() (*gql.transport.transport.Transport method*), 61
close() (*gql.transport.websockets.WebsocketsTransport method*), 76
close() (*gql.transport.websockets_base.WebsocketsTransportBase method*), 78
close_async() (*gql.Client method*), 51
close_async() (*gql.client.Client method*), 56
close_sync() (*gql.Client method*), 51
close_sync() (*gql.client.Client method*), 56
code (*gql.transport.exceptions.TransportServerError attribute*), 68
connect() (*gql.client.SyncClientSession method*), 59
connect() (*gql.transport.aiohttp.AIOHTTPTransport method*), 63
connect() (*gql.transport.appsync_websockets.AppSyncWebsocketsTranspo method*), 67
connect() (*gql.transport.async_transport.AsyncTransport method*), 61
connect() (*gql.transport.httpx.HTTPXAsyncTransport method*), 73
connect() (*gql.transport.httpx.HTTPXTransport method*), 72
connect() (*gql.transport.local_schema.LocalSchemaTransport method*), 62
connect() (*gql.transport.phoenix_channel_websockets.PhoenixChannelW method*), 69
connect() (*gql.transport.requests.RequestsHTTPTransport method*), 71
connect() (*gql.transport.transport.Transport method*), 61
connect() (*gql.transport.websockets.WebsocketsTransport method*), 76
connect() (*gql.transport.websockets_base.WebsocketsTransportBase method*), 78
connect_async() (*gql.Client method*), 51
connect_async() (*gql.client.Client method*), 56
connect_sync() (*gql.Client method*), 51
connect_sync() (*gql.client.Client method*), 56
create_aiohttp_closed_event() (*gql.transport.aiohttp.AIOHTTPTransport static method*), 63

D

data (*gql.transport.exceptions.TransportQueryError attribute*), 68
default() (*gql.dsl.DSLVariable method*), 86
document (*gql.GraphQLRequest attribute*), 53
dsl_gql() (*in module gql.dsl*), 79

DSLExecutable (*class in gql.dsl*), 81
DSLField (*class in gql.dsl*), 88
DSLFieldSelector (*class in gql.dsl*), 87
DSLFragment (*class in gql.dsl*), 91
DSLFragmentSelector (*class in gql.dsl*), 87
DSLInlineFragment (*class in gql.dsl*), 90
DSLMetaField (*class in gql.dsl*), 89
DSLMutation (*class in gql.dsl*), 84
DSLOperation (*class in gql.dsl*), 82
DSLQuery (*class in gql.dsl*), 83
DSLRootFieldSelector (*class in gql.dsl*), 81
DSLSchema (*class in gql.dsl*), 80
DSLSelectable (*class in gql.dsl*), 87
DSLSelectableWithAlias (*class in gql.dsl*), 88
DSLSelector (*class in gql.dsl*), 80
DSLSubscription (*class in gql.dsl*), 85
DSLType (*class in gql.dsl*), 86
DSLVariable (*class in gql.dsl*), 86
DSLVariableDefinitions (*class in gql.dsl*), 86

E

errors (*gql.transport.exceptions.TransportQueryError attribute*), 68
executable_ast (*gql.dsl.DSLExecutable property*), 81
executable_ast (*gql.dsl.DSLFragment property*), 91
executable_ast (*gql.dsl.DSLMutation property*), 84
executable_ast (*gql.dsl.DSLOperation property*), 82
executable_ast (*gql.dsl.DSLQuery property*), 83
executable_ast (*gql.dsl.DSLSubscription property*), 85
execute() (*gql.Client method*), 51
execute() (*gql.client.AsyncClientSession method*), 53
execute() (*gql.client.Client method*), 56
execute() (*gql.client.ReconnectingAsyncClientSession method*), 58
execute() (*gql.client.SyncClientSession method*), 59
execute() (*gql.transport.aiohttp.AIOHTTPTransport method*), 63
execute() (*gql.transport.appsync_websockets.AppSyncWebsocketsTransport method*), 67
execute() (*gql.transport.async_transport.AsyncTransport method*), 61
execute() (*gql.transport.httpx.HTTPXAsyncTransport method*), 73
execute() (*gql.transport.httpx.HTTPXTransport method*), 72
execute() (*gql.transport.local_schema.LocalSchemaTransport method*), 62
execute() (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport method*), 70
execute() (*gql.transport.requests.RequestsHTTPTransport method*), 71
execute() (*gql.transport.transport.Transport method*), 61
execute() (*gql.transport.websockets.WebsocketsTransport method*), 76
execute() (*gql.transport.websockets_base.WebsocketsTransportBase method*), 78
execute_batch() (*gql.Client method*), 52
execute_batch() (*gql.client.Client method*), 57
execute_batch() (*gql.client.SyncClientSession method*), 60
execute_batch() (*gql.transport.httpx.HTTPXTransport method*), 73
execute_batch() (*gql.transport.requests.RequestsHTTPTransport method*), 72
execute_batch() (*gql.transport.transport.Transport method*), 61
extensions (*gql.transport.exceptions.TransportQueryError attribute*), 68

F

fetch_schema() (*gql.client.AsyncClientSession method*), 54
fetch_schema() (*gql.client.ReconnectingAsyncClientSession method*), 58
fetch_schema() (*gql.client.SyncClientSession method*), 60
field (*gql.dsl.DSLField attribute*), 88
field (*gql.dsl.DSLMetaField attribute*), 90
file_classes (*gql.transport.aiohttp.AIOHTTPTransport attribute*), 62
file_classes (*gql.transport.httpx.HTTPXAsyncTransport attribute*), 74
file_classes (*gql.transport.httpx.HTTPXTransport attribute*), 73
file_classes (*gql.transport.requests.RequestsHTTPTransport attribute*), 70

G

get() (*gql.transport.websockets_base.ListenerQueue method*), 76
get_auth_url() (*gql.transport.appsync_auth.AppSyncApiKeyAuthentication method*), 64
get_auth_url() (*gql.transport.appsync_auth.AppSyncAuthentication method*), 64
get_auth_url() (*gql.transport.appsync_auth.AppSyncIAMAuthentication method*), 65
get_auth_url() (*gql.transport.appsync_auth.AppSyncJWTAuthentication method*), 65
get_headers() (*gql.transport.appsync_auth.AppSyncApiKeyAuthentication method*), 64
get_headers() (*gql.transport.appsync_auth.AppSyncAuthentication method*), 64
get_headers() (*gql.transport.appsync_auth.AppSyncIAMAuthentication method*), 65
get_headers() (*gql.transport.appsync_auth.AppSyncJWTAuthentication method*), 65

get_introspection_query_ast() (*in module gql.utilities*), 92
gql
    module, 50
gql() (*in module gql*), 53
gql.client
    module, 53
gql.dsl
    module, 79
gql.transport.aiohttp
    module, 62
gql.transport.appsync_auth
    module, 64
gql.transport.appsync_websockets
    module, 66
gql.transport.exceptions
    module, 67
gql.transport.httpx
    module, 72
gql.transport.phoenix_channel_websockets
    module, 69
gql.transport.requests
    module, 70
gql.transport.websockets
    module, 74
gql.transport.websockets_base
    module, 76
gql.utilities
    module, 92
GraphQLRequest (*class in gql*), 53
GRAPHQLWS_SUBPROTOCOL (*gql.transport.websockets.WebsocketsTransport attribute*), 75

H

HTTPXAsyncTransport (*class in gql.transport.httpx*), 73
HTTPXTransport (*class in gql.transport.httpx*), 72

I

is_valid_field() (*gql.dsl.DSLExecutable method*), 81
is_valid_field() (*gql.dsl.DSLField method*), 89
is_valid_field() (*gql.dsl.DSLFieldSelector method*), 87
is_valid_field() (*gql.dsl.DSLFragment method*), 91
is_valid_field() (*gql.dsl.DSLFragmentSelector method*), 87
is_valid_field() (*gql.dsl.DSLInlineFragment method*), 91
is_valid_field() (*gql.dsl.DSLMetaField method*), 90
is_valid_field() (*gql.dsl.DSLMutation method*), 84
is_valid_field() (*gql.dsl.DSLOperation method*), 82
is_valid_field() (*gql.dsl.DSLQuery method*), 83
is_valid_field() (*gql.dsl.DSLRootFieldSelector method*), 81
is_valid_field() (*gql.dsl.DSLSelector method*), 80
is_valid_field() (*gql.dsl.DSLSubscription method*), 85

L

ListenerQueue (*class in gql.transport.websockets_base*), 76
LocalSchemaTransport (*class in gql.transport.local_schema*), 61

M

meta_type (*gql.dsl.DSLMetaField attribute*), 89
module
    gql, 50
    gql.client, 53
    gql.dsl, 79
    gql.transport.aiohttp, 62
    gql.transport.appsync_auth, 64
    gql.transport.appsync_websockets, 66
    gql.transport.exceptions, 67
    gql.transport.httpx, 72
    gql.transport.phoenix_channel_websockets, 69
    gql.transport.requests, 70
    gql.transport.websockets, 74
    gql.transport.websockets_base, 76
    gql.utilities, 92

N

name (*gql.dsl.DSLExecutable attribute*), 81
name (*gql.dsl.DSLFragment attribute*), 91
name (*gql.dsl.DSLMutation attribute*), 85
name (*gql.dsl.DSLOperation attribute*), 83
name (*gql.dsl.DSLQuery attribute*), 84
name (*gql.dsl.DSLSubscription attribute*), 85
node_tree() (*in module gql.utilities*), 92

O

on() (*gql.dsl.DSLFragment method*), 91
on() (*gql.dsl.DSLInlineFragment method*), 91
operation_name (*gql.GraphQLRequest attribute*), 53
operation_type (*gql.dsl.DSLMutation attribute*), 84
operation_type (*gql.dsl.DSLOperation attribute*), 82
operation_type (*gql.dsl.DSLQuery attribute*), 83
operation_type (*gql.dsl.DSLSubscription attribute*), 85

P

parse_result() (*in module gql.utilities*), 92
payloads (*gql.transport.appsync_websockets.AppSyncWebsocketsTranspor attribute*), 67
payloads (*gql.transport.phoenix_channel_websockets.PhoenixChannelWe attribute*), 70

payloads (*gql.transport.websockets.WebsocketsTransport attribute*), 76
payloads (*gql.transport.websockets_base.WebsocketsTransportBase attribute*), 77
PhoenixChannelWebsocketsTransport (*class in gql.transport.phoenix_channel_websockets*), 69
ping_received (*gql.transport.websockets.WebsocketsTransport attribute*), 75
pong_received (*gql.transport.websockets.WebsocketsTransport attribute*), 75
put() (*gql.transport.websockets_base.ListenerQueue method*), 76

Q

query_id (*gql.transport.exceptions.TransportQueryError attribute*), 68

R

ReconnectingAsyncClientSession (*class in gql.client*), 57
RequestsHTTPTransport (*class in gql.transport.requests*), 70
response_headers (*gql.transport.httpx.HTTPXAsyncTransport attribute*), 74
response_headers (*gql.transport.httpx.HTTPXTransport attribute*), 73

S

select() (*gql.dsl.DSLExecutable method*), 81
select() (*gql.dsl.DSLField method*), 89
select() (*gql.dsl.DSLFieldSelector method*), 87
select() (*gql.dsl.DSLFragment method*), 91
select() (*gql.dsl.DSLFragmentSelector method*), 87
select() (*gql.dsl.DSLInlineFragment method*), 91
select() (*gql.dsl.DSLMetaField method*), 90
select() (*gql.dsl.DSLMutation method*), 84
select() (*gql.dsl.DSLOperation method*), 83
select() (*gql.dsl.DSLQuery method*), 84
select() (*gql.dsl.DSLRootFieldSelector method*), 82
select() (*gql.dsl.DSLSelector method*), 80
select() (*gql.dsl.DSLSubscription method*), 85
selection_set (*gql.dsl.DSLExecutable attribute*), 81
selection_set (*gql.dsl.DSLField attribute*), 89
selection_set (*gql.dsl.DSLFieldSelector attribute*), 88
selection_set (*gql.dsl.DSLFragment attribute*), 91
selection_set (*gql.dsl.DSLFragmentSelector attribute*), 87
selection_set (*gql.dsl.DSLInlineFragment attribute*), 91
selection_set (*gql.dsl.DSLMetaField attribute*), 90
selection_set (*gql.dsl.DSLMutation attribute*), 85
selection_set (*gql.dsl.DSLOperation attribute*), 83
selection_set (*gql.dsl.DSLQuery attribute*), 84
selection_set (*gql.dsl.DSLRootFieldSelector attribute*), 82
selection_set (*gql.dsl.DSLSelector attribute*), 80
selection_set (*gql.dsl.DSLSubscription attribute*), 86
send_ping() (*gql.transport.websockets.WebsocketsTransport method*), 75
send_pong() (*gql.transport.websockets.WebsocketsTransport method*), 76
serialize_value() (*in module gql.utilities*), 92
serialize_variable_values() (*gql.GraphQLRequest method*), 53
serialize_variable_values() (*in module gql.utilities*), 93
set_exception() (*gql.transport.websockets_base.ListenerQueue method*), 76
set_type() (*gql.dsl.DSLVariable method*), 86
start_connecting_task() (*gql.client.ReconnectingAsyncClientSession method*), 58
stop_connecting_task() (*gql.client.ReconnectingAsyncClientSession method*), 58
subscribe() (*gql.Client method*), 52
subscribe() (*gql.client.AsyncClientSession method*), 54
subscribe() (*gql.client.Client method*), 57
subscribe() (*gql.client.ReconnectingAsyncClientSession method*), 58
subscribe() (*gql.transport.appsync_websockets.AppSyncWebsocketsTran method*), 66
subscribe() (*gql.transport.async_transport.AsyncTransport method*), 61
subscribe() (*gql.transport.local_schema.LocalSchemaTransport method*), 62
subscribe() (*gql.transport.phoenix_channel_websockets.PhoenixChannel method*), 70
subscribe() (*gql.transport.websockets.WebsocketsTransport method*), 76
subscribe() (*gql.transport.websockets_base.WebsocketsTransportBase method*), 77
Subscription (*class in gql.transport.phoenix_channel_websockets*), 69
SyncClientSession (*class in gql.client*), 59

T

to_ast_type() (*gql.dsl.DSLVariable method*), 86
Transport (*class in gql.transport.transport*), 61
transport (*gql.client.AsyncClientSession property*), 55
transport (*gql.client.ReconnectingAsyncClientSession property*), 59
transport (*gql.client.SyncClientSession property*), 60
TransportAlreadyConnected, 68
TransportClosed, 68

TransportError, 67
TransportProtocolError, 67
TransportQueryError, 68
TransportServerError, 68

U

update_schema_enum() (*in module gql.utilities*), 93
update_schema_scalar() (*in module gql.utilities*), 93
update_schema_scalars() (*in module gql.utilities*), 94

V

variable_definitions (*gql.dsl.DSLExecutable attribute*), 81
variable_definitions (*gql.dsl.DSLFragment attribute*), 91
variable_definitions (*gql.dsl.DSLMutation attribute*), 85
variable_definitions (*gql.dsl.DSLOperation attribute*), 83
variable_definitions (*gql.dsl.DSLQuery attribute*), 84
variable_definitions (*gql.dsl.DSLSubscription attribute*), 85
variable_values (*gql.GraphQLRequest attribute*), 53

W

wait_closed() (*gql.transport.appsync_websockets.AppSyncWebsocketsTransport method*), 67
wait_closed() (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport method*), 70
wait_closed() (*gql.transport.websockets.WebsocketsTransport method*), 76
wait_closed() (*gql.transport.websockets_base.WebsocketsTransportBase method*), 78
WebsocketsTransport (*class in gql.transport.websockets*), 74
WebsocketsTransportBase (*class in gql.transport.websockets_base*), 77
with_traceback() (*gql.transport.exceptions.TransportAlreadyConnected method*), 69
with_traceback() (*gql.transport.exceptions.TransportClosed method*), 68
with_traceback() (*gql.transport.exceptions.TransportError method*), 67
with_traceback() (*gql.transport.exceptions.TransportProtocolError method*), 67
with_traceback() (*gql.transport.exceptions.TransportQueryError method*), 68
with_traceback() (*gql.transport.exceptions.TransportServerError method*), 68