GraphQL Documentation

Xledger has recently added a GraphQL API, which opens a lot of integration possibilities. If you are unfamiliar with the concepts in GraphQL, we suggest checking out the tutorials here. In this document, we'll assume you are familiar with the basics, and cover:

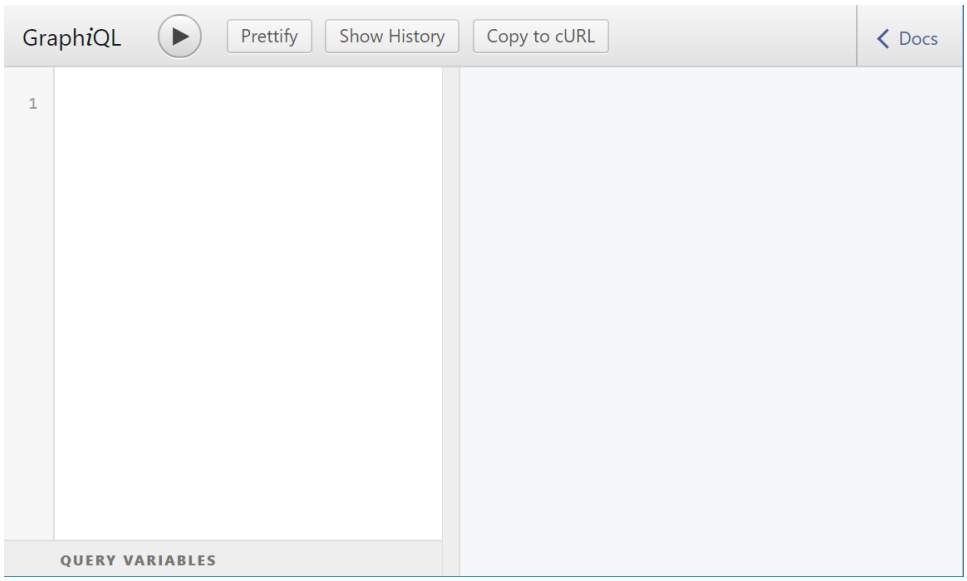
- How to explore the Xledger Schema
- How to connect to the API programmatically
- Querying: Filtering, sorting, pagination
- A few Xledger specific concepts

Note: This document will assume you are trying to connect to www.xledger.net. If you are connecting to a test environment (for example, test.xledger.net), replace the hostname of the URLs in this document accordingly.

Table of Contents

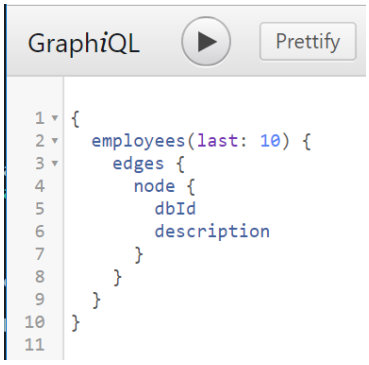
Exploring the Schema

To be able to start exploring the schema, you need to have access to the "Administrator", “Domain Administrator”, or “Implementation Manager” roles in Xledger. After you have logged in and switched to that role, navigate to www.xledger.net/GraphQL to begin. You should then see something like this:



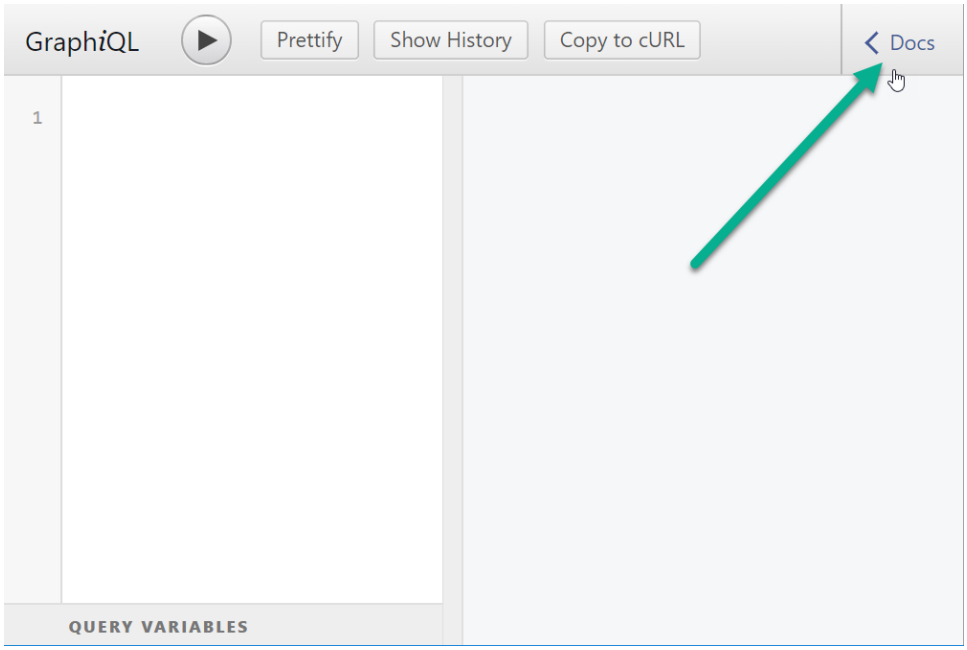
Note: If you have not logged in with the right credentials, you'll get an error saying you do not have access to GraphQL.

Try typing in a query like this, and then hit the Play button:

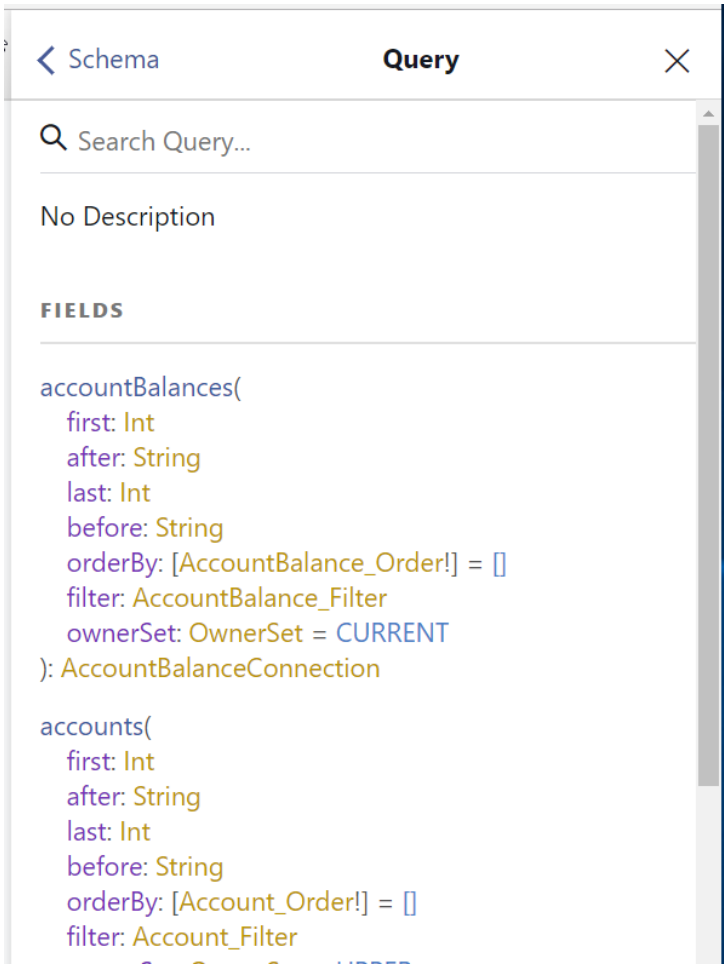


You should see a JSON response of employees to the right.

To view the schema, click on the "Docs" button in the top right:



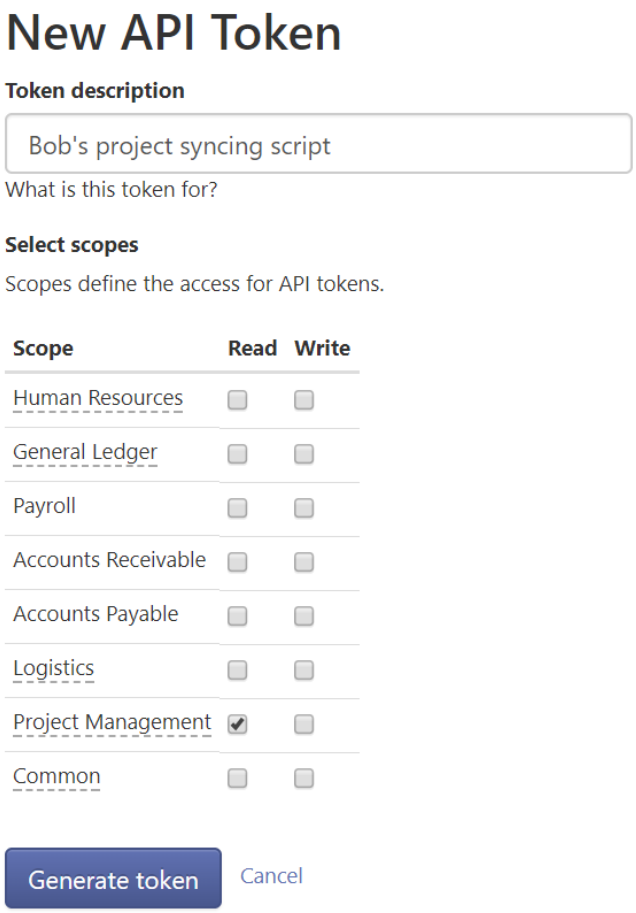
If you then click on query, you'll see a list of the top-level fields that are available:



Connecting to the API Programmatically

To connect to the API programmatically, first go to this screen to create an API token:

https://www.xledger.net/f/api-tokens



You will be able to create a token, give it a name, and select what type of information it has access to. Once you have a token, you can make a request like this:

Uri	`https://www.xledger.net/graphql`	
HTTP Method	POST	
Header: Authorization	`token <your-token-here>`	
Body	`{ "query": "<your-query-string>", "variables": <variables object or null>, "operationName": <operation name or null>}`	JSON

Example using cURL:

```
curl -H "Authorization: token <your-token-here>"
      https://master.xlabs.com/graphql --data-binary '{"query": "{
      employees(last: 10) { edges { node { dbId description } } }
      }","variables":null,"operationName":null}'
```

Here is a basic example that shows how to connect to the API, and loop through a list of employees in C# (open in LINQPad).

Querying: Sorting and Filtering

GraphQL queries in Xledger come in two forms: one for fetching a single record, and one for fetching multiple. For example, the type "Employee" has both top level fields, "employee(…)" for fetching 1 by id, and "employees(…)" to fetch many.

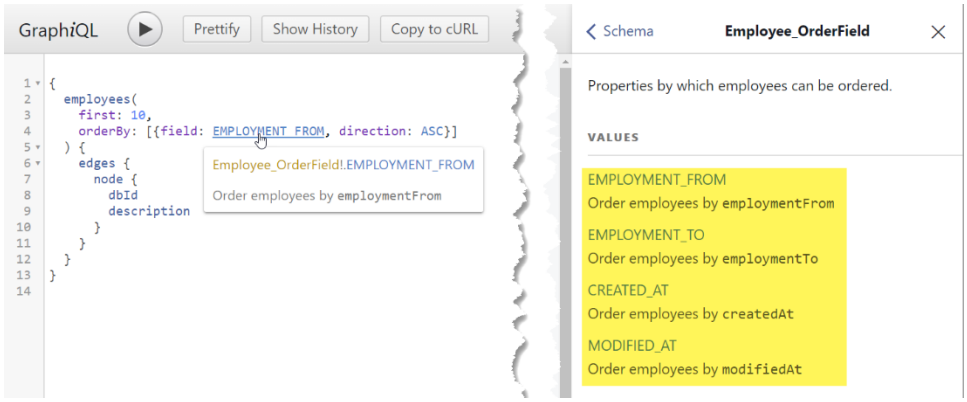
Fetching one by Id	Fetch many

`employee(dbId: 130482) {`
`  dbId`
`  description`
`  dateFrom`
`  dateTo`
`}`
	`employees(last: 10) {`
	`  edges {`
	`  node {`
	`  dbId`
	`  description`
	`  }`
	`  }`
	`}`

When fetching many, for some types you can specify the order to get them back in. For example, if you wanted to fetch the 10 employees with the earliest start date, you would issue this query:

```
{
  employees(
    first: 10,
    orderBy: [{field: EMPLOYMENT_FROM, direction: ASC}]
  ) {
    edges {
      node {
        dbId
        description
      }
    }
  }
}
```

If you click on the "EMPLOYMENT_FROM" field, you can see a list of other fields you can sort on:



To filter the results, you can specify the "filter" argument. For example, to only return results modified since January 1, 2017, issue this query:

```
{
  employees(
    first: 10,
    orderBy: [{field: EMPLOYMENT_FROM, direction: ASC}],
    filter: { modifiedAt_gte: "2017-01-01"}
  ) {
    edges {
      node {
        dbId
        description
      }
    }
  }
}
```

Filters can be combined with other filters by using "AND" or "OR" and then specifying a list of conditions. For example, this query will get rows that either have been modified since January 1, 2017, or where the `dateTo` is less than December 31, 2017:

```
{
  employees(
    first: 10,
    orderBy: [{field: EMPLOYMENT_FROM, direction: ASC}],
    filter:
     { OR: [{ modifiedAt_gte: "2017-01-01" }
            { dateTo_lt: "2017-12-31" }]}
  ) {
    edges {
      node {
        dbId
        description
      }
    }
  }
}
```

Querying: OwnerSet and Object Status

Many concepts in Xledger are defined in a hierarchical fashion, so that a company can see rows defined above or below them in the owner hierarchy. For types where this makes sense, we add an argument for "ownerSet". For example, is a query for products that only looks at the current level (instead of UPPER, which is the default in this case):

```
{
  products(
    first: 10
    ownerSet: CURRENT
  ) {
    edges {
      node {
        id
        description
      }
    }
  }
}
```

Similarly, many concepts in Xledger have the notion of an "object status", where objects can be closed or opened at will. For types where this applies, we add an "objectStatus" field which can be either OPEN (the default), CLOSED, or ALL.

Querying: Pagination

For pagination, Xledger supports the Relay pagination specification. To give this a try, modify the earlier employee query, adding these parts:

```
{
  employees(first: 10) {
    pageInfo {
      hasNextPage
    }
    edges {
      cursor
      node {
        dbId
        description
      }
    }
  }
}
```

When you hit play, you will then get a field indicating whether there is a next page of results, as well as a cursor. If there is a next page, and you want to get the next one, specify the "after" field, and use one of the cursors (for example, the last one) to get the first N records after that one. For example:

```
{
  employees(first: 10, after: "124813.124929.124998") {
    pageInfo {
      hasNextPage
    }
    edges {
      cursor
      node {
        dbId
        description
      }
    }
  }
}
```

If you want to go backwards instead, you can the "last" and "before" fields instead of "first" and "after".

Querying: History

Some resources in Xledger are audit friendly, in that changes made to them get logged. For these, you can access the changes by using the `_changes` meta field:

```
{
  subledgers(last: 1000) {
    edges {
      node {
        id
        _changes(last: 10) {
          edges {
            node {
              description
              code
              modifiedAt
              changeType
              changedByUser { description }
            }
          }
        }
      }
    }
  }
}
```

The results will be sorted chronologically, and will have a non-null value for the fields you fetch (e.g., `description`) if that field was inserted or updated. If it did not change in an update, it will be `null`. The `changeType` special field will always have a value.

Alternatively, if you want to see all changes to a resource, not starting from an individual instance, you can use the appropriate connection field, e.g., `subledger_changes`.

Rate Limiting

Requests against the GraphQL API are rate limited, so that each company can only make so many requests each hour. Each query or mutation has a cost, which depends on the number of records and fields requested. To see how much a query costs, and how many credits your company has remaining, query for the special field `rateLimit`:

```
{
  rateLimit {
    cost
    limit
    resetAt
    remaining
  }

  employees(first: 10) {
    edges {
      cursor
      node {
        dbId
        description
      }
    }
  }
}
```

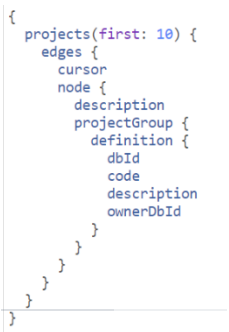
Mutations also have a cost, but unfortunately, you cannot query this field in a mutation, since one request cannot mix queries and mutations in GraphQL.

Conserving Credits

To conserve credits, be tactical with the kind of queries you make.

If you need to sync data, instead of querying for and updating every record, consider asking for only the records that have been modified in the last N (hours/days).

When asking for data in related registers, make sure you are not repeating information. For example, instead of writing queries like Figure 1, write a query that just fetches the projects and the dbId of the projectGroups, and then fetch the projectGroups in the next query.



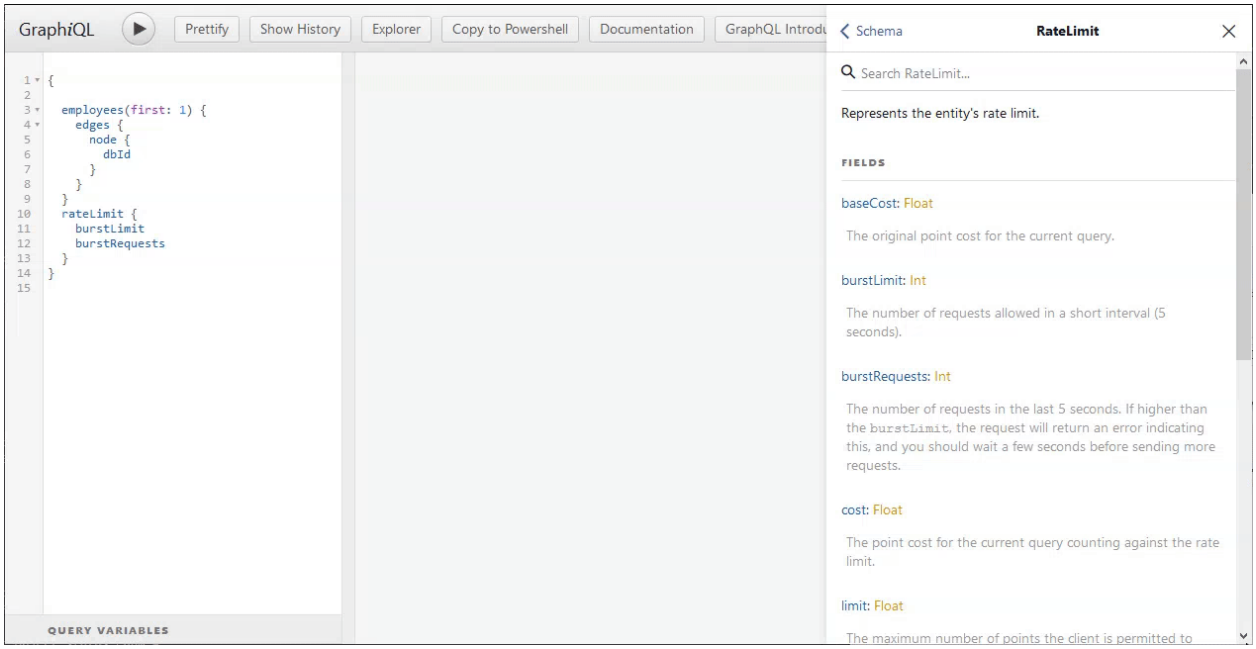
In some cases, you can also start from “header” rows, and then get the details for each (like SalesOrder and SalesOrderDetail), which would achieve the same result.

When asking for a lot of data, (e.g., downloading all your transactions), make sure you are using the biggest page size you can (10 000). The query cost does not scale linearly with the number of rows, so this will be much more efficient credits-wise.

When in doubt, keep asking for the rateLimit field, and pay attention to the “cost” in the response as you make changes to the query.

Burst Traffic Limit

In addition to the credit limit, there is one other limit to be aware of. In order to keep Xledger responsive for everyone, there is a limit to how many requests we accept in a short time window (5 seconds). Too much burst traffic will result in an error, and you should wait a few seconds before sending more requests.



Example Application

Here is an example application that randomly generates and inserts contacts, all while respecting the credit and burst limit, and firing requests as quickly as possible:

http://share.linqpad.net/cbcexe.linq

Note how this program doesn't hardcode all the delays - it just keeps making requests until it gets told to wait, and then it adds a delay before continuing. As you can see, this is simple to do by just reacting to the error messages. The benefit of this approach is that it will keep acting optimally if the GraphQL rate limits are relaxed(*).

* They are relaxed for credits at night time CET - see 'slackPeriods' in the schema documentation.

Mutations

Mutations in Xledger are fairly self-explanatory, but here are a couple of examples.

Adding a new supplier:

```
mutation {
  addSupplier(code: "WATTO", description: "Watto's Emporium") {
    dbId
  }
}
```

Updating a supplier by Id:

```
mutation {
  updateSupplier(dbId: 23538892, code: "WATTOS" ) {
    dbId
  }
}
```

File Attachments

For some mutations, you may want to attach a file with the request. Examples of this would be `InvoiceBase` and `ExpenseBase`. To do this:

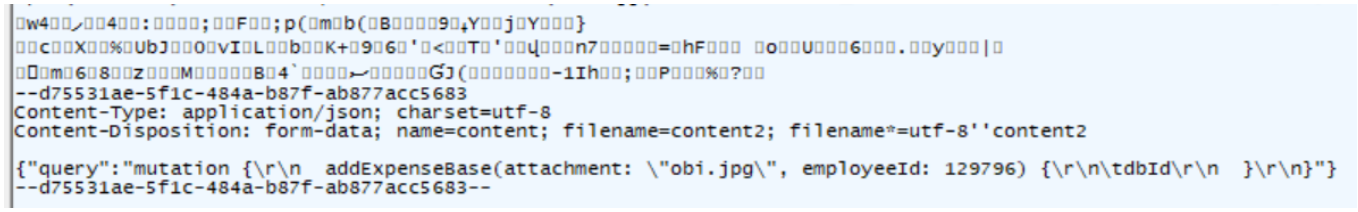
1. Instead of sending a pure JSON request, change to sending a Multipart request.
2. Add one or more file content parts
3. Add a final part (with `Content-Type: "application/json"`). This part can refer to filenames you provided in step 2.

Here is an example in C#: (LINQPad link)

```
async Task<string> MakeRequest() {
    var image_path = @"C:\Users\Bob\Desktop\obi.jpg";

    using (var client = new HttpClient())
    using (var content = new MultipartFormDataContent()) {
        if (!client.DefaultRequestHeaders.TryAddWithoutValidation("Authorization", $"token {TOKEN}")) {
            throw new Exception("Could not add header");
        }

        content.Add(new StreamContent(new FileStream(image_path, FileMode.Open)), "obi", "obi.jpg");

        var json = new Dictionary<string, object> {
            ["query"] = @"mutation {
  addExpenseBase(attachment: ""obi.jpg"", employeeId: 129796) {
    dbId
  }
}"
        };

        var json_str = Newtonsoft.Json.JsonConvert.SerializeObject(json);

        content.Add(new StringContent(json_str, Encoding.UTF8, "application/json"), "content", "content2");

        using (var message = await client.PostAsync("https://www.xledger.net/graphql", content)) {
            var input = await message.Content.ReadAsStringAsync();
            return input;
        }
    }
}
```

If you have done everything correctly, the raw request will look something like this:

Figur 1 - The beginning of the request



Figur 2 - The end of the request

Updating multiple records in one request

To create or update multiple records in one request, use the bulk mutation fields (e.g., addProjects, updateProjects) and send multiple inputs. For example:

```
mutation ($newProjects: [AddProjectsInput!]!) {
  addProjects(inputs: $newProjects) {
    edges { node { dbId }}
  }
}
```

QUERY VARIABLES

```
{
  "newProjects": [
    {"clientId": "p1", "node": {"description": "Pump Dogecoin", "subledger": {"code": "ELON"}}},
    {"clientId": "p2", "node": {"description": "Water Sanitization", "projectManager": {"code": "FLINTMI"}}},
    {"clientId": "p3", "node": {"description": "Jeff Bezos PR support", "subledger": {"code": "WASHPO"}}}
  ]
}
```

You can use what you pass for clientId to correlate the results in the response.

Subscriptions (beta)

Subscriptions is a feature of GraphQL that allows you to get notifications in real time in response to events (such as records being inserted, updated or deleted). The GraphQL specification does not specify which transport mechanism one should use for subscriptions, but we decided on starting with websockets, since that is the easiest to implement robustly - we think both for us, and for our customers.

Getting started with subscriptions

Here are the steps you need to take to start using subscriptions:

1. Send a WebSocket handshake request (GET /graphql) to the endpoint you want to connect to (for example: www.xledger.net)
2. Send a "connection_init" JSON text message, like this:

```
{
    "type":"connection_init",
    "payload":{
        "headers":{
            "Accept":"application/json",
            "Content-Type":"application/json",
            "Authorization": "token {apiToken}"
        }
    }
}
```

3. After connecting, you have 30 seconds to start a subscription before your inactive connection will be closed by the server. To start a subscription, send a message in the following form:

```
{
        "type":"start",
        "id":1,
        "payload":{
            "query":"subscription { projectsMutated { edges { node {
description } } } }",
            "variables":null
        }
}
```

The **query** above should of course be the subscription query that you want to get notifications for. The **id** above is unique in the context of a connection, and allows a client to start many concurrent subscriptions on a single socket, and then identify which subscription they get a message for, as well as stopping it at any time. After starting a subscription like the above, you will get messages like this when the relevant events occur:

```
{
    "type":"data",
    "id":1,
    "payload":{
        "data":{
```

```
                "projectsMutated":{
                    "edges":[
                        {"node":{"description":"Acme Company"}}
                    ]
                }
            }
        }
    }
```

4. If you want to stop a subscription, you can send a message like this with the **id** of the subscription you want to stop:

```
{
    "type":"stop",
    "id":1
}
```

Note: If your websocket connection is closed, all subscriptions are stopped automatically.

Special messages

If an error occurs, you will get a message in this form:

```
{
    "type":"error",
    "payload":{
        "errors":[
            {
                "message":"Something went awry."
            }
        ]
    }
}
```

Another message you may get is a 'keep-alive' message, which we may send periodically to prevent the websocket connection from closing. It will look like this:

```
{ "type": "ka" }
```

You may safely ignore it.

Subscriptions and API Credits

API Credits are charged with subscriptions in three different ways:

1. A client is charged when a web socket is opened (5 credits).

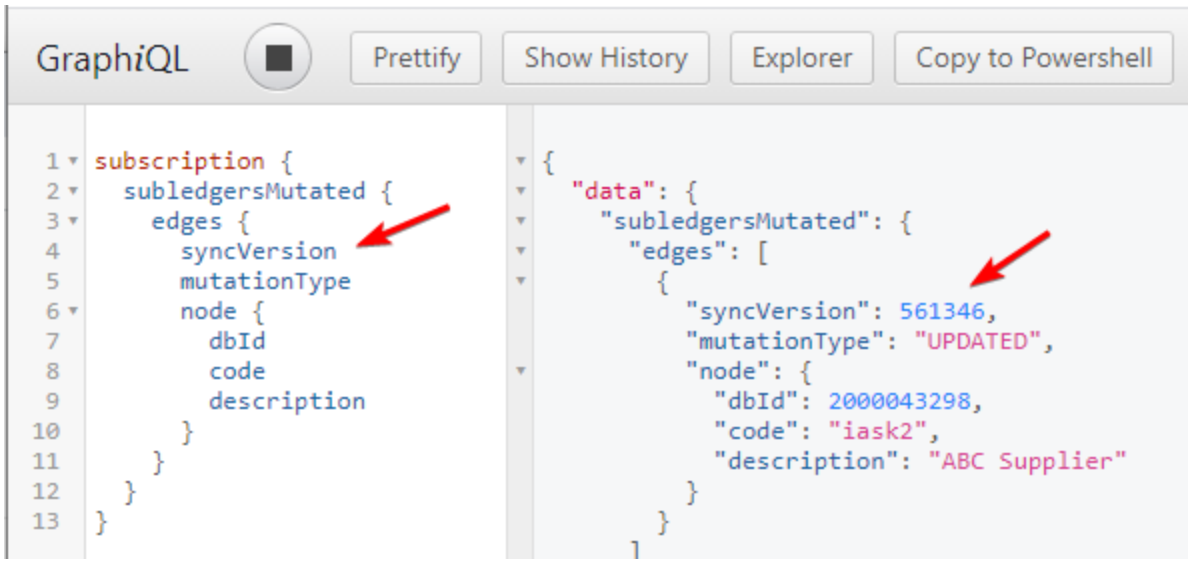
2. A client is charged for each minute a subscription is active (0.5 credits).

3. A client is charged by the complexity of the query when returning data for a subscription (as though the subscription were a regular query).

If the server's attempt to charge a client one of those credit costs would take that client beyond their hourly limit, the server will report an error and close the connection.

Surviving disconnects

To make you get all messages, and can process all notifications in case you get disconnected, ask for the syncVersion field:



If you keep track of the latest syncVersion you have seen, you can provide it when starting the subscription (via the "lastSyncVersion" argument), and we will send you all the notifications that have happened since then. You have up to 3 days to reconnect and get caught up. After that, the notifications will not be available anymore.

Example subscription program

Here is a link to an example program that listens for changes in subledgers and prints the messages to the console. It is runnable with Linqpad version **5**.

http://share.linqpad.net/n84q4b.linq

Try running this program, then making changes to suppliers via the UI. You will see notifications in the console about the changes.

Recent Changes

Note: This section only covers general changes, not changes to specific fields or mutations.

2021 R2 - Nov 20th 2021 (planned)

This release, we added a way to redirect which owner to charge for query credits. You can redirect credits that would otherwise be charged to the owner of the API key you are querying with by adding a header called `X-XL-Owner-To-Charge`, with the value being the `dbId` of the owner that should be charged instead.

2021 R1 - June 12th 2021

New bulk mutation support

This release, we added fields such as "addProjects", which allows you to insert many records at once. The limit on the number is usually 500, but is lower (5) for some fields such as "addUsers".

```
mutation {
  addProjects(inputs: [
    {clientId: "1", node: {code: "GH", description: "Integrate with Github" }}
    {clientId: "2", node: {code: "EP2021", description: "End of year party for 2021" }}
  ]) {
    edges {
      clientId
      node { dbId }
    }
  }
}
```

These new bulk mutation fields are more efficient, run in a transaction (i.e., all of the updates succeed, or none), cost you less credits per item, and should be easier to use.

Improved error messages for reference input arguments

When using our mutation fields and you provided an invalid value, up until this release, we could only provide error messages such as "The value 456 is not valid or allowed" for a reference input argument like "projectDbId". With this release, we either provide an explanation for the failure, or give a few examples of values that are valid.

```
{
  "errors": [
    {
      "path": [
        "addEmployee"
      ],
      "message": "Argument: \"glObject1DbId\" - the value 1 is not valid or allowed.",
      "extensions": {
        "exampleValidInputs": [
          {
            "dbId": 4024082,
            "text": "Consulting and Support"
          },
```

Alternative input methods for reference input arguments

You can now refer to other types of objects (e.g., Project, Employee) when doing a mutation without necessarily using our dbIds. For example, if you have a code, and that is enough to uniquely identify an object, you can use that:

Old way	`mutation { addProject(code: "Colonize Mars", subledgerDbId: 6237215) { dbId } }`
New way	`mutation { addProjects(inputs: [{node: {description: "Colonize Mars", subledger: {code: "ELON"}}}]) { edges { node { dbId }} } }`