# aiographql-client

Mar 02, 2020

# Contents

CHAPTER 1

# Usage Examples

Many of the following examples make use of GitHub GrpahQL API. You can retrieve a token as detailed here.

## 1.1 Simple Query

The *aiographql.client.GraphQLClient* can be used to store headers like *Authorization* that need to be sent with every request made.

```
client = GraphQLClient(
    endpoint="https://api.github.com/graphql",
    headers={"Authorization": f"Bearer {TOKEN}"},
)
request = GraphQLRequest(
    query="""
        query {
          viewer {
            login
          }
        }
    """
)
response = await client.query(request=request)
```

If you have a valid token, the above query will return the following data.

```
>>> response.data
{'viewer': {'login': 'username'}}
```

If you intend to only make use of the query once, ie. it is not re-used, you may forgo the creation of an *aiographql.client.GraphQLRequest* instance and pass in the query string direct to *aiographql.client.GraphQLClient.query()*.

```
await client.query(request="{ viewer { login } }")
```

## 1.2 Client Side Query Validation

Because we make use of GraphQL Core 3, client-side query validation is performed by default.

```
request = GraphQLRequest(
    query="""
        query {
          bad {
            login
          }
        }
    """
)
response: GraphQLResponse = await client.query(request=request)
```

This will raise *aiographql.client.GraphQLClientValidationException*. The message will also contain information about the error.

```
aiographql.client.exceptions.GraphQLClientValidationException: Query validation failed

Cannot query field 'bad' on type 'Query'.

GraphQL request:3:15
2 |              query {
3 |                bad {
  |                ^
4 |                  login

Process finished with exit code 1
```

## 1.3 Server Side Query Validation

You can skip the client side validation, forcing server side validation instead by setting the `aiographql.client.GraphQLRequest.validate` to *False* before making the request.

```
request = GraphQLRequest(
    query="""
        query {
          bad {
            login
          }
        }
    """,
    validate=False
)
response: GraphQLResponse = await client.query(request=request)
```

```
>>> response.data
{}
```

(continues on next page)

(continued from previous page)

```
>>> response.errors
[GraphQLError(extensions={'code': 'undefinedField', 'typeName': 'Query', 'fieldName':
↪'bad'}, locations=[{'line': 3, 'column': 15}], message="Field 'bad' doesn't exist␣
↪on type 'Query'", path=['query', 'bad'])]
```

## 1.4 Query Variables

```
request = GraphQLRequest(
    query="""
        query($number_of_repos:Int!) {
          viewer {
             repositories(last: $number_of_repos) {
               nodes {
                 name
                 isFork
               }
             }
           }
        }
    """,
    variables={"number_of_repos": 3},
)
response: GraphQLResponse = await client.query(request=request)
```

You can override default values specified in the prepared request too. The values are upserted into the existing defaults.

```
response: GraphQLResponse = await client.query(request=request, variables={
    "number_of_repos": 1
})
```

## 1.5 Specifying Operation Name

You can use a single *aiographql.client.GraphQLRequest* object to stop a query wit multiple operations.

```
request = GraphQLRequest(
    query="""
        query FindFirstIssue {
          repository(owner:"octocat", name:"Hello-World") {
            issues(first:1) {
              nodes {
                id
                url
              }
            }
          }
        }

        query FindLastIssue {
          repository(owner:"octocat", name:"Hello-World") {
            issues(last:1) {
              nodes {
```

(continues on next page)

(continued from previous page)

```
                id
                url
              }
            }
          }
        }
    """,
    operation="FindFirstIssue",
)

# use the default operation (FindFirstIssue)
response = await client.query(request=request)

# use the operation FindLastIssue
response = await client.query(
    request=request,
    operation="FindLastIssue"
)
```

## 1.6 Subscriptions

The following example makes use of the Hasura World Database Demo application as there aren't many public GraphQL schema that allow subscriptions for testing. You can use the project's provided docker compose file to start an instance locally.

**By default the subscription is closed if any of the following event type is received.**

1. *aiographql.client.GraphQLSubscriptionEventType.ERROR*
2. *aiographql.client.GraphQLSubscriptionEventType.CONNECTION_ERROR*
3. *aiographql.client.GraphQLSubscriptionEventType.COMPLETE*

The following example will subscribe to any change events and print the event as is to stdout when either *aiographql.client.GraphQLSubscriptionEventType.DATA* or *aiographql.client.GraphQLSubscriptionEventType.ERROR* is received.

```
request = GraphQLRequest(
    query="""
    subscription {
      city(where: {name: {_eq: "Berlin"}}) {
        name
        id
      }
    }
"""
)
# subscribe to data and error events, and print them
subscription = await client.subscribe(
    request=request, on_data=print, on_error=print
)
# unsubscribe
await subscription.unsubscribe_and_wait()
```

### 1.6.1 Callback Registry

Subscriptions make use of `cafeteria.asyncio.callbacks.CallbackRegistry` internally to trigger registered callbacks when an event of a particular type is encountered. You can also register a *Coroutine* if required.

```
# both the following statements have the same effect
subscription = await client.subscribe(
    request=request, on_data=print, on_error=print
)
subscription = await client.subscribe(
    request=request, callbacks={
        GraphQLSubscriptionEventType.DATA: print,
        GraphQLSubscriptionEventType.ERROR: print,
    }
)

# this can also be done as below
registry = CallbackRegistry()
registry.register(GraphQLSubscriptionEventType.DATA, print)
registry.register(GraphQLSubscriptionEventType.ERROR, print)
```

If you'd like a single callback for all event types or any "unregistered" event, you can simply set the event type to *None* when registering the callback.

```
>>> registry.register(None, print)
```

Here is an example that will print the timestamp every time a keep-alive event is received.

```
subscription.callbacks.register(
    GraphQLSubscriptionEventType.KEEP_ALIVE,
    lambda x: print(f"Received keep-alive at {datetime.utcnow().isoformat()}")
)
```

CHAPTER 2

# Client API

## 2.1 GraphQLClient

**class** aiographql.client.**GraphQLClient**(*endpoint: str*, *headers: Optional[Mapping[str, str]] = None*, *method: Optional[str] = None*, *schema: Optional[graphql.type.schema.GraphQLSchema] = None*, *session: Optional[aiohttp.client.ClientSession] = None*)

Client implementation handling all interactions with a specified endpoint. The following example shows how to make a simple query.

```
client = GraphQLClient(
    endpoint="http://127.0.0.1:8080/v1/graphql",
    headers={"Authorization": "Bearer <token>"},
)
response: GraphQLResponse = await client.query("{ city { name } }")
```

You can also use an application scoped `aiohttp.ClientSession` throughout the life of the client as show below.

```
async with aiohttp.ClientSession() as session:
    client = GraphQLClient(
        endpoint="http://127.0.0.1:8080/v1/graphql",
        session=session
    )
```

**Parameters**

- **endpoint** – URI of graph api.
- **headers** – Default headers to use for every request made by this client. By default the client adds 'Content-Type: application/json' and 'Accept-Encoding: gzip' to all requests. These can be overridden by specifying then here.

- **method** – Default method to use when submitting a GraphQL request to the specified *endpoint*.
- **session** – Optional *aiohttp.ClientSession* to use when making requests. This is expected to be externally managed.

**get**(*request: aiographql.client.request.GraphQLRequest*, *headers: Optional[Dict[str, str]] = None*, *operation: Optional[str] = None*, *variables: Optional[Dict[str, Any]] = None*, *session: Optional[aiohttp.client.ClientSession] = None*) → aiographql.client.response.GraphQLResponse

Helper method that wraps :method: *GraphQLClient.query* with method explicitly set as *GraphQLQueryMethod.get*.

**Parameters**

- **request** – Request to send to the GraphQL server.
- **headers** – Additional headers to be set when sending HTTP request.
- **operation** – GraphQL operation name to use if the *GraphQLRequest.query* contains named operations. This will override any default operation set.
- **variables** – Query variables to set for the provided request. This will override the default values for any existing variables in the request if set.
- **session** – Optional *aiohttp.ClientSession* to use for requests

**Returns** The resulting *GraphQLResponse* object.

**get_schema**(*refresh: bool = False*, *headers: Optional[Dict[str, str]] = None*) → graphql.type.schema.GraphQLSchema

Get the introspected schema for the endpoint used by this client. If an unexpired cache exists, this is returned unless the *refresh* parameter is set to True.

**Parameters**

- **refresh** – Refresh the cached schema by forcing an introspection of the GraphQL endpoint.
- **headers** – Request headers

**Returns** The GraphQL schema as introspected. This maybe a previously cached value.

**introspect**(*headers: Optional[Dict[str, str]] = None*) → graphql.type.schema.GraphQLSchema

Introspect the GraphQL endpoint specified for this client and return a *graphql.GraphQLSchema* object specifying the schema associated with this endpoint.

**Returns** GraphQL schema for the configured endpoint

**post**(*request: aiographql.client.request.GraphQLRequest*, *headers: Optional[Dict[str, str]] = None*, *operation: Optional[str] = None*, *variables: Optional[Dict[str, Any]] = None*, *session: Optional[aiohttp.client.ClientSession] = None*) → aiographql.client.response.GraphQLResponse

Helper method that wraps *GraphQLClient.query* with method explicitly set as *GraphQLQueryMethod.post*.

**Parameters**

- **request** – Request to send to the GraphQL server.
- **headers** – Additional headers to be set when sending HTTP request.
- **operation** – GraphQL operation name to use if the *GraphQLRequest.query* contains named operations. This will override any default operation set.
- **variables** – Query variables to set for the provided request. This will override the default values for any existing variables in the request if set.

- **session** – Optional *aiohttp.ClientSession* to use for requests

**Returns** The resulting *GraphQLResponse* object.

**query**(*request: Union[aiographql.client.request.GraphQLRequest, str], method: Optional[str] = None, headers: Optional[Dict[str, str]] = None, operation: Optional[str] = None, variables: Optional[Dict[str, Any]] = None, session: Optional[aiohttp.client.ClientSession] = None*) → aiographql.client.response.GraphQLResponse

Method to send provided `GraphQLRequest` to the configured endpoint as an HTTP request. This method handles the configuration of headers HTTP method specific handling of request parameters and/or data as required.

**The order of precedence, least to greatest, of headers is as follows,**

1. client headers (`GraphQLClient.headers`)
2. request headers (`GraphQLRequest.headers`)
3. *headers* specified as method parameter

In accordance to the GraphQL specification, any non 2XX response is treated as an error and raises *GraphQLTransactionException* instance.

**Parameters**

- **request** – Request to send to the GraphQL server.
- **method** – HTTP method to use when submitting request (POST/GET). If once is not specified, the client default (*GraphQLClient.method*) is used.
- **headers** – Additional headers to be set when sending HTTP request.
- **operation** – GraphQL operation name to use if the *GraphQLRequest.query* contains named operations. This will override any default operation set.
- **variables** – Query variables to set for the provided request. This will override the default values for any existing variables in the request if set.
- **session** – Optional *aiohttp.ClientSession* to use for requests

**Returns** The resulting response object.

**subscribe**(*request: aiographql.client.request.GraphQLRequest, headers: Optional[Dict[str, str]] = None, operation: Optional[str] = None, variables: Optional[Dict[str, Any]] = None, callbacks: Union[cafeteria.asyncio.callbacks.CallbackRegistry, Dict[aiographql.client.subscription.GraphQLSubscriptionEventType, Union[Callable, Coroutine[T_co, T_contra, V_co], Callback, List[Union[Callable, Coroutine[T_co, T_contra, V_co], Callback]]]], None] = None, on_data: Union[Callable, Coroutine[T_co, T_contra, V_co], Callback, None] = None, on_error: Union[Callable, Coroutine[T_co, T_contra, V_co], Callback, None] = None, session: Optional[aiohttp.client.ClientSession] = None*) → aiographql.client.subscription.GraphQLSubscription

Create and initialise a GraphQL subscription. Once subscribed and a known event is received, all registered callbacks for the event type is triggered with the `aiographql.client.GraphQLSubscriptionEvent` instance passed in the first argument.

The following example will start a subscription that prints all data events as it receives them.

```
# initialise and subscribe to events in the background
subscription: GraphQLSubscription = await client.subscribe(
    request="{ notifications: { id, summary } }",
    on_data=lambda event: print(f"Data: {event}"),
    on_error=lambda event: print(f"Error: {event}"),
)
```

(continues on next page)

(continued from previous page)

```
# process events for 10 seconds then unsubscribe
await asyncio.wait(subscription.task, timeout=10)
subscription.unsubscribe()
```

**Parameters**

- **request** – Request to send to the GraphQL server.
- **headers** – Additional headers to be set when sending HTTP request.
- **operation** – GraphQL operation name to use if the *GraphQLRequest.query* contains named operations. This will override any default operation set.
- **variables** – Query variables to set for the provided request. This will override the default values for any existing variables in the request if set.
- **session** – Optional *aiohttp.ClientSession* to use for requests
- **callbacks** – Custom callback registry mapping an event to one more more callback methods. If not provided, a new instance is created.
- **on_data** – Callback to use when data event is received.
- **on_error** – Callback to use when an error occurs.
- **session** – Optional session to use for connecting the graphql endpoint, if one is not provided, a new session is created for the duration of the subscription.

**Returns** The resulting *GraphQLResponse* object.

**Returns** The initialised subscription.

**validate**(*request: aiographql.client.request.GraphQLRequest*, *schema: Optional[graphql.type.schema.GraphQLSchema] = None*, *headers: Optional[Dict[str, str]] = None*, *force: bool = False*) → None

Validate a given request against a schema (provided or fetched). Validation is skipped if the request's *validate* property is set to *False* unless forced.

**Parameters**

- **request** – Request that is to be validated.
- **schema** – Schema against which provided request should be validated, if different from *GraphQLRequest.schema* or as fetched from the client endpoint.
- **headers** – Headers to be set when fetching the schema from the client endpoint. If provided, request headers are ignored.
- **force** – Force validation even if the provided request has validation disabled.

## 2.2 GraphQLSubscription

**class** aiographql.client.**GraphQLSubscription**(*request: GraphQLRequest*, *headers: InitVar[Optional[Dict[str, str]]] = None*, *operation: InitVar[Optional[str]] = None*, *variables: InitVar[Optional[Dict[str, Any]]] = None*, *callbacks: Optional[CallbacksType] = <factory>*, *stop_event_types: List[GraphQLSubscriptionEventType] = <factory>*)

Subscription container, with an attached `cafeteria.asyncio.callbacks.CallbackRegistry`. When subscribed, the *task* will be populated with the `asyncio.Task` instance.

By default the subscription will be stopped, if an error, connection error or complete (*GraphQLSubscriptionEventType*) is received.

Subscription instances are intended to be used as immutable objects. However, *callbacks* and *stop_event_types* can be updated after initialisation.

**Parameters**

- **id** – A unique subscription identifier that will be passed into any events generated by this subscription.
- **callbacks** – A `CallbackRegistry` containing a mapping of *GraphQLSubscriptionEventType* callback methods to trigger.
- **stop_event_types** – Events that cause the subscription to stop. By default, connection error, query error or connection complete events received are considered stop events.

**active**() → bool

Check if the subscription is active.

**Returns** *True* if subscribed and active.

**connection_init_request**() → Dict[str, Any]

Connection init payload to use when initiating a new subscription.

**Returns** Connection initialise payload.

**connection_start_request**() → Dict[str, Any]

Connection start payload to use when starting a subscription.

**Returns** Connection start payload.

**connection_stop_request**() → Dict[str, Any]

Connection stop payload to use when stopping a subscription.

**Returns** Connection stop payload.

**handle**(*event: aiographql.client.subscription.GraphQLSubscriptionEvent*) → NoReturn

Helper method to dispatch any configured callbacks for the specified event type.

**Parameters** **event** – Event to dispatch callbacks for.

**is_stop_event**(*event: aiographql.client.subscription.GraphQLSubscriptionEvent*) → bool

Check if the provided *event* is configured as a stop even for this subscription.

**Parameters** **event** – Event to check.

**Returns** *True* if *event* is in *stop_event_types*.

**subscribe**(*endpoint: str*, *force: bool = False*, *session: Optional[aiohttp.client.ClientSession] = None*) → None
Create a websocket subscription and set internal task.

**Parameters**

- **endpoint** – GraphQL endpoint to subscribe to
- **force** – Force re-subscription if already subscribed
- **session** – Optional session to use for requests

**unsubscribe**() → None
Unsubscribe current websocket subscription if active and clear internal task.

CHAPTER 3

# Data Containers

## 3.1 GraphQLRequest

**class** aiographql.client.**GraphQLRequest**(*query: str*, *operation: InitVar[Optional[str]] = None*, *variables: Dict[str, Any] = <factory>*, *validate: bool = True*, *headers: Dict[str, str] = <factory>*)

GraphQL Request object that can be reused or used to store multiple named queries with default *operationName', *variables* and *header* to use.

**Parameters**

- **query** – GraphQL query string.
- **operation** – Optional name of operation to use from the query.
- **variables** – Variable dictionary pass with the query to the server.
- **validate** – If *True*, the request query is validated against the latest available schema from the server.
- **headers** – Headers to use, in addition to client default headers when making the HTTP request.

## 3.2 GraphQLResponse

**class** aiographql.client.**GraphQLResponse**(*request: GraphQLRequest*, *headers: InitVar[Optional[Dict[str, str]]] = None*, *operation: InitVar[Optional[str]] = None*, *variables: InitVar[Optional[Dict[str, Any]]] = None*, *json: Dict[str, Any] = <factory>*)

GraphQL Response object wrapping response data and any errors. This object also contains the a copy of the *GraphQLRequest* that produced this response.

**data**
The data payload the server responded with.

**errors**
A list of *GraphQLError* objects if server responded with query errors.

**query**
The query string used to produce this response.

## 3.3 GraphQLSubscriptionEvent

**class** aiographql.client.**GraphQLSubscriptionEvent**(*request: GraphQLRequest, headers: InitVar[Optional[Dict[str, str]]] = None, operation: InitVar[Optional[str]] = None, variables: InitVar[Optional[Dict[str, Any]]] = None, json: Dict[str, Any] = <factory>, subscription_id: Optional[str] = None*)

GraphQL subscription event wrapping the payload received from the server.

**Parameters** **subscription_id** – The id of the subscription that generated this event.

**id**
The id of the event, if available.

**payload**
The id of the subscription that generated this event.

**type**
The type of event (*GraphQLSubscriptionEventType*).

## 3.4 GraphQLError

**class** aiographql.client.**GraphQLError**(*extensions: Dict[str, Any] = <factory>, locations: Optional[List[Dict[str, int]]] = None, message: Optional[str] = None, path: Optional[List[Union[str, int]]] = None*)

GraphQL error response object.

**locations = None**

**message = None**

**path = None**

CHAPTER 4

# Constants

## 4.1 GraphqQLQueryMethod

**class** aiographql.client.**GraphQLQueryMethod**(*post: 'str' = 'post'*, *get: 'str' = 'get'*)

**get = 'get'**

**post = 'post'**

## 4.2 GraphQLSubscriptionEventType

**class** aiographql.client.**GraphQLSubscriptionEventType**

GraphQL Subscription Event Types

**COMPLETE = 'complete'**

**CONNECTION_ACK = 'connection_ack'**

**CONNECTION_ERROR = 'connection_error'**

**CONNECTION_INIT = 'connection_init'**

**CONNECTION_TERMINATE = 'connection_terminate'**

**DATA = 'data'**

**ERROR = 'error'**

**KEEP_ALIVE = 'ka'**

**START = 'start'**

**STOP = 'stop'**

CHAPTER 5

# Exceptions

## 5.1 GraphQLClientException

**class** aiographql.client.**GraphQLClientException**

## 5.2 GraphQLClientValidationException

**class** aiographql.client.**GraphQLClientValidationException**(**args*)

## 5.3 GraphQLRequestException

**class** aiographql.client.**GraphQLRequestException**(*response: GraphQLResponse*)

## 5.4 GraphQLIntrospectionException

**class** aiographql.client.**GraphQLIntrospectionException**(*message: Optional[str] = None*)

CHAPTER 6

# Indices and tables

- genindex
- modindex
- search

# Index

## K



## L



## M



## P



## Q



## S



## T



## U



## V