# Using the ScienceLogic GraphQL API

SL1 version 12.3.0

# Table of Contents


Introduction 5
What is GraphQL? 5
How Does SL1 Use GraphQL? 6
Authentication and User Access 6
Using Request Headers 6
The GQL Browser 6
GraphQL Terminology and Punctuation 8
GraphQL Terminology 8
Schema 8
Types 9
Operations 9
Queries 9
Mutations 9
Object Types 9
Fields 9
Arguments 9
Return Types 10
Connections 10
Edges 10
Nodes 10
Variables 10
Fragments 10
Directives 11
Case-Sensitivity 11
Punctuation 11
Using the GQL Browser 13
GQL Browser and GraphiQL User Interface Elements 14
Query Pane 14
Variables Pane 14
Headers Pane 15
Results Pane 15

Docs Pane ..... 15
Additional Options ..... 15
The GraphQL Schema ..... 16
Exploring the GraphQL Schema in the Docs Pane ..... 17
Example 1: Using the Schema to Execute a Simple Query ..... 18
Example 2: Using the Schema to Execute a Complex Query ..... 22
Example 3: Using the Schema to Execute a Mutation ..... 32
Queries ..... 36
What Are Queries? ..... 37
Basic Query Syntax ..... 37
Example: Querying a Single Device ..... 37
Example: A Simple Query without the Operation Type "query" ..... 38
Connections, Edges, and Nodes ..... 39
Example: A Query with Edges and Nodes ..... 39
Variables ..... 41
Example: Querying Basic Device Information Using a Variable ..... 42
Fragments ..... 43
Example: Creating a Fragment for Account Fields ..... 44
Inline Fragments ..... 44
Example: Using an Inline Fragment ..... 45
Directives ..... 45
Example: Using a Directive with a Variable ..... 46
Search Filters ..... 46
Filter Operators ..... 47
Example: Querying Devices from the Same Device Class ..... 48
Example: Querying a List of Specific Devices ..... 48
Query Pagination ..... 49
Example: Querying the First Five Devices Listed in SL1 ..... 50
Example: Determining the Total Number of Query Results ..... 50
Example: Using Cursor-based Query Pagination ..... 52
Querying the ScienceLogic GraphQL API from an External Application ..... 54
Mutations ..... 55

What are Mutations? 55
Mutation Types 56
Basic Mutation Syntax 56
Example: Updating a User's Name 56
Variables 57
Example: Using Variables in a Mutation 57
Fragments 58
Example: Using a Fragment in a Mutation 58
Directives 59
Example: Using a Directive in a Mutation 59

# Chapter 1

# Introduction

## Overview

This manual describes the functionality of the SL1 GraphQL API and is intended for developers who are responsible for integrating SL1 with external systems.

This manual includes the following topics:

This chapter covers the following topics:


*What is GraphQL?* ........ 5

*How Does SL1 Use GraphQL?* ........ 6

*Authentication and User Access* ........ 6

*The GQL Browser* ........ 6


## What is GraphQL?

***GraphQL*** is a data query and manipulation language for application programming interfaces (APIs) and a server-side runtime for executing queries and mutations with your data. It lets you define the exact data that you want to fetch from the API and, unlike REST, enables you to retrieve data from multiple sources in a single query.

GraphQL—sometimes abbreviated "GQL"—has two main operation types that are used in SL1:

- ***Queries***, which read and return data
- ***Mutations***, which write and modify data

Typically used over HTTP, GraphQL lets you make a POST request using a JSON structure to an HTTP endpoint on your server; the server then processes the request, calls the database or other source to fetch the requested data, and then returns the data to you in the same JSON structure as your request.

GraphQL is data source-agnostic. This means it can query data from any source: Databases, REST APIs, microservices, flat files, or even other GraphQL servers.

> **NOTE:** For more information about GraphQL, see the GraphQL documentation.

# How Does SL1 Use GraphQL?

The ***SL1 GraphQL API*** allows external systems to programmatically access data in SL1 using the GraphQL query language. The API gives access to entities in the platform, such as devices and collected data, using standard HTTP request/response protocols.

The default SL1 user interface, also referred to as "AP2", uses GraphQL exclusively to request and access SL1 data from multiple sources and consolidate it together into a unified JSON response. The default SL1 user interface does not make any REST API calls.

# Authentication and User Access

The SL1 GraphQL API uses the same authentication profiles and credentials as SL1. When you log in to SL1, you are also authenticated in the SL1 GraphQL API. It does not require an additional login.

All users have access to the SL1 GraphQL API—not only administrators—but users' ability to perform actions in GraphQL is controlled in the same way as in other parts of SL1:

- A user can interact only with entities associated with their organizations. Entities are either explicitly aligned with organizations, aligned with organizations based on the user that created the entity, or are not aligned with an organization.
- Administrator users can perform all actions on all resources, regardless of organization membership.
- A user must be granted the appropriate access keys to interact with specific parts of SL1 or perform specific actions in SL1.

## Using Request Headers

Using a request, or "run-as", header allows you to run your query as another user when it is passed to the GraphQL API. The `X-sl-run-as` header is only available for use as an administrator; other users attempting to use it will receive a 404 error. To use the `X-sl-run-as` header you must set the value of the header to the user you want to run as.

# The GQL Browser

To explore and perform operations in the SL1 GraphQL API, you can use one of the following options, depending on which version of SL1 you are using:

- The **GQL Browser** page (System > Tools > GQL Browser) is a page built into SL1 versions 12.3.0 and later.

- ***GraphiQL*** is a browser-based user interface that is used in SL1 versions 12.2.x and earlier. To access the GraphiQL interface, type the URL or IP address for SL1 in a browser, add **/gql** to the end of the URL or IP address, and press **[Enter]**.

> **NOTE:** GraphiQL is not maintained by ScienceLogic. You can access its documentation at https://github.com/graphql/graphiql.

Both of these methods use similar elements that enable you to interactively explore the capabilities of, and execute queries against, the SL1 GraphQL API. Unlike the SL1 REST API, all requests to the GraphQL API on a given SL1 system use the same URI.

For more information about using the GQL Browser or GraphiQL, see the chapter on ***Using the GQL Browser***.

Chapter

2

# GraphQL Terminology and Punctuation

## Overview

This chapter defines some of the most commonly used GraphQL terminology and syntax punctuation.

This chapter covers the following topics:

*GraphQL Terminology* ........ 8

*Case-Sensitivity* ........ 11

*Punctuation* ........ 11

## GraphQL Terminology

This section defines some of the common terminology you will encounter when using GraphQL in SL1.

> **NOTE:** This is not an exhaustive list of GraphQL terminology. For other GraphQL terminology definitions, see the GraphQL documentation.

### Schema

A ***schema*** describes all of the system data that you can request or modify in GraphQL. It defines the full hierarchy of types and fields, and all of the possible queries and mutations that are available in the GraphQL API.

For more information, see the section on ***The GraphQL Schema***.

## Types

***Types*** are the fundamental units of any GraphQL schema. There are multiple kinds of types that are defined in the schema; many of these types are defined below and described in further detail throughout this manual.

## Operations

***Operations*** are executable actions within a GraphQL document. There are three types of operations in GraphQL: Queries, mutations, and subscriptions. The SL1 GraphQL API supports queries and mutations, which are described in greater detail in this manual; it does not support subscriptions. For more information about subscriptions, see the GraphQL documentation.

### Queries

***Queries*** are GraphQL operations that search for and return data from fields in your schema. A single query can search for and return data for a single field or multiple fields.

For more information, see the chapter on ***Queries***.

### Mutations

***Mutations*** are GraphQL operations that modify data in the system and then return a value. Typically, mutations create new data, update existing data, or delete existing data.

For more information, see the chapter on ***Mutations***.

## Object Types

***Object types*** are the most basic components of a schema, representing objects that you can fetch from SL1. All object types have a name and one or more fields.

For more information, see the section on ***Basic Query Syntax***.

## Fields

***Fields*** are units of data within object types. Each field has a name, potentially a list of arguments, and a return type. Every GraphQL query requests one or more fields from an object, while mutations modify one or more fields.

For more information, see the section on ***Basic Query Syntax***.

## Arguments

***Arguments*** are parameters, written as `field name: value type` pairs, that are passed into a field in a query or mutation. They help you define exactly what data you want GraphQL to search for and return in your query or provide specific parameters for your mutation.

Most argument fields in the schema include one of the following defined scalar types as a value type:

- *Int*: A signed, 32-bit integer
- *Float*: A signed, double-precision, fractional value
- *String*: Textual data, represented as UTF-8 character sequences
- *Boolean*: `true` or `false`
- *ID*: A unique identifier, often used to fetch an object or as key for a cache.

Additionally, there are other value types in the SL1 GraphQL schema beyond those scalar types listed above. Examples of these other types include ***connection types*** and ***search types***.

For more information, see the section on ***Basic Query Syntax***.

## Return Types

***Return types*** specify the manner in which field data resolves in an operation. Typically, return types indicate the specific fields that you want GraphQL to return values for when you execute the operation.

For more information, see the section on ***Basic Query Syntax***.

## Connections

***Connections*** are types within the GraphQL schema that are used to connect other defined elements in the schema. A connection consists of a group of related edges.

For more information, see the section on ***Connections, Edges, and Nodes***.

## Edges

***Edges*** are types that connect two nodes, representing some sort of relationship between them. Edge types must have at least two fields: `node` and `cursor`.

For more information, see the section on ***Connections, Edges, and Nodes***.

## Nodes

***Nodes*** are individual object types that are defined in the schema, consisting of one or more fields.

For more information, see the section on ***Connections, Edges, and Nodes***.

## Variables

***Variables*** are dynamic values that can be used to replace arguments in your operation, enabling you to reuse the operation for multiple objects by changing the variable value.

For more information, see the section on ***Variables***.

## Fragments

***Fragments*** are reusable units that you can include in multiple queries or mutations. Each fragment consists of a group of fields that are all associated with the same type.

For more information, see the section on ***Fragments***.

## Directives

***Directives*** are keywords that you can use to make GraphQL perform custom logic in your operations. They can be attached to a field or a fragment that you are including in your operation, and can affect the operation execution and the results that GraphQL fetches.

For more information, see the section on ***Directives***.

# Case-Sensitivity

The GraphQL schema is full of named elements. Operations, objects, fields, arguments, types, variables, fragments, and directives all have names.

Names in GraphQL are limited to ASCII characters and are case-sensitive. For example, in the SL1 GraphQL schema, the object name `account` is different from the type name `Account`. Therefore, when forming queries and mutations, it is important to double-check that all of the included elements are using the correct case.

# Punctuation

GraphQL uses certain special characters as punctuation to help define the structure of data. This section defines some of the common punctuation you will encounter when using GraphQL in SL1.

**NOTE:** This is not an exhaustive list of GraphQL punctuation. For other GraphQL punctuation and special characters, see the GraphQL documentation.

When browsing the schema or writing your queries or mutations, you might encounter or use the following punctuation:

| Punctuation | Function |
|---|---|
| `{ }` | Pass input objects and fields to the query or mutation object. They also represent the indentation levels within the hierarchy of the query or mutation. |
| `( )` | Pass parameters such as arguments to query or mutation objects. |
| `:` | Define a field or a type, such as in a `field: value` pair used in an argument or when defining a variable. |

| Punctuation | Function |
|---|---|
| `!` | Indicates a non-nullable value type.<br><br>By default, all value types in GraphQL can result in a null value. If a value type includes an exclamation point, it means that value cannot be null. The implications of a non-nullable type vary slightly based on the context in which it is used:<br><br>• When used with an input field, such as an argument or variable, it means that field is required; it must be included and have a value.<br>• When used with an output field, such as in a return type field, it means that GraphQL cannot return a null value for that field. If it does return a null value, the response will fail validation and trigger a GraphQL execution error, indicating that something has gone wrong.<br><br>Can also be used as a ***filter operator***. |
| `[ ]` | Indicates a list value type. The implications of a list type vary slightly based on the context in which it is used:<br><br>• When used with an input field, such as an argument, GraphQL accepts the list values only when each item in the list can be accepted by the list's item type. For example, if the list's expected item type is an integer (`Int`), GraphQL would accept the list values `[1, 2, 3]` but it would not accept the list values `[1, 2, c]`.<br>• When used with an output field, such as in a return type field, GraphQL must return an ordered list as a result of the list type. If it does not return an ordered list, the response will fail validation and trigger a GraphQL execution error, indicating that something has gone wrong. |
| `$` | Indicates a variable name. For more information about variables, see the section on ***Variables***. |
| `=` | Indicates the default value of a variable. For more information about variables, see the section on ***Variables***.<br><br>Can also be used as a ***filter operator***. |
| `< >` | Can be used together or separately as ***filter operators***. |
| `...` | Indicates a fragment name. For more information about fragments, see the section on ***Fragments***. |
| `@` | Indicates a directive, such as the default directives `@include` and `@skip`. For more information about directives, see the section on ***Directives***. |
| `#` | Used at the beginning of a line within an operation to indicate a line that GraphQL can ignore when executing that operation.<br><br>For example, you could include comments in your query by preceding them with the `#` symbol at the beginning of the line and GraphQL will ignore them when executing the query. |

Chapter

# 3

# Using the GQL Browser

## Overview

This chapter describes the user interface options that are available for using the GraphQL API in SL1.

To explore and perform operations in the SL1 GraphQL API, you can use one of the following options, depending on which version of SL1 you are using:

- The **GQL Browser** page (System > Tools > GQL Browser) is a page built into SL1 versions 12.3.0 and later.
- ***GraphiQL*** is a browser-based user interface that is used in SL1 versions 12.2.x and earlier. To access the GraphiQL interface, type the URL or IP address for SL1 in a browser, add **/gql** to the end of the URL or IP address, and press **[Enter]**.

> **NOTE:** GraphiQL is not maintained by ScienceLogic. You can access its documentation at [https://github.com/graphql/graphiql](https://github.com/graphql/graphiql).

Both of these methods use similar elements that enable you to interactively explore the capabilities of, and execute queries against, the SL1 GraphQL API. Unlike the SL1 REST API, all requests to the GraphQL API on a given SL1 system use the same URI.

This chapter covers the following topics:

*GQL Browser and GraphiQL User Interface Elements* ..... 14

*The GraphQL Schema* ..... 16

# GQL Browser and GraphiQL User Interface Elements

The **GQL Browser** page (System > Tools > GQL Browser) and GraphiQL user interface enable you to write and execute operations, define variables, view operation results, explore your schema, and more.

The following sections describe all of the elements you can interact with on these pages.

## Query Pane

The query pane is where you write GraphQL queries and mutations. It includes several features to help you properly formulate your queries and mutations:

- As you type, the pane intuitively suggests potential fields, arguments, and syntax based on what you've already written and your defined schema.
- Additionally, the pane automatically validates your query or mutation as you type, highlighting any errors in syntax or other elements that would cause your query or mutation to fail.
- The pane highlights fields, arguments, syntax, and other important elements in the query pane to make the query or mutation easier to read.
- If you need to insert notes or comments in a query or mutation, add them on a line that starts with the `#` symbol above or below the query or mutation. The pane ignores any lines that start with `#` when executing an operation.
- As you edit a query or mutation, the browser automatically updates the URL that appears in the address bar to reflect what you have entered. It preserves every part of your entry in the URL, including whitespace, comments, and invalid syntax or elements. This feature enables you to share your query or mutation publicly or with colleagues for distribution or debugging purposes.
- You can use the following keyboard shortcuts in the query pane:
  - Shift + Ctrl + P: Prettify the query or mutation
  - Shift + Ctrl + M: Merge the query or mutation
  - Shift + Ctrl + C: Copy the query or mutation
  - Ctrl + Enter: Execute the query or mutation
  - Ctrl + Space: Auto-complete the query or mutation

## Variables Pane

You can use the **Variables** pane to define variables that you are using in your query or mutation. You should write variables in JSON format. This pane is at the bottom of the query pane and might be collapsed when you first access the page. You can click one of the following options to expand and use it, depending on which page you are using:

- On the **GQL Browser** page (System > Tools > GQL Browser), click **Variables**.
- In GraphiQL, click **Query Variables**.

For more information about using variables, see the section on *Variables*.

## Headers Pane

You can use the **Headers** pane to define a request (or "run-as") header that allows you to run your query as another user when it is passed to the GraphQL API. This pane is at the bottom of the query pane and might be collapsed when you first access the page. To expand it on the **GQL Browser** page (System > Tools > GQL Browser), click **Headers**.

> **NOTE:** The **Headers** pane is not available in GraphiQL.

For more information about headers, see the section on ***Using Request Headers***.

## Results Pane

The right-hand pane is the results pane. This is where the results of the most recently executed query or mutation display in JSON format.

Alternatively, if your query or mutation is not properly formed, the results pane displays the errors that are causing the query or mutation to fail.

Fields, arguments, syntax, and other important elements are highlighted in the results pane to make the results easier to read.

The results that display in this pane are read-only, so you cannot accidentally edit or delete them.

## Docs Pane

The **Docs** (or **Documentation Explorer**) pane enables you to search or browse through the GraphQL schema. This allows you to explore all of the possible queries, mutations, fields, types, arguments, and other schema elements that you can use. You can click one of the following options to expand and use it, depending on which page you are using:

- On the **GQL Browser** page (System > Tools > GQL Browser), click the Show Documentation Explorer icon () on the left side of the page.
- In GraphiQL, click the **[Docs]** button on the right side of the page.

For more information about using the **Docs** (or **Documentation Explorer**) pane to browse the GraphQL schema, see the section on ***The GraphQL Schema***.

## Additional Options

The **GQL Browser** page (System > Tools > GQL Browser) and GraphiQL user interface both include additional options that you can use when utilizing GQL.

- On the **GQL Browser** page, these options display as icons on either side of the query pane.
- In GraphiQL, these options display as toolbar buttons at the top of the browser.

A few of these options are unique to the **GQL Browser** page, but most of them are common to both pages:

- Click the execute query icon ( ) to run the query or mutation in the main query pane.
  - In GraphiQL, this appears as the execute query icon ( ).
- Click the prettify query icon ( ) to re-format the text in the main query pane to make it easier to read. If your query or mutation is not properly formed, the prettify query icon turns pink when you click it and does not format the text. If there is only one unnamed query in the query pane, then clicking the prettify query icon also removes the `query` operation type from the query.
  - In GraphiQL, this appears as the **[Prettify]** button.
- Click the merge fragments into query icon ( ) to flatten a query that contains defined fragments.
  - In GraphiQL, this appears as the **[Merge]** button.
- Click the copy query icon ( ) to copy all of the text in the main query pane to your clipboard, which enables you to paste your query or mutation elsewhere.
  - In GraphiQL, this appears as the **[Copy]** button.
- Click the show history icon ( ) to open a new pane on the left side of the page that contains a list of the previous queries you have run in the GQL browser. You can click a query to add it to the main query pane. You can also edit, favorite, or delete the query from your history, or clear your history altogether.
  - In GraphiQL, this appears as the **[History]** button.
- Click the show font size icon ( ) to view and, if necessary, adjust the current font size for the browser. In the ***Font Size*** field, you can type a new numerical value or use the up and down arrows to adjust the font size.
- Click the re-fetch GraphQL schema icon ( ) to refresh the schema in the **Docs** pane with the latest changes.
- Click the open short keys dialog icon ( ) to open a dialog that lists the keyboard shortcuts that you can use in the browser.
- Click the open settings dialog icon ( ) to open a dialog where you can adjust the following GQL browser settings:
  - ***Persist Headers***. Toggle on if you want to save your headers upon reloading the browser, or toggle off if you do not. You should only toggle this setting on if you trust the device you are working on.
  - ***Clear Storage***. Click **[Clear data]** to remove all data that is locally stored on your GQL browser.

# The GraphQL Schema

A common refrain you might hear when people discuss GraphQL is that it is "self-documenting". This is because of two key features in GraphQL:

- GraphQL's intuitive suggestions and validation capabilities as you type your query or mutation in the ***query pane***
- The GraphQL ***schema***, which explicitly defines all of the possible queries, mutations, fields, types, arguments, and other schema elements that you can use

In all, the GraphQL schema provides the following information:

- All of the possible queries you can make
- All of the possible mutations you can perform
- All of the types that you can query or mutate
- All of the fields within a given type
- All of the potential arguments for a given field
- The accepted value types for each argument
- The relationships between types

You can access the full schema in the **Docs** (or **Documentation Explorer**) pane. This pane is always synced with GraphQL, automatically adding new fields, types, and arguments as they are added to the schema. It also automatically removes deprecated fields from the schema.

In this pane, you can search for specific elements within the schema, or you can click one of the two root types—`query` or `mutation`—and drill down from there by clicking the fields, arguments, and types that display.

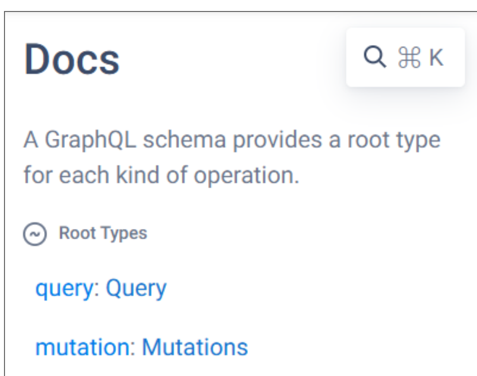
## Exploring the GraphQL Schema in the Docs Pane

To get the most value out of the **Docs** pane, you must understand how to explore the SL1 GraphQL schema that it documents.

> **TIP:** When using the **Docs** pane to explore the GraphQL schema, keep in mind this basic rule: Everything you click in the schema will result in either a definition or an additional set of fields or types that represent additional layers in the schema (and, by extension, your query or mutation).

> **NOTE:** This section and the examples that follow reference the **Docs** pane on the **GQL Browser** page (System > Tools > GQL Browser), which is available in SL1 versions 12.3.0 and later. However, the functionality works the same way in the **Documentation Explorer** in GraphiQL, which is used in SL1 versions 12.2.x and earlier.

When you first open the **Docs** pane, you will see that the schema begins with the two root operation types: `query` and `mutation`:



When you click one of the root operation types, a list of objects that are available for that operation type displays. Each object lists the object name, the arguments that can be passed into that object, the object's return type, and a definition of that object.

In the following example, let's say you clicked `Query` and then scrolled down to the following section:

```
event(
  globalManagerStackSearch:
  GlobalManagerStackSearch
  id: ID!
  isGlobalManagerRequest: Boolean
): Event

Get an individual event
```

- `event` is a query object. It represents an object that can be queried from the SL1 API. As indicated by the definition below it, you can use this particular object to get information about an individual event.
- The next three lines, which are surrounded by parentheses, are all of the possible arguments that you can pass into the `event` query object. Each argument is displayed in a `field name: value type` pair, where the field name indicates an API field associated with the object and the value type indicates how you must define that field if you include that argument in your query. Arguments are used to define your query parameters. This example includes the following arguments:
  - `globalManagerStackSearch: GlobalManagerStackSearch`. This argument has a "search" value type. If you include this argument, you must also include a nested set of search parameter fields below it. If you click `GlobalManagerStackSearch` in the schema, it displays all of the nested fields that you can include in the `globalManagerStackSearch` argument, along with their value types. If you click on their value types, the schema might display even more fields and their value types.
  - `id: ID!`. When you use this argument, you must supply a unique identifier (as indicated by the `ID` value type) for the event `id` API field. The ***exclamation point*** indicates that you ***must*** include this argument in the query.
  - `isGlobalManagerRequest: Boolean`. If you use this argument in the query, you must include `true` or `false` (as indicated by the `Boolean` value type) as a value for the event `isGlobalManagerRequest` API field.
- `Event` is the return type for the `event` query object. The return type consists of one or more fields related to the event that you specified in your argument; you are requesting that GraphQL return values for these fields when you execute the query. The schema lists all of the fields that belong to the `Event` return type, along with their value types.

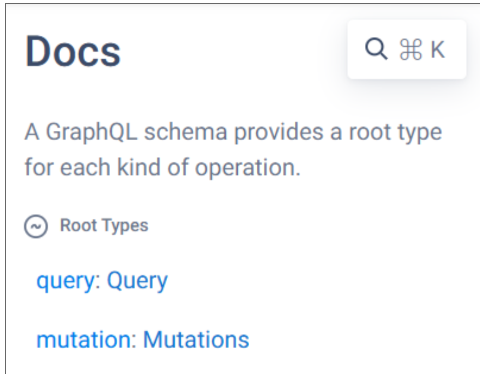
## Example 1: Using the Schema to Execute a Simple Query

This example describes how to use the **Docs** pane to explore the SL1 GraphQL schema and determine the specific objects, arguments, and return fields that we need to include in a query.

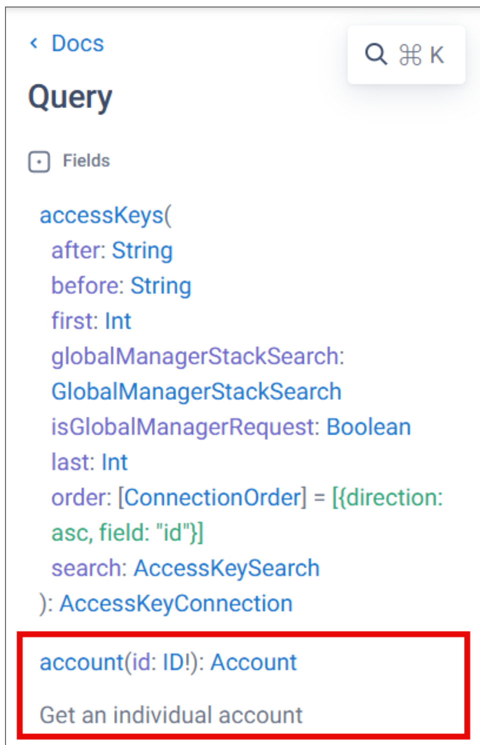
For this example, we are executing a simple query: We are asking GraphQL to return some basic information about a single user account in SL1.

To use the schema to execute this simple query:

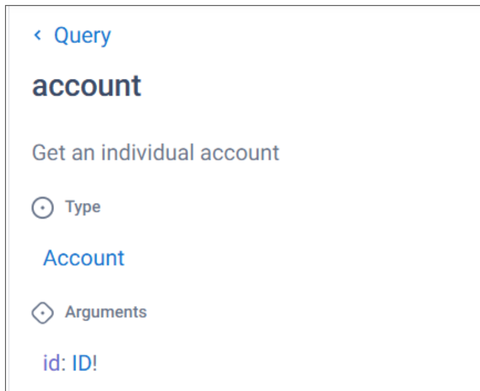
1. On the **GQL Browser** page (System > Tools > GQL Browser), click the Show Documentation Explorer icon (▤) on the left side of the page to expand the **Docs** pane. The main schema page displays:



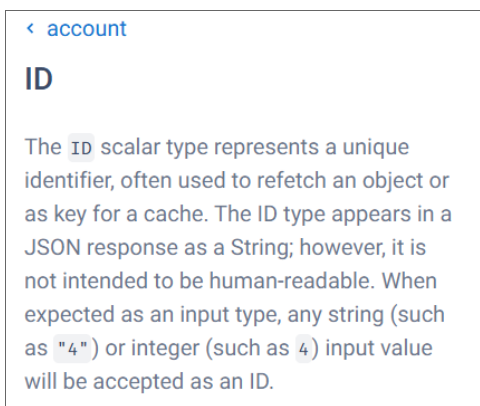

2. Because we are executing a query, click the `Query` type link. A list of all potential query objects displays in the **Docs** pane.
3. Because we want to query a specific user account, browse the list or use the search feature to locate the entry for the `account` query object:



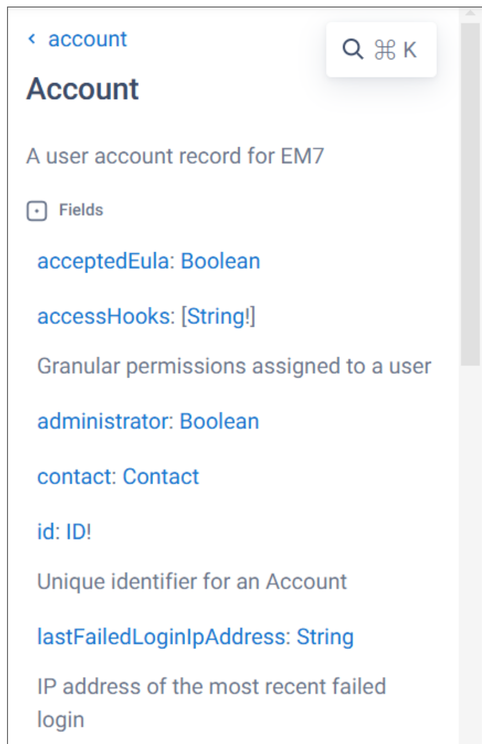
4. Click the `account` object link that displays in blue text. The schema displays a more readable version of the information that appeared for the `account` object in the list of query objects. The schema defines the object, shows that it has a return type of `Account`, and indicates that it has just a single argument (`id: ID!`):




5. Click the `ID!` argument value type link. The schema displays a definition for the ID scalar type that specifies the type of value you must enter in the argument for the `id` field:



6. Click the `account` object link in the top-left corner of the pane to return to the `account` object definition, then click on the `Account` return type link. The schema displays a list of all the potential fields that you can ask GraphQL to return in your `account` query, along with their value types:




7. Click through the return fields and value types as needed to view their definitions. For this example, let's assume we clicked the definitions for `id: ID!`, `user: String`, and `administrator: Boolean` for additional information about those fields, if needed.

8. Having reviewed the schema in the **Docs** pane, type the query in the query pane. In this query, we are asking GraphQL to return the user ID number and username for the user account with the ID "25", and indicate whether that user account is an administrator:

```
query basicAccountInfo {
  account(id: 25) {
    id
    user
    administrator
  }
}
```

> **TIP:** As you type your query, the query pane suggests potential fields, arguments, and syntax based on what you've already written and your defined schema; validates your query or mutation as you type; and uses different colors to highlight fields, arguments, syntax, and other important elements to make the query easier to read.

9. Click the execute query icon ( ). The query results display in the results pane. Having reviewed the schema, we can expect GraphQL to return an ID number for the id field, a string for the user field, and a boolean value for the administrator field:

```
{
  "data": {
    "account": {
      "id": "25",
      "user": "example",
      "administrator": false
    }
  }
}
```

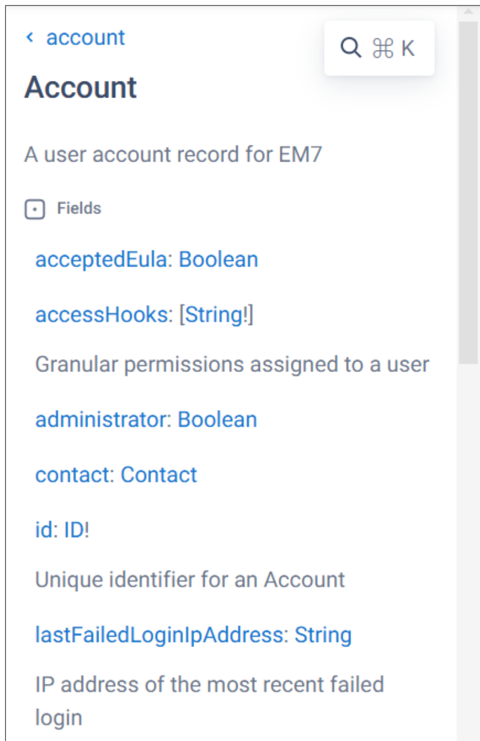
## Example 2: Using the Schema to Execute a Complex Query

This example expands on the previous example to execute a more complex query about the same user account.

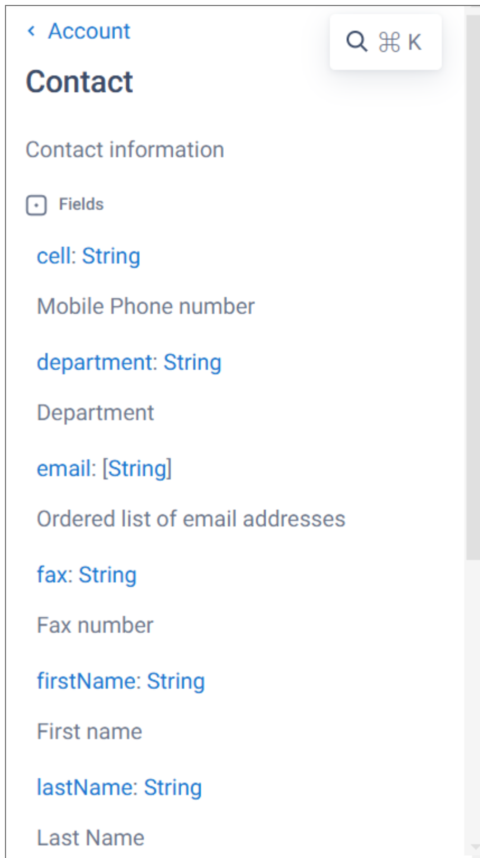
For this example, we are asking GraphQL to return additional information about the user account from Example 1, including information about the user's contact information, physical location, and organization membership. This example will require us to dig deeper into the schema in the **Docs** pane to successfully execute a more advanced query.

To use the schema to execute this more advanced query:

1. Complete the steps in ***Example 1***.
2. In the **Docs** pane, click the `Account` return type link in the top-left corner of the pane to return to the `Account` return type definition and fields:



3. In this example, one of the things we want to add to the query is the user's contact information. From the list of `Account` return fields and value types, locate the `contact: Contact` field: value type pair, and then click the `Contact` type link. The schema displays a list of all the potential contact information fields that you can ask GraphQL to return in your `account` query, along with their value types:

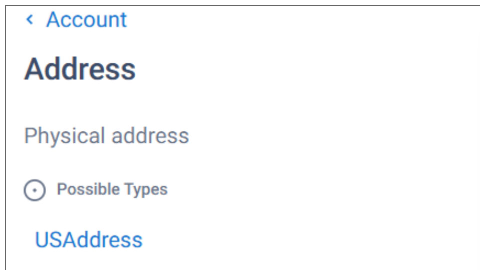



4. Click through the return fields and value types as needed to view their definitions. For this example, let's assume that we do not want GraphQL to return values for every possible contact return field, but that instead we want GraphQL to return values for just a few of them: The user's first name, last name, and email address. Therefore, let's assume we clicked the definitions for `firstName: String`, `lastName: String`, and `email: [String]` for additional information about those fields, if needed.

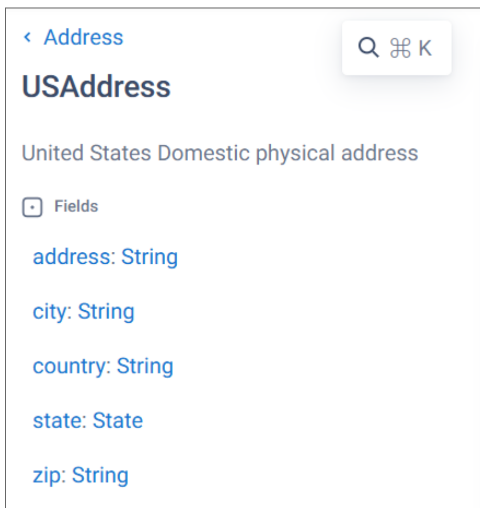
> **TIP:** Because the value type for the `contact` return field is the custom `Contact` type, which itself consists of these additional return fields, we will need to nest these additional return fields under the `contact` return field when we form the query. If we include only the `contact` return field in our query without specifying any nested return fields under it, GraphQL will return values for every possible contact field.

5. Click the `Account` return type link in the top-left corner of the pane to return to the `Account` return type definition and fields.

6. In this example, we also want to add information about the user's physical location to the query. From the list of `Account` return fields and value types, locate `location: Address` and then click the `Address` type link. This time, however, the schema does not display any fields for the `Address` type; instead, it indicates that the `Address` type itself has a possible type of `USAddress`:




7. Click the `USAddress` type link. The schema displays a list of all the potential U.S. Address fields that you can ask GraphQL to return in your `account` query, along with their value types:

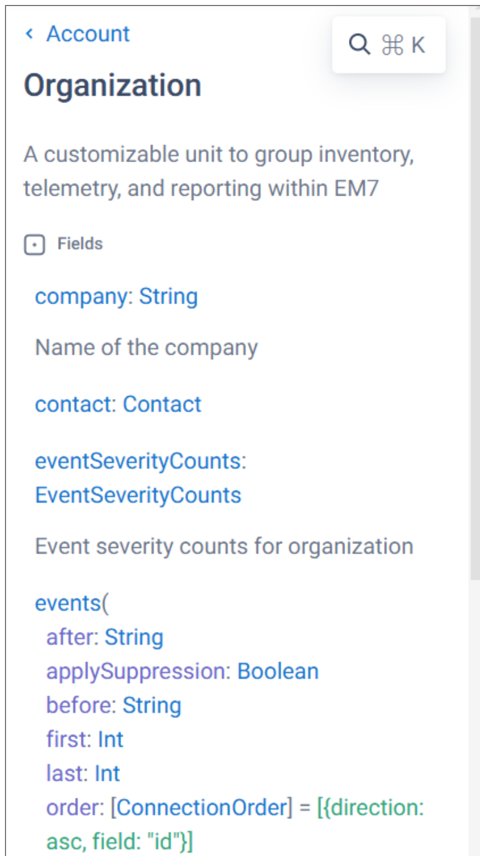



8. Click through the return fields and value types as needed to view their definitions. For this example, let's assume that we clicked the definitions for `address: String`, `city: String`, `state: String`, and `zip: String` for additional information about those fields, if needed.

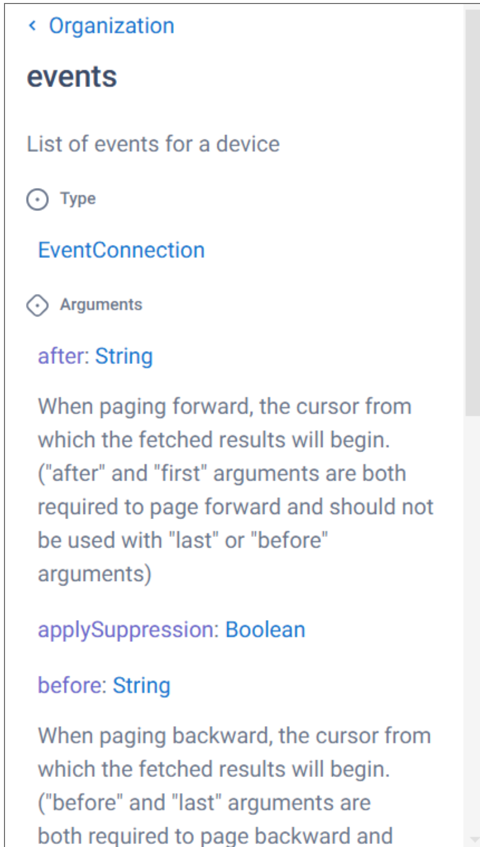
> **TIP:** Because the value type for the `location` return field is the custom `Address` type, which in turn has the possible custom `USAddress` type that consists of these additional return fields, we will need to nest these additional return fields under the `location` return field when we form the query. To do this, we will need to include an ***inline fragment*** that allows us to specify that the location return fields that we are including in the query belong to the `USAddress` type within the schema. We can do this using the format "`... on USAddress`" in our query.

9. Click the `Address` return type link in the top-left corner of the pane and then the `Account` return type link to return one final time to the `Account` return type definition and fields

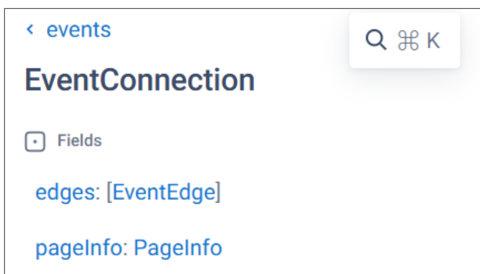
10. Lastly, in this example, we want to add information about the user's organization to the query. From the list of `Account` return fields and value types, locate `organization: Organization` and then click the `Organization` type link. The schema displays a list of all the potential organization fields that you can ask GraphQL to return in your `account` query, along with their value types:



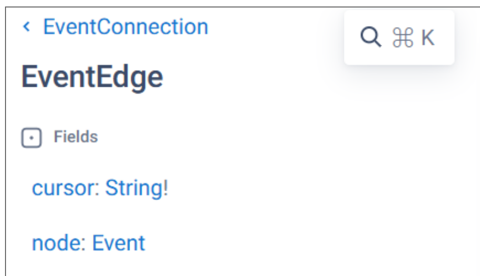
11. Click through the return fields and value types as needed to view their definitions. For this example, let's assume that we clicked the definitions for `id: ID!`, `company: String`, and `events` for additional information about those fields, if needed. If you do so, you discover that those first two are field: value type pairs like we have encountered throughout this example, but the `events` return field includes its own list of arguments and a connection type, `EventConnection`:



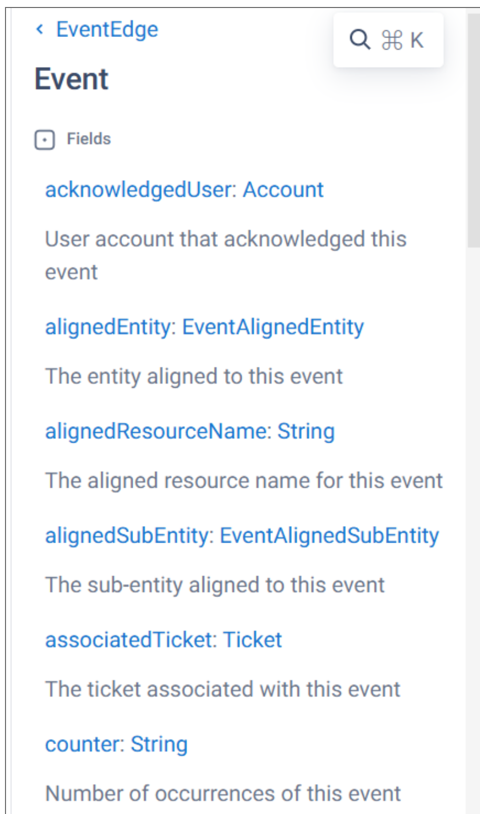

12. For this example, we want to query GraphQL to return some basic information about the first three events aligned to the user's organization. To do this, we will use the `first: Int` argument, and then we will query several fields under the `EventConnection` connection type. Click the `EventConnection` type link:



13. `EventConnection` is a connection type. *Connection types consist of one or more edges that connect two nodes*. The schema reflects this, so from `EventConnection`, click `[EventEdge]`:




14. From `EventEdge`, click `Event`. The schema displays a list of all the potential event fields that you can ask GraphQL to return from the `Event` node in your `account` query, along with their value types:




15. Click through the return fields and value types as needed to view their definitions. For this example, let's assume that we clicked the definitions for `id: ID!`, `severityLevel: SeverityLevel`, `message: String`, and `dateAcknowledged: Int` for additional information about those fields, if needed. Let's assume that we also clicked the definition for `source: EventSource`, which we discovered had additional nested fields, including `id: ID!`.

16. Having reviewed the schema in the **Docs** pane, type the query in the query pane. In this query, in addition to the information we queried in *Example 1*, we are asking GraphQL to return:

    - The user's contact information, including first name, last name, and email address
    - Information about the user's physical location, including address, city, state, and ZIP code
    - Information about the user's organization, including its organization ID and company name, along with the event ID, severity level, source ID, event message, and acknowledged date of the first 3 events aligned to that organization

```
query basicAccountInfo {
  account(id: 25) {
    id
    user
    administrator
    contact {
      firstName
      lastName
      email
    }
    location {
      ... on USAddress {
        address
        city
        state
        zip
      }
    }
    organization {
      id
      company
      events(first: 3) {
        edges {
          node {
            id
            severityLevel
            source {
              id
            }
            message
            dateAcknowledged
          }
        }
```

```
            }
        }
    }
}
```

**TIP:** As you type your query, the query pane automatically suggests potential fields, arguments, and syntax based on what you've already written and your defined schema; validates your query or mutation as you type; and uses different colors to highlight fields, arguments, syntax, and other important elements to make the query easier to read.

17. Click the execute query icon (▶). The query results display in the results pane. Having reviewed the schema, we can expect GraphQL to return data for all of the fields in the format of their respective defined value types:


```
{
  "data": {
    "account": {
      "id": "25",
      "user": "example",
      "administrator": false,
      "contact": {
        "firstName": "Example",
        "lastName": "User",
        "email": [
          "example@sciencelogic.com"
        ]
      },
      "location": {
        "address": "10700 Parkridge Blvd",
        "city": "Reston",
        "state": "VA",
        "zip": "20191"
      },
      "organization": {
        "id": "0",
        "company": "System",
        "events": {
          "edges": [
            {
              "node": {
                "id": "154134",
                "severityLevel": "Minor",
                "source": {
                  "id": "2"
                },
                "message": "Ticket Created: 95",
                "dateAcknowledged": null
              }
            },
            {... 2 more nodes
            }
```

```
                    ]
                }
            }
        }
    }
}
```

> **NOTE:** The above result has been truncated to make it easier to read in this documentation. When you actually complete this example, information about the second and third events will display in the results in the place of `{... 2 more nodes}`.

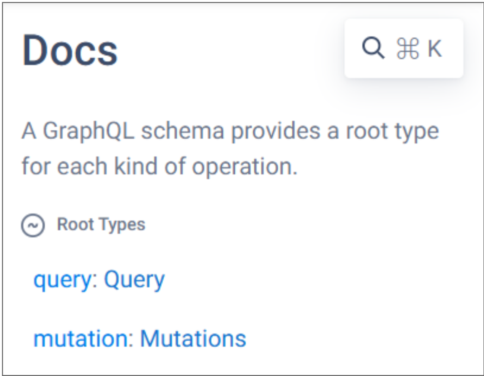
## Example 3: Using the Schema to Execute a Mutation

This example describes how to use the **Docs** pane to explore the SL1 GraphQL schema and determine the specific objects, arguments, and return fields we need to include in a mutation.

For this example, we are telling GraphQL to create a custom link to appear in the SL1 Action Runner.

To use the schema to execute this mutation:

1. On the **GQL Browser** page (System > Tools > GQL Browser), click the Show Documentation Explorer icon ( ) on the left side of the page to expand the **Docs** pane. The main schema page displays:




2. Because we are executing a mutation, click the `Mutations` type link. A list of all potential mutation objects displays in the **Docs** pane.

3. Because we want to create a new custom link, browse the list or use the search feature to locate the entry for the `createCustomLink` mutation object:

```
createCustomLink(
  entity: CustomLinkEntity!
  isGlobalManagerRequest: Boolean
  name: String
  stack: ID
  url: String!
): CustomLink

Create a new Custom Link.
```

4. Click the `createCustomLink` mutation object. The schema displays a more readable version of the information that appeared for the `createCustomLink` object in the list of mutation objects. The schema defines the object, displays its potential arguments, and shows that it has a return type of `CustomLink`:

‹ Mutations

**createCustomLink**

Create a new Custom Link.

Type

CustomLink

Arguments

entity: CustomLinkEntity!

The type of entity to align the custom link to.

isGlobalManagerRequest: Boolean

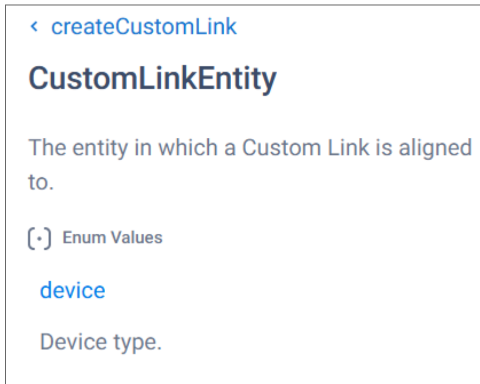
Set to true for global manager requests. (Required only for global manager requests)

name: String

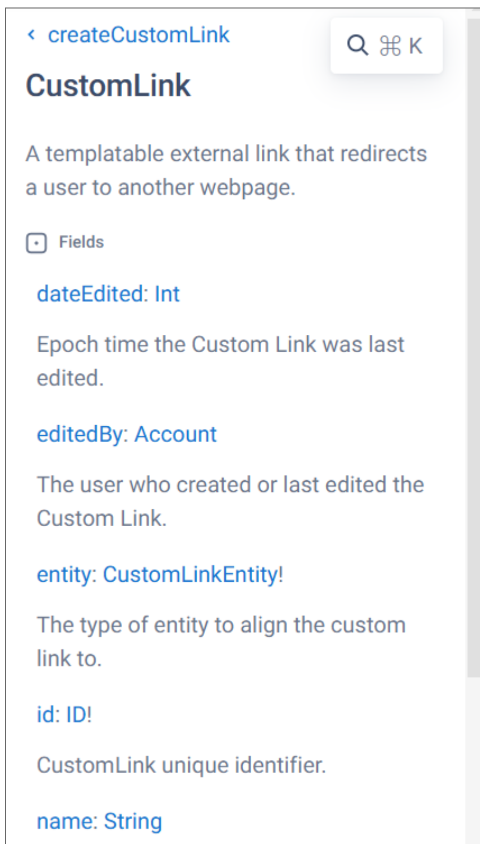
Human readable name to display the link as.

stack: ID

5. Click the `CustomLinkEntity!` argument value type link. The schema displays a definition that specifies the type of value you must enter in the argument for the `entity` field. According to the schema, the only acceptable value is `device`.




6. Use the navigation in the top-left corner of the pane to return to the `createCustomLink` object definition, then click through the other argument value types as needed to view their definitions.
7. Click on the `CustomLink` return type link. The schema displays a list of all the potential fields that you can ask GraphQL to return when it executes your `createCustomLink` mutation, along with their value types:



8. Click through the return fields and value types as needed to view their definitions.
9. Having reviewed the schema in the **Docs** pane, type the mutation in the query pane. In this mutation:
    - The custom link would display for all devices that used the custom attribute "Restorepoint ID". It would not display for devices that do not use that custom attribute.
    - When creating the mutation, you would replace `<Restorepoint Hostname or IP>` in the `url` field with the Restorepoint hostname or IP address that you want the link to open.
    - When clicked, the link would open the URL https://<Restorepoint Hostname or IP>/ui/devices/{attributeLabel.Restorepoint ID}, where <Restorepoint Hostname or IP> would already be defined, and SL1 would automatically replace {attributeLabel.Restorepoint ID} with the Restorepoint ID custom attribute value of the device from which the link was clicked.
    - The custom link would display in the Action Runner as "Restorepoint".

    This is the mutation we would type in the query pane:

```
mutation createCustomLink {
  createCustomLink(
    entity: device
    url: "https://<Restorepoint Hostname or IP>/ui/devices/
{attributeLabel.Restorepoint ID}"
    name: "Restorepoint"
  ) {
    id
    entity
    url
    name
    editedBy {
       user
    }
    dateEdited
  }
}
```

> **TIP:** As you type your mutation, the query pane automatically suggests potential fields, arguments, and syntax based on what you've already written and your defined schema; validates your mutation as you type; and uses different colors to highlight fields, arguments, syntax, and other important elements to make the mutation easier to read.

10. Click the execute query icon (▶). The mutation executes, and GraphQL displays the data you requested in the results pane.

# Chapter 4

# Queries

## Overview

This chapter describes how to form a query in GraphQL to search for and return data from fields in SL1. It describes how to form basic query syntax and use connections, variables, fragments, directives, search filters, and pagination options in your queries. Each section includes one or more examples to further illustrate each topic.

> **NOTE:** GraphQL is a complex language, and the information in this chapter is not an exhaustive list of every possible option at your disposal when writing and executing a query. For additional information about any of the subjects in this chapter, or for information not covered in this chapter, see the GraphQL documentation.

This chapter covers the following topics:


*What Are Queries?* 37

*Basic Query Syntax* 37

*Connections, Edges, and Nodes* 39

*Variables* 41

*Fragments* 43

*Inline Fragments* 44

*Directives* 45

*Search Filters* 46

*Query Pagination* 49

*Querying the ScienceLogic GraphQL API from an External Application* 54

# What Are Queries?

***Queries*** are GraphQL operations that search for and return data from fields in your schema. A single query can search for and return data for a single field or multiple fields.

When building a query, it should match the shape of the object types in your schema. GraphQL returns the results in the same shape as your query.

To see a full list of queries that are available for SL1, see the schema in the GQL browser.

# Basic Query Syntax

The most basic syntax of a query involves the following elements:

- ***Operation type***. For queries, this is `query`. If only one query is present in your GraphQL browser, specifying the `query` operation type at the beginning of your query is optional; if you do not specify an operation type, GraphQL assumes that you are performing a query by default.
- ***Operation name***. A name that you define for the operation. An operation name is required if you are including ***variables*** in your query. While operation names are not always required in other circumstances, they are still helpful to have for logging and debugging purposes.
- ***Object***. The object you are querying, as defined in the schema. This is typically the object's name, written in camel case.
- ***Argument(s)***. The data that you want to pass to the object in the query. These are defined in the following structure: `(fieldName: "value")`. The schema lists the argument field name and return type pairs that are available for each object. Some objects can have multiple arguments, in which case the argument structure is `(fieldName: "value", fieldName: "value")`, with any additional arguments separated by additional commas. When including multiple arguments, the arguments can be provided in any order and GraphQL will consider them to be semantically identical.
- ***Return field(s)***. The fields that you want GraphQL to return values for when you execute the query.

**TIP:** If you are unsure of which arguments or return fields are required for a query object, type `query {objectName}` in the query pane, replacing `objectName` with the name of the query object, and then click the execute query icon (▶). GraphQL will automatically add any required return fields to the query, while the results pane will display an error that lists any required arguments.

**NOTE:** If you query multiple objects, GraphQL executes the queries on all of those objects at the same time, in parallel.

## Example: Querying a Single Device

Here is an example of a basic query. In this example, we are searching for information about a specific device in our SL1 system:

```
query getDevice {
  device(id: 3) {
    ip
    deviceClass {
      class
      description
    }
  }
}
```

In this example:

- `query` is the operation type.
- `getDevice` is the operation name that we defined for this specific operation.
- `device` is the object. We are asking GraphQL to search for and return data about a device in our SL1 system.
- `id:3` is the argument for `device`, with `id` being the field and `3` being the value in the `field: value` pair that makes up an argument. We are asking GraphQL to search for and return data about the device with device ID "3".
- `ip` and `deviceClass` are the return fields. Additionally, `deviceClass` has two nested fields, `class` and `description`. We are asking GraphQL to return the device IP address as well as the class name and description of the device's device class.

When we execute this query, GraphQL returns exactly the data we asked for, and in the exact structure we specified:

```
{
  "data": {
    "device": {
      "ip": "10.100.100.26",
      "deviceClass": {
        "class": "ScienceLogic, Inc.",
        "description": "EM7 Data Collector"
      }
    }
  }
}
```

## Example: A Simple Query without the Operation Type "query"

If only one query is present in your GraphQL browser, you do not need to specify the `query` operation type at the beginning of your query. Because a query is GraphQL's default operation type, GraphQL assumes that you are performing a query if you do not specify otherwise.

Here is the same query as the previous example, but without the `query` operation type specified:

```
{
  device(id: 3) {
    ip
    deviceClass {
      class
      description
    }
  }
}
```

When we execute this query, GraphQL returns the exact same data as was returned in the previous example, and in the same structure.

# Connections, Edges, and Nodes

The SL1 GraphQL schema uses connections, edges, and nodes to connect defined elements and to handle ***pagination***.

- A ***connection*** is a type within the GraphQL schema that is used to connect other defined elements in the schema. A connection consists of a group of related edges. Any type with a name that ends in "Connection" is a connection type.
- An ***edge*** is a type that connects two nodes, representing some sort of relationship between them. An edge also has a ***cursor*** in addition to the underlying node.
- A ***node*** is an individual object type that is defined in the schema, consisting of one or more fields.

For example, in the SL1 GraphQL schema:

- The query object `devices` includes a connection type, `DeviceConnection`.
    - The `DeviceConnection` type includes edges.
        - Those edges link the `devices` node to the `Device` node.
            - The `Device` node includes multiple fields that are used to define an individual device in SL1.

When you execute a query, you define the shape in which you want GraphQL to return data, but that shape must ultimately conform to the shape of data as it has been defined in the schema. That means you will often need to include edges and nodes in your query, as they are part of the structure within the SL1 GraphQL schema. You do not need to include the connection type, as it is part of the query object definition.

## Example: A Query with Edges and Nodes

Here is an example of a query that includes edges and nodes, because those things are defined in the schema as being part of the data shape:

```
query defineDevices{
  devices(first: 5) {
    edges {
      node {
        id
        ip
        deviceClass {
          class
          description
        }
      }
    }
  }
}
```

When we execute this query, GraphQL returns the requested data for each node:


```
{
  "data": {
    "devices": {
      "edges": [
        {
          "node": {
            "id": "3",
            "ip": "10.100.100.26",
            "deviceClass": {
              "class": "ScienceLogic, Inc.",
              "description": "EM7 Data Collector"
            }
          }
        },
        {
          "node": {
            "id": "27",
            "ip": "10.2.5.72",
            "deviceClass": {
              "class": "VMWare",
              "description": "vCenter Server Appliance"
            }
          }
        },
```

```
        {...3 more nodes
        }
      ]
    }
  }
}
```

# Variables

***Variables*** are dynamic values that can be used to replace arguments in your query, enabling you to reuse the query for multiple objects simply by changing the variable value.

When using variables, keep the following in mind:

- Variable names are always preceded by a dollar sign (`$`). For example: `$variableName`
- Before you can use a variable in a query, you must first declare the variable acceptable to use in the query.
  - Place the declared variable in parentheses immediately after the operation name.
  - Use the format `$variableName: scalarType`, where `$variableName` represents the name of the variable and `scalarType` represents the acceptable scalar type for that variable.
  - Variable scalar types must match the type of the arguments that they will replace. For example, if you are replacing a field that has a scalar type of "String", then your variable must also use the "String" scalar type.
  - If you declare a variable after the operation name, you must then use it in your query. You cannot declare a variable and then not use it in the query.
  - Example: `query findUserAccount($userID: ID)`
- Optionally, when declaring a variable, you can assign the variable a default value.
  - Place the default value immediately after the scalar type in the declaration, preceded by an equal sign (`=`).
  - Use the format `$variableName: scalarType = defaultValue`, where `$variableName` represents the name of the variable, `scalarType` represents the acceptable scalar type for that variable, and `defaultValue` represents the default value for that variable.
  - Any variables that you do not define in the **Variables** pane will use the default value.
  - Any variable values that you define in the **Variables** pane will override the default value.
  - Example: `query findUserAccount($userID: ID = 15)`

- You must then insert the declared variable into your query.
    - Place the variable in parentheses immediately after the query object name, just like you would a regular argument.
    - Use the format `fieldName: $variableName`, where `fieldName` represents the argument field name and `$variableName` represents the variable name.
    - Example: `account(id: $userID)`
- Finally, you must define the variable.
    - In the **Variables** pane at the bottom of the GQL browser, define the variable value as a JSON object.
    - Use the format `"variableName": "value"`, where `"variableName"` represents the variable name and `"value"` represents the value that you want to use in place of the variable in the query.
    - If you included a default value in your variable declaration, and the variable is using the default value, you do not need to define the variable in the **Variables** pane.
    - Example: `{ "userID": "23" }`

## Example: Querying Basic Device Information Using a Variable

Here is an example of a query that uses a variable. In this example, we are asking GraphQL to return some basic information about a device, using a variable `$deviceID` that will be defined at the time we run the query to replace the value in the `id` argument:

```
query deviceBasicInfo($deviceID: ID!) {
  device(id: $deviceID) {
    id
    name
    ip
    organization {
      id
    }
    deviceClass {
      class
      description
    }
    state
    active {
      userDisabled
      unavailable
```

```
      maintenance
      systemDisabled
      userInitiatedMaintenance
    }
  }
}
```

We must then define this variable. To do so, we would type its value into the **Variables** pane in the GQL browser in the following format:

```
{
  "deviceID": "27"
}
```

# Fragments

***Fragments*** are reusable units that you can include in multiple queries or mutations. Each fragment consists of a group of fields that are all associated with the same type.

You might use fragments if you want to reuse the same set of fields for multiple objects. Rather than retyping the same set of fields throughout your query, you can define those fields as a fragment and then just insert that fragment in your query or mutation as needed.

Fragments are defined in the following format, where `fragmentName` represents the name you are giving the fragment, `Type` is the type to which the fragment belongs, and `fieldNames` are the names of one or more fields that belong to that type:

```
fragment fragmentName on Type {
  fieldNames
}
```

After you have defined your fragment, you can then use it in any query or mutation in the same location where the fields contained within the fragment would normally go, using the format `...fragmentName`.

When using fragments in your queries and mutations, keep the following in mind:

- If you use multiple named fragments in the same document, each fragment's name must be unique. Inline fragments do not have this requirement.
- All fragments, whether named or inline, can be declared only on objects, unions, and interfaces. (For more information about unions and interfaces, see the GraphQL documentation.)
- If you define a fragment, it must be used in your query or mutation.

- You can use variables inside fragments. When you do so, add the variable to the appropriate field in the fragment definition. Then, rather than defining the variable in the **Variables** pane, you can define it when you declare it at the beginning of the query.

### Example: Creating a Fragment for Account Fields

This query example demonstrates how to create and use a fragment within a query. In this example, we are creating a fragment called `accountInfo` that is associated with the *Account* type. The fragment includes the fields `id` and `user`, plus the nested fields `firstName`, `lastName`, and `email` under the field `contact`:

```
fragment accountInfo on Account {
  id
  user
  contact {
    firstName
    lastName
    email
  }
}

query basicAccountInfo {
  accounts(search: {organization: {has: {company: {eq: "System"}}}}) {
    edges {
      node {
        ...accountInfo
      }
    }
  }
}
```

## Inline Fragments

If you are querying a field that returns an interface or union type, you must use an inline fragment for GraphQL to return data from one of those types.

**NOTE:** For more information about unions and interfaces, see the GraphQL documentation.

When you use an inline fragment, you do not define the fragment separately from the query or mutation in which it is being used. Instead, you define the fragment inline within a selection set.

Within a query or mutation, inline fragments are written in the following format, where `Type` is the type to which the fragment fields belong, and `fieldNames` are the names of one or more fields that belong to that type:

```
... on Type {
  fieldNames
}
```

### Example: Using an Inline Fragment

The following example uses an inline fragment. In the SL1 GraphQL API, the `location` field has the custom `Address` type as a return type. In turn, the `Address` type has the possible custom `USAddress` type, which consists of additional return fields. Due to this structure, if we want to query those fields, we will need to nest the fields under the `location` field using an inline fragment when we form the query, specifying that the location fields that we are including in the query belong to the `USAddress` type within the schema:

```
query additionalAccountInfo {
  account(id: 25) {
    location {
      ... on USAddress {
        address
        city
        state
        zip
      }
    }
  }
}
```

## Directives

***Directives*** are keywords that you can use to make GraphQL perform custom logic in your queries. They can be attached to a field or a fragment that you are including in your query, and can affect the query execution and the results that GraphQL fetches.

This is useful for situation where you otherwise would need to manually add or remove fields in your query based on specific circumstances.

Each directive can appear only after the field or fragment that it decorates. They are preceded by the "@" character and can include their own arguments. They are often used in conjunction with variables to create queries with more complex dynamic logic.

GraphQL includes the following default operational directives:

| Directive | Argument | Description |
|---|---|---|
| `@skip` | `(if: Boolean!)` | If `true`, the field or fragment the directive decorates is skipped and not resolved by GraphQL. |
| `@include` | `(if: Boolean!)` | If `true`, the field or argument the directive decorates is resolved and included in the operation results. |

### Example: Using a Directive with a Variable

The following example illustrates a query that uses a directive with a variable. In this example, we have declared a variable, `$withIcon`, and given it a default value of `Boolean = true`. We have then included the directive `@include(if: $withIcon)` on the `icon` field. Because we have defined the default value of the `$withIcon` variable as `true`, we are telling GraphQL to include the icon field in the query results if the device has an icon; otherwise, if the device does not have an icon, GraphQL will still return data about that device but it will not include data about the icon in the results for that device, since it does not have one:

```
query devices($withIcon: Boolean = true) {
  devices(first: 30) {
    edges {
      node {
        id
        name
        ip
        icon @include(if: $withIcon) {
          id
        }
      }
    }
  }
}
```

**NOTE:** As with any variable, you can override the default value by defining a different value for the variable in the **Variables** pane.

## Search Filters

GraphQL lets you add search expressions that filter the data returned in queries based on fields and parameters that you specify. So, for instance, let's say that you want GraphQL to query only those devices that belong to a specific device class. You could use a filter to limit the query to just the devices within that specific device class.

Filters are passed through fields as arguments. The basic syntax of a filter argument is:

```
(search: {<field>: {<operator>: <value>}})
```

Depending on your schema, your search argument could be longer, with additional fields and operators that ultimately precede the filter value.

If your search argument includes a list, it might also include an array, which is wrapped in square brackets, `[` and `]`. For example:

```
(search: {<field>: {<operator>: [<value 1>, <value 2>, <value 3>]}})
```

## Filter Operators

The fields in your search filter have specific allowable operators, which are defined for each field in the schema.

The following table describes the operators that you can use in GraphQL filters.

| Operator | Function |
| --- | --- |
| `and` | returns data only where all parts separated by `and` are true |
| `or` | returns data if any parts separated by `or` are true |
| `not` | returns data only where the parts following `not` are not true |
| `eq` | equals |
| `neq` | does not equal |
| `gt` | greater than |
| `lt` | less than |
| `gte` | greater than or equal to |
| `lte` | less than or equal to |
| `beginsWith` | string begins with |
| `endsWith` | string ends with |
| `contains` | string contains |
| `doesNotBeginWith` | string does not begin with |
| `doesNotEndWith` | string does not end with |
| `doesNotContain` | string does not contain |
| `in` | value is included in an explicit set |
| `notIn` | value is not included in an explicit set |
| `from` | value is included in a calculated set |
| `has` | non-null calculated set intersection |

| Operator | Function |
| --- | --- |
| `isNull` | value is null |
| `isNotNull` | value is not null |
| `isTrue` | value is true |
| `isFalse` | value is false |

## Example: Querying Devices from the Same Device Class

Here is an example of a query that uses a search filter. In this example, we are asking GraphQL to return the device IDs of all the devices in our SL1 system that have the device class "ScienceLogic, Inc.":

```
query getSL1Appliances {
  devices(search: {deviceClass: {has: {class: {eq: "ScienceLogic,
Inc."}}}}) {
    edges {
      node {
        id
      }
    }
  }
}
```

## Example: Querying a List of Specific Devices

Here is an example of a query that uses a search filter that includes a list of specific devices. In this example, we are asking GraphQL to return the device IDs, IP addresses, and device class names and descriptions for three specific devices in SL1—the devices with device IDs 3, 27, and 2132:

```
query specificDevices {
  devices(search: {id: {in: [3, 27, 2132]}}) {
    edges {
      node {
        id
        ip
        deviceClass {
          class
          description
        }
      }
    }
```

```
  }
}
```

# Query Pagination

When you write and execute a GraphQL query, GraphQL might return a large number of data results matching that query. Obviously, it would be difficult to wade through an extremely long list of results. Pagination can help make the list of results more manageable and readable.

The SL1 GraphQL API uses ***cursor pagination***, which uses several possible arguments to set a unique identifier—a cursor—that maps to a specific record in your data set. GraphQL can then use that cursor as a bookmark to determine where pagination should begin or end. The identifier type varies based on the data field and its scalar type; it might be an integer or a string. You can also use variables to determine the cursor position.

You can use the following arguments in cursor pagination:

- The `after` argument indicates the unique identifier of the record you want to establish as the cursor, after which you want GraphQL to start returning results. Here are a few examples:
  - The argument `(after: 5)` would return results starting with the sixth record in the list that matched your query.
  - The argument `(after: "example_cursor")` would return results that matched your query starting with the record after the one that included the field determined by the cursor you enter.
- The `first` argument indicates the number of records to return from the record established as the cursor. Here are a few examples:
  - The argument `(first: 10)` would return the first 10 records that matched your query.
  - The argument `(first: 25, after: "example_cursor")` would return results for the first 25 records that matched your query after the one that included the field determined by the cursor you enter.
- The `before` argument indicates the unique identifier of the record you want to establish as the cursor, before which you want GraphQL to start returning results. Here are a few examples:
  - The argument `(before: 15)` would return the first 14 records that matched your query.
  - The argument `(before: "example_cursor")` would return results that matched your query up to but not including the record that included the field determined by the cursor you enter and working backwards towards the beginning of the data set.
- The `last` argument indicates the number of records to return prior to the cursor.
  - The argument `(last: 10)` would return the last 10 records that matched your query.
  - The argument `(last: 30, before: "example_cursor")` would return results for the last 30 records that matched your query before the one that included the field determined by the cursor you enter..

There is one additional important piece to query pagination: The `pageInfo` field. The `pageInfo` field typically goes at the end of your query, below the data fields that you want to query, and can display information about the total number of results that match your query parameters, as well as information about whether the page of results that displays when you execute the query has additional pages of results before or after it.

You can use one or more of the following nested fields under the `pageInfo` field:

- The `hasPreviousPage` field indicates if there are additional results prior to the page of fetched data results. It will display `true` in the data results if there are additional pages; otherwise, it will display `false`.
- The `hasNextPage` field indicates if there are additional results after the page of fetched data results. It will display `true` in the data results if there are additional pages; otherwise, it will display `false`.
- The `matchCount` field indicates the total number of records that can be fetched for your query. This is helpful, as it lets you know how many batches you will need to fetch.

When paging backwards over a data set, the `hasPreviousPage` field nested under `pageInfo` is used in conjunction with the `last` and `before` arguments in other parts of your query.

When paging forward over a data set, the `hasNextPage` field nested under `pageInfo` is used in conjunction with the `first` and `after` arguments.

**NOTE:** The `first`, `after`, and `hasNextPage` arguments should be used individually or together in your query, as should the `last`, `before`, and `hasPreviousPage` arguments. The `first` and `before` arguments and `last` and `after` arguments should not be used together.

## Example: Querying the First Five Devices Listed in SL1

Here is an example of a query that uses a simple pagination method. In this example, we are searching for information about the first five devices listed in our SL1 system:

```
query firstFiveDevices {
  devices(first: 5) {
    edges {
      node {
        id
        name
      }
    }
  }
}
```

## Example: Determining the Total Number of Query Results

Here is an example of a query that uses the pageInfo field and its nested field matchCount to determine the total number of results for this query:

```
query howManyPages {
  devices(first: 3) {
    edges {
      node {
        id
```

```
      }
    }
    pageInfo {
      hasPreviousPage
      hasNextPage
      matchCount
    }
  }
}
```

In the query data results that display when we execute the query, we can see from the `matchCount` field value that there are 25 devices that match our query:

```
{
  "data": {
    "devices": {
      "edges": [
        {
          "node": {
            "id": "3"
          }
        },
        {
          "node": {
            "id": "27"
          }
        },
        {
          "node": {
            "id": "2132"
          }
        }
      ],
      "pageInfo": {
        "hasPreviousPage": false,
        "hasNextPage": true,
        "matchCount": 25
      }
    }
  }
}
```

## Example: Using Cursor-based Query Pagination

Here is an example of a query that uses a more complex cursor-based pagination method. In this example, we are searching for information about the first 10 devices that appear in the data set after a specific device, which we will define with a variable.

First, we must determine the cursor value for the record that we want to establish as the cursor. In this example, we want to use the 15th device as the cursor, so we would first do the following in GraphQL:

```
query getTenDevices {
  devices(first: 15, after:"") {
    edges {
      node {
        id
        ip
        name
        state
      }
      cursor
    }
    pageInfo {
      hasNextPage
    }
  }
}
```

That query returns the following data results:

```
{
  "data": {
    "devices": {
      "edges": [
        {
          ...first 14 records
        },
        {
          "node": {
            "id": "27",
            "ip": "10.2.5.72",
            "name": "10.2.5.72",
            "state": "0"
          },
          "cursor": "OpcmVjdGlvbiI6ImFzYyJ9XX0="
        }
```

```
      ],
      "pageInfo": {
        "hasNextPage": true
      }
    }
  }
}
```

In these results, we see that the cursor value for the 15th device is `"OpcmVjdGlvbiI6ImFzYyJ9XX0="`. We now need to use that cursor value in combination with a variable to get the first 10 device records after that 15th device. To do this, we will create a new query that uses a variable `$after` as the value of the argument `after`:

```
query getTenDevices($after: String!) {
  devices(first: 10, after: $after) {
    edges {
      node {
        id
        ip
        name
        state
      }
      cursor
    }
    pageInfo {
      hasNextPage
    }
  }
}
```

Note that, in the query above, the `hasNextPage` field nested under `pageInfo` is used in conjunction with the `first` and `after` arguments passed through the `devices` field. When paging forward through a data set using cursor-based pagination, the `hasNextPage` field is always used in conjunction with the `first` and `after` arguments; when paging backwards through a data set, the `hasPreviousPage` field nested under `pageInfo` is always used in conjunction with the `last` and `before` arguments.

Finally, we must define the variable `$after` with the cursor value that we previously determined. To do so, we would type its value into the **Variables** pane in the GQL browser in the following format:

```
{
  "after": "OpcmVjdGlvbiI6ImFzYyJ9XX0="
}
```

When we execute the query, GraphQL returns the requested data about the first 10 records after the cursor.

# Querying the ScienceLogic GraphQL API from an External Application

After you have determined the queries you want to execute using the GQL browser, you can execute those queries from an external application by performing an HTTPS request to the GraphQL URI for your SL1 system. To execute a GraphQL query using an HTTPS request:

- Use the POST method.
- Set the content-type header to "application/json".
- Specify the username and password of an appropriate user account.
- In the request content, send JSON in the following structure:

```
{
  "query":"[GraphQL query]"
}
```

For example, to execute the original example query, you would POST the following JSON:

```
{
  "query":"query devices { devices { edges { node { id name} } } }"
}
```

# Chapter

# 5

# Mutations

## Overview

This chapter describes how to form mutations in GraphQL to create, update, or delete data in SL1. It describes how to form basic mutation syntax and use variables, fragments, and directives in your mutations. Each section includes one or more examples to further illustrate each topic.

> NOTE: GraphQL is a complex language, and the information in this chapter is not an exhaustive list of every possible option at your disposal when writing and executing a mutation. For additional information about any of the subjects in this chapter, or for information not covered in this chapter, see the GraphQL documentation.

This chapter covers the following topics:


*What are Mutations?* ..... 55

*Mutation Types* ..... 56

*Basic Mutation Syntax* ..... 56

*Variables* ..... 57

*Fragments* ..... 58

*Directives* ..... 59


## What are Mutations?

***Mutations*** are GraphQL operations that modify data in the system and then return a value. Typically, mutations create new data, update existing data, or delete existing data.

## Mutation Types

Most mutations fall into one of three types:

- ***Create***. These mutations add new data to your system.
- ***Update***. These mutations update existing data on your system.
- ***Delete***. These mutations remove existing data from your system.

There are also less common mutation types, such as ***Validate***, ***Set***, ***Upgrade***, ***Duplicate***, and ***Save***.

To see a full list of mutations that are available for SL1, see the schema in the GQL browser.

## Basic Mutation Syntax

The most basic syntax of a mutation involves the following elements:

- ***Operation type***. For mutations, this is `mutation`. Unlike queries, the operation type is required when executing a mutation.
- ***Operation name***. A name that you define for the operation. While operation names are not always required, they are helpful to have for logging and debugging purposes.
- ***Object***. The mutation object, as defined in the schema. This is written as a verb—typically "create", "update", or "delete"—followed by the name of the object you are mutating, written together in camel case. For example, if you wanted to create a new physical device, you would use the object `createPhysicalDevice`. If you wanted to update an existing system process monitoring policy, you would use the object `updateMonitorSystemProcess`.
- ***Argument(s)***. The data that you want to pass to the database through the mutation. These are defined in the following structure: `(fieldName: value")`. The schema lists the argument field name and value type pairs that are available for each mutation field. Depending on the specific mutation, it might have no arguments, one argument, or multiple arguments. If it has multiple arguments, the argument structure is `(fieldName: "value", fieldName: "value")`, with any additional arguments separated by additional commas.
- ***Return field(s)***. The fields that you want GraphQL to return when you execute the mutation. For example, you might include a return field if you want to see the new state of an object after an update.

**NOTE:** If you mutate multiple objects, GraphQL executes the mutations on those objects in sequence, one after the other.

### Example: Updating a User's Name

Here is an example of a basic mutation. In this example, we are updating the name of an existing SL1 user:

```
mutation updateUserID {
  updateAccount(id: "11", user: "Example User", acceptedEula: true) {
```

```
    id
    user
    acceptedEula
  }
}
```

In this example:

- `mutation` is the operation type.
- `updateUserName` is the operation name that we defined for this specific operation.
- `updateAccount` is the object. We are telling GraphQL that we want to update an existing user account in SL1.
- `id: "11"`, `user: "Example User"`, and `acceptedEula: true` are the arguments that we are passing to the database. We are telling GraphQL that the user account we want to update has the user ID "11", and that we want to update that user's name to "Example User" and indicate that the user has accepted the SL1 end user license agreement.
- `id`, `user`, and `acceptedEula` are the return fields. We are telling GraphQL that, after it mutates the specified user account, we want it to return the user's ID and name, as well as confirmation the user is marked as having accepted the SL1 end user license agreement, so that we can verify the information.

When we execute this mutation, GraphQL returns the following, confirming that the mutation has updated the fields that we told it to:

```
{
  "data": {
    "updateAccount": {
      "id": "11",
      "user": "Example User",
      "acceptedEula": true
    }
  }
}
```

# Variables

Just like with queries, you can use ***variables*** to dynamically replace arguments in mutations, enabling you to reuse the mutation for multiple objects simply by changing the variable value.

The same rules apply to using variables in mutations as in queries.

## Example: Using Variables in a Mutation

In this example, we are taking the same mutation that we executed in the ***previous example*** and replacing the defined argument values with variables. This enables us to use the same mutation numerous times to add multiple new users by simply re-defining the variable values as needed:

```
mutation updateUserID($id: ID!, $userName: String!, $Eula: Boolean = true)
{
  updateAccount(id: $id, user: $userName, acceptedEula: $Eula) {
    id
    user
    acceptedEula
  }
}
```

We must then define these variables. To do so, we would type their values into the **Variables** pane in the GQL browser in the following format:

```
{"id": "11", "userName": "Example User"}
```

**NOTE:** In the above example, we did not define the `$Eula` variable because we declared a default value of `true` for it in our mutation. If we had wanted instead to pass a value of `false`, then we would have specified that with the other variable definitions in the **Variables** pane.

# Fragments

Similar to their use in queries, you can use ***fragments*** in mutations to group together a set of fields associated with a particular type that you can then reuse to define return fields in multiple mutations.

The same rules apply to using fragments in mutations as in queries.

## Example: Using a Fragment in a Mutation

In this example, we are taking the same mutation that we executed in the ***previous example*** and replacing the return fields `id`, `user`, and `acceptedEula` with the fragment `accountInfo`. This enables us to use the same return fields every time we use that mutation to add a new user:

```
fragment accountInfo on Account {
  id
  user
  acceptedEula
}

mutation updateUserID($id: ID!, $userName: String!, $Eula: Boolean = true)
{
  updateAccount(id: $id, user: $userName, acceptedEula: $Eula) {
    ...accountInfo
  }
}
```

# Directives

You can use ***directives*** to make GraphQL perform custom logic in your mutations, just as you can in queries. They can be attached to a field or a fragment that you are including in your mutation, and can affect the data that GraphQL displays when you execute the mutation.

The same rules apply to using fragments in mutations as in queries.

## Example: Using a Directive in a Mutation

In this example, we are taking the same scenario and fields as in the ***first example***, with one exception: this time, we are not marking the user as having accepted the SL1 end user license agreement by passing the argument `acceptedEula: true` to the `updateAccount` mutation, as we did previously.

We are still going to include the `acceptedEula` return field, however. This will inform us whether the user has already accepted the SL1 end user license agreement without us having done it for them.

But in this scenario, we are only concerned with users who have ***not*** accepted the license agreement. Therefore, we are going to include a directive to skip the inclusion of the `acceptedEula` return field if GraphQL determines that the value is already `true`:

```
mutation updateUserID {
  updateAccount(id: "11", user: "Example User") {
    id
    user
    acceptedEula @skip(if: true)
  }
}
```

As you can see, because that user account has already accepted the license agreement, GraphQL does not return the `acceptedEula` field when we execute the mutation:

```
{
  "data": {
    "updateAccount": {
      "id": "11",
      "user": "Example User"
    }
  }
}
```

© 2003 - 2024, ScienceLogic, Inc.

All rights reserved.

LIMITATION OF LIABILITY AND GENERAL DISCLAIMER

ALL INFORMATION AVAILABLE IN THIS GUIDE IS PROVIDED "AS IS," WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED. SCIENCELOGIC™ AND ITS SUPPLIERS DISCLAIM ALL WARRANTIES, EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE OR NON-INFRINGEMENT.

Although ScienceLogic™ has attempted to provide accurate information on this Site, information on this Site may contain inadvertent technical inaccuracies or typographical errors, and ScienceLogic™ assumes no responsibility for the accuracy of the information. Information may be changed or updated without notice. ScienceLogic™ may also make improvements and / or changes in the products or services described in this Site at any time without notice.

## Copyrights and Trademarks

ScienceLogic, the ScienceLogic logo, and EM7 are trademarks of ScienceLogic, Inc. in the United States, other countries, or both.

Below is a list of trademarks and service marks that should be credited to ScienceLogic, Inc. The ® and ™ symbols reflect the trademark registration status in the U.S. Patent and Trademark Office and may not be appropriate for materials to be distributed outside the United States.

- ScienceLogic™
- EM7™ and em7™
- Simplify IT™
- Dynamic Application™
- Relational Infrastructure Management™

The absence of a product or service name, slogan or logo from this list does not constitute a waiver of ScienceLogic's trademark or other intellectual property rights concerning that name, slogan, or logo.

Please note that laws concerning use of trademarks or product names vary by country. Always consult a local attorney for additional guidance.

## Other

If any provision of this agreement shall be unlawful, void, or for any reason unenforceable, then that provision shall be deemed severable from this agreement and shall not affect the validity and enforceability of any remaining provisions. This is the entire agreement between the parties relating to the matters contained herein.

In the U.S. and other jurisdictions, trademark owners have a duty to police the use of their marks. Therefore, if you become aware of any improper use of ScienceLogic Trademarks, including infringement or counterfeiting by third parties, report them to Science Logic's legal department immediately. Report as much detail as possible about the misuse, including the name of the party, contact information, and copies or photographs of the potential misuse to: legal@sciencelogic.com. For more information, see https://sciencelogic.com/company/legal.

ScienceLogic

800-SCI-LOGIC (1-800-724-5644)

International: +1-703-354-1010