
gql 3

Release 3.0.0b1

graphql-python.org

Nov 22, 2021

CONTENTS

1	Contents	3
1.1	Introduction	3
1.2	Usage	4
1.3	Async vs Sync	15
1.4	Transports	16
1.5	Advanced	22
1.6	gql-cli	31
1.7	Reference	33
2	Indices and tables	55
	Python Module Index	57
	Index	59

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 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 --pre gql[aiohttp]
```

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

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

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 directly after the first connection to the backend.

1.2.3 Subscriptions

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

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

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

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

query = gql(''
```

(continues on next page)

(continued from previous page)

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

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

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

1.2.4 Using variables

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

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

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

params = {"code": "EU"}

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

params = {"code": "AF"}

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

1.2.5 HTTP Headers

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

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

1.2.6 File uploads

GQL supports file uploads with the *aiohttp transport* and the *requests 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')

client = Client(transport=transport)

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

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

    params = {"file": f}

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

File list

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

```
transport = AIOHTTPTransport(url='YOUR_URL')
# Or transport = RequestsHTTPTransport(url='YOUR_URL')

client = Client(transport=transport)

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

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

(continues on next page)

(continued from previous page)

```
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:
        chunk = await f.read(64*1024)
        while chunk:
            yield chunk
            chunk = await f.read(64*1024)
```

(continues on next page)

(continued from previous page)

```

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 amount 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:
    return value.isoformat()
```

(continues on next page)

(continued from previous page)

```

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
```

(continues on next page)

(continued from previous page)

```
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() }
```

(continues on next page)

(continued from previous page)

```

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

```

1.3 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,

```

(continues on next page)

(continued from previous page)

```
) as session:

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

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

asyncio.run(main())
```

IPython

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

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

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

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

```
await main()
```

OR:

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

1.4 Transports

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

1.4.1 Async Transports

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

AIOHTTPTransport

This transport uses the [aiohttp](#) library and allows you to send GraphQL queries using the HTTP protocol.

Reference: `gql.transport.aiohttp.AIOHTTPTransport`

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())
```

Authentication

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

1. Using HTTP Headers

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

2. Using HTTP Cookies

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

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

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

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

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

WebsocketsTransport

The websockets transport supports both:

- the [Apollo websockets transport protocol](#).
- the [GraphQL-ws websockets transport protocol](#)

It will detect the backend supported protocol from the response http headers returned.

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
```

(continues on next page)

(continued from previous page)

```

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

```

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,
)
```

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.

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)
```

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 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)
```

Disabling logs

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

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

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

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

For the RequestsHTTPTransport:

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

For the WebsocketsTransport:

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

1.5.3 Execution on a local schema

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

```
from gql import gql, Client

from .someSchema import SampleSchema

client = Client(schema=SampleSchema)

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

result = client.execute(query)
```

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

1.5.4 Compose queries dynamically

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

The following code:

```
ds = DSLSchema(StarWarsSchema)

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

will generate a query equivalent to:

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

How to use

First generate the root using the *DSLSchema*:

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

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

```
ds.Query
```

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

```
ds.Query.hero
```

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

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

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

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

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

Or do it sequentially:

```
hero_query = ds.Query.hero

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

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

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

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

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

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

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

result = client.execute(query)
```

Arguments

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

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

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

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

Aliases

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

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

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

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

Or using keyword arguments in the *select* method:

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

Mutations

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

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

Variable arguments

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

- 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  
    )  
)
```

(continues on next page)

(continued from previous page)

```
)
op.variable_definitions = var
query = dsl_gql(op)
```

will generate a query equivalent to:

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

Subscriptions

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

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

Multiple fields in an operation

It is possible to create an operation with multiple fields:

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

Operation name

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

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

will generate the request:

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

Multiple operations in a document

It is possible to create an Document with multiple operations:

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

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)
)
```

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")
```

(continues on next page)

(continued from previous page)

```

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

# 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 3.6+ 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 ...]] [-H [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[Union[int,
                  float]] = 10, serialize_variables: bool = False, parse_results: bool = False)
```

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[Union[int, float]] = 10, serialize_variables: bool
          = False, parse_results: bool = False)
```

Initialize the client with the given parameters.

Parameters

- **schema** – an optional GraphQL Schema for local validation See *Schema validation*
- **transport** – The provided *transport*.

- **fetch_schema_from_transport** – Boolean to indicate that if we want to fetch the schema from the transport using an introspection query
- **execute_timeout** – The maximum time in seconds for the execution of a request before a `TimeoutError` is raised. Only used for async transports. 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 back-end. Can be used to unserialize custom scalars or enums.

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

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

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

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

This method will:

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

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

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

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

Execute a GraphQL subscription with a python generator.

We need an async transport for this functionality.

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

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

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

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

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

1.7.2 Sub-Packages

`gql.client`

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

Bases: `object`

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

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

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

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

async execute (*document*: *graphql.language.ast.DocumentNode*, **args*, *variable_values*: *Optional[Dict[str, Any]] = None*, *operation_name*: *Optional[str] = None*, *serialize_variables*: *Optional[bool] = None*, *parse_result*: *Optional[bool] = None*, ***kwargs*) → Dict

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. Default: False.
- **parse_result** – Whether gql will unserialize the result. By default use the *parse_results* attribute of the client.

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

subscribe (*document*: *graphql.language.ast.DocumentNode*, **args*, *variable_values*: *Optional[Dict[str, Any]] = None*, *operation_name*: *Optional[str] = None*, *serialize_variables*: *Optional[bool] = None*, *parse_result*: *Optional[bool] = None*, ***kwargs*) → AsyncGenerator[Dict, 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. Default: False.
- **parse_result** – Whether gql will unserialize the result. By default use the *parse_results* attribute of the client.

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: Op-
                        tional[Union[int, float]] = 10, serialize_variables: bool = False,
                        parse_results: bool = False)
```

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[Union[int, float]] = 10, serialize_variables: bool
         = False, parse_results: bool = False)
```

Initialize the client with the given parameters.

Parameters

- **schema** – an optional GraphQL Schema for local validation See [Schema validation](#)
- **transport** – The provided *transport*.
- **fetch_schema_from_transport** – Boolean to indicate that if we want to fetch the schema from the transport using an introspection query
- **execute_timeout** – The maximum time in seconds for the execution of a request before a `TimeoutError` is raised. Only used for async transports. 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 back-end. Can be used to unserialize custom scalars or enums.

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*, *variable_values*: *Optional*[Dict[str, Any]] = *None*, *operation_name*: *Optional*[str] = *None*, *serialize_variables*: *Optional*[bool] = *None*, *parse_result*: *Optional*[bool] = *None*, ***kwargs*) → Dict
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. Default: `False`.
- **parse_result** – Whether `gql` will unserialize the result. By default use the `parse_results` attribute of the client.

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: `object`

close ()

Close the transport

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

connect ()

Establish a session with the transport.

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

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

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

Returns *ExecutionResult*

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

Bases: *gql.transport.async_transport.AsyncTransport*

A transport for executing GraphQL queries against a local schema.

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

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

async close ()
No close needed on local transport

async connect ()
No connection needed on local transport

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

subscribe (*document*: *graphql.language.ast.DocumentNode*, *args, **kwargs) → *AsyncGenerator[graphql.execution.execute.ExecutionResult, None]*
Send a subscription and receive the results using an async generator
The results are sent as an *ExecutionResult* object

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

Bases: *gql.transport.transport.Transport*

Sync Transport used to execute GraphQL queries on remote servers.

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

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

Parameters

- **url** – The GraphQL server URL.

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

close()

Closing the transport by closing the inner session

connect()

Establish a session with the transport.

execute (*document: graphql.language.ast.DocumentNode, variable_values: Optional[Dict[str, Any]] = None, operation_name: Optional[str] = None, timeout: Optional[int] = None, extra_args: Optional[Dict[str, Any]] = None, upload_files: bool = False*) → `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`).
- **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.

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

class `gql.transport.async_transport.AsyncTransport`
Bases: `object`

abstract async close()

Coroutine used to Close an established connection

abstract async connect()

Coroutine used to create a connection to the specified address

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

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

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

Send a query and receive the results using an async generator

The query can be a graphql query, mutation or subscription

The results are sent as an *ExecutionResult* object

```
class gql.transport.aiohttp.AIOHTTPTransport (url: str, headers: Optional[Union[Mapping[Union[str, multidict._multidict.istr], str], multidict._multidict.CIMultiDict, multidict._multidict.CIMultiDictProxy]] = None, cookies: Optional[Union[Mapping[str, Union[str, BaseCookie[str], Morsel[Any]]], Iterable[Tuple[str, Union[str, BaseCookie[str], Morsel[Any]]], BaseCookie[str]]] = None, auth: Optional[aiohttp.helpers.BasicAuth] = None, ssl: Union[ssl.SSLContext, bool, aiohttp.client_reqrep.Fingerprint] = False, timeout: Optional[int] = None, client_session_args: Optional[Dict[str, Any]] = None)
```

Bases: *gql.transport.async_transport.AsyncTransport*

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

This transport use the aiohttp library with asyncio.

```
__init__ (url: str, headers: Optional[Union[Mapping[Union[str, multidict._multidict.istr], str], multidict._multidict.CIMultiDict, multidict._multidict.CIMultiDictProxy]] = None, cookies: Optional[Union[Mapping[str, Union[str, BaseCookie[str], Morsel[Any]]], Iterable[Tuple[str, Union[str, BaseCookie[str], Morsel[Any]]], BaseCookie[str]]] = None, auth: Optional[aiohttp.helpers.BasicAuth] = None, ssl: Union[ssl.SSLContext, bool, aiohttp.client_reqrep.Fingerprint] = False, timeout: Optional[int] = None, client_session_args: Optional[Dict[str, 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
- **ssl** – ssl_context of the connection. Use ssl=False to disable encryption
- **client_session_args** – Dict of extra args passed to *aiohttp.ClientSession*

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 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.

async execute (*document*: `graphql.language.ast.DocumentNode`, *variable_values*: `Optional[Dict[str, Any]] = None`, *operation_name*: `Optional[str] = None`, *extra_args*: `Optional[Dict[str, Any]] = None`, *upload_files*: `bool = False`) → `graphql.execution.execute.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.

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

```
class gql.transport.websockets.WebsocketsTransport (url: str, headers: Optional[Union[websockets.datastructures.Headers, Mapping[str, str], Iterable[Tuple[str, str]]]] = None, ssl: Union[ssl.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] = {})
```

Bases: `gql.transport.async_transport.AsyncTransport`

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[websockets.datastructures.Headers, Mapping[str, str], Iterable[Tuple[str, str]]]] = None, ssl: Union[ssl.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] = {}) → 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.
- **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

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: graphql.language.ast.DocumentNode, variable_values: Optional[Dict[str, Any]] = None, operation_name: Optional[str] = None) → graphql.execution.execute.ExecutionResult
```

Execute the provided document AST 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.

payloads: Dict[str, Any]

payloads is a dict which will contain the payloads received with the graphql-ws protocol. Possible keys are: 'ping', 'pong', 'connection_ack'

ping_received: asyncio.locks.Event

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

pong_received: asyncio.locks.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

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

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

Bases: `gql.transport.websockets.WebsocketsTransport`

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

APOLLO_SUBPROTOCOL = 'graphql-ws'

GRAPHQLWS_SUBPROTOCOL = 'graphql-transport-ws'

__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

ack_timeout: `Optional[Union[int, float]]`

answer_pings: `bool`

check_keep_alive_task: `Optional[asyncio.Future]`

async close (`)` $\rightarrow$ `None`
Coroutine used to Close an established connection

close_exception: `Optional[Exception]`

close_task: `Optional[asyncio.Future]`

close_timeout: `Optional[Union[int, float]]`

async connect (`)` $\rightarrow$ `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

connect_timeout: `Optional[Union[int, float]]`

async execute (*document:* `graphql.language.ast.DocumentNode`, *variable_values:* `Optional[Dict[str, Any]] = None`, *operation_name:* `Optional[str] = None`) $\rightarrow$ `graphql.execution.execute.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.

headers: `Optional[HeadersLike]`

init_payload: `Dict[str, Any]`

keep_alive_timeout: `Optional[Union[int, float]]`

listeners: `Dict[int, ListenerQueue]`

next_query_id: `int`

payloads: `Dict[str, Any]`
payloads is a dict which will contain the payloads received with the graphql-ws protocol. Possible keys are: 'ping', 'pong', 'connection_ack'

ping_interval: `Optional[Union[int, float]]`

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

pong_timeout: `Optional[Union[int, float]]`

receive_data_task: `Optional[asyncio.Future]`

async send_ping (*payload:* `Optional[Any] = None`) $\rightarrow$ `None`
Send a ping message for the graphql-ws protocol


```

send_ping_task: Optional[asyncio.Future]

async send_pong (payload: Optional[Any] = None) → None
    Send a pong message for the graphql-ws protocol

ssl: Union[SSLContext, bool]

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

url: str

async wait_closed () → None

websocket: Optional[WebSocketClientProtocol]

```

gql.dsl

```

gql.dsl.ast_from_serialized_value_untyped (serialized: Any) → Optional[graphql.language.ast.ValueNode]

```

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

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

Anything resembling 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[graphql.type.definition.GraphQLScalarType,
                                                graphql.type.definition.GraphQLEnumType,
                                                graphql.type.definition.GraphQLInputObjectType,
                                                graphql.type.definition.GraphQLWrappingType]) → Optional[graphql.language.ast.ValueNode]

```

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

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

Produce a GraphQL Value AST given a Python object.

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

```

gql.dsl.dsl_gql (*operations: gql.dsl.DSLExecutable, **operations_with_name: gql.dsl.DSLExecutable) → graphql.language.ast.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: graphql.type.schema.GraphQLSchema)

Bases: object

The DSLSchema is the root of the DSL code.

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

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

Initialize the DSLSchema with the given schema.

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

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

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

Bases: abc.ABC

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

Inherited by *DSLOperation* and *DSLFragment*

abstract property executable_ast

Generates the ast for *dsl_gql*.

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

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

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

Parameters

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

Raises

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

name: Optional[str]

variable_definitions: *gql.dsl.DSLVariableDefinitions*

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

Bases: *gql.dsl.DSLExecutable*

Interface for GraphQL operations.

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

operation_type: `graphql.language.ast.OperationType`

property executable_ast

Generates the ast for *dsl_gql*.

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

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

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

Parameters

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

Raises

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

variable_definitions: `gql.dsl.DSLVariableDefinitions`

name: `Optional[str]`

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

Bases: *gql.dsl.DSLOperation*

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

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

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

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

Parameters

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

Raises

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

property executable_ast

Generates the ast for *dsl_gql*.

variable_definitions: `gql.dsl.DSLVariableDefinitions`

name: `Optional[str]`

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

Bases: *gql.dsl.DSLOperation*

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

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

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

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

Parameters

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

Raises

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

```
property executable_ast
```

Generates the ast for *dsl_gql*.

```
variable_definitions: gql.dsl.DSLVariableDefinitions
```

```
name: Optional[str]
```

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

Bases: *gql.dsl.DSLOperation*

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

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

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

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

Parameters

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

Raises

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

```
property executable_ast
```

Generates the ast for *dsl_gql*.

```
variable_definitions: gql.dsl.DSLVariableDefinitions
```

```
name: Optional[str]
```

```
class gql.dsl.DSLVariable (name: str)
```

Bases: *object*

The *DSLVariable* represents a single variable defined in a GraphQL operation

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

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

```

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

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

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

```

class `gql.dsl.DSLVariableDefinitions`

Bases: `object`

The `DSLVariableDefinitions` represents variable definitions in a GraphQL operation

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

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

```

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

```

```

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

```

Bases: `object`

The `DSLType` represents a GraphQL type for the DSL code.

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

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

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

```

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

```

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

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

class `gql.dsl.DSLSelectable`

Bases: `abc.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[graphql.language.ast.FieldNode, graphql.language.ast.InlineFragmentNode]

```

class `gql.dsl.DSLSelector`

Bases: `abc.ABC`

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

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

```

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

```

selection_set: `graphql.language.ast.SelectionSetNode`

select (*fields: `gql.dsl.DSLSelectable`, **fields_with_alias: `gql.dsl.DSLSelectableWithAlias`) → `gql.dsl.DSLSelector`

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

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

Parameters

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

Returns itself

Raises `TypeError` – if any of the provided fields are not instances of the `DSLSelectable` class.

class `gql.dsl.DSLSelectableWithAlias`

Bases: `gql.dsl.DSLSelectable`

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

ast_field: `graphql.language.ast.FieldNode`

alias (alias: `str`) → `gql.dsl.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`, graphql_type: `Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType]`, graphql_field: `graphql.type.definition.GraphQLField`)

Bases: `gql.dsl.DSLSelectableWithAlias`, `gql.dsl.DSLSelector`

The `DSLField` represents a GraphQL field for the DSL code.

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

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

__init__ (name: `str`, graphql_type: `Union[graphql.type.definition.GraphQLObjectType, graphql.type.definition.GraphQLInterfaceType]`, graphql_field: `graphql.type.definition.GraphQLField`)
Initialize the `DSLField`.

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

Parameters

- **name** – the name of the field

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

field: `graphql.type.definition.GraphQLField`

ast_field: `graphql.language.ast.FieldNode`

args (***kwargs*) → *gql.dsl.DSLField*

Set the arguments of a field

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

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

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

Returns itself

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

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

Calling *select* method with corrected typing hints

alias (*alias:* *str*) → *gql.dsl.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

selection_set: `graphql.language.ast.SelectionSetNode`

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

Bases: *gql.dsl.DSLSelectable*, *gql.dsl.DSLSelector*

DSLInlineFragment represents an inline fragment for the DSL code.

__init__ (**fields:* *gql.dsl.DSLSelectable*, ***fields_with_alias:* *gql.dsl.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: `graphql.language.ast.InlineFragmentNode`

select (*fields: gql.dsl.DSLSelectable, **fields_with_alias: gql.dsl.DSLSelectableWithAlias) → *gql.dsl.DSLInlineFragment*
Calling *select* method with corrected typing hints

on (type_condition: gql.dsl.DSLType) → *gql.dsl.DSLInlineFragment*
Provides the GraphQL type of this inline fragment.

selection_set: graphql.language.ast.SelectionSetNode

class gql.dsl.DSLFragment (name: str, *fields: gql.dsl.DSLSelectable, **fields_with_alias: gql.dsl.DSLSelectableWithAlias)

Bases: *gql.dsl.DSLSelectable*, *gql.dsl.DSLSelector*, *gql.dsl.DSLExecutable*

DSLFragment represents a named GraphQL fragment for the DSL code.

selection_set: graphql.language.ast.SelectionSetNode

variable_definitions: *gql.dsl.DSLVariableDefinitions*

__init__ (name: str, *fields: gql.dsl.DSLSelectable, **fields_with_alias: gql.dsl.DSLSelectableWithAlias)
Initialize the DSLFragment.

Parameters

- **name** (*str*) – the name of the fragment
- ***fields** (*DSLSelectable* (*DSLField*, *DSLFragment* or *DSLInlineFragment*)) – new children fields
- ****fields_with_alias** (*DSLField*) – new children fields with alias as key

name: str

property ast_field

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: gql.dsl.DSLSelectable, **fields_with_alias: gql.dsl.DSLSelectableWithAlias) → *gql.dsl.DSLFragment*
Calling *select* method with corrected typing hints

on (type_condition: gql.dsl.DSLType) → *gql.dsl.DSLFragment*
Provides the GraphQL type of this fragment.

Parameters **type_condition** (*DSLType*) – the provided type

property executable_ast

Generates the ast for *dsl_gql*.

Raises **AttributeError** – if a type has not been provided

gql.utilities

gql.utilities.parse_result (schema: graphql.type.schema.GraphQLSchema, document: graphql.language.ast.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_*: *graphql.type.definition.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*: *graphql.type.schema.GraphQLSchema*,
document: *graphql.language.ast.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 dictionary of variable values which needs to be serialized.
- **operation_name** – the optional operation_name for the query.

`gql.utilities.update_schema_enum` (*schema*: *graphql.type.schema.GraphQLSchema*, *name*:
str, *values*: *Union[Dict[str, Any], Type[enum.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 default behaviour when passing an Enum to a GraphQLEnumType.

```
gql.utilities.update_schema_scalar(schema: graphql.type.schema.GraphQLSchema, name: str, scalar: graphql.type.definition.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: graphql.type.schema.GraphQLSchema, scalars: List[graphql.type.definition.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.

INDICES AND TABLES

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

PYTHON MODULE INDEX

g

`gql`, [33](#)

`gql.client`, [34](#)

`gql.dsl`, [45](#)

`gql.utilities`, [52](#)

Symbols

`__init__()` (*gql.Client* method), 33
`__init__()` (*gql.client.AsyncClientSession* method), 34
`__init__()` (*gql.client.Client* method), 36
`__init__()` (*gql.client.SyncClientSession* method), 37
`__init__()` (*gql.dsl.DSLExecutable* method), 46
`__init__()` (*gql.dsl.DSLField* method), 50
`__init__()` (*gql.dsl.DSLFragment* method), 52
`__init__()` (*gql.dsl.DSLInlineFragment* method), 51
`__init__()` (*gql.dsl.DSLMutation* method), 47
`__init__()` (*gql.dsl.DSLOperation* method), 47
`__init__()` (*gql.dsl.DSLQuery* method), 47
`__init__()` (*gql.dsl.DSLSchema* method), 46
`__init__()` (*gql.dsl.DSLSelector* method), 49
`__init__()` (*gql.dsl.DSLSubscription* method), 48
`__init__()` (*gql.dsl.DSLType* method), 49
`__init__()` (*gql.dsl.DSLVariable* method), 48
`__init__()` (*gql.dsl.DSLVariableDefinitions* method), 49
`__init__()` (*gql.transport.aiohttp.AIOHTTPTransport* method), 40
`__init__()` (*gql.transport.local_schema.LocalSchemaTransport* method), 38
`__init__()` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* method), 43
`__init__()` (*gql.transport.requests.RequestsHTTPTransport* method), 38
`__init__()` (*gql.transport.websockets.WebsocketsTransport* method), 42

A

`ack_timeout` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 43
`AIOHTTPTransport` (class in *gql.transport.aiohttp*), 40
`alias()` (*gql.dsl.DSLField* method), 51
`alias()` (*gql.dsl.DSLSelectableWithAlias* method), 50
`answer_pings` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44
`APOLLO_SUBPROTOCOL` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport*

attribute), 43
`APOLLO_SUBPROTOCOL` (*gql.transport.websockets.WebsocketsTransport* attribute), 41
`args()` (*gql.dsl.DSLField* method), 51
`ast_field` (*gql.dsl.DSLField* attribute), 51
`ast_field` (*gql.dsl.DSLInlineFragment* attribute), 51
`ast_field` (*gql.dsl.DSLSelectable* attribute), 49
`ast_field` (*gql.dsl.DSLSelectableWithAlias* attribute), 50
`ast_field()` (*gql.dsl.DSLFragment* property), 52
`ast_from_serialized_value_untyped()` (in module *gql.dsl*), 45
`ast_from_value()` (in module *gql.dsl*), 45
`AsyncClientSession` (class in *gql.client*), 34
`AsyncTransport` (class in *gql.transport.async_transport*), 39

C

`check_keep_alive_task` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44
`Client` (class in *gql*), 33
`Client` (class in *gql.client*), 35
`close()` (*gql.transport.aiohttp.AIOHTTPTransport* method), 40
`close()` (*gql.transport.async_transport.AsyncTransport* method), 39
`close()` (*gql.transport.local_schema.LocalSchemaTransport* method), 38
`close()` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* method), 44
`close()` (*gql.transport.requests.RequestsHTTPTransport* method), 39
`close()` (*gql.transport.transport.Transport* method), 37
`close()` (*gql.transport.websockets.WebsocketsTransport* method), 42
`close_exception` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44
`close_task` (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

close_timeout (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

connect () (*gql.transport.aiohttp.AIOHTTPTransport* method), 41

connect () (*gql.transport.async_transport.AsyncTransport* method), 39

connect () (*gql.transport.local_schema.LocalSchemaTransport* method), 38

connect () (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* method), 44

connect () (*gql.transport.requests.RequestsHTTPTransport* method), 39

connect () (*gql.transport.transport.Transport* method), 37

connect () (*gql.transport.websockets.WebsocketsTransport* method), 42

connect_timeout (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

D

dsl_gql () (*in module gql.dsl*), 45

DSLExecutable (*class in gql.dsl*), 46

DSLField (*class in gql.dsl*), 50

DSLFragment (*class in gql.dsl*), 52

DSLInlineFragment (*class in gql.dsl*), 51

DSLMutation (*class in gql.dsl*), 47

DSLOperation (*class in gql.dsl*), 46

DSLQuery (*class in gql.dsl*), 47

DSLSchema (*class in gql.dsl*), 46

DSLSelectable (*class in gql.dsl*), 49

DSLSelectableWithAlias (*class in gql.dsl*), 50

DSLSelector (*class in gql.dsl*), 49

DSLSubscription (*class in gql.dsl*), 48

DSLType (*class in gql.dsl*), 49

DSLVariable (*class in gql.dsl*), 48

DSLVariableDefinitions (*class in gql.dsl*), 49

E

executable_ast () (*gql.dsl.DSLExecutable* property), 46

executable_ast () (*gql.dsl.DSLFragment* property), 52

executable_ast () (*gql.dsl.DSLMutation* property), 48

executable_ast () (*gql.dsl.DSLOperation* property), 47

executable_ast () (*gql.dsl.DSLQuery* property), 47

executable_ast () (*gql.dsl.DSLSubscription* property), 48

execute () (*gql.Client* method), 34

execute () (*gql.client.AsyncClientSession* method), 35

execute () (*gql.client.Client* method), 36

execute () (*gql.client.SyncClientSession* method), 37

execute () (*gql.transport.aiohttp.AIOHTTPTransport* method), 41

execute () (*gql.transport.async_transport.AsyncTransport* method), 39

execute () (*gql.transport.local_schema.LocalSchemaTransport* method), 38

execute () (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* method), 44

execute () (*gql.transport.requests.RequestsHTTPTransport* method), 39

execute () (*gql.transport.transport.Transport* method), 37

execute () (*gql.transport.websockets.WebsocketsTransport* method), 42

F

fetch_schema () (*gql.client.AsyncClientSession* method), 35

fetch_schema () (*gql.client.SyncClientSession* method), 37

field (*gql.dsl.DSLField* attribute), 51

file_classes (*gql.transport.aiohttp.AIOHTTPTransport* attribute), 41

file_classes (*gql.transport.requests.RequestsHTTPTransport* attribute), 39

G

gql

- module, 33

gql () (*in module gql*), 34

gql.client

- module, 34

gql.dsl

- module, 45

gql.utilities

- module, 52

GRAPHQLWS_SUBPROTOCOL

- (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 43

GRAPHQLWS_SUBPROTOCOL

- (*gql.transport.websockets.WebsocketsTransport* attribute), 41

H

headers (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

I

init_payload (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

K

keep_alive_timeout

(*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

L

listeners (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

LocalSchemaTransport (class in *gql.transport.local_schema*), 38

M

module

- gql*, 33
- gql.client*, 34
- gql.dsl*, 45
- gql.utilities*, 52

N

name (*gql.dsl.DSLExecutable* attribute), 46

name (*gql.dsl.DSLFragment* attribute), 52

name (*gql.dsl.DSLMutation* attribute), 48

name (*gql.dsl.DSLOperation* attribute), 47

name (*gql.dsl.DSLQuery* attribute), 47

name (*gql.dsl.DSLSubscription* attribute), 48

next_query_id (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

O

on () (*gql.dsl.DSLFragment* method), 52

on () (*gql.dsl.DSLInlineFragment* method), 52

operation_type (*gql.dsl.DSLMutation* attribute), 47

operation_type (*gql.dsl.DSLOperation* attribute), 47

operation_type (*gql.dsl.DSLQuery* attribute), 47

operation_type (*gql.dsl.DSLSubscription* attribute), 48

P

parse_result () (in module *gql.utilities*), 52

payloads (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

payloads (*gql.transport.websockets.WebsocketsTransport* attribute), 43

PhoenixChannelWebsocketsTransport (class in *gql.transport.phoenix_channel_websockets*), 43

ping_interval (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

ping_received (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

ping_received (*gql.transport.websockets.WebsocketsTransport* attribute), 43

pong_received (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

pong_timeout (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

R

receive_data_task (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

RequestsHTTPTransport (class in *gql.transport.requests*), 38

S

select () (*gql.dsl.DSLField* method), 51

select () (*gql.dsl.DSLFragment* method), 52

select () (*gql.dsl.DSLInlineFragment* method), 51

select () (*gql.dsl.DSLSelector* method), 50

selection_set (*gql.dsl.DSLField* attribute), 51

selection_set (*gql.dsl.DSLFragment* attribute), 52

selection_set (*gql.dsl.DSLInlineFragment* attribute), 52

selection_set (*gql.dsl.DSLSelector* attribute), 49

send_ping () (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* method), 44

send_ping () (*gql.transport.websockets.WebsocketsTransport* method), 43

send_ping_task (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 44

send_pong () (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* method), 45

send_pong () (*gql.transport.websockets.WebsocketsTransport* method), 43

serialize_value () (in module *gql.utilities*), 53

serialize_variable_values () (in module *gql.utilities*), 53

set_type () (*gql.dsl.DSLVariable* method), 49

ssl (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* attribute), 45

subscribe () (*gql.Client* method), 34

subscribe () (*gql.client.AsyncClientSession* method), 35

subscribe () (*gql.client.Client* method), 37

subscribe () (*gql.transport.async_transport.AsyncTransport* method), 40

subscribe () (*gql.transport.local_schema.LocalSchemaTransport* method), 38

subscribe () (*gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport* method), 45

subscribe () (*gql.transport.websockets.WebsocketsTransport* method), 43

SyncClientSession (class in *gql.client*), 37

to_ast_type () (*gql.dsl.DSLVariable* method), 49

`Transport` (*class in `gql.transport.transport`*), 37
`transport()` (*`gql.client.AsyncClientSession` property*), 35
`transport()` (*`gql.client.SyncClientSession` property*), 37

U

`update_schema_enum()` (*in module `gql.utilities`*), 53
`update_schema_scalar()` (*in module `gql.utilities`*), 54
`update_schema_scalars()` (*in module `gql.utilities`*), 54
`url` (*`gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport` attribute*), 45

V

`variable_definitions` (*`gql.dsl.DSLExecutable` attribute*), 46
`variable_definitions` (*`gql.dsl.DSLFragment` attribute*), 52
`variable_definitions` (*`gql.dsl.DSLMutation` attribute*), 48
`variable_definitions` (*`gql.dsl.DSLOperation` attribute*), 47
`variable_definitions` (*`gql.dsl.DSLQuery` attribute*), 47
`variable_definitions` (*`gql.dsl.DSLSubscription` attribute*), 48

W

`wait_closed()` (*`gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport` method*), 45
`wait_closed()` (*`gql.transport.websockets.WebsocketsTransport` method*), 43
`websocket` (*`gql.transport.phoenix_channel_websockets.PhoenixChannelWebsocketsTransport` attribute*), 45
`WebsocketsTransport` (*class in `gql.transport.websockets`*), 41