
gql 3

Release 3.0.0a4

graphql-python.org

Nov 01, 2020

CONTENTS

1	Contents	3
1.1	Introduction	3
1.2	Usage	4
1.3	Async vs Sync	8
1.4	Transports	9
1.5	Advanced	13
1.6	gql-cli	15
1.7	Reference	17
2	Indices and tables	25
	Python Module Index	27
	Index	29

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

CONTENTS

1.1 Introduction

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

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

1.1.1 Installation

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

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

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

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

Less dependencies

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

If for example you only need the [AIOHTTPTransport](#), which needs the `aiohttp` dependency, then you can install GQL with:

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

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

Extra dependency	Transports
<code>aiohttp</code>	AIOHTTPTransport
<code>websockets</code>	WebsocketsTransport PhoenixChannelWebsocketsTransport
<code>requests</code>	RequestsHTTPTransport

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

1.1.2 Reporting Issues and Contributing

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

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

1.2 Usage

1.2.1 Basic usage

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

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

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

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

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

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

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

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

1.2.2 Schema validation

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

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

Using a provided schema

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

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

client = Client(schema=schema_str)
```

OR can be created using python classes:

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

client = Client(schema=SampleSchema)
```

See [tests/starwars/schema.py](#) for an example of such a schema.

Using introspection

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

1.2.3 Subscriptions

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

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

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

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

query = gql(''
```

(continues on next page)

(continued from previous page)

```
        subscription yourSubscription {  
            ...  
        }  
    }  
'''  
  
for result in client.subscribe(query):  
    print (result)
```

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

1.2.4 Using variables

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

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

```
query = gql(  
    """  
    query getContinentName ($code: ID!) {  
        continent (code: $code) {  
            name  
        }  
    }  
    """  
)  
  
params = {"code": "EU"}  
  
# Get name of continent with code "EU"  
result = client.execute(query, variable_values=params)  
print(result)  
  
params = {"code": "AF"}  
  
# Get name of continent with code "AF"  
result = client.execute(query, variable_values=params)  
print(result)
```

1.2.5 HTTP Headers

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

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

1.2.6 File uploads

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

Single File

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

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

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

client = Client(transport=sample_transport)

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

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

    params = {"file": f}

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

File list

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

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

client = Client(transport=sample_transport)

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

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

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

result = client.execute(
```

(continues on next page)

(continued from previous page)

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

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

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.

Note: GraphQL subscriptions are not supported on the HTTP transport. For subscriptions you should use the [websockets transport](#).

```
import asyncio

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

async def main():

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

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

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

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

asyncio.run(main())
```

WebsocketsTransport

The websockets transport implements the [Apollo websockets transport protocol](#).

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

```
import asyncio
import logging

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

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

(continues on next page)

(continued from previous page)

```

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)

asyncio.run(main())

```

Websockets SSL

If you need to connect to an ssl encrypted endpoint:

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

```

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

```

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

```
import pathlib
import ssl

ssl_context = ssl.SSLContext(ssl.PROTOCOL_TLS_CLIENT)
localhost_pem = pathlib.Path(__file__).with_name("YOUR_SERVER_PUBLIC_CERTIFICATE.pem")
ssl_context.load_verify_locations(localhost_pem)

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

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

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

Websockets authentication

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

1. Using HTTP Headers

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

2. With a payload in the connection_init websocket message

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

PhoenixChannelWebsocketsTransport

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

1.4.2 Sync Transports

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

RequestsHTTPTransport

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

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

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

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

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

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

1.5 Advanced

1.5.1 Async advanced usage

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

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

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

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

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

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

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

(continues on next page)

(continued from previous page)

```

# Then create a coroutine 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))
        task4 = asyncio.create_task(execute_subscription2(session))

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

asyncio.run(graphql_connection())

```

Subscriptions tasks can be stopped at any time by running

```
task.cancel()
```

1.5.2 Logging

GQL use the python `logging` module.

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

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

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

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

1.5.3 Execution on a local schema

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

```

from gql import gql, Client

from .someSchema import SampleSchema

client = Client(schema=SampleSchema)

query = gql('')

```

(continues on next page)

(continued from previous page)

```

    {
        hello
    }
'''
)

result = client.execute(query)

```

See `tests/starwars/test_query.py` for an example

1.5.4 Compose queries dynamically

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

```

from gql.dsl import DSLSchema

client = Client(schema=StarWarsSchema)
ds = DSLSchema(client)

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

```

will create a query equivalent to:

```

hero {
  id
  name
  friends {
    name
  }
}

```

Warning: Please note that the DSL module is still considered experimental in GQL 3 and is subject to changes

1.6 gql-cli

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

This script supports http(s) or websockets protocols.

1.6.1 Usage

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

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

Positional Arguments

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

Named Arguments

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

1.6.2 Examples

Simple query using https

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

Simple query using websockets

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

Query with variable

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

Interactive usage

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

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

Execute query saved in a file

Put the query in a file:

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

Then execute query from the file:

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

1.7 Reference

1.7.1 Top-Level Functions

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

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

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

Bases: object

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

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

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

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

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

Initialize the client with the given parameters.

Parameters

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

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

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

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

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

This method will:

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

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

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

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

Execute a GraphQL subscription with a python generator.

We need an async transport for this functionality.

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

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

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

Returns a Document which can be later executed or subscribed by a [Client](#), by an [async session](#) or by a [sync session](#)

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

1.7.2 Sub-Packages

Client

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

Bases: `object`

An instance of this class is created when using `async` with on a [client](#).

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

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

Parameters **client** – the [client](#) used

async execute (*document: graphql.language.ast.DocumentNode, *args, **kwargs*) → Dict
Coroutine to execute the provided document AST asynchronously using the async transport.

The extra arguments are passed to the transport execute method.

async fetch_and_validate (*document: graphql.language.ast.DocumentNode*)
Fetch schema from transport if needed and validate document.

If no schema is present, the validation will be skipped.

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

subscribe (*document: graphql.language.ast.DocumentNode, *args, **kwargs*) → AsyncGenerator[Dict, None]
Coroutine to subscribe asynchronously to the provided document AST asynchronously using the async transport.

The extra arguments are passed to the transport subscribe method.

property transport

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

Bases: object

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

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

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

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

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

Initialize the client with the given parameters.

Parameters

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

execute (*document: graphql.language.ast.DocumentNode, *args, **kwargs*) → Dict
Execute the provided document AST against the remote server using the transport provided during init.

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

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

This method will:

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

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

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

subscribe (*document*: *graphql.language.ast.DocumentNode*, **args*, ***kwargs*) → Generator[Dict, None, None]
Execute a GraphQL subscription with a python generator.

We need an async transport for this functionality.

class `gql.client.SyncClientSession` (*client*: `gql.client.Client`)
Bases: `object`

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

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

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

Parameters *client* – the *client* used

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

fetch_schema () → None

Fetch the GraphQL schema explicitly using introspection.

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

property `transport`

Transport

class `gql.transport.transport.Transport`
Bases: `object`

__init__ ()

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

close ()

Close the transport

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

connect ()

Establish a session with the transport.

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

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

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

Returns ExecutionResult

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

Bases: `gql.transport.async_transport.AsyncTransport`

A transport for executing GraphQL queries against a local schema.

```
__init__ (schema: graphql.type.schema.GraphQLSchema)
    Initialize the transport with the given local schema.
```

Parameters **schema** – Local schema as GraphQLSchema object

```
async close ()
    No close needed on local transport
```

```
async connect ()
    No connection needed on local transport
```

```
async execute (document:    graphql.language.ast.DocumentNode,    *args,    **kwargs) →
    graphql.execution.execute.ExecutionResult
    Execute the provided document AST for on a local GraphQL Schema.
```

```
subscribe (document:    graphql.language.ast.DocumentNode,    *args,    **kwargs) → AsyncGenera-
    tor[graphql.execution.execute.ExecutionResult, None]
    Send a subscription and receive the results using an async generator
```

The results are sent as an ExecutionResult object

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

Bases: `gql.transport.transport.Transport`

Sync Transport used to execute GraphQL queries on remote servers.

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

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

Parameters

- **url** – The GraphQL server URL.
- **headers** – Dictionary of HTTP Headers to send with the Request (Default: None).
- **cookies** – Dict or CookieJar object to send with the Request (Default: None).
- **auth** – Auth tuple or callable to enable Basic/Digest/Custom HTTP Auth (Default: None).

- **use_json** – Send request body as JSON instead of form-urlencoded (Default: True).
- **timeout** – Specifies a default timeout for requests (Default: None).
- **verify** – Either a boolean, in which case it controls whether we verify the server's TLS certificate, or a string, in which case it must be a path to a CA bundle to use. (Default: True).
- **retries** – Pre-setup of the requests' Session for performing retries
- **method** – HTTP method used for requests. (Default: POST).
- **kwargs** – Optional arguments that `request` takes. These can be seen at the [requests](#) source code or the official [docs](#)

close()

Closing the transport by closing the inner session

connect()

Establish a session with the transport.

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

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

Parameters

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

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

class `gql.transport.async_transport.AsyncTransport`

Bases: `object`

__init__()

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

abstract async close()

Coroutine used to Close an established connection

abstract async connect()

Coroutine used to create a connection to the specified address

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

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

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

Send a query and receive the results using an async generator

The query can be a graphql query, mutation or subscription
The results are sent as an ExecutionResult object

INDICES AND TABLES

- `genindex`
- `modindex`
- `search`

PYTHON MODULE INDEX

g

`gql`, [17](#)

`gql.client`, [18](#)

Symbols

[__init__\(\)](#) (*gql.Client* method), 17
[__init__\(\)](#) (*gql.client.AsyncClientSession* method), 18
[__init__\(\)](#) (*gql.client.Client* method), 19
[__init__\(\)](#) (*gql.client.SyncClientSession* method), 20
[__init__\(\)](#) (*gql.transport.async_transport.AsyncTransport* method), 22
[__init__\(\)](#) (*gql.transport.local_schema.LocalSchemaTransport* method), 22
[__init__\(\)](#) (*gql.transport.local_schema.LocalSchemaTransport* method), 21
[__init__\(\)](#) (*gql.transport.requests.RequestsHTTPTransport* method), 21
[__init__\(\)](#) (*gql.transport.transport.Transport* method), 20

A

[AsyncClientSession](#) (class in *gql.client*), 18
[AsyncTransport](#) (class in *gql.transport.async_transport*), 22

C

[Client](#) (class in *gql*), 17
[Client](#) (class in *gql.client*), 19
[close\(\)](#) (*gql.transport.async_transport.AsyncTransport* method), 22
[close\(\)](#) (*gql.transport.local_schema.LocalSchemaTransport* method), 21
[close\(\)](#) (*gql.transport.requests.RequestsHTTPTransport* method), 22
[close\(\)](#) (*gql.transport.transport.Transport* method), 20
[connect\(\)](#) (*gql.transport.async_transport.AsyncTransport* method), 22
[connect\(\)](#) (*gql.transport.local_schema.LocalSchemaTransport* method), 21
[connect\(\)](#) (*gql.transport.requests.RequestsHTTPTransport* method), 22
[connect\(\)](#) (*gql.transport.transport.Transport* method), 20

E

[execute\(\)](#) (*gql.Client* method), 18
[execute\(\)](#) (*gql.client.AsyncClientSession* method), 18
[execute\(\)](#) (*gql.client.Client* method), 19
[execute\(\)](#) (*gql.client.SyncClientSession* method), 20
[execute\(\)](#) (*gql.transport.async_transport.AsyncTransport* method), 22
[execute\(\)](#) (*gql.transport.local_schema.LocalSchemaTransport* method), 21
[execute\(\)](#) (*gql.transport.requests.RequestsHTTPTransport* method), 20
[execute\(\)](#) (*gql.transport.transport.Transport* method), 20

F

[fetch_and_validate\(\)](#) (*gql.client.AsyncClientSession* method), 19
[fetch_schema\(\)](#) (*gql.client.AsyncClientSession* method), 19
[fetch_schema\(\)](#) (*gql.client.SyncClientSession* method), 20

G

[gql](#) module, 17
[gql\(\)](#) (in module *gql*), 18
[gql.client](#) module, 18

L

[LocalSchemaTransport](#) (class in *gql.transport.local_schema*), 21

M

[module](#) *gql*, 17
[module](#) *gql.client*, 18

R

[RequestsHTTPTransport](#) (class in *gql.transport.requests*), 21

S

`subscribe()` (*gql.Client method*), [18](#)
`subscribe()` (*gql.client.AsyncClientSession method*),
[19](#)
`subscribe()` (*gql.client.Client method*), [20](#)
`subscribe()` (*gql.transport.async_transport.AsyncTransport method*), [22](#)
`subscribe()` (*gql.transport.local_schema.LocalSchemaTransport method*), [21](#)
`SyncClientSession` (*class in gql.client*), [20](#)

T

`Transport` (*class in gql.transport.transport*), [20](#)
`transport()` (*gql.client.AsyncClientSession property*), [19](#)
`transport()` (*gql.client.SyncClientSession property*),
[20](#)