



SIEM Integration Guide

HOW-TO FOR THIRD PARTY INTEGRATORS

Version: 1



WORKING WITH THE CATO API.....	3
OVERVIEW.....	3
API ENDPOINT	4
AUTHENTICATION	4
YOUR FIRST REQUEST: CURL ONE-LINER	5
SCHEMA.....	6
API PLAYGROUND.....	7
DOCUMENTATION	9
SAMPLE CODE	10
ERRORS.....	12
<i>HTTP 422</i>	12
<i>Authentication</i>	14
<i>Internal server errors</i>	14
RATE LIMITS	15
GETTING ACCOUNT INFORMATION VIA ENTITYLOOKUP	16
SCHEMA.....	16
USE CASES.....	17
EXECUTION	17
GETTING STATUS DATA WITH ACCOUNTSNAPSHOT	18
SCHEMA.....	18
USE CASES.....	18
EXECUTION	19
GETTING PERFORMANCE METRICS VIA ACCOUNTMETRICS.....	21
SCHEMA.....	21
USE CASES.....	22
EXECUTION	22
<i>Timeframe</i>	22
<i>Metrics</i>	22
<i>Timeseries Data</i>	25
GETTING SECURITY AND CONNECTIVITY EVENTS VIA EVENTSFEED	27
SCHEMA.....	27
USE CASES.....	27
EXECUTION	28
<i>Event Lifecycle</i>	28
<i>Enabling Events Feed in the Cato Management Application</i>	29
<i>Event Record Formats</i>	29
<i>Cato Event Fields, Types and Subtypes</i>	32
<i>Pagination and Markers</i>	34
<i>Filtering</i>	35
<i>Using the Sample Script</i>	38
EXAMPLE: SENDING EVENTS TO MICROSOFT SENTINEL.....	39
<i>Overview</i>	39
1. <i>Validate our Sentinel environment.</i>	40

2.	<i>Launch a Linux VM in Azure.</i>	40
3.	<i>Transfer the sample script to the Linux VM.</i>	42
4.	<i>Set up the sample script.</i>	42
5.	<i>Test the sample script.</i>	43
6.	<i>Check that we can see Cato events in Sentinel.</i>	43
7.	<i>Schedule the sample script for continuous feed.</i>	44
APPENDIX A – SAMPLE EVENTS		47
SECURITY		47
	<i>Internet Firewall</i>	47
	<i>IPS</i>	47
	<i>Anti Malware</i>	47
	<i>NG Anti Malware</i>	48
	<i>RPF</i>	48
CONNECTIVITY		48
	<i>Connected</i>	48
	<i>Cato Management Application</i>	48
APPENDIX B – EVENTSFEED.PY		49

Working with the Cato API

Overview

Cato customers can programmatically download configuration, status and event data via our GraphQL API. GraphQL is a modern REST-like (but not REST) API suitable for enterprise-grade web applications. GraphQL was created by Facebook as a query language for use in high volume APIs and is comprehensively documented at the GraphQL Foundation's website <https://graphql.org/>. Requests are to a single URL, with a JSON-like query in the body of the POST request; the response is always in strict JSON. The main difference between REST and GraphQL is that GraphQL requires callers to specify the fields they want to be returned in the response, which allows callers to only request the fields which they are interested in receiving, rather than having a rigid response format containing many irrelevant fields.

API users who are coming to the Cato API from a REST background can sometimes overestimate the effort required to work with a GraphQL API. The response format is the same as REST; the principles are mostly the same as REST, the query language is simple and the fields are mostly self-documenting.

Cato Networks provides a comprehensive suite of converged WAN, Internet access and network security solutions ranging from basic network connectivity through to CASB, DLP and ZTNA. For most of our customers "we are the network" which gives us both broad and deep visibility into network traffic and security posture. That visibility results in a high volume of configuration and event data. The largest Cato customers have thousands of sites and users generating over 1 billion events and nearly 900GB of log data each week, making it important for third party integrators to consider what sort of information they need to download from Cato and design their queries accordingly.

API users who are coming to the Cato API from a REST background often underestimate the volume of data generated by their Cato accounts. They are familiar with traditional on-premise systems where limitations in processing and storage mean that relatively few events are logged, whereas Cato's cloud-first architecture enables everything to be logged. When asking a third party integrator "What sort of events are you interested in?" it's not uncommon for them to respond with "Give us everything" thinking that they might get a few thousand events every 24 hours. It's also not uncommon for them to quickly walk that back to "We just want IPS blocks and VPN user logins" once they realise that "everything" can mean several thousand events each minute.

Currently, there are five public API queries:

- `accountMetrics` – real-time and historic performance metrics for sites and VPN users, including throughput, packet loss, jitter and latency.
- `accountSnapshot` – real-time performance metrics, health status and some configuration information.
- `auditFeed` – near-real-time and historic audit trail events.

- entityLookup – retrieve a list of the account’s administrators, sites or VPN users.
- eventsFeed – retrieve security, connectivity and health events for consumption by a third party SIEM or analytics platform.

Additional API calls (including write APIs) are currently being developed. Cato customers who require functionality outside of the five API calls listed above are encouraged to raise a Request For Enhancement (RFE) through their Cato account team or Cato Technical Support.

API Endpoint

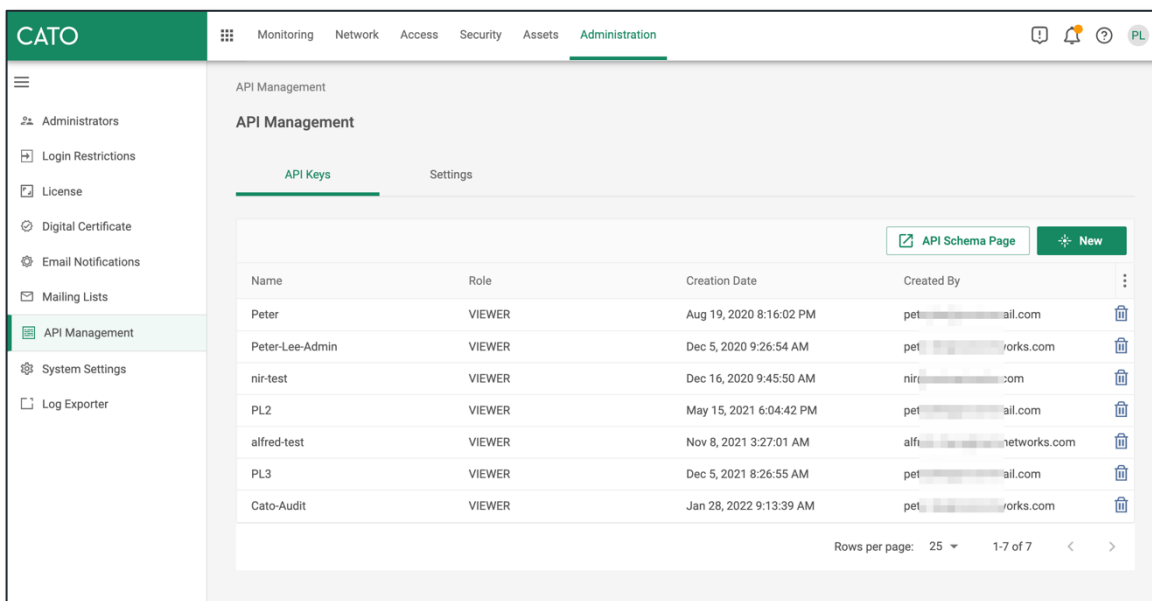
There is only one API endpoint to which users make requests:

<https://api.catonetworks.com/api/v1/graphql2>

Each request is a POST with the GraphQL query in the body of the request. The content type must be “application/json”. Responses are always UTF-8 JSON.

Authentication

All access to the Cato API requires an API key. Keys are generated in the Cato Management Application (CMA) by account administrators who have the Editor privilege. Keys do not automatically expire but can be revoked at any time. The list of current API keys can be viewed in the CMA by any account administrator with at least Viewer privilege:

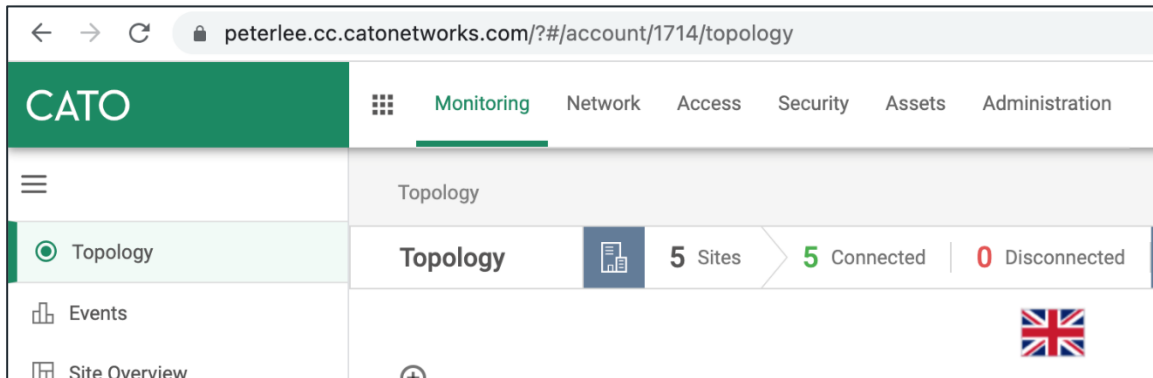


If you are a third party without administrative access to the Cato customer’s console (such as a SOC provider) you will need to ask your customer to generate an API key for you.

API keys are 512 bits expressed as a 32 character hexadecimal string. They are only shown at the time of creation, after which they are unrecoverable and if lost must be revoked and replaced with

a new key. The key string is included with each API request as the value of a custom header called "x-api-key".

All API calls also require an account ID parameter. The easiest way to locate the account ID for an account is to look at the four-digit integer in the browser address bar when logged into the CMA. For example, the ID of this account is 1714:



Your first request: curl one-liner

One of the simplest requests possible with the Cato API is to call a query like `accountSnapshot` and ask for one field to be returned, the ID field. The ID is also one of the inputs to the query so there's no new information being returned, but it is a useful way of verifying that you have the right account ID, that your API key is valid, that you have the right API endpoint URL and that you're able to successfully make API calls. The full curl request, with the lines broken up to more easily see the individual parameters, is:

```
curl -s https://api.catonetworks.com/api/v1/graphql2 \
  -H Content-Type:application/json \
  -H x-api-key:D7912C93E1B14117EFDD51464FEC485F \
  -d '{"query":"{accountSnapshot(id:1714){id}}"}'
```

The field in **red** is the API key and the field in **green** is the account ID. Please refer to the section on **Authentication** if you do not have either of these and remember that if you are a third party working with a Cato customer, you will need to obtain these from your customer.

If you are running on Linux or Mac and don't have access to curl then you can skip this section and proceed to **API Playground** below where we will be running this same query in a browser. Alternatively, you could take the time to spin up a Linux VM or download curl for Windows. It is an extremely useful tool which is the de facto standard for making HTTP requests from a command line and is well worth getting to know.

If the curl command works then you should see something like this at your CLI:

```
$ curl -s https://api.catonetworks.com/api/v1/graphql \
> -H Content-Type:application/json \
> -H x-api-key:D7912C93E1B14117EFDD51464FEC485F \
> -d '{"query":"{accountSnapshot(id:1714){id}}"}'
{"data":{"accountSnapshot":{"id":"1714"}}}sh-3.2$
```

The actual response is:

```
{"data":{"accountSnapshot":{"id":"1714"}}}
```

Responses are always in UTF-8 so non-ASCII characters should appear fine provided your query tool/script supports Unicode strings:

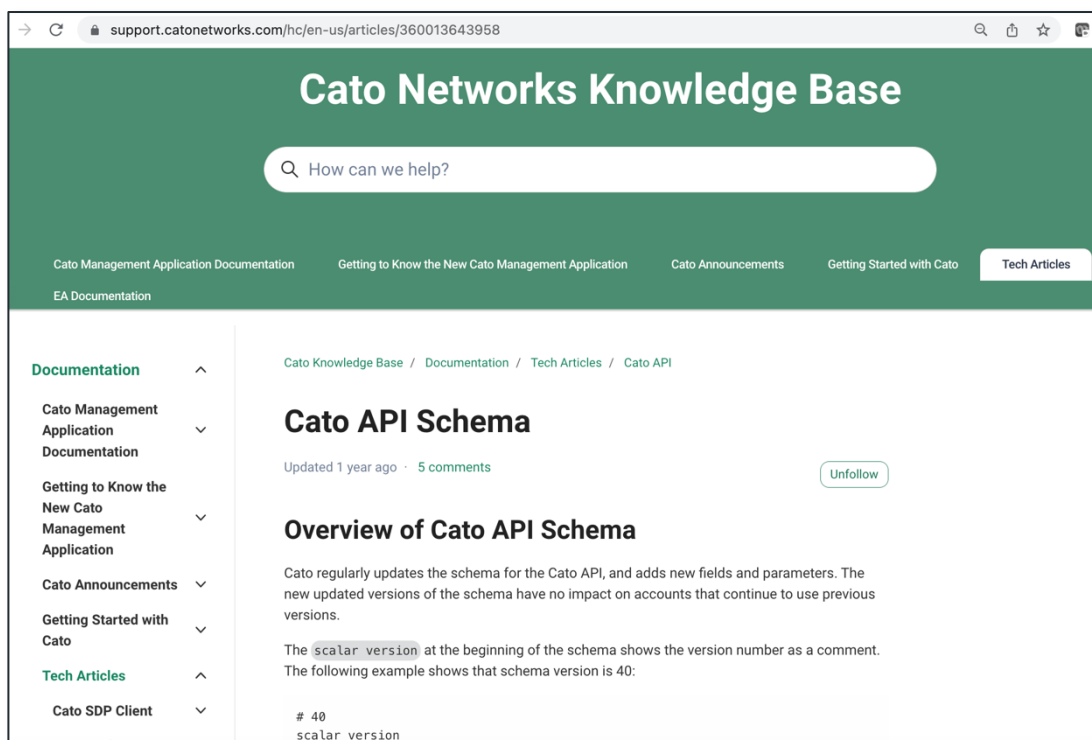
```
{"data":{"accountSnapshot":{"users":[{"name":"Peter Leé"}]}}}
```

An unsuccessful request due to invalid authentication will look like this:

```
{"errors":[{"message":"authorization error",
"path":["accountSnapshot","id"]}], "data":{"accountSnapshot":null}}
```

Schema

The schema defines the types which make up the GraphQL interface. It is written in a JSON-like GraphQL definition language and can be downloaded from the Cato Networks Knowledge Base (<https://support.catonetworks.com/hc/en-us/articles/360013643958>):



Inside the schema, you will see the five queries plus definitions of each field and type. Some of the queries are fairly small, such as entityLookup, but some (such as accountSnapshot) return objects which contain other objects which contain other objects and so on, resulting in hundreds of fields to choose from when designing your query.

A full explanation of how to design queries from the schema is outside of the scope of this document. Users who want to learn more about GraphQL schemas and types are encouraged to read the GraphQL Foundation's documentation, available here: <https://graphql.org/learn/schema/>.

All of the sample Python scripts available from the Cato Networks Knowledge Base can display a full debug output, including GraphQL queries and responses, so the quickest way for a Cato API user to get hold of a working query for any Cato API request is to run the sample script with the -V parameter. For example:

```
$ python3 entityLookup.py -I 1714 -K D7912C93E1B14117EFDD51464FEC485F -t admin -V
LOG 2022-04-27 14:11:18.375965>
```

```
{
  entityLookup(accountID:1714 type:admin from:0 limit:1000) {
    total
    items {
      entity {
        id
        name
        type
      }
      description
      helperFields
    }
  }
}
```

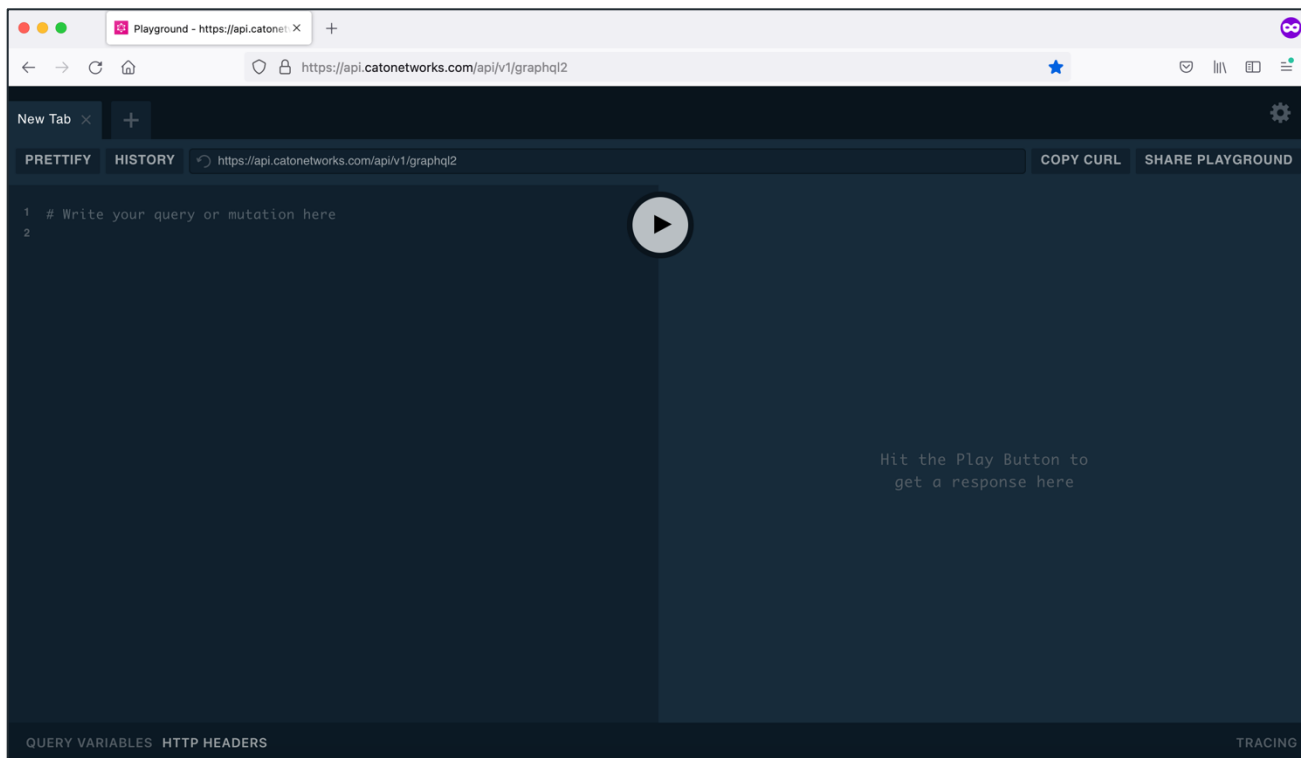
```
LOG 2022-04-27 14:11:20.075263> {'data': {'entityLookup': {'total': 4, 'items':
[{'entity': {'id': '4472', 'name': 'peterjlee@xxx.com', 'type': 'admin'},
'description': 'Peter Lee', 'helperFields': {}}, {'entity': {'id': '4571', 'name':
'peter.lee@xxx.com', 'type': 'admin'}, 'description': 'Peter Lee', 'helperFields':
{}}, {'entity': {'id': '6173', 'name': 'phil_keeling@xxx.com', 'type': 'admin'},
'description': 'Phil', 'helperFields': {}}, {'entity': {'id': '258', 'name':
'suri@xxx.com', 'type': 'admin'}, 'description': 'Suri', 'helperFields': {}}]}}
```

In this example, the actual GraphQL query is marked in green and the actual GraphQL response is the text in blue.

API Playground

One of the biggest challenges when creating a GraphQL query is understanding what the different fields do and what to expect from the output, and then if something isn't working in your script, understanding if this is a problem with your GraphQL query or your script. To make it easier for API users to develop and troubleshoot queries, Cato provides an online web-based GraphQL query tool called the "API Playground".

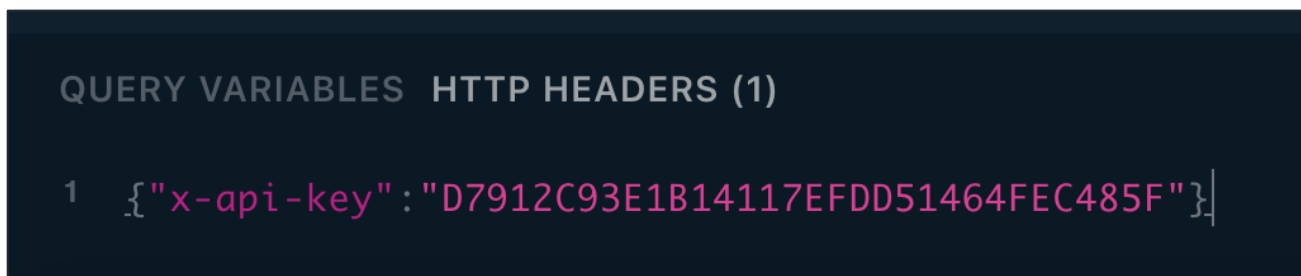
You can access the playground at this URL: <https://api.catonetworks.com/api/v1/graphql2> which just happens to be the same URL accessed by your scripts, i.e. the API endpoint. When accessed via a web browser the endpoint knows to return the playground interface; when accessed via a script or CLI tool (with the right headers) the endpoint knows to return an API response.



The first thing you will need to do is to enter in your API key, which you do down in the “HTTP HEADERS” section. The format has to be like this:

```
{"x-api-key": "D7912C93E1B14117EFDD51464FEC485F"}
```

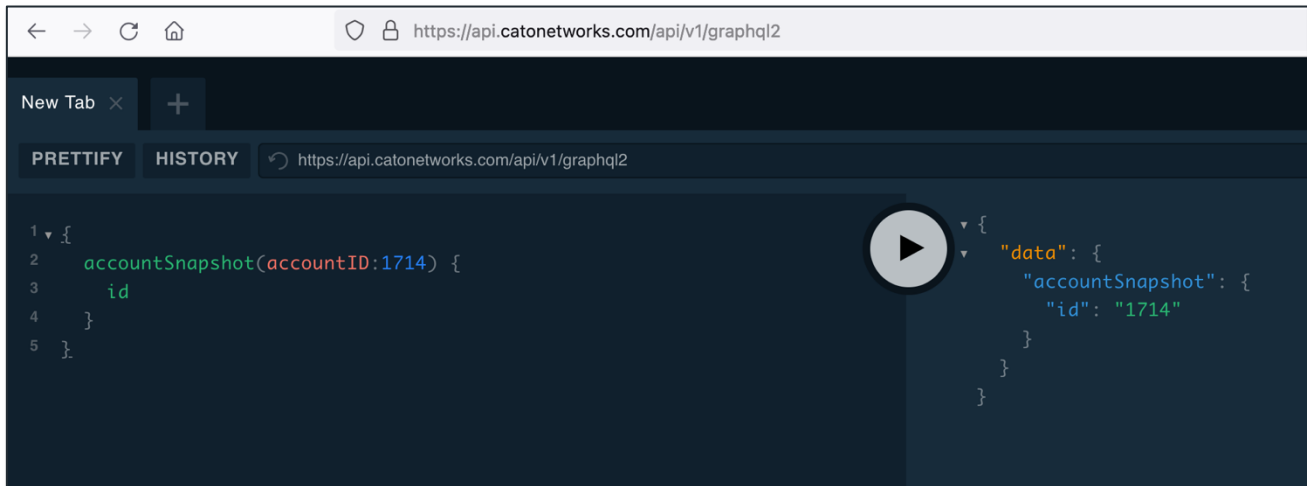
Note that those are standard ASCII double-quote characters, not new-fangled Unicode quotes, “Smart” quotes or single quotes. Once you have entered your API key, it should look like this (**don’t forget to replace this demo key with your own key**):



The next thing is to enter a GraphQL query into the query window. The query from the curl one-liner is:

```
{
  accountSnapshot(accountID:1714) {
    id
  }
}
```

If you enter this into the query window (don't forget to replace my demo id 1714 with your Cato account ID) and press the play button, you should see something similar to this:



The API Playground is a very useful tool for validating queries, troubleshooting queries which are failing, and reproducing issues when engaging with Cato Technical Support.

There are many other tools which can be used to access APIs, such as Postman and Altair, some of which are even documented in the Cato Networks Knowledge base. Some API users make the mistake of initially using one of these tools when they are getting started because they provide a slick interface and claim to hide much of the complexity of the API. Although these are excellent tools which have their place, they are not good choices for getting started with the Cato API. If you experience an issue while accessing the Cato API using any tool other than the Cato API Playground, please migrate your query to the API Playground and try to reproduce your issue in this environment before engaging with Cato Support.

Documentation

There are two sources for Cato Networks API documentation:

- The Cato Networks Knowledge Base.
- The official sample scripts.

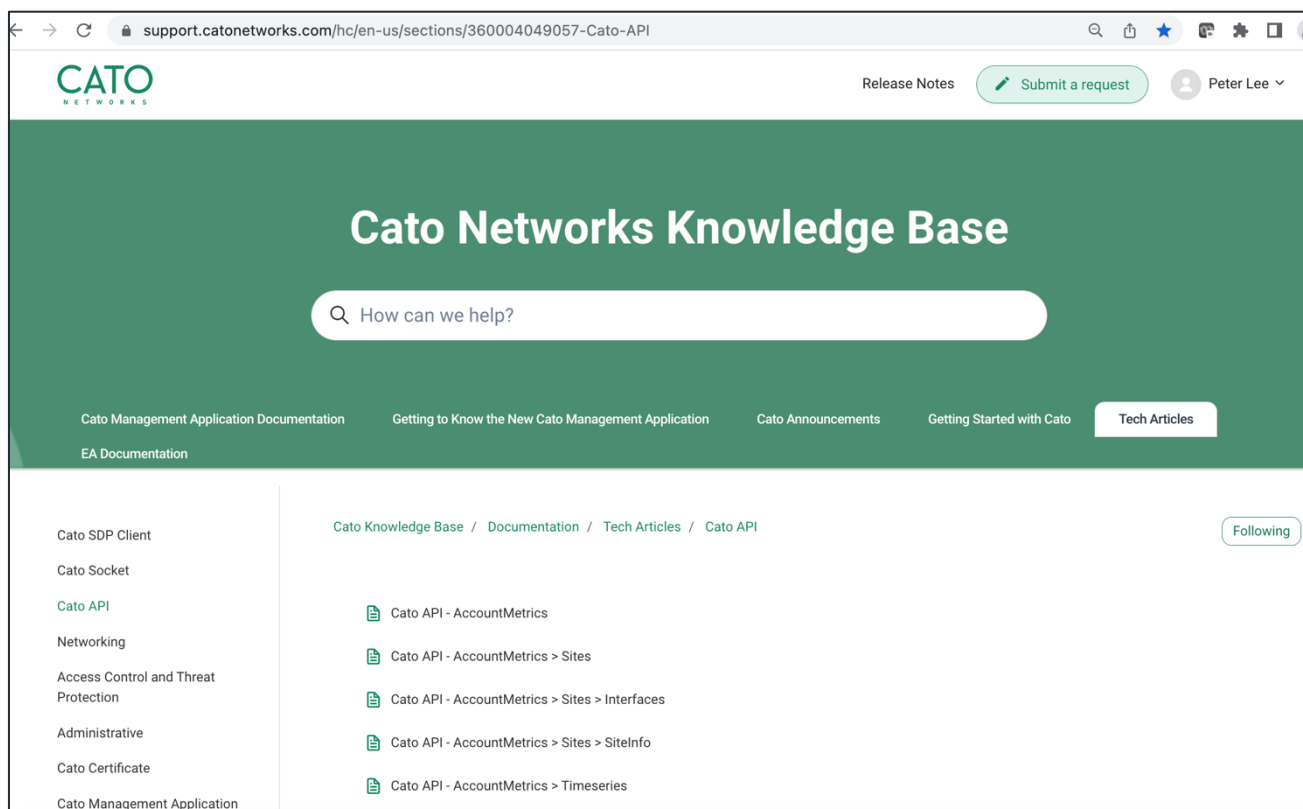
We will cover the sample scripts in the next section. The Knowledge Base (KB) is an online resource available to all Cato account administrators. If you are a Cato customer yourself or a

Cato customer's authorized partner, you will have direct access to the KB. If you are a third party integrator you probably won't have access to the KB because you won't have a CMA administrator login, but you can ask your Cato customer to export KB articles in PDF format for you to read offline.

There are detailed articles covering each query, the request and response fields, how to run queries and troubleshoot API calls. The API section starts at this link:

<https://support.catonetworks.com/hc/en-us/sections/360004049057-Cato-API>

And looks like this:



Sample Code

Cato provides sample scripts for retrieving data via the Cato API. There is one script for each public API call which can be downloaded from the Cato Networks Knowledge Base. The scripts are written for Python 3 and the Python Standard Library, so there is no need to install additional third-party modules. The scripts are designed to be simple, easy to use and ready as a drop-in method for accessing the Cato API:

- They use a very plain, standard Python which should be compatible with any relatively recent Python 3 runtime.
- The lack of third party imports eliminates the need to install additional packages.

- The code is meant to be simple and easy even for novice programmers to understand.
- They include support for handling API errors and rate-limiting.
- Extensive use of CLI parameters means that no code changes are necessary for most situations.
- They use simple, non-parameterised GraphQL queries which eliminate some of the up-front complexity of moving to GraphQL.

The screenshot shows the Cato Networks Knowledge Base interface. At the top is a search bar with the text "How can we help?". Below the search bar is a navigation menu with links for "Cato Management Application Documentation", "Getting to Know the New Cato Management Application", "Cato Announcements", "Getting Started with Cato", and "Tech Articles". The "Tech Articles" link is highlighted. On the left side, there is a sidebar menu with categories like "Documentation", "Getting to Know the New Cato Management Application", "Cato Announcements", "Getting Started with Cato", "Tech Articles", "Cato SDP Client", and "Cato Socket". The main content area displays an article titled "Example Scripts: Using the Cato API with Python" with a "Follow" button and a "Contents" table of contents listing items like "Overview", "accountMetrics", "accountSnapshot", "auditFeed", "entityLookup", and "eventsFeed".

Each script includes extensive documentation at the beginning of the file and if executed without any parameters, will print a help statement detailing all possible parameters. For example:

```
$ python3 entityLookup.py
Usage: entityLookup.py [options]
```

Options:

- h, --help show this help message and exit
- K API_KEY API key
- I ID Account ID
- P Prettify output
- p Print entity records
- t ENTITY_TYPE Entity type. Supported types are site, vpnUser, admin
- v Print debug info
- V Print detailed debug info

Customers are encouraged to download the scripts, use them to access the API and study them for guidance on how to customise their code. Customers with support issues will often be asked to download and try to reproduce the issue with the sample script.

Although the sample scripts are “battle-tested” and intended for use in a production environment, they are not an official Cato product and are not within scope for support by Cato Technical Support. Any questions or concerns regarding the sample scripts should be sent to api@catonetworks.com.

Customers who intend to use the scripts in production should satisfy themselves as to their production readiness. Although they include some error-detection and logging capabilities, this might not be sufficient for all environments. They also do not validate the endpoint TLS certificate which could represent a security risk in some environments. Customers for whom this is not acceptable are welcome to modify the script to suit their own risk profile.

Errors

Although the API endpoint is robust and reliable and rarely a source of API errors, API users should ensure that their scripts and tools can detect and handle an API error. Users are much more likely to encounter errors caused by their own code or by transient network events (such as a loss of network connectivity at the client end, or high query rates causing rate-limiting) than a problem with the API endpoint itself. Rate limiting will be covered in the next section. In the meantime, the most commonly-encountered errors and how to handle them are:

HTTP 422

If the user sends a malformed GraphQL query string or a query that the endpoint can't understand, it may respond with an HTTP error code 422 “Unprocessable Entity”. The standard sample scripts are configured to catch the exceptions thrown by HTTP errors, on which they sleep for 2 seconds and then retry up to a maximum of 10 times before bailing out.

```
while True:
    if retry_count > 10:
        print("FATAL ERROR retry count exceeded")
        sys.exit(1)
    try:
        request =
urllib.request.Request(url='https://api.catonetworks.com/api/v1/graphql2',
                        data=json.dumps(data).encode("ascii"),headers=headers)
        response = urllib.request.urlopen(request, context=no_verify, timeout=30)
        api_call_count += 1
    except Exception as e:
        log(f"ERROR {retry_count}: {e}, sleeping 2 seconds then retrying")
        time.sleep(2)
        retry_count += 1
        continue
```

If you call a sample script with the -v parameter you can see this in action.

For example, say that you want to use the entityLookup.py sample script to retrieve a list of sites in your account. You want to refresh your memory as to what parameters are required, so you first run the script with no parameters:

```
$ python3 entityLookup.py
Usage: entityLookup.py [options]
```

Options:

```
-h, --help          show this help message and exit
-K API_KEY          API key
-I ID               Account ID
-P                 Prettify output
-p                 Print entity records
-t ENTITY_TYPE     Entity type. Supported types are site, vpnUser, admin
-v                 Print debug info
-V                 Print detailed debug info
```

You realise that you need to call it with the -I and -K parameters for authentication, the -p parameter to see the records, and the -t parameter, but when you run it nothing happens for a while until the script stops with a fatal error:

```
$ python3 entityLookup.py -I 1714 -K D7912C93E1B14117EFDD51464FEC485F -t sites -p
FATAL ERROR retry count exceeded
```

So you run it again and this time you add the -v parameter to see debug output:

```
$ python3 entityLookup.py -I 1714 -K D7912C93E1B14117EFDD51464FEC485F -t sites -p -v
LOG 2022-04-27 19:19:09.990518> ERROR 0: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:12.233893> ERROR 1: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:14.421091> ERROR 2: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:16.670238> ERROR 3: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:18.905914> ERROR 4: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:21.447685> ERROR 5: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:23.699166> ERROR 6: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:26.751635> ERROR 7: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:29.590447> ERROR 8: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:31.861476> ERROR 9: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
LOG 2022-04-27 19:19:34.020327> ERROR 10: HTTP Error 422: Unprocessable Entity,
sleeping 2 seconds then retrying
FATAL ERROR retry count exceeded
```

Seeing that there is a problem with the query, you take a closer look and realise that you have called it with “-t sites” instead of “-t site”. You run it again with the correct parameter and this time, it works:

```
$ python3 entityLookup.py -I 1714 -K D7912C93E1B14117EFDD51464FEC485F -t site -p -v
LOG 2022-04-27 19:23:48.711794> iteration:1 count:5 total_count:5 Total:5 Lechlade -
Sydney
{"entity": {"id": "34639", "name": "Lechlade", "type": "site"}, "description": "BRANCH
", "helperFields": {"connectionType": "Socket X1500", "creationDate": "Jan 10, 2022
1:40:24 PM", "description": "", "isHA": "false", "lastModified": "Mar 17 2022 1:36:24
PM", "operationalStatus": "active", "protoID": "1000000212", "type": "BRANCH"}}
{"entity": {"id": "37867", "name": "London", "type": "site"}, "description": "CLOUD_DC
", "helperFields": {"connectionType": "vSocket AWS", "creationDate": "Mar 28, 2022
9:03:42 AM", "description": "", "isHA": "false", "lastModified": "Mar 28 2022 10:23:22
AM", "operationalStatus": "active", "protoID": "1000000214", "type": "CLOUD_DC"}}
```

Authentication

If the API Endpoint cannot match the account ID with the API key, the result will be an error message in JSON format:

```
$ python3 accountSnapshot.py -I 1714 -K not_the_right_key -p
{'errors': [{'message': 'authorization error', 'path': ['accountSnapshot',
'accountID']}], 'data': {'accountSnapshot': None}}
```

Unlike the HTTP error above, the response code for this error will be 200 OK. This makes it necessary to check the response body for error conditions. The sample scripts do this by converting the JSON response string into a Python dictionary, and checking for an “errors” dictionary key:

```
result = json.loads(result_data.decode('utf-8','replace'))
if "errors" in result:
    log(f"API error: {result_data}")
```

Internal server errors

Input problems can also result in an “internal server error” which will also be detected by the above code snippet. For example, one of the parameters to auditFeed.py is a timeframe parameter:

```
$ python3 auditFeed.py
Usage: auditFeed.py [options]
```

Options:

```
-h, --help      show this help message and exit
-K API_KEY      API key
-I ID           Account ID
-P             Prettify output
-p             Print audit events
-t TIMEFRAME    timeframe for query. Default is last 5 minutes (last.PT5M)
-v             Print debug info
-V             Print detailed debug info
```

If auditFeed is called with a mangled parameter, the result is an internal server error:

```
$ python3 auditFeed.py -I 1714 -K D7912C93E1B14117EFDD51464FEC485F -t last.PT5X
{'errors': [{'message': 'internal server error', 'path': ['auditFeed', 'timeFrame']}],
'data': {'auditFeed': None}}
```

Rate Limits

To maintain the availability of the Cato API Endpoint infrastructure, rate limits are applied when users send too many queries in too short a time. When a rate limit is imposed, the response is a 200 OK with an error string in JSON:

```
{"errors":[{"message":"rate limit for operation: <call> was reached"}],"data":null}
```

The sample script captures this error and sleeps for five seconds before retrying:

```
result_data = response.read()
if result_data[:48] == b'{"errors":[{"message":"rate limit for operation:':
    log("RATE LIMIT sleeping 5 seconds then retrying")
    time.sleep(5)
    continue
```

Rate limits are applied on a per-query, per-account basis. This means that for each query there is an individual counter, but it applies to all API keys querying that account. So two different users calling two different queries would not impact each other, but if two different users were calling the same query, their queries would be counted together for the purposes of rate-limiting and so one user can cause another user to experience rate limiting.

The Cato API back end is highly available and elastic, so the rate limits are a guaranteed minimum rather than an absolute maximum. For example, that auditFeed has a rate limit of 5 per minute means that auditFeed can be called for an account at least five times every 60 seconds without being rate limited. In reality, customers can often call it much more frequently than this, but the guaranteed minimum rate of auditFeed calls is five per minute. This is still subject to the account-wide counter, so if you have five different users who all want to query auditFeed, to guarantee that they will never be rate-limited you would need to ensure that each user isn't calling the query more than once every 60 seconds.

Limits are:

accountMetrics:	120/minute
accountSnapshot:	120/minute
auditFeed:	5/minute
entityLookup:	30/minute
eventsFeed:	75/minute

Getting account information via entityLookup

Schema

```
entityLookup(  
  # The account ID (or 0 for non-authenticated requests)  
  accountID: ID!  
  type: EntityType!  
  # Sets the maximum number of items to retrieve  
  limit: Int = 50  
  # Sets the offset number of items (for paging)  
  from: Int = 0  
  # Return items under a parent entity (can be site, vpn user, etc),  
  # used to filter for networks that belong to a specific site for example  
  parent: EntityInput  
  # Adds additional search parameters for the lookup. Available options:  
  # country lookup: "removeExcluded" to return only allowed countries  
  # countryState lookup: country code ("US", "CN", etc) to get country's states  
  search: String = ""  
  # Adds additional search criteria to fetch by the selected list of entity IDs.  
  This option is not  
  # universally available, and may not be applicable specific Entity types. If used  
  # on non applicable entity  
  # type, an error will be generated.  
  entityIDs: [ID!]  
  # Adds additional sort criteria(s) for the lookup.  
  # This option is not universally available, and may not be applicable specific  
  Entity types.  
  sort: [SortInput]  
): EntityLookupResult!  
  
type EntityLookupResult {  
  items: [EntityInfo!]!  
  total: Int  
}  
  
type EntityInfo {  
  entity: Entity!  
  description: String!  
  helperFields: Map!  
}  
  
type Entity {  
  id: ID!  
  name: String  
  type: EntityType!  
}
```

Use Cases

The entityLookup query serves two primary use cases:

- To map object names to object IDs for use in other queries.
- To retrieve a list of objects with basic configuration data.

Some of the other queries have objects which can be filtered by ID. Object IDs are in general not shown in the CMA, making entityLookup one of the few ways for Cato customers to get them. For example, the accountSnapshot query returns a list of site snapshot objects and a list of user snapshot objects. All sites are included regardless of status but by default, only users who are currently logged in are available in the snapshot. To force accountSnapshot to include users who are not currently logged in the API caller can add a list of user IDs to the query, which is where entityLookup comes in.

The query's helperFields field is a set of key:value pairs which add data to the object. The precise nature of the data varies according to the query type. The "admin" type has no helper fields, but the vpnUser includes several useful items:

```
{"entity": {"id": "122364", "name": "Jack Jarvis", "type": "vpnUser"}, "description": "jack@peterljames.int | DISABLED, since Jul 27, 2019 11:15:16 PM", "helperFields": {"creationDate": "Jul 27, 2019 11:15:16 PM", "email": "jack@peterljames.int", "importType": "LDAP", "lastModified": 1643321818619, "operationalStatus": "disabled", "prefix": "AD:", "protoID": "4"}}
```

Execution

This query is subject to pagination. If the total number of entities to retrieve exceeds the maximum which can be returned in a single query (defined by the "limit" input parameter), the caller needs to keep submitting the query with an updated "from" parameter until the total number of entities retrieved matches the "total" field. An example of this logic can be seen in the sample entityLookup.py script (unnecessary lines removed for clarity):

```
total_count = 0
while True:
    query = '''
{
    entityLookup(accountID:''' + options.ID + ''' type:''' + options.entity_type + '''
from:''' + str(total_count) + ''' limit:1000) {
    total
    items { entity { id } }
}
'''
    success, resp = send(query)
    count = len(resp["data"]["entityLookup"]["items"])
    total_count += count
    total = int(resp["data"]["entityLookup"]["total"])
    if total_count >= total:
        break
```

Getting status data with accountSnapshot

Schema

```
# Account current snapshot based on Near Realtime Stats.
# Provides information similar to the topology page of the account.
accountSnapshot(
  # For the transition period of id been deprecated,
  # both accountID and id would be optional
  # if both are provided, accountID would take precedence.
  accountID: ID
  id: ID
): AccountSnapshot

type AccountSnapshot {
  id: ID
  # Sites includes information about online as well as offline sites
  sites(
    # For the transition period of id been deprecated,
    # both siteIDs and ids would be optional
    # if both are provided, siteIDs would take precedence.
    siteIDs: [ID!]
    ids: [Int!]
  ): [SiteSnapshot!]
  # VPN users information includes only connected users by default (Unlike sites),
  unless specific ID is requested
  users(
    # request specific IDs, regardless of if connected or not
    userIDs: [ID!]
    # For the transition period of ids been deprecated,
    # both userIDs and ids would be optional
    # if both are provided, siteIDs would take precedence.
    ids: [Int!]
  ): [UserSnapshot!]
  timestamp: Time
}
```

The SiteSnapshot and UserSnapshot objects can be looked up in the schema. A full response from accountSnapshot includes nearly 400 fields. There is an example of the full response in the sample accountSnapshot.py file.

Use Cases

The accountSnapshot query is primarily used to iterate over an account's sites and users, retrieving asset information such as socket IDs and VPN user devices, current connection status, and a metrics snapshot. The metrics are from a fixed timeframe of "the last 30 seconds" which is not massively useful in most integration scenarios. Integrators who want to retrieve performance metrics should use the accountMetrics query instead.

Execution

API users will typically call `accountSnapshot` to retrieve a list of `siteSnapshot` and/or `userSnapshot` objects containing fields of interest. For sites, these are likely to be the connectivity and HA status for health check purposes or socket device details for inventory purposes. If the list of sites is requested, it will always include all sites regardless of their connectivity status. If users are requested this will by default only include users who are currently connected – to retrieve data for other specific users (or all users regardless of connection status) the `entityLookup` query should be used first to build a list of user IDs.

For example, say you want to report on all devices used by VPN users. You want to know the device type, Cato Client version, OS version and when the device was last used. This code uses `entityLookup` to build the user list, then calls `accountSnapshot` with that list and prints the data. Some lines were removed for brevity:

```
total_count = 0
users = {}
while True:
    user_query = '''
{
    entityLookup(accountID:'''+options.ID+''' type:vpnUser from:''' +
str(total_count) + ''' limit:1000) {
        total items { entity { id } helperFields } } }
'''
    result,resp = send(user_query)
    total_count += len(resp["data"]["entityLookup"]["items"])
    total = int(resp["data"]["entityLookup"]["total"])
    for item in resp["data"]["entityLookup"]["items"]:
        users[item["entity"]["id"]] = item
    if total_count >= total:
        break

selected_ids = [int(k) for k,v in users.items()]
device_query = '''
{
    accountSnapshot(accountID:'''+options.ID+''') {
        users(userIDs:'''+str(selected_ids)+''') {
            id
            name
            devices {id name lastConnected osType osVersion version } } } }
'''
    result,resp = send(device_query)
    for u in sorted(resp["data"]["accountSnapshot"]["users"],key=lambda x: x["name"]):
        if u["devices"]:
            user = users[u["id"]]
            for device in u["devices"]:

print(u["id"],u["name"],user["helperFields"]["email"],device["name"],device["osType"],
        device["osVersion"],device["version"],device["lastConnected"],sep=",")
```

Running the full script:

```
$ python3 GetVPNDevices.py -I 1714 -K D7912C93E1B14117EFDD51464FEC485F
277009,Peter James,p@x.com,Peter's MacBook,OS_MAC,11.6.0,4.5.1,2021-12-23T15:23:17Z
277009,Peter James, p@x.com,Joburg-Win10a,OS_WINDOWS,10,4.5.102.705,2020-12-20T23:57:56Z
277009,Peter James, p@x.com,DESKTOP-E8HLQ2V,OS_WINDOWS,10,4.7.106.794,2021-09-07T13:48:34Z
277009,Peter James, p@x.com,n/a,OS_ANDROID,8.0.0,4.2.1.92,2021-09-22T15:37:52Z
277009,Peter James, p@x.com,Pune-Win,OS_WINDOWS,10,4.7.106.794,2021-11-12T15:59:45Z
277009,Peter James, p@x.com,PETERLEE59FD,OS_WINDOWS,10,4.7.106.794,2022-01-11T13:27:01Z
277009,Peter James, p@x.com,Peter's MacBook Pro,OS_MAC,11.6.0,4.5.2,2022-04-27T07:44:34Z
277009,Peter James, p@x.com,Valentina's MacBook Air,OS_MAC,11.6.0,4.5.0,2022-04-24T22:15:40Z
277009,Peter James, p@x.com,DESKTOP-VVOT9GB,OS_WINDOWS,10,4.7.106.794,2021-04-09T16:43:00Z
277009,Peter James, p@x.com,Valentina's MacBook Pro,OS_MAC,10.13.6,4.4.2,2021-02-12T21:23:40Z
631047,Peter Leé, p2@x.com,Peter's MacBook Pro,OS_MAC,11.6.0,4.5.2,None
631047,Peter Leé, p2@x.com,PeterLe0-PC,OS_WINDOWS,10,4.7.106.794,2022-04-26T12:02:06Z
```

Getting performance metrics via accountMetrics

Schema

```
# Account time-frame based metrics based on Near Realtime Stats
# Provides information similar to the connectivity pages for each site.
accountMetrics(
  # For the transition period of id been deprecated,
  # both accountID and id would be optional
  # if both are provided, accountID would take precedence.
  accountID: ID
  id: ID
  timeFrame: TimeFrame!
  # Aggregate results to all interfaces as a single all interface. Note that some
information (such as RTT will
  # become inconsistent. When results are aggregated, all interfaces will have
a single generic name `all`
  #
  # When querying for VPN users, this attribute is always evaluated to `true`
  groupInterfaces: Boolean
  # Aggregate results to all devices. `groupDevices` only becomes distinguishable
when there are
  # multiple devices in the site. When set to false in HA site, interface
names will be prefixed by
  # "Primary - " and "Secondary - " etc. It is important multiple devices
(such as HA pairs ) have consistent
  # interface names and functionality.
  #
  # When querying for VPN users, this attribute is always evaluated to `true`
  # __This attribute can only be set to `false` for a query on a single site__
  groupDevices: Boolean
): AccountMetrics

type AccountMetrics {
  id: ID
  from: Time
  to: Time
  # The size of a single time bucket in seconds
  granularity: Int
  # If no IDs are given, all sites will be retrieved (but not VPN users). For specific
IDs VPN Users metrics
  # can also be requested.
  sites(
    # For the transition period of id been deprecated,
    # both siteIDs and ids would be optional
    # if both are provided, siteIDs would take precedence.
    siteIDs: [ID!]
    ids: [String!]
  ): [SiteMetrics!]
  timeseries(
    labels: [TimeseriesMetricType!]
```

```

    # number of buckets, defaults to 10, max 1000
    buckets: Int
  ): [Timeseries!]
}

```

Use Cases

The accountMetrics query is probably the most complex but also one of the most valuable. As well as the most common use case of retrieving near-real-time performance metrics for sites and users, it can be used to go back in time to audit usage versus licence, to perform trend analysis for capacity planning and even to measure user productivity versus connectivity time.

Execution

Timeframe

The query takes a timeframe parameter which is a duration in ISO 8601 format. For more detailed information on this format, see https://en.wikipedia.org/wiki/ISO_8601#Durations. Effectively users have a choice between an absolute timeframe (“from this datetime to that datetime”) and a relative timeframe (“the last N minutes/hours/days/months”). Example timeframe parameters are:

Last five minutes:	last.PT5M
Last hour:	last.PT1H
Last 3 days:	last.P3D
Last 2 months:	last.P2M
UTC 1st June 2021 06:15:30 – 14:06:23:	utc.2021-06-01/{06:15:30--14:06:23}
UTC 1st May 2021 06:15:30 – 1st June 2021 14:06:23:	utc.2021-{05-01/06:15:30--06-01/14:06:23}

The maximum timeframe duration is 90 days, and the earliest the timeframe can start is 180 days ago from the current date.

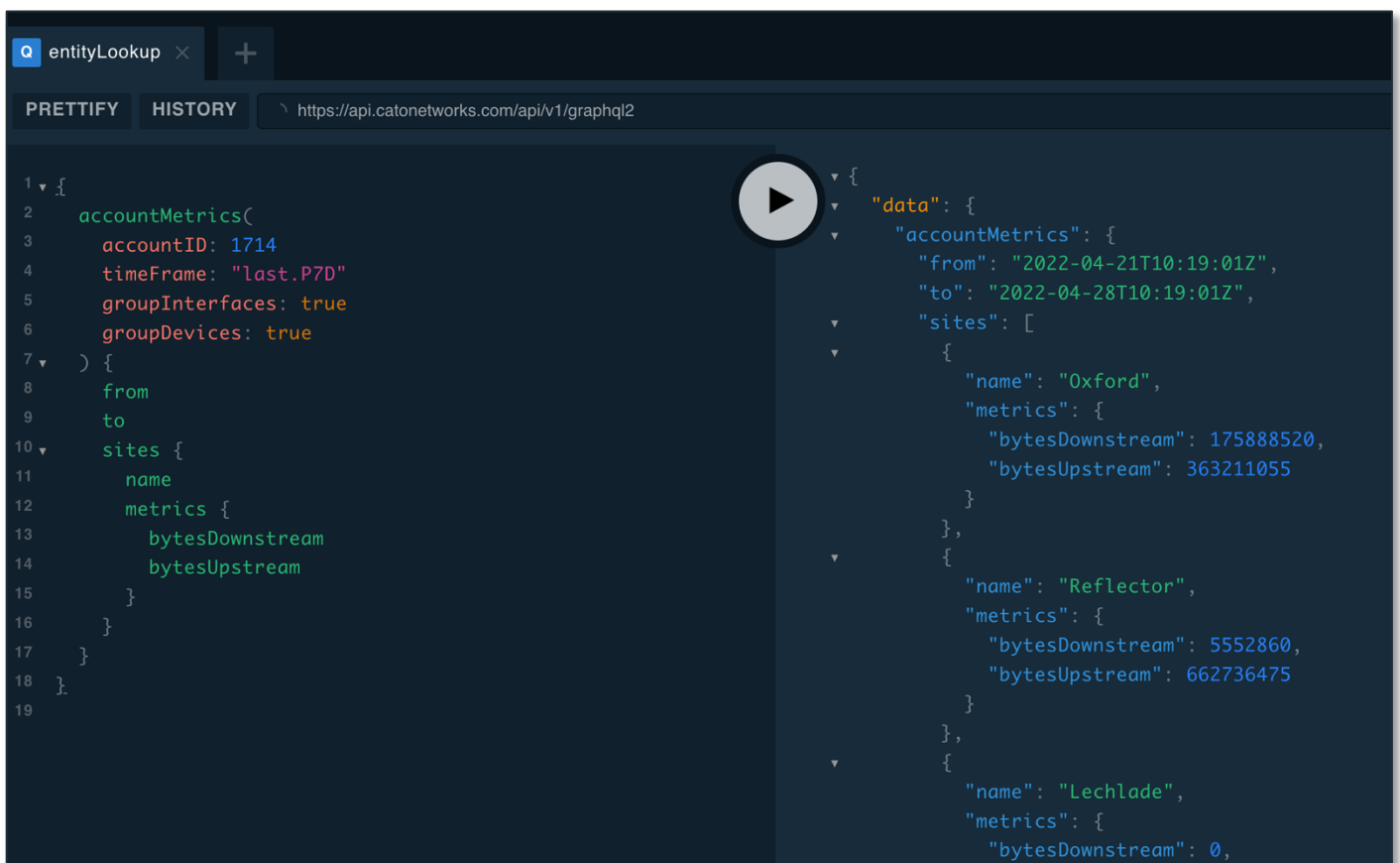
Metrics

The query can return two different types of metrics: a set of total metrics for each site/user for the specified timeframe, and a set of timeseries metrics which break the timeframe down into several buckets and provide subtotals within those buckets. If requesting data at the site level, the groupInterfaces and groupDevices parameters should both be set to true to aggregate data from all devices and interfaces.

Example: you have been asked to provide a report of total upstream and downstream data transferred for all sites for the last 7 days. The query (for account with ID=1714) would look like this:

```
{
  accountMetrics(accountID:1714 timeFrame:"last.P7D" groupInterfaces:true
groupDevices:true) {
    from
    to
    sites {
      name
      metrics {
        bytesDownstream
        bytesUpstream
      }
    }
  }
}
```

And the whole query/response looks like this in the API playground:



If you only wanted metrics for one particular site and you know (perhaps from entityLookup) that the site's ID is 9668, you can supply a list of site IDs as a parameter to the sites field. Querying just for site ID=9668 looks like this in the API Playground:


```

1 {
2   accountMetrics(
3     accountID: 1714
4     timeframe: "last.P7D"
5     groupInterfaces: true
6     groupDevices: true
7   ) {
8     from
9     to
10    sites (siteIDs:[9668]) {
11      name
12      metrics {
13        bytesDownstream
14        bytesUpstream
15      }
16    }
17  }
18 }
19

```

```

{
  "data": {
    "accountMetrics": {
      "from": "2022-04-21T10:21:47Z",
      "to": "2022-04-28T10:21:47Z",
      "sites": [
        {
          "name": "Oxford",
          "metrics": {
            "bytesDownstream": 190466000,
            "bytesUpstream": 365389445
          }
        }
      ]
    }
  }
}

```

The business likes the report so much that they ask you to also provide a per device, per-interface breakdown. So you set `groupInterfaces` and `groupDevices` to `false`, and add the query fields relating to interface metrics. While your Oxford site only has one device and interface, the data is the same as the aggregate total:

```

1 {
2   accountMetrics(
3     accountID: 1714
4     timeframe: "last.P7D"
5     groupInterfaces: false
6     groupDevices: false
7   ) {
8     from
9     to
10    sites(siteIDs: [9668]) {
11      name
12      metrics {
13        bytesDownstream
14        bytesUpstream
15      }
16      interfaces {
17        name
18        socketInfo {
19          isPrimary
20        }
21        metrics {
22          bytesDownstream
23          bytesUpstream
24        }
25      }
26    }
27  }
}

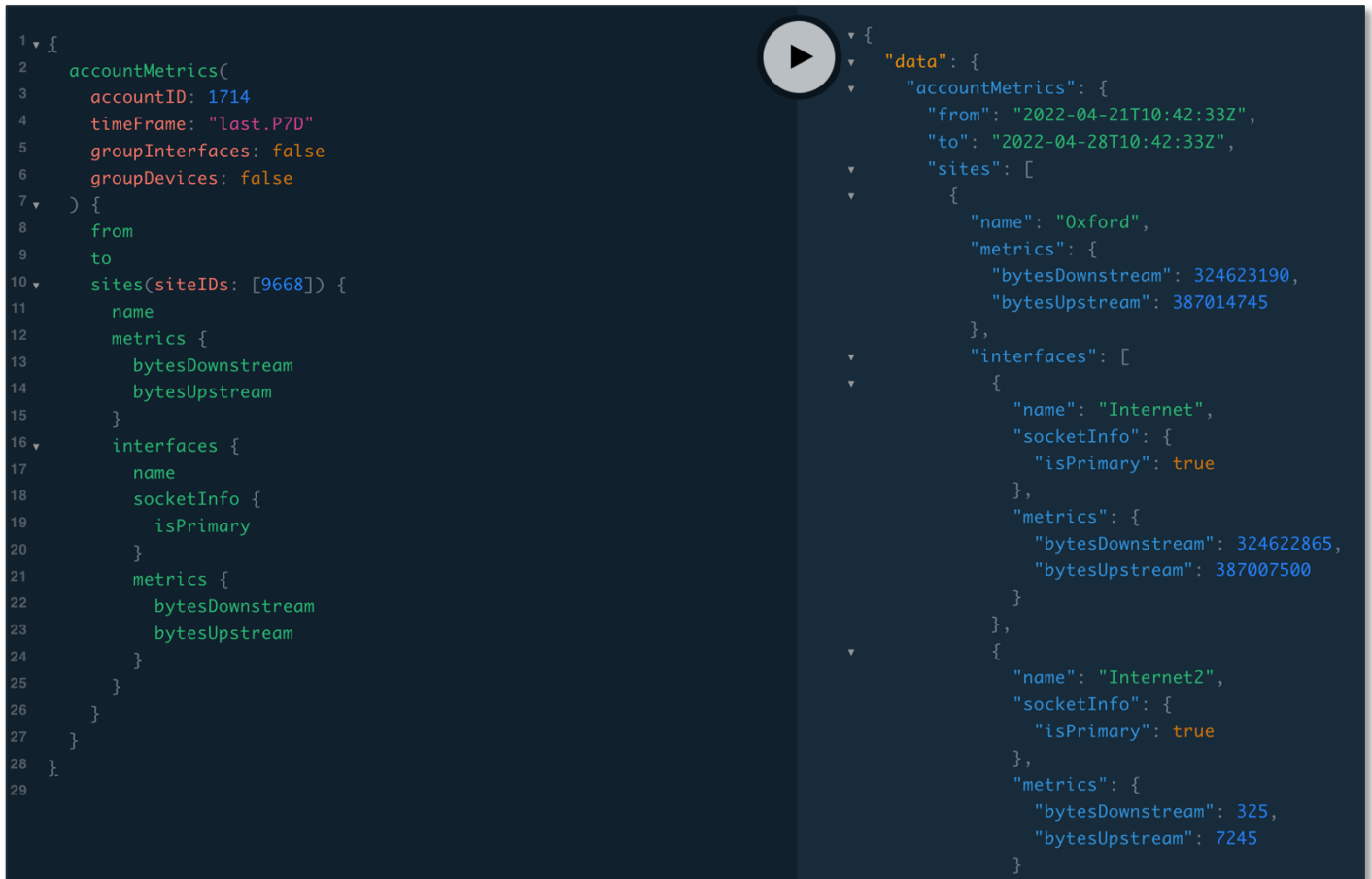
```

```

{
  "data": {
    "accountMetrics": {
      "from": "2022-04-21T10:37:26Z",
      "to": "2022-04-28T10:37:26Z",
      "sites": [
        {
          "name": "Oxford",
          "metrics": {
            "bytesDownstream": 284076205,
            "bytesUpstream": 381522385
          },
          "interfaces": [
            {
              "name": "Internet",
              "socketInfo": {
                "isPrimary": true
              },
              "metrics": {
                "bytesDownstream": 284076205,
                "bytesUpstream": 381522385
              }
            }
          ]
        }
      ]
    }
  }
}

```

Then the business decides to add a second Internet connection to Oxford, and now you see two interfaces which together add up to the site totals:



```
1 {
2   accountMetrics(
3     accountID: 1714
4     timeframe: "last.P7D"
5     groupInterfaces: false
6     groupDevices: false
7   ) {
8     from
9     to
10    sites(siteIDs: [9668]) {
11      name
12      metrics {
13        bytesDownstream
14        bytesUpstream
15      }
16      interfaces {
17        name
18        socketInfo {
19          isPrimary
20        }
21        metrics {
22          bytesDownstream
23          bytesUpstream
24        }
25      }
26    }
27  }
28 }
29
```

```
{
  "data": {
    "accountMetrics": {
      "from": "2022-04-21T10:42:33Z",
      "to": "2022-04-28T10:42:33Z",
      "sites": [
        {
          "name": "Oxford",
          "metrics": {
            "bytesDownstream": 324623190,
            "bytesUpstream": 387014745
          },
          "interfaces": [
            {
              "name": "Internet",
              "socketInfo": {
                "isPrimary": true
              },
              "metrics": {
                "bytesDownstream": 324622865,
                "bytesUpstream": 387007500
              }
            },
            {
              "name": "Internet2",
              "socketInfo": {
                "isPrimary": true
              },
              "metrics": {
                "bytesDownstream": 325,
                "bytesUpstream": 7245
              }
            }
          ]
        }
      ]
    }
  }
}
```

Timeseries Data

As well as the total metrics within the given timeframe, the query can return time series data which divides the timeframe duration into a specified number of buckets, returning a bucket timestamp and metric for that bucket. This is extremely useful for creating throughput graphs and measuring metrics down to very low durations – as low as 5 seconds.

Continuing the previous example, the business asks you to create some 24 throughput graphs for specific sites. Your Excel expertise means that as long as you can get 24x1 hour individual readings for any 24 hour period, you will be able to turn that into a beautiful bar chart in Excel. After studying the schema and experimenting in the API Playground, you get what looks like the right data:

Getting security and connectivity events via eventsFeed

Schema

```
eventsFeed(  
  # account IDs  
  accountIDs: [ID!]  
  filters: [EventFeedFieldFilterInput!]  
  # Marker to use to get results from  
  marker: String  
): EventsFeedData  
  
type EventsFeedData {  
  marker: String  
  fetchedCount: Int!  
  accounts: [EventsFeedAccountRecords]  
}  
  
type EventsFeedAccountRecords {  
  id: ID  
  errorString: String  
  records(fieldNames: [EventFieldName!]): [EventRecord!]  
}  
  
type EventRecord {  
  time: Time  
  # record fields in full object format  
  fields: [EventField!]  
  # fields in map format (see Map scalar)  
  fieldsMap: Map  
  # Simplified fields, as array of name value tuples, e.g: [ [ "name", "val" ], [ "name2", "val2" ] ... ]  
  flatFields: [[String!]]  
}
```

Use Cases

The eventsFeed query has one purpose in life, and only one – it is for downloading recent events from Cato for immediate consumption by a third party SIEM or analytics platform. It is not a general event query tool. For ad hoc event extraction use cases, Cato admins can download events via the Cato Management Application. The lack of granular filtering and short time frame for retrieval makes it unsuitable for general ad hoc queries. On the other hand, they allow for guaranteed non-duplicate delivery of events into a customer’s SIEM, making it the right tool for that job.

Execution

Event Lifecycle

The lifecycle for event data at Cato is:

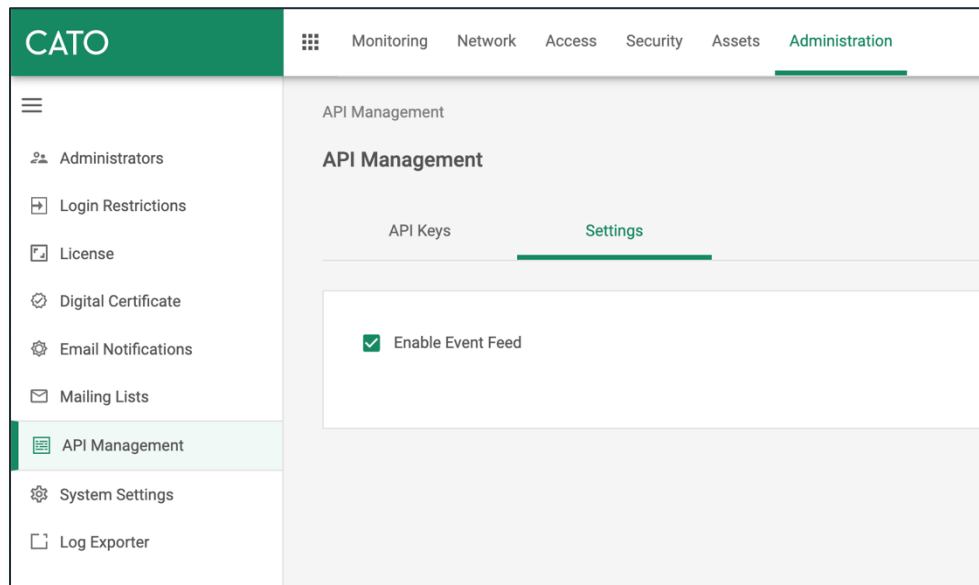
1. Events are generated in the Cato POPs by customer activity. Some activities will always generate an event (such as CMA logins) while some are generated according to policies which customers can tune (such as firewall events).
2. Each POP forwards events to Cato's centralized big data infrastructure.
3. The big data infrastructure processes events into the data warehouse for access via the Cato Management Application.
4. If the event belongs to an account which has an enabled Event Feed, the event is also added to the end of a Kafka queue for retrieval by the eventsFeed API query.
5. After the event has been in the Kafka queue for 7 days it is purged from the queue.
6. After six months the event is purged from the data warehouse.

This means that:

- Customers who wish to retrieve events via API must first enable the Event Feed in the Cato Management Application (CMA).
- Events will be visible in the CMA for six months, but only available for API retrieval for seven days.
- Retrieving an event from the queue does not remove it from the queue. It is perfectly safe for multiple different users to query the same queue.

Enabling Events Feed in the Cato Management Application

The first thing integrators should do is ensure that the Event Feed has been enabled in the Cato Management Application:



It is not enabled by default and cannot be retrospectively applied to events which are already in the data warehouse. If you are a third party integrator without direct access to your customer's CMA console, you will need to ask them to enable this for you.

Event Record Formats

According to the schema, there are two main output options for event records:

```
type EventRecord {
  time: Time
  # record fields in full object format
  fields: [EventField!]
  # fields in map format (see Map scalar)
  fieldsMap: Map
  # Simplified fields, as array of name value tuples, e.g: [ [ "name", "val" ], [
"name2", "val2" ] ... ]
  flatFields: [[String!]]
}
```

Both have advantages and disadvantages depending on which query tool or language you are using to transform the data before pushing it into your SIEM. The fieldsMap is a set of key:value pairs, a data structure known in Go as a “map” and in Python as a “dictionary”:

```

1 {
2   eventsFeed(accountIDs: [1714], marker: "", filters: []) {
3     marker
4     fetchedCount
5     accounts {
6       records {
7         time
8         fieldsMap
9       }
10    }
11  }
12 }
13

```

```

{
  "data": {
    "eventsFeed": {
      "marker":
"W3siVG9waWMiOiIxNzE0IiwUGFydG10aW9uIjowLjZmZzZXQ0jgxmzc1NzF9XQ==",
      "fetchedCount": 1000,
      "accounts": [
        {
          "records": [
            {
              "time": "2022-04-21T02:10:48Z",
              "fieldsMap": {
                "account_id": "1714",
                "action": "Block",
                "dest_ip": "10.64.4.10",
                "dest_is_site_or_vpn": "Site",
                "dest_port": "22",
                "event_count": "1",
                "event_sub_type": "IPS",
                "event_type": "Security",
                "internalId": "G0B0MwhgrM",
                "ip_protocol": "TCP",
                "mitre_attack_subtechniques": "",
                "mitre_attack_tactics": "",
                "mitre_attack_techniques": "",
                "os_type": "OS_UNKNOWN",
                "pop_name": "Dublin",

```

The flatFields object is a list of double-element lists (a data structure which a Python programmer would describe as a “list of tuples”) where the first element of each sub-list is the field name, and the second element is the field value:

```

1 {
2   eventsFeed(accountIDs: [1714], marker: "", filters: []) {
3     marker
4     fetchedCount
5     accounts {
6       records {
7         time
8         flatFields
9       }
10    }
11  }
12 }
13

```

```

{
  "data": {
    "eventsFeed": {
      "marker":
"W3siVG9waWMiOiIxNzE0IiwUGFydG10aW9uIjowLjZmZzZXQ0jgxmzc1NzF9XQ==",
      "fetchedCount": 1000,
      "accounts": [
        {
          "records": [
            {
              "time": "2022-04-21T02:10:48Z",
              "flatFields": [
                [
                  "src_country",
                  "China"
                ],
                [
                  "src_is_site_or_vpn",
                  "Site"
                ],
                [
                  "os_type",
                  "OS_UNKNOWN"
                ],
                [
                  "mitre_attack_techniques",
                  ""
                ],

```

My own experience has been that fieldsMap works well for almost all purposes including Python, where it converts easily to a dictionary. flatFields has been a handy alternative where fieldsMap does not readily convert into a suitable data structure.

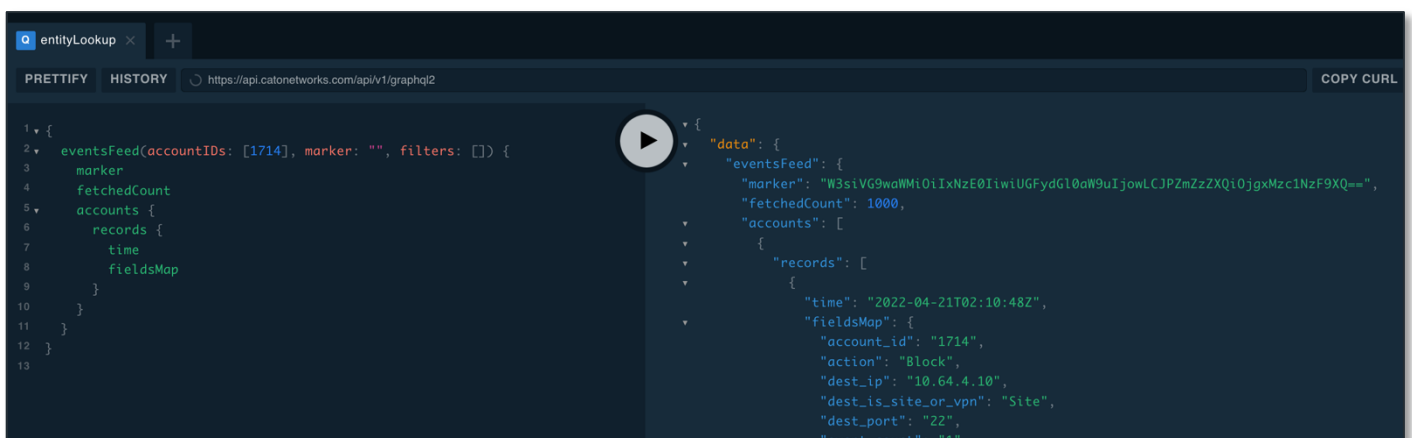
Each event includes a “time” field, which is the time the event was generated in the Unix epoch format. To convert this to a human-readable string in Excel, use a formula like this:

	A	B	C
1	1650507048441	21/04/2022 02:10	
2			

The same thing can be achieved in Python using the datetime module’s fromtimestamp method:

```
$ python3
Python 3.8.0 (v3.8.0:fa919fdf25, Oct 14 2019, 10:23:27)
[Clang 6.0 (clang-600.0.57)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> import datetime
>>> EventDate = 1650507048441
>>> HumanDate = datetime.datetime.fromtimestamp(EventDate/1000,datetime.timezone.utc)
>>> print(HumanDate.strftime("%Z %Y-%m-%d %H:%M:%S"))
UTC 2022-04-21 02:10:48
```

Alternatively, each event record includes a human-readable “time” field at the same level as the flatFields/fieldsMap objects:



A useful tip for integrators is to add this field to the event record before sending to SIEM. This ensures that there is always a human readable timestamp in every event. The Python sample script does this, calling the new field “event_timestamp”:


```
for event in resp["data"]["eventsFeed"]["accounts"][0]["records"]:
    event["fieldsMap"]["event_timestamp"] = event["time"]
```

Cato Event Fields, Types and Subtypes

Event records are a set of fields. Some of the fields are common across all event types, while others are specific to the type of event. It is usually obvious from the field name what the purpose of the field is. The Cato Networks Knowledge Base contains articles on eventsFeed including detailed listings of possible fields. If you are a third party integrator without access to the KB, please ask your customer to export them to PDF and send them to you.

The most important fields to be aware of are event_type and event_sub_type. They define a hierarchy of event types which provides some structure and are also the only fields by which events can be filtered at retrieval time.

Cato Networks does not publish a fixed data dictionary of event types, subtypes and fields. Cato is an agile vendor with an accelerated roadmap of new features being introduced every quarter. New features sometimes mean new subtypes and fields, and new types may be added in future. Integrators who are trying to retrieve all events for a particular type or types should ensure that their integration can automatically accommodate new subtypes, and all integrators should be alert to the possibility of additional fields suddenly appearing in known subtypes.

At the time of writing (April 2022) the event_types are Security, Connectivity, System, Routing, and Sockets Management. The most commonly encountered subtypes within each with a description and a rough estimate of how many events to expect:

event_type	event_sub_type	Description
Security	Internet Firewall	Outbound network flows (site or VPN user to an external destination). Very high volume, up to hundreds of millions of events per day.
	WAN Firewall	WAN network flows (site/VPN to site/VPN). Very high volume, up to hundreds of millions of events per day.
	IPS	Intrusion Prevention Events (can be outbound, inbound or WAN). Moderate volume, several thousand events per day.
	Anti Malware	Signature-based malware scans and detections. Moderate volume, up to tens of thousands of events per day.
	NG Anti Malware	Heuristic malware detections. Very low volume, several hundred events per month.
	RPF	Incoming connections from external sources to internal resources. High volume, up to millions of events every day.

	SDP Activity	Clientless VPN activity. Variable from very low to very high.
	TLS	Blocked TLS connections.
	Misclassification	End-user requests for secure web gateway domain name reclassifications.
Connectivity	Connected	Site or user connects to a POP (both successful and unsuccessful). Moderate volume – several thousand per day.
	Reconnected	Site or user reconnects to a POP. Moderate volume – several thousand per day.
	Disconnected	Site or user disconnects. Moderate volume – several thousand per day.
	Changed POP	Site or user moves to a different POP. Moderate volume – several thousand per day.
	Off-Cloud Recovery	Socket falls back to off-cloud transport for site-to-site traffic. Very low volume, several events per day.
	Last-Mile Quality	Link health notifications. Low volume, several dozen events per day.
	Cato Management Application	Administrator logins to CMA. Low volume, several dozen events per day.
	LAN Monitoring	LAN host unreachable notifications. Very low volume, several dozen events per month (under normal circumstances).
	Socket Fail-Over	Socket HA failover. Very low volume under normal circumstances (several events per month in a large account).
System	SCIM Provisioning	VPN user provisioning via SCIM success and failure notifications. several hundred events per month in a large account.
	QUOTA LIMIT	Events quota exceeded for a particular type. Moderate volume, several hundred events per month in a large account.
	Multiple Users Detected	Unable to map a LAN IP to one single user due to multiple user sessions emanating from a single IP. Moderate volume, several hundred events per month.
	DC Connectivity Failure	User Awareness is unable to connect to LAN Domain Controller. Very low volume, several events per month under normal circumstances.
Routing	BGP Routing	BGP route changes. High volume, several thousand events per month in a large account.

	BGP Session	BGP peering changes. High volume, several thousand events per month in a large account.
Sockets Management	Socket WebUI Access	Administrative connections to socket web UI. Low volume, several hundred events per month in a large account.
	Socket Upgrade	Socket upgrade success/failure notifications. Very low to low.

Examples of the most commonly encountered events are provided in Appendix A.

Pagination and Markers

Although each customer's Kafka queue only holds the last 7 days' worth of events, in a large customer this can be over 1 billion events. Even a fairly small customer can generate thousands of events every few minutes if they are logging all network flows. To prevent API users from making a call which tries to pull down millions of events and eventually times out, there is a built-in limit of 1000 events for each eventsFeed request. One of the fields returned by the call is a marker string, which is effectively the index of the last returned event. To successfully retrieve all events in the queue, the logic should look like this:

```
Read the last marker from local storage (or use an empty string to fetch from the
start of the queue)
```

```
While true:
```

```
    Send the query with the marker value
    Read the marker value from the response
    Update the marker
    Process any events received
    If we hit a stop condition:
        Break out of the loop
```

```
Write the final marker to local storage
```

This is the only way to successfully use eventsFeed to reliably retrieve data from Cato. Failure to preserve the marker between runs will result in the caller starting at the beginning of the queue each time, resulting in excessive network consumption, duplicate events and missed events.

There are several different strategies for deciding when to stop the current processing run, write the marker to disk and halt execution. Most Cato customers want to retrieve events on a near-real-time schedule, so they might schedule a script to run every 60 seconds. If their scheduler operates in a way which guarantees that only one copy of the script will ever be executing (i.e. no concurrent scheduling) then that user might configure their script to stop when a fetch retrieves zero events (this is how the official sample script works by default). If the customer is unable to prevent concurrent script execution, they might add a timer threshold which forces the script to stop after more than 55 seconds (the official sample script can do this with the -r parameter). A user who is testing the API and wants to only run a single execution might configure their script to

stop after the first fetch, but still write the marker to disk so they can start from where they left off on the next test run (the official sample script can do this with the -f parameter). Finally, in a busy account with thousands of events flooding in each minute, you can end up in a situation where the script starts off fetching batches of 1000 events until it catches up to the end of the queue. Then it can go into a very fast, tight loop where it is fetching small numbers each time and frequently hitting the rate limiter. In this situation, the user might tell the script to stop if a fetch retrieves less than say 100 events (again, the -f option lets the official sample do this).

Filtering

The eventsFeed script includes a “filters” parameter which can be used to filter events based on the values of the event_type and/or event_sub_type fields. These are often used by third-party integrators who are often interested in only a specific set of events. In most Cato accounts, more than 99% of events are of event_type=“Security”, and more than 99% of those security events are event_sub_type=“Internet Firewall” or event_sub_type=“WAN Firewall”. If you are a third party integrator who is primarily interested in higher-value security events like IPS or anti-malware, or you want to track administrator logins to Cato, or you are providing a NOC service which tracks site connectivity, then you can vastly reduce the number of events and volume of data you retrieve by using the right filter.

According to the schema, the parameter takes a list of type “EventFeedFieldFilterInput”:

```
filters: [EventFeedFieldFilterInput!]
```

Again we check the schema to find out what that is:

```
input EventFeedFieldFilterInput {
  fieldName: EventFeedFilterFieldName!
  # Use event_type and event_sub_type for events
  operator: EventFeedFilterOperator!
  values: [String!]
}
```

So it’s a list of objects, each one of which has a fieldname, an operator, and a list of values. So the structure of the parameter will look like this:

```
filters: [{fieldName:name1,operator:operator1,values:["value1","value2","valueN"]},{fieldName:name2,operator:operator2,values:["value3"]}]
```

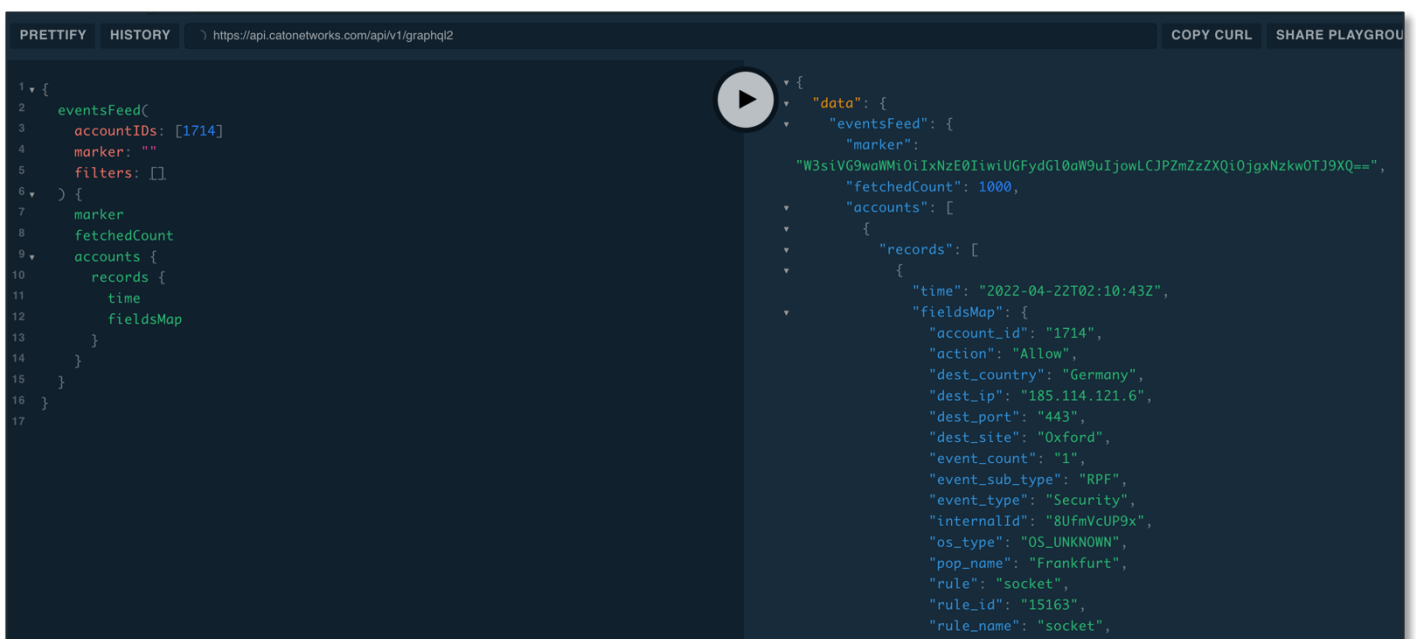
To determine the valid values for “EventFeedFilterFieldName” we check the schema and see there are two which are, as expected, event_type and event_sub_type:

```
enum EventFeedFilterFieldName {
  # Sub-type for Routing, Security, Connectivity, System or Sockets Management event
  event_sub_type
  # Routing, Security, Connectivity, System or Sockets Management event
  event_type
}
```

The operators are:

```
# Search operators on Event Feed
enum EventFeedFilterOperator {
  is
  is_not
  in
  not_in
}
```

The API Playground is a useful tool for testing complex input parameters like the filter. Let's say we're only interested in tracking administrator logins to the CMA. We start with an empty filter and see that when the marker is an empty string (start of the queue) we get 1000 events:



The screenshot shows the API Playground interface with a query on the left and its JSON response on the right. The query is:

```
1 {
2   eventsFeed(
3     accountIDs: [1714]
4     marker: ""
5     filters: []
6   ) {
7     marker
8     fetchedCount
9     accounts {
10      records {
11        time
12        fieldsMap
13      }
14    }
15  }
16 }
17
```

The response is:

```
{
  "data": {
    "eventsFeed": {
      "marker":
"W3siVG9waWMiOiIxNzE0IiwUGFydG10aW9uIjowLCJPZmZzZXQiOjg0NzkwOTJ9XQ==",
      "fetchedCount": 1000,
      "accounts": [
        {
          "records": [
            {
              "time": "2022-04-22T02:10:43Z",
              "fieldsMap": {
                "account_id": "1714",
                "action": "Allow",
                "dest_country": "Germany",
                "dest_ip": "185.114.121.6",
                "dest_port": "443",
                "dest_site": "Oxford",
                "event_count": "1",
                "event_sub_type": "RPF",
                "event_type": "Security",
                "internalId": "8UfmVcUP9x",
                "os_type": "OS_UNKNOWN",
                "pop_name": "Frankfurt",
                "rule": "socket",
                "rule_id": "15163",
                "rule_name": "socket",

```

Although it is possible that there are exactly 1000 events currently in the queue and we grabbed them all on that single fetch, this is extremely unlikely. When you see eventsFeed return 1000 events, it means that there are almost certainly more to be fetched and if you resubmit the query with the marker value received from the first fetch, you will get the second batch of 1000, and so on. A quick scroll through the returned events shows that they are all type=Security, so we decide to exclude those (and once again fetch from the start of the queue). From studying the schema we think a filter value of:

```
{ fieldName: event_type, operator: is_not, values: ["Security"]} might be worth a try.
```

So we add that and rerun the query:

```

PRETTIFY HISTORY https://api.catonetworks.com/api/v1/graphql2 COPY CURL SHARE PLAYGROU
1 {
2   eventsFeed(
3     accountIDs: [1714]
4     marker: ""
5     filters: [{ fieldName: event_type, operator: is_not, values: ["Security"] }]
6   ) {
7     marker
8     fetchedCount
9     accounts {
10      records {
11        time
12        fieldsMap
13      }
14    }
15  }
16 }
17
18 {
19   "data": {
20     "eventsFeed": {
21       "marker":
22         "W3siVG9waWMiOiIxNzE0IiwUGFydGl0aW9uIjowLCJpZmZzZXQiOjg0ODY3Njd9XQ==",
23       "fetchedCount": 48,
24       "accounts": [
25         {
26           "records": [
27             {
28               "time": "2022-04-22T08:00:30Z",
29               "fieldsMap": {
30                 "account_id": "1714",
31                 "action": "Succeeded",
32                 "authentication_type": "Password",
33                 "event_count": "1",
34                 "event_sub_type": "Cato Management Application",
35                 "event_type": "Connectivity",
36                 "internalId": "Qvil0CXQzn",
37                 "login_type": "Admin Login",
38                 "src_country": "United Kingdom of Great Britain and Northern Ireland",
39                 "src_ip": "185.69.144.37",
40                 "src_is_site_or_vpn": "VPN User",
41                 "src_site": "4472",
42                 "time": "1650614430222",
43                 "user_name": "Peter Lee",

```

The first thing we notice is that the event count for that first fetch is only 48. The filter has had a massive impact on the volume of data. The second thing we notice is that the first event is the exact subtype we're interested in. However, when we scroll further down we see that most of the rest of the events are VPN user connection and disconnection, which we're not interested in. We could just ignore them after we've retrieved them, but for all we know, the account might only be ramping up. Although 48 events isn't a lot now, it might be many times that in future. So we decide to add a filter to only include the CMA events:

```

PRETTIFY HISTORY https://api.catonetworks.com/api/v1/graphql2 COPY CURL SHARE PLAYGROU
1 {
2   eventsFeed(
3     accountIDs: [1714]
4     marker: ""
5     filters: [{ fieldName: event_type, operator: is_not, values: ["Security"] },
6             { fieldName: event_sub_type, operator: is, values: ["Cato Management Application"]} ]
7   ) {
8     marker
9     fetchedCount
10    accounts {
11      records {
12        time
13        fieldsMap
14      }
15    }
16  }
17 }
18
19 {
20   "data": {
21     "eventsFeed": {
22       "marker":
23         "W3siVG9waWMiOiIxNzE0IiwUGFydGl0aW9uIjowLCJpZmZzZXQiOjg0ODY3Njd9XQ==",
24       "fetchedCount": 8,
25       "accounts": [
26         {
27           "records": [
28             {
29               "time": "2022-04-22T08:00:30Z",
30               "fieldsMap": {
31                 "account_id": "1714",
32                 "action": "Succeeded",
33                 "authentication_type": "Password",
34                 "event_count": "1",
35                 "event_sub_type": "Cato Management Application",
36                 "event_type": "Connectivity",
37                 "internalId": "Qvil0CXQzn",
38                 "login_type": "Admin Login",
39                 "src_country": "United Kingdom of Great Britain and Northern Ireland",
40                 "src_ip": "185.69.144.37",
41                 "src_is_site_or_vpn": "VPN User",
42                 "src_site": "4472",

```

Which gets us to our desired effect. We also subsequently realise that we no longer need to exclude Security types if we are specifically including only the CMA subtype, so we remove the type=Security filter.

We can be fairly confident if we receive 1000 events in a fetch that there are more in the queue. But does the inverse hold true, i.e. if we receive less than 1000 events, can we be confident that there are no more waiting to be fetched?

Unfortunately, if we are using filters, this is not the case, especially if we are filtering for a type or subtype with a very small number of events within a much larger feed. This is because the `eventsFeed` call includes a timeout on the back end which will stop the query and return whatever data we've been able to gather if the query is taking too long to finish. For example, say that a customer queue has 100 million firewall events and three logins to the Cato Management Application – one near the start of the queue, and two at the end. If we call `eventsFeed` with an empty marker (start of the queue) and filter so we only get CMA logins, we will probably only get the first event. How then do we know that there are more to be retrieved? We don't, but what we do get back with that first event is a marker value which is the place in the queue where the query was when it timed out. So if we were to keep querying with the right logic, updating and resubmitting the marker each time, within a few iterations we would have caught up to the end of the queue and retrieved the other two events.

Using the Sample Script

There are really good reasons for using the official sample `eventsFeed.py` script, especially when you are starting the integration journey:

- It has been battle-tested across many different SIEMs and platforms.
- The only requirement is Python3, so the deployment is straightforward even on platforms like Windows.
- It provides multiple options for stopping the fetch loop.
- It can provide extensive log output to help you monitor and debug pagination.
- It includes command-line options for filtering by type and subtype and can print the queries out before sending them, which can help you debug your query filters.

Even if you have no intentions of ever using the sample script, it can be a useful reference to ways of calling `eventsFeed` which are known to work in other production environments. You don't have to be a Python guru to know what this code does:

```
# increment counter and check if we hit any limits for stopping
iteration += 1
if fetched_count < FETCH_THRESHOLD:
    log(f"Fetched count {fetched_count} less than threshold {FETCH_THRESHOLD}, stopping")
    break
elapsed = datetime.datetime.now() - start
if elapsed.total_seconds() > RUNTIME_LIMIT:
    log(f"Elapsed time {elapsed.total_seconds()} exceeds runtime limit {RUNTIME_LIMIT}, stopping")
    break
```


Example: sending events to Microsoft Sentinel

Overview

One of the most commonly requested platforms for integration is Microsoft Sentinel (formerly known as Azure Sentinel). The official sample script includes a Sentinel output option which uses the Sentinel API to send Cato events natively into the Sentinel Log Analytics Workspace for analysis by your Sentinel team. The steps we will follow are:

1. Validate our Sentinel environment.
2. Launch a Linux VM in Azure.
3. Transfer the sample script to the Linux VM.
4. Set up the sample script.
5. Test the sample script.
6. Check that we can see Cato events in Sentinel.
7. Schedule the sample script for continuous feed.

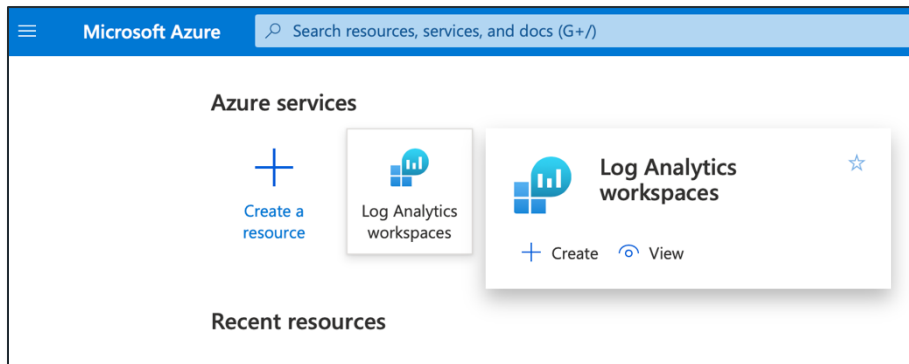
WARNING: Sentinel charges occur as data is ingested. Based on the worst-case PAYG pricing published by Microsoft at time-of-writing (April 2022) and the average size of Cato events, a rough estimate of the cost of ingesting Cato events is approximately \$2 per million Cato events. That doesn't sound like much, but some Cato customers generate over 1 billion events every seven days. Effective use of strategies such as filtering, fetch limits and data normalization (removing unwanted fields prior to ingestion into Sentinel) can help to avoid unexpectedly high Azure charges, especially in test environments.

Cato Networks is not responsible for any costs associated with the steps in this example.

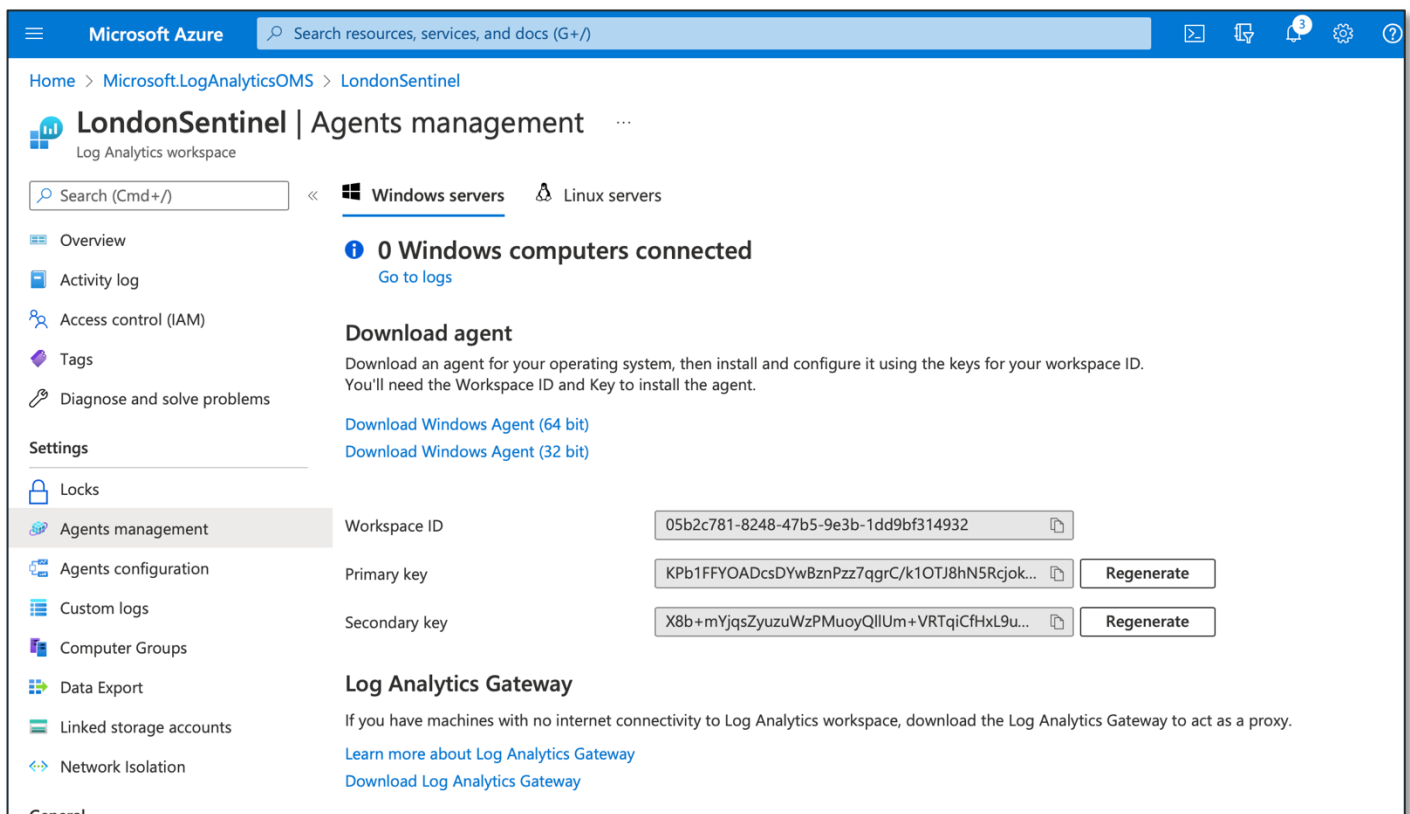
This is a high-level overview of one way to get Cato events into Microsoft Sentinel. It is not a Sentinel, Azure or Linux tutorial, and Cato cannot provide detailed technical support for these technologies. Customers who are unfamiliar with them should seek professional assistance.

1. Validate our Sentinel environment.

Most integrators will be working with an existing log analytics workspace, but we will start in an empty Azure account by creating a new one:



Once you have access to a workspace, on the Agents Management page, note the workspace ID and at least one of the keys. You will need these to be able to send events to Sentinel:



2. Launch a Linux VM in Azure.

The official sample script can be run from any host which has access to the Internet. During test and development, you will often run it from a laptop or user workstation but a production Sentinel integration needs a more reliable environment. Many customers already have existing Linux VMs in an Azure virtual network which are used for various logging operations, but for this example, we will create a new Linux (ubuntu 20.04) VM on a Standard DS1 v2 (1 vcpu, 3.5 GiB memory)

instance in an Azure virtual network, with a public IP of 51.142.202.227. Our Sentinel deployment is in UK-South, as is our VM, but they don't have to be.

Once you have the VM up and running:

1. SSH to the VM.
2. Create a new group called "cato":

```
peter@Sentinel-VM:~$ sudo addgroup cato
Adding group `cato' (GID 1001) ...
Done.
```

3. Create a directory called /opt/cato:

```
peter@Sentinel-VM:~$ sudo mkdir /opt/cato
```

4. Change the /opt/cato group to cato:

```
peter@Sentinel-VM:~$ sudo chgrp cato /opt/cato
```

5. Allow group to write to the directory:

```
peter@Sentinel-VM:~$ sudo chmod g+w /opt/cato
```

6. Create a new user called "cato_svc":

```
peter@Sentinel-VM:~$ sudo adduser cato_svc
Adding user `cato_svc' ...
Adding new group `cato_svc' (1002) ...
Adding new user `cato_svc' (1001) with group `cato_svc' ...
Creating home directory `/home/cato_svc' ...
Copying files from `/etc/skel' ...
New password:
Retype new password:
passwd: password updated successfully
Changing the user information for cato_svc
Enter the new value, or press ENTER for the default
  Full Name []:
  Room Number []:
  Work Phone []:
  Home Phone []:
  Other []:
Is the information correct? [Y/n] Y
```

7. Add the new user to the cato group:

```
peter@Sentinel-VM:~$ sudo adduser cato_svc cato
Adding user `cato_svc' to group `cato' ...
Adding user cato_svc to group cato
Done.
```

3. Transfer the sample script to the Linux VM.

If you are a third party integrator working with a Cato customer, please ask your customer to obtain a copy of the latest sample script from the Cato Networks Knowledge Base. If you are an independent SIEM vendor creating an official Cato integration, please contact api@catonetworks.com to arrange for a copy to be provided to you.

Once you have the script, the easiest way to copy it up to the VM is to SCP using the `catosvc` account you created in the previous step:

```
sh-3.2$ scp ./eventsFeed.py cato_svc@51.142.202.227:/opt/cato
cato_svc@51.142.202.227's password:
eventsFeed.py                               100% 15KB 193.5KB/s 00:00
```

4. Set up the sample script.

In this scenario, we are going to set the script up to be run every minute with a scheduler. The command line will contain several parameters, some of which have long values, so to make it easy to consistently execute the script, we will place the Python command with all parameters inside a shell script, and then call the shell script.

How you create the shell script will depend on how comfortable you are with editing files on the Linux command line. Worst case scenario – you can create and edit the file on your laptop, then scp it up to the VM after every change. This is how my initial file looks:

```
sh-3.2$ cat script.sh
python3 /opt/cato/eventsFeed.py \
-I 1714 \
-K D7912C93E1B14117EFDD51464FEC485F \
-c /opt/cato/config.txt \
-v \
-z 05b2c781-8248-47b5-9e3b-
1dd9bf314932:KPb1FFY0ADcsDYwBznPzz7qgrC/k10TJ8hN5RcjokKuD3xDs lbXPyQe1m/30bQ98jLHPX0DfV
1N2u4WxVkrV9g== \
-f 1001
```

The parameters are:

- I Cato account ID.
- K Cato API key.
- c Location for the config file which stores the marker between runs.
- v Enable log output to standard output.
- z The workspace id, followed by a :, followed by one of the agent keys noted in step 1.
- f Calling this parameter with 1001 forces the script to exit after the first fetch.

We will replace the `-f` parameter with a `-r` parameter once we are happy that the script works, but while we are testing we want the script to halt between fetches, so we start with `-f 1001`.

5. Test the sample script.

Make sure that you are logged on to the Linux VM as the `cato_svc` user. Change the directory to the `/opt/cato` directory, do a directory listing and ensure that the directory looks like this:

```
cato_svc@Sentinel-VM:/opt/cato$ ls -al
total 28
drwxrwxr-x 2 root    cato      4096 Apr 29 10:51 .
drwxr-xr-x 3 root    root      4096 Apr 29 10:11 ..
-rwxr-xr-x 1 cato_svc cato_svc 15522 Apr 29 10:29 eventsFeed.py
-rw-r--r-- 1 cato_svc cato_svc   298 Apr 29 10:51 script.sh
```

Let us now run the script and see what happens. If the script runs successfully, you should see something like this:

```
cato_svc@Sentinel-VM:/opt/cato$ sh script.sh
LOG 2022-04-29 10:55:44.268953> Using config file from -c parameter:
/opt/cato/config.txt
LOG 2022-04-29 10:55:44.269069> No marker value supplied, setting marker = ""
LOG 2022-04-29 10:55:44.269119> Config file does not exist, sticking with default
marker
LOG 2022-04-29 10:55:44.665144> iteration:1 fetched:1000 total_count:1000
marker:W3siVG9waWMiOiIxNzE0IiwUGFydGl0aW9uIjowLCJZmZzZXQiOjg0NzkwOTJ9XQ== 2022-04-
22T02:10:43Z 2022-04-22T02:46:55Z
LOG 2022-04-29 10:55:44.746240> Fetched count 1000 less than threshold 1001, stopping
LOG 2022-04-29 10:55:44.746355> OK 1000 events from 1 API calls in 0:00:00.477683
```

If you don't see something like that, if you see errors then you need to troubleshoot. It could be that you have the wrong Cato API key, or you omitted a step when setting up the service account under which we are running the script.

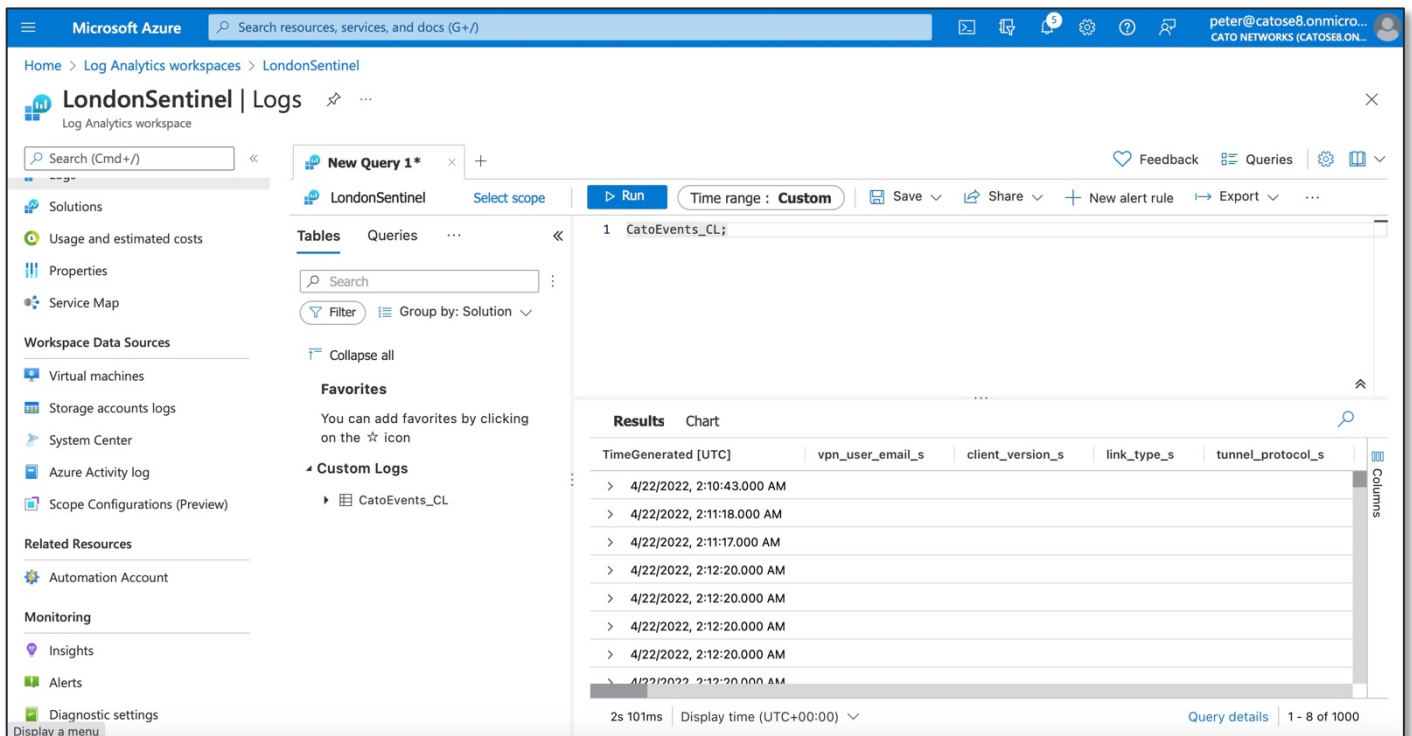
If you do another directory listing, you should now see a `config.txt` file which contains the marker:

```
cato_svc@Sentinel-VM:/opt/cato$ ls -al
total 32
drwxrwxr-x 2 root    cato      4096 Apr 29 10:55 .
drwxr-xr-x 3 root    root      4096 Apr 29 10:11 ..
-rw-rw-r-- 1 cato_svc cato_svc   68 Apr 29 10:55 config.txt
-rwxr-xr-x 1 cato_svc cato_svc 15522 Apr 29 10:29 eventsFeed.py
-rw-r--r-- 1 cato_svc cato_svc  300 Apr 29 10:55 script.sh
```

6. Check that we can see Cato events in Sentinel.

Once you believe the script to be working, go back to the Azure portal and have a look in the Logs window. If the push into Sentinel has succeeded you should see a new custom table called

“CatoEvents_CL”. If you run a query in that table with a time range far back enough in time to cover the earliest events in the queue (at least 8 days) then you should see something like this:



7. Schedule the sample script for continuous feed.

Once you are seeing Cato events in Sentinel from your test run, the final step is to schedule the script to run once each minute. To guarantee that we won't have multiple copies of the script running simultaneously, which could cause problems with the marker, we will replace the "-f 1001" parameter with "-r 55":

```
cato_svc@Sentinel-VM:/opt/cato$ cat script.sh
python3 /opt/cato/eventsFeed.py \
  -I 1714 \
  -K D7912C93E1B14117EFDD51464FEC485F \
  -c /opt/cato/config.txt \
  -v \
  -z 05b2c781-8248-47b5-9e3b-
1dd9bf314932:KPb1FFY0ADcsDYwBznPzz7qgrC/k10TJ8hN5RcjokKuD3xDs lbXPYqe1m/30bQ98jLHPX0DfV
1N2u4WxVkrV9g== \
  -r 55
```

Now if we run the script we should see it run multiple fetches until the timer kicks in and stops execution after 55 seconds have elapsed:

```
cato_svc@Sentinel-VM:/opt/cato$ sh script.sh
LOG 2022-04-29 12:06:56.948601> Using config file from -c parameter:
/opt/cato/config.txt
LOG 2022-04-29 12:06:56.948730> No marker value supplied, setting marker = ""
LOG 2022-04-29 12:06:56.948783> Found config file: /opt/cato/config.txt
LOG 2022-04-29 12:06:56.948882> Read marker from config_file:
W3siVG9waWMi0iIxNzE0IiwUGFydGluaW9uIjowLCJpZmZzZXQiOjgkNzk0Tj9XQ==
```

```
LOG 2022-04-29 12:06:57.361064> iteration:1 fetched:1000 total_count:1000
marker:W3siVG9waWmi0iIxNzE0IiwiUGFydGl0aW9uIjowLCJpZmZzZXQi0jgx0DAwOTJ9XQ== 2022-04-
22T02:46:54Z 2022-04-22T03:32:04Z
LOG 2022-04-29 12:06:57.879856> iteration:2 fetched:1000 total_count:2000
marker:W3siVG9waWmi0iIxNzE0IiwiUGFydGl0aW9uIjowLCJpZmZzZXQi0jgy0DEwOTJ9XQ== 2022-04-
22T03:32:04Z 2022-04-22T03:58:31Z
<snip>
LOG 2022-04-29 12:07:51.445828> iteration:117 fetched:1000 total_count:117000
marker:W3siVG9waWmi0iIxNzE0IiwiUGFydGl0aW9uIjowLCJpZmZzZXQi0jgy0TYwOTJ9XQ== 2022-04-
25T09:35:57Z 2022-04-25T10:01:31Z
LOG 2022-04-29 12:07:51.893965> iteration:118 fetched:1000 total_count:118000
marker:W3siVG9waWmi0iIxNzE0IiwiUGFydGl0aW9uIjowLCJpZmZzZXQi0jgy0TcwOTJ9XQ== 2022-04-
25T10:01:30Z 2022-04-25T10:30:15Z
LOG 2022-04-29 12:07:52.003234> Elapsed time 55.054903 exceeds runtime limit 55,
stopping
LOG 2022-04-29 12:07:52.003379> OK 118000 events from 118 API calls in 0:00:55.055059
```

If the script doesn't stop, use Control-C to stop it manually and then inspect your file to see where you've gone wrong. Once you have a script which runs for 55 seconds and then stops, the final step is to schedule it using something like cron:

```
cato_svc@Sentinel-VM:/opt/cato$ crontab -e
no crontab for cato_svc - using an empty one
```

Select an editor. To change later, run 'select-editor'.

1. /bin/nano <---- easiest
2. /usr/bin/vim.basic
3. /usr/bin/vim.tiny
4. /bin/ed

```
Choose 1-4 [1]: 2
```

Crontab configuration is outside of scope for this document, but essentially you want to end up with a crontab which looks like this:

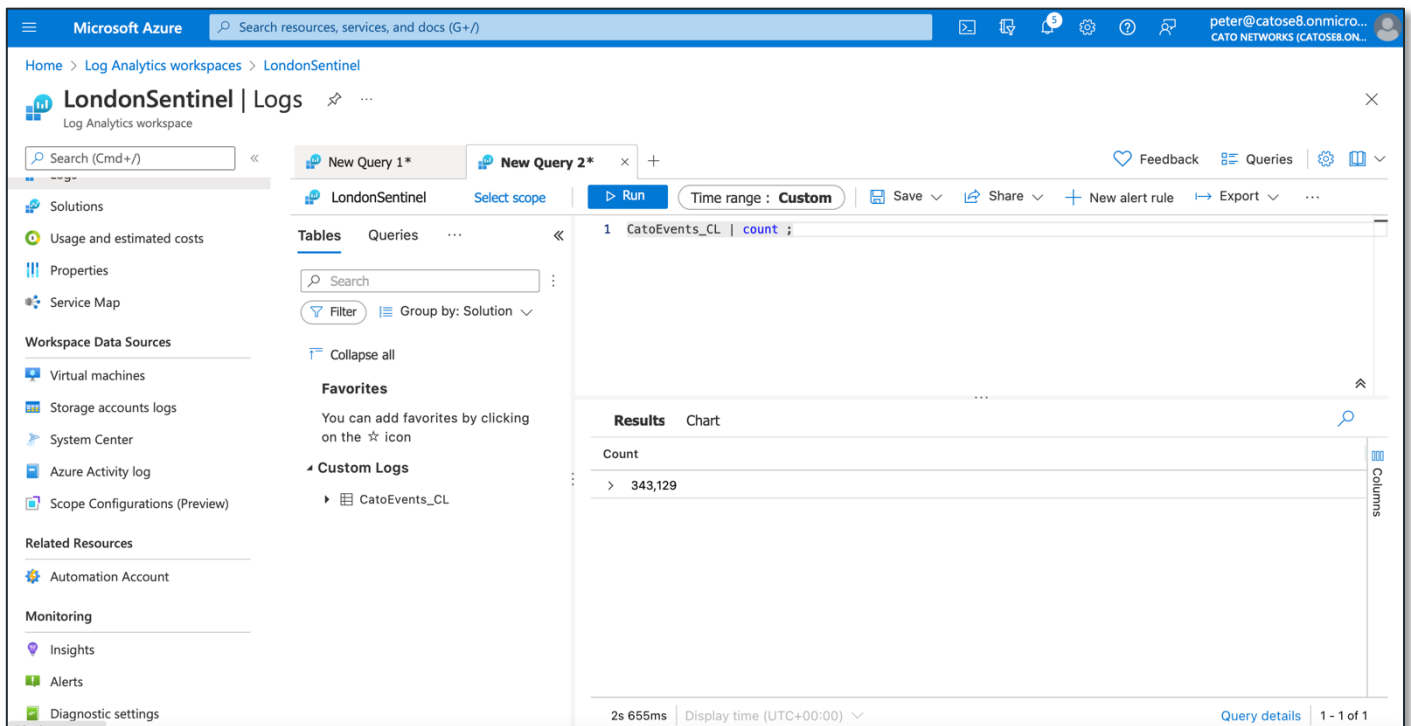
```
cato_svc@Sentinel-VM:/opt/cato$ crontab -l
# Edit this file to introduce tasks to be run by cron.
#
# Each task to run has to be defined through a single line
# indicating with different fields when the task will be run
# and what command to run for the task
#
# To define the time you can provide concrete values for
# minute (m), hour (h), day of month (dom), month (mon),
# and day of week (dow) or use '*' in these fields (for 'any').
#
# Notice that tasks will be started based on the cron's system
# daemon's notion of time and timezones.
#
# Output of the crontab jobs (including errors) is sent through
```

```

# email to the user the crontab file belongs to (unless redirected).
#
# For example, you can run a backup of all your user accounts
# at 5 a.m every week with:
# 0 5 * * 1 tar -zcf /var/backups/home.tgz /home/
#
# For more information see the manual pages of crontab(5) and cron(8)
#
# m h dom mon dow  command
* * * * * sh /opt/cato/script.sh

```

As soon as the scheduler is installed, you should see the event count in Sentinel climbing as the script works through the queue:



Appendix A – Sample events

These are samples of commonly encountered event types and subtypes. For additional samples, please check the CMA Events page or ask your customer to provide samples if you don't have access to CMA.

Security

Internet Firewall

```
{"ISP_name": "Vodafone Ltd", "account_id": "1714", "action": "Monitor", "application": "Technological apps", "categories": "Computers and Technology", "custom_categories": "Allowed Internet for Guests, Domain User Internet", "dest_country": "United States of America", "dest_ip": "44.240.37.33", "dest_port": "443", "domain_name": "push.services.mozilla.com", "event_count": "1", "event_sub_type": "Internet Firewall", "event_type": "Security", "internalId": "UK8P5Uy7ms", "ip_protocol": "TCP", "is_sanctioned_app": "false", "os_type": "OS_MAC", "os_version": "11.6.0", "pop_name": "Melbourne", "rule": "Track All", "rule_id": "5957", "rule_name": "Track All", "src_country": "United Kingdom of Great Britain and Northern Ireland", "src_ip": "10.41.169.183", "src_is_site_or_vpn": "VPN User", "src_isp_ip": "185.69.144.161", "src_site": "Peter James", "time": "1650741710842", "vpn_user_email": "peter@xxx.com", "event_timestamp": "2022-04-23T19:21:50Z"}
```

IPS

```
{"account_id": "1714", "action": "Block", "dest_ip": "10.64.4.10", "dest_is_site_or_vpn": "Site", "dest_port": "22", "event_count": "1", "event_sub_type": "IPS", "event_type": "Security", "internalId": "M9w5A3mkAa", "ip_protocol": "TCP", "mitre_attack_subtechniques": "", "mitre_attack_tactics": "", "mitre_attack_techniques": "", "os_type": "OS_UNKNOWN", "pop_name": "Dublin", "risk_level": "Medium", "rule": "3605", "rule_id": "3605", "signature_id": "feed_ips15_ssh", "src_country": "China", "src_ip": "61.177.173.13", "src_is_site_or_vpn": "Site", "src_port": "47046", "src_site": "Reflector", "threat_name": "IP reputation based signature – Network Scanner", "threat_reference": "https://support.catonetworks.com/hc/en-us/articles/360011568478", "threat_type": "Reputation", "time": "1650596005910", "traffic_direction": "INBOUND", "event_timestamp": "2022-04-22T02:53:25Z"}
```

Anti Malware

```
{"account_id": "1714", "action": "Block", "application": "Suspected apps", "dest_country": "Ireland", "dest_ip": "52.51.102.52", "dest_port": "443", "domain_name": "reflector.peterljames.org", "event_count": "1", "event_sub_type": "Anti Malware", "event_type": "Security", "file_hash": "275a021bbfb6489e54d471899f7db9d1663fc695ec2fe2a2c4538aabf651fd0f", "file_name": "eicar.txt", "file_size": "68", "internalId": "MeEqGJfDwP", "os_type": "OS_MAC", "os_version": "11.6.0", "pop_name": "London", "rule": "0", "rule_id": "0", "src_ip": "10.41.173.156", "src_is_site_or_vpn": "VPN User", "src_site": "Peter James", "threat_name": "EICAR-Test-File", "threat_verdict": "virus_found", "time": "1651045480533", "url": "https://reflec.xxx.com/eicar.txt", "vpn_user_email": "peter@xxx.com", "event_timestamp": "2022-04-27T07:44:40Z"}
```


NG Anti Malware

```
{"account_id": "1714", "action": "Block", "application": "Suspected apps",  
"dest_country": "Ireland", "dest_ip": "52.51.102.52", "dest_port": "443",  
"domain_name": "reflector.peterljames.org", "event_count": "1", "event_sub_type": "NG  
Anti Malware", "event_type": "Security", "file_hash":  
"70355dcf91019652e32eba67140a2708232a1fa786f90446d7984afe314f63f3", "file_name":  
"eicar.exe", "file_size": "68", "indicator": "EICAR-SENTINEL-ANTIVIRUS-TEST-FILE",  
"internalId": "QCzt6ht6GY", "os_type": "OS_MAC", "os_version": "11.6.0", "pop_name":  
"London", "rule": "0", "rule_id": "0", "src_ip": "10.41.173.156",  
"src_is_site_or_vpn": "VPN User", "src_site": "Peter James", "threat_name": "malware",  
"threat_verdict": "virus_found", "time": "1651045480623", "url":  
"https://reflec.xxx.com /eicar.exe", "vpn_user_email": "peter@xxx.com",  
"event_timestamp": "2022-04-27T07:44:40Z"}
```

RPF

```
{"account_id": "1714", "action": "Allow", "dest_country": "United Kingdom of Great  
Britain and Northern Ireland", "dest_ip": "85.255.16.36", "dest_port": "22",  
"dest_site": "Reflector", "event_count": "6", "event_sub_type": "RPF", "event_type":  
"Security", "internalId": "cAmaGkX3na", "os_type": "OS_UNKNOWN", "pop_name": "London",  
"rule": "RPF22", "rule_id": "15366", "rule_name": "RPF22", "src_country": "China",  
"src_ip": "61.177.173.13", "time": "1650618945981", "event_timestamp": "2022-04-  
22T09:15:45Z"}
```

Connectivity

Connected

```
{"ISP_name": "Vodafone Ltd", "account_id": "1714", "action": "Succeeded",  
"client_version": "4.5.2", "device_name": "Peter's MacBook Pro", "event_count": "1",  
"event_sub_type": "Connected", "event_type": "Connectivity", "internalId":  
"qV6DEyT6wP", "link_type": "Cato", "os_type": "OS_MAC", "os_version": "11.6.0",  
"pop_name": "London", "src_country": "United Kingdom of Great Britain and Northern  
Ireland", "src_ip": "10.41.6.171", "src_is_site_or_vpn": "VPN User", "src_isp_ip":  
"185.69.145.183", "src_site": "Peter James", "time": "1651172220000",  
"tunnel_protocol": "DTLS", "vpn_user_email": "peter@xxx.com", "event_timestamp":  
"2022-04-28T18:57:00Z"}
```

Cato Management Application

```
{"account_id": "1714", "action": "Succeeded", "authentication_type": "Password",  
"event_count": "1", "event_sub_type": "Cato Management Application", "event_type":  
"Connectivity", "internalId": "fN6RlumJ1s", "login_type": "Admin Login",  
"src_country": "United Kingdom of Great Britain and Northern Ireland", "src_ip":  
"185.69.144.176", "src_is_site_or_vpn": "VPN User", "src_site": "4472", "time":  
"1651158043764", "user_name": "Peter Lee", "vpn_user_email": "peter@xxx.com",  
"event_timestamp": "2022-04-28T15:00:43Z"}
```

Appendix B – eventsFeed.py

```
# eventsFeed.py
#
#
# Version: 20220412
# Author: Peter Lee, April 2022
#
# This script takes as input an API key and account ID, and returns events in JSON format
# from the event queue associated with that account ID. Requires events feed to be enabled
# in the Cato management console before events will be placed in the queue. Optional parameters
# include a marker value to initialise with and the path to a file in which to store the marker
# between runs (necessary to preserve place in the event queue). Also an event type and event
# subtype filter, for retrieving events only for certain types/subtypes.
#
# The eventsFeed API query supports multiple output formats – the script uses the fieldsMap format
# which displays nicely as a JSON key:value collection.
#
# The script provides the -n option for sending events as a stream to a TCP socket, and the -z option
# for sending events directly into Microsoft Sentinel.
#
# Usage: eventsFeed.py [options]
#
# Options:
# -h, --help          show this help message and exit
# -K API_KEY          API key
# -I ID               Account ID
# -P                 Prettify output
# -p                 Print event records
# -n STREAM_EVENTS   Send events over network to host:port TCP
# -z SENTINEL        Send events to Sentinel customerid:sharedkey
# -m MARKER           Initial marker value (default is "", which means start
#                    of the queue)
# -c CONFIG_FILE     Config file location (default ./config.txt)
# -t EVENT_TYPES     Comma-separated list of event types to filter on
# -s EVENT_SUB_TYPES Comma-separated list of event sub types to filter on
# -f fetch_limit     Stop execution if a fetch returns less than this number
#                    of events (default=1)
# -r RUNTIME_LIMIT   Stop execution if total runtime exceeds this many
#                    seconds (default=infinite)
# -v                 Print debug info
# -V                 Print detailed debug info
#
# Examples:
#
# To run the script with key=YOURAPIKEY for account ID 1714 for the first time, pulling all events
# and storing the marker in the default location (./config.txt) without displaying events:
# python3 eventsFeed.py -K YOURAPIKEY -I 1714 -m ""
#
# Running the script from the start with debug enabled so you can see the fetch logic in action:
# python3 eventsFeed.py -K YOURAPIKEY -I 1714 -m "" -v
#
# To show events, use the -p parameter:
# python3 eventsFeed.py -K YOURAPIKEY -I 1714 -p
#
# For more human readable events, use -pP
# python3 eventsFeed.py -K YOURAPIKEY -I 1714 -pP
#
# To print human readable events to screen and send raw events to TCP port 8000 on 192.168.1.1:
```

```

# python3 eventsFeed.py -K YOURAPIKEY -I 1714 -pP -n 192.168.1.1:8000
#
# To only see connectivity type events:
# python3 eventsFeed.py -K YOURAPIKEY -I 1714 -p -t Connectivity
#
# To only see NG Anti Malware and Anti Malware subtype events:
# python3 eventsFeed.py -K YOURAPIKEY -I 1714 -p -s "NG Anti Malware,Anti Malware"
#
# This script is supplied as a demonstration of how to access the Cato API with Python. It
# is not an official Cato release and is provided with no guarantees of support. Error handling
# is restricted to the bare minimum required for the script to work with the API, and may not be
# sufficient for production environments.
#
# All questions or feedback should be sent to api@catonetworks.com

```

```

import base64
import datetime
import hmac
import hashlib
import json
import os
import socket
import ssl
import sys
import time
import urllib.parse
import urllib.request
from optparse import OptionParser

```

```

#####
#####
#####
# Helper functions and globals

```

```

# log debug output
def log(text):
    if options.verbose or options.veryverbose:
        print(f"LOG {datetime.datetime.utcnow()}> {text}")

```

```

# log detailed debug output
def logd(text):
    if options.veryverbose:
        log(text)

```

```

# send GraphQL query string to API, return JSON
# if we hit a network error, retry ten times with a 2 second sleep
def send(query):
    global api_call_count
    retry_count = 0
    data = {'query':query}
    headers = { 'x-api-key': options.api_key, 'Content-Type': 'application/json' }
    no_verify = ssl._create_unverified_context()
    while True:
        if retry_count > 10:
            print("FATAL ERROR retry count exceeded")
            sys.exit(1)
        try:
            request = urllib.request.Request(url='https://api.catonetworks.com/api/v1/graphql2',
                data=json.dumps(data).encode("ascii"),headers=headers)

```

```

        response = urllib.request.urlopen(request, context=no_verify, timeout=30)
        api_call_count += 1
    except Exception as e:
        log(f"ERROR {retry_count}: {e}, sleeping 2 seconds then retrying")
        time.sleep(2)
        retry_count += 1
        continue
    result_data = response.read()
    if result_data[:48] == b'{"errors":[{"message":"rate limit for operation:':
        log("RATE LIMIT sleeping 5 seconds then retrying")
        time.sleep(5)
        continue
    break
result = json.loads(result_data.decode('utf-8','replace'))
if "errors" in result:
    log(f"API error: {result_data}")
    return False,result
return True,result

#####
#####
#####
# Azure Sentinel functions
# Taken from Microsoft sample here:
# https://docs.microsoft.com/en-gb/azure/azure-monitor/logs/data-collector-api
#
# Main change is to replace requests with urllib

# Build the API signature
def build_signature(customer_id, shared_key, date, content_length):
    x_headers = 'x-ms-date:' + date
    string_to_hash = f"POST\n{content_length}\napplication/json\n{x_headers}\n/api/logs"
    bytes_to_hash = bytes(string_to_hash, encoding="utf-8")
    decoded_key = base64.b64decode(shared_key)
    encoded_hash = base64.b64encode(hmac.new(decoded_key, bytes_to_hash,
digestmod=hashlib.sha256).digest()).decode()
    authorization = "SharedKey {}:{}".format(customer_id,encoded_hash)
    return authorization

# Build and send a request to the POST API
def post_data(customer_id, shared_key, body):

    rfc1123date = datetime.datetime.utcnow().strftime('%a, %d %b %Y %H:%M:%S GMT')
    content_length = len(body)
    signature = build_signature(customer_id, shared_key, rfc1123date, content_length)
    headers = {
        'content-type': 'application/json',
        'Authorization': signature,
        'Log-Type': 'CatoEvents',
        'Time-generated-field': 'event_timestamp',
        'x-ms-date': rfc1123date
    }
    no_verify = ssl._create_unverified_context()
    try:
        request = urllib.request.Request(url='https://' + customer_id +
'.ods.opinsights.azure.com/api/logs?api-version=2016-04-01',
        data=body,headers=headers)
        response = urllib.request.urlopen(request, context=no_verify)
    except urllib.error.URLError as e:
        print(f"Azure API ERROR:{e}")

```

```

    sys.exit(1)
except OSError as e:
    print(f"Azure API ERROR: {e}")
    sys.exit(1)
return response.code

#####
#####
#####
# start of the main program

api_call_count = 0
start = datetime.datetime.now()

# Process options
parser = OptionParser()
parser.add_option("-K", dest="api_key", help="API key")
parser.add_option("-I", dest="ID", help="Account ID")
parser.add_option("-P", dest="prettify", action="store_true", help="Prettify output")
parser.add_option("-p", dest="print_events", action="store_true", help="Print event records")
parser.add_option("-n", dest="stream_events", help="Send events over network to host:port TCP")
parser.add_option("-z", dest="sentinel", help="Send events to Sentinel customerid:sharedkey")
parser.add_option("-m", dest="marker", help="Initial marker value (default is \\", which means start of the queue)")
parser.add_option("-c", dest="config_file", help="Config file location (default ./config.txt)")
parser.add_option("-t", dest="event_types", help="Comma-separated list of event types to filter on")
parser.add_option("-s", dest="event_sub_types", help="Comma-separated list of event sub types to filter on")
parser.add_option("-f", dest="fetch_limit", help="Stop execution if a fetch returns less than this number of events (default=1)")
parser.add_option("-r", dest="runtime_limit", help="Stop execution if total runtime exceeds this many seconds (default=infinite)")
parser.add_option("-v", dest="verbose", action="store_true", help="Print debug info")
parser.add_option("-V", dest="veryverbose", action="store_true", help="Print detailed debug info")
(options, args) = parser.parse_args()
if options.api_key is None or options.ID is None:
    parser.print_help()
    sys.exit(1)

# either use the default marker or load from config file
config_file = "./config.txt"
marker = ""
if options.config_file is None:
    log(f"No config file specified, using default: {config_file}")
else:
    config_file = options.config_file
    log(f"Using config file from -c parameter: {config_file}")
if options.marker is None:
    log("No marker value supplied, setting marker = \\")
    # does the config file exist, if so load the marker value
    if os.path.isfile(config_file):
        log(f"Found config file: {config_file}")
        with open(config_file,"r") as File:
            try:
                marker = File.readlines()[0].strip()
            except IndexError as E:
                log(str(E))
                log(f"Couldn't read marker from config file, leaving marker as {marker}")
        log(f"Read marker from config_file: {marker}")
    else:

```

```

        log("Config file does not exist, sticking with default marker")
else:
    marker = options.marker
    log(f"Using marker value from -m parameter: {marker}")

# process event_type filters
if options.event_types is not None:
    log(f"Event type filter parameter: {options.event_types}")
    event_type_strings = options.event_types.split(',')
    log("Event type strings:" + str(event_type_strings).replace('\ ', ''))
    event_filter_string =
"{fieldName:event_type,operator:in,values:"+str(event_type_strings).replace('\ ', '')+"}"
    log(f"Event filter string: {event_filter_string}")
else:
    event_filter_string = ""

# process event_sub_type filters
if options.event_sub_types is not None:
    log(f"Event sub type filter parameter: {options.event_sub_types}")
    event_subtype_strings = options.event_sub_types.split(',')
    log("Event sub type strings:" + str(event_subtype_strings).replace('\ ', ''))
    event_subfilter_string =
"{fieldName:event_sub_type,operator:in,values:"+str(event_subtype_strings).replace('\ ', '')+"}"
    log(f"Event sub filter string: {event_subfilter_string}")
else:
    event_subfilter_string = ""

# process network options
if options.stream_events is not None:
    network_elements = options.stream_events.split(":")
    if len(network_elements) != 2:
        print("Error: -n value must be in the form of host:port")
        parser.print_help()
        sys.exit(1)

# process sentinel options
if options.sentinel is not None:
    sentinel_elements = options.sentinel.split(":")
    if len(sentinel_elements) != 2:
        print("Error: -z value must be in the form of customerid:sharedkey")
        parser.print_help()
        sys.exit(1)

# fetch count
if options.fetch_limit is None:
    FETCH_THRESHOLD = 1
else:
    FETCH_THRESHOLD = int(options.fetch_limit)

# runtime threshold
if options.runtime_limit is None:
    RUNTIME_LIMIT = sys.maxsize
else:
    RUNTIME_LIMIT = int(options.runtime_limit)

# API call loop
iteration = 1
total_count = 0
while True:
    query = ''

```

```

{
  eventsFeed(accountIDs:['' + options.ID + '']
    marker:'' + marker + '')
    filters:['' + event_filter_string + "," + event_subfilter_string + ''])
{
  marker
  fetchedCount
  accounts {
    id
    records {
      time
      fieldsMap
    }
  }
}
}
}'''

logd(query)
success,resp = send(query)
if not success:
  print(resp)
  sys.exit(1)
logd(resp)
marker = resp["data"]["eventsFeed"]["marker"]
fetched_count = int(resp["data"]["eventsFeed"]["fetchedCount"])
total_count += fetched_count
line = f"iteration:{iteration} fetched:{fetched_count} total_count:{total_count} marker:{marker}"
if len(resp["data"]["eventsFeed"]["accounts"][0]["records"]) > 0:
  line += " "+resp["data"]["eventsFeed"]["accounts"][0]["records"][0]["time"]
  line += " "+resp["data"]["eventsFeed"]["accounts"][0]["records"][-1]["time"]
log(line)

# print output
if options.print_events:
  for event in resp["data"]["eventsFeed"]["accounts"][0]["records"]:
    event["fieldsMap"]["event_timestamp"] = event["time"]
    if options.prettify:
      print(json.dumps(event["fieldsMap"],indent=2, ensure_ascii=False))
    else:
      print(json.dumps(event["fieldsMap"], ensure_ascii=False))

# network stream
if options.stream_events is not None:
  logd(f"Sending events to {network_elements[0]}:{network_elements[1]}")
  with socket.socket(socket.AF_INET, socket.SOCK_STREAM) as s:
    s.connect((network_elements[0], int(network_elements[1])))
    for event in resp["data"]["eventsFeed"]["accounts"][0]["records"]:
      event["fieldsMap"]["event_timestamp"] = event["time"]
      s.sendall(json.dumps(event["fieldsMap"]).encode("utf-8"))

# send to Microsoft Sentinel
if options.sentinel is not None:
  logd(f"Sending events to Azure workspace ID {sentinel_elements[0]}")
  body = []
  for event in resp["data"]["eventsFeed"]["accounts"][0]["records"]:
    event["fieldsMap"]["event_timestamp"] = event["time"]
    body.append(event["fieldsMap"])
  response_status =
post_data(sentinel_elements[0],sentinel_elements[1],json.dumps(body).encode('ascii'))
  if response_status < 200 or response_status > 299:
    print(f"Send to Azure returned {response_status}, exiting")

```

```
        sys.exit(1)
    logd(f"Send to Azure response code:{response_status}")

# write marker back out
logd("Writing marker to " + config_file)
with open(config_file,"w") as File:
    File.write(marker)

# increment counter and check if we hit any limits for stopping
iteration += 1
if fetched_count < FETCH_THRESHOLD:
    log(f"Fetched count {fetched_count} less than threshold {FETCH_THRESHOLD}, stopping")
    break
elapsed = datetime.datetime.now() - start
if elapsed.total_seconds() > RUNTIME_LIMIT:
    log(f"Elapsed time {elapsed.total_seconds()} exceeds runtime limit {RUNTIME_LIMIT}, stopping")
    break

end = datetime.datetime.now()
log(f"OK {total_count} events from {api_call_count} API calls in {end-start}")
```