Async GraphQL Client

Arun Neelicattu, Maria Soulountsi, Josha Inglis

Mar 25, 2024

INTRODUCTION

An asynchronous GraphQL client built on top of AIOHTTP and GraphQL-core 3.

Feature Highlights

- client side schema introspection and validation
- `aiohttp` inspired API
- GraphQL query, mutation and subscription support

CHAPTER

ONE

DOCUMENTATION CONTENT

1.1 Getting started

This package is intended to be used as a client library for when your application requires interacting with a GraphQL server.

Getting started is as simple as passing your GraphQL query to `aiographql.client.GraphQLClient.query()`.

```
async def get_logged_in_username(token: str) -> GraphQLResponse:
    client = GraphQLClient(
        endpoint="https://api.github.com/graphql",
        headers={"Authorization": f"Bearer {token}"},
    )
    return await client.query("query { viewer { login } }")
```

For more detailed examples on how to use the library, see *Usage Examples*.

Hint: The JS GraphQL plugin allows for easier working with GraphQL and also adds auto-complete during development.

1.1.1 Adding to your project

You can add the the package to your project by specifying a dependency to aiographql-client.

If you are using Poetry to manage your project, the following command should do the trick.

```
poetry add aiographql-client
```

When using pip you can do the following.

```
pip install aiographql-client
```

1.2 Usage Examples

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

1.2.1 Queries

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 } }")
```

Query Variables

```
request = GraphQLRequest(
    query="""
        query($number_of_repos:Int!) {
          viewer {
             repositories(last: $number_of_repos) {
               nodes {
                 name
                 isFork
               }
             }
```

(continues on next page)

(continued from previous page)

```
            }
        }
    """,
    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
})
```

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 {
                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"
)
```

1.2.2 Subscriptions

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()
```

Hint: In the case you want to specify the GraphQL over WebSocket sub-protocol to use, you may do so by setting `aiographql.client.GraphQLSubscription.protocols`. For example, `await client.subscribe(..., protocols="graphql-ws")`. This is required for certain server implementations like Apollo Server as it supports multiple implementations.

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={
```

(continues on next page)

(continued from previous page)

```
        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()}")
)
```

1.2.3 Validation

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
```

(continues on next page)

(continued from previous page)

```
Process finished with exit code 1
```

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
{}
>>> 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'])]
```

1.3 Configuring HTTP Transport

1.3.1 Custom HTTP Client Sessions

The client allows you to specify a aiohttp Client Session for use at various levels. Including per query and/or for all queries made by the client.

This can be done so by passing in the session when doing any of the following;

1. creating a client

```
aiographql.GraphQLClient(
    endpoint="http://127.0.0.1:8080/v1/graphql", session=session
)
```

2. making a query

```
await client.query(
    request=request, session=session
)
```

3. creating a subscription

```
await client.subscribe(
    request=request, session=session
)
```

1.3.2 Using Behind SOCK Proxies

In order use via a socks proxy, you will need to custom connector, like the one provided by aiohttp-socks.

Here is an example code snippet using this library.

```
connector = aiohttp_socks.ProxyConnector(
    proxy_type=aiohttp_socks.ProxyType.SOCKS5,
    host="127.0.0.1",
    port=1080,
    rdns=True,
)
async with aiohttp.ClientSession(connector=connector) as session:
    client = GraphQLClient(
        endpoint="http://gql.example.com/v1/graphql", session=session
    )
    await client.query(request="query { city { name } }")
```

1.4 Python API

1.4.1 Client

GraphQLClient

GraphQLSubscription

1.4.2 Data Containers

GraphQLRequest

GraphQLResponse

GraphQLSubscriptionEvent

GraphQLError

1.4.3 Constants

GraphQLQueryMethod

GraphQLSubscriptionEventType

1.4.4 Exceptions

GraphQLClientException

GraphQLClientValidationException

GraphQLRequestException

GraphQLIntrospectionException

1.5 Contributing

1.5.1 Reporting Issues

If you are using this package and are encountering issues, please feel free to raise them at on our issue tracker.

1.5.2 Providing Feedback

This client was developed with a limited set of use cases in mind. And therein, can be biased in various places. Any feedback regarding what could be improved, added or changes can also be submitted similar to reporting issues, on our issue tracker.

1.5.3 Code Contributions

Code contributions in the form of bugfixes, improvements or new features are always welcome. All that we ask, is that you do make sure the rationale for the change is described if it is more than a simple change. The current open change requests can be seen are available here.

1.5.4 Documentation

Improvements and additions to our documentation are always welcome. From typo fixes, to documenting undocumented features, or structural changes. Everything is welcome.

1.6 Index

- genindex