13.2 Front-end API Guide

Front-end API Guide

Contents

1 Introduction

Some users will want to automate interaction with the Insight front-end, which uses a business intelligence application called Metabase. The most common interaction is to retrieve data, for example to export all rows of one or more tables, or the results of a given SQL statement. Other interactions may include tasks like creating similar metrics (aggregations) or segments (filters) across a range of tables.

This guide is aimed at describing how to interact with the Metabase API, highlighting the relevant endpoints, with reference to common programming tools.

The intended audience is statisitical or generalist programmers.

The Metabase API is not currently guaranteed by OpenClinica to be stable, fully validated, or have full backwards compatibility. In the future we plan to incorporate functional testing and change management of the API into our validation & release process but have not done so yet. We've found it to be reliable and useful thus far, but ensure you perform testing appropriate for your intended use case(s).

Metabase Documentation

The Metabase API lists the available endpoints, accepted HTTP verbs, parameters, and purpose of each endpoint: https://github.com/metabase/metabase/blob/master/docs/api-documentation.md

As a technical reference, the API docs do not cover the how-to's of using the API. In some cases it does not fully describe parameters, for example when the required value is a JSON object that may itself contain arrays or other objects.

This guide aims to fill some of these gaps, but in general the recommended strategy is to observe how the browser client app interacts with the API. In other words, since the Metabase UI that we interact with in the browser is essentially a ReactJS client app, we can learn how to perform an interaction by observing the client app uses the API to interact with the Metabase backend app.

Other Resources

On GitHub there are a few open source Metabase API client libraries, as listed below. These have not been vetted /endorsed for use with Insight, and have various levels of endpoint and test coverage, but may serve as either useful tools or examples:

- C# https://github.com/elevate/elevate.metabase.tools
- Clojure https://github.com/OpenCHS/metabase-migrator
- Golang https://github.com/kucuny/go-metabase
- JavaScript https://github.com/callawaywilson/metabasecloner
- PHP https://github.com/Germanaz0/php-metabase-sdk
- Python https://github.com/STUnitas/metabase-py
- Python https://github.com/mertsalik/metabasepy
- Ruby https://github.com/shimoju/metabase-ruby

Statistical programmers could either use a generalist language (+/- the above libraries) to perform the export then load the data into their stats software, or utilise the relevant capabilities in their stats software for interacting with HTTP APIs. For example:

- SAS has a PROC HTTP for JSON data: https://blogs.sas.com/content/sasdummy/2016/12/02/json-libname-engine-sas/
- Stata can call external programs using `shell` for example to invoke http tools like `curl`, `wget`, or `powershell` to actually perform the API calls, and then import the result CSV data with `insheet`, Excel data with `import excel`, or parse JSON data with ssc modules like `jsonio`.
- R has many packages on CRAN for working with HTTP APIs, such as `httr` and `RCurl`.

For integrating Metabase dashboards reports with other applications, there are also example embedding apps for JavaScript (Node), Python (Django), and Ruby (Rails) at: https://github.com/metabase/embedding-reference-apps

2 Authentication

The first interaction will always be authentication. The endpoint is POST `/api/session`, which accepts a username (email) and password. This email must correspond to an active Metabase user. The response will be a JSON dict with an `id` key, e.g. `{"id": "abcd"}` where the value is the session key.

This session key should then be included in the HTTP header of all subsequent requests, using the key `X-Metabase-Session`, e.g. `X-Metabase-Session: abcd`. An alternative approach is to save it as a session cookie value, with the the key `metabase.SESSION_ID`.

In preparing scripts, be sure to log in once then re-use the session key, ideally retaining it until it has expired. Metabase has non-configurable login throttling, such that even valid login requests can be rejected if they are sent too often.

To check if a session key is still valid, you can request any endpoint and check for HTTP 403, but a good generic one to use is GET `/api/user/current` and check for HTTP 200.

3 Retrieving Data

To retrieve data, the easiest endpoint to use is called `dataset`, which allows exporting ad-hoc query results as CSV, JSON, or XLSX. The other option is the `query` sub-path of `card`, which allows exporting results of saved questions.

The first task is to find the relevant Metabase ID for the database being queried, from GET `/api/database`, and filter the response JSON by `name` to find the integer `id`.

To use the `dataset` endpoint, send a POST to a sub-path of `/api/dataset`. The sub-path specifies the result format. For example to get results as CSV, send to `/api/dataset/csv`, for JSON `/api/dataset/json`, for XLSX `/api/dataset/xlsx`. The request parameters differ based on whether you intend to retrieve a table based on the Metabase ID, or if you intend to retrieve results from an SQL statement. In both cases, the request body should be of the type `x-www-form-urlencoded`, and the request body data should include a key `query` with a value as described below.

To query for table data by Metabase ID, first get the list of tables from GET `/api/tables`, then filter the response JSON by `schema` and table `name` to find the table `id`. Next, send a POST to `/api/dataset/<format>`. The request body value for `query` should be of the format `{"database":2,"type":"query","query": {"source_table":1234},"parameters":[]}`, where the `2` is the database ID found earlier, and `1234` is the table ID.

To query for SQL statement results, send a POST to `/api/dataset/<format>`. The request body value for `query` should be of the format `{"database":2,"type":"native","native":{"query":"SELECT * FROM s_mystudy.participant","template_tags":{}},"parameters":[]}`, where `2` is the database ID found earlier, and the inner `"query"` value is the SQL statement to execute; in this case it is `SELECT * FROM s_mystudy.participant`.

The keys for `"parameters"` and `"template_tags"` relate to the use of variables and filters, which are used in reports created in Metabase. However in this context the simplest approach would be to vary the provided SQL statement to include the appropriate parameters.

3.1 Example R Syntax

```
# Set as needed for a particular query.
metabase_url <- "customer-name.insight.openclinica.io"
session_id <- "a120c64b-bf33-41f3-ac9f-6121ac7f8c79"
database_id <- 2
sql <- "SELECT a as col1, a + 1 as col2 FROM generate_series(1, 10) as t(a)"

# Generic steps. Requires "httr" and "jsonlite" packages.
dataset_url <- paste0(metabase_url, "/api/dataset/json")
query <- list(
  native=list(query=sql, "template-tags"=c()),
  type="native",
  database=database_id,
  parameters=list()
)
response <- httr::POST(
  url=dataset_url,
```

```
  httr::add_headers(c("X-Metabase-Session"=session_id)),
  body=list(query=jsonlite::toJSON(query, auto_unbox=TRUE)),
  encode="form"
)

# Contains observations and columns as per input SQL query.
query_data <- jsonlite::fromJSON(httr::content(response, as="text"))
```

4 Getting All Data

As mentioned in the SQL Guide, the Postgres `pg_catalog` allows end users to retrieve information about the database structure. This can be useful in scripting data retrieval, for example to get all data from all tables at once, in combination with the above techniques for data retrieval.

The following catalog query returns a list of all non-system tables by their schema name. To get a list of schema names to filter the below query further, use `SELECT nspname FROM pg_catalog.pg_namespace`.

```
SELECT
  nsp.nspname AS schema_name,
  cls.relname AS table_name
FROM pg_catalog.pg_class AS cls
INNER JOIN pg_catalog.pg_namespace AS nsp
  ON nsp.oid = cls.relnamespace
WHERE cls.relkind = 'r'
  AND nsp.nspname NOT IN ('public', 'pg_catalog', 'information_schema')
  AND nsp.nspname NOT LIKE 'pg_temp%'
  AND nsp.nspname NOT LIKE 'pg_toast%'
```

Sending the above query to `/api/dataset/json` then provides the names of all the schemas and tables, which can in turn be used to generate the relevant SQL statements for a series of further calls. For example, we may receive an array of JSON objects like:

```
[
  {"schema_name":"s_study1","table_name":"crf"},
  {"schema_name":"s_study1", table_name":"event_crf"},
  {"schema_name":"s_study1","table_name":"item_data}
]
```

Which can then be processed into a series of request parameters:

```
{"database":2,"type":"native","native":{"query":"SELECT * FROM
s_study1.crf","template_tags":{}},"parameters":[]}
{"database":2,"type":"native","native":{"query":"SELECT * FROM
s_study1.event_crf","template_tags":{}},"parameters":[]}
{"database":2,"type":"native","native":{"query":"SELECT * FROM
s_study1.item_data","template_tags":{}},"parameters":[]}
```

Which in turn can be send to `/api/dataset/<format>` to fetch all table data. The results could then be saved for future reference, loaded into some other system, or analysed.

Bulk Export Consistency

Something to be aware of is how the Insight data refresh process and the above batch export job may affect data consistency among the exported results files.

Each API call would ultimately be executed in it's own "read committed" database transaction, while the Insight data refresh is in one "serializable" transaction so that users see a consistent snapshot. However, if some API calls execute before a refresh starts, and some after it finishes, the retrieved copy of the latter tables may refer to data not present in the retrieved copy of the former tables.

For example, if the `event_crf` table is exported before a refresh commits and the `item_data` table after, there may be some newly added `item_data` rows referring to `event_crf` rows that weren't present at the time the `event_crf` table was exported.

A strategy to avoid this issue is to be aware of the schema's refresh frequency, either by querying the schema's `refresh_log` table, or asking Customer Support, and scheduling / running batch exports such that they complete before a refresh completes (or begin immediately after).

If this is not practicable, other options include a custom-developed solution, or using a direct back-end database connection. In these scenarios, it's then possible to open a single database transaction to retrieve all data, i.e. using "repeatable read" or "serializable" transaction isolation to ensure that all SELECT statements se the same atomic version of the data.