GraphQL API + R Integration

Technical Brief

Last Updated: 05/04/2021

Verified using R 3.6.3 and ghql 0.1.1.93

Table of Contents

Introduction

To demonstrate the process for accessing data from the Pluralsight GraphQL API with R, we'll take a look at pulling Pluralsight's skill assessment catalog into a dataframe in R.

For documentation related to querying Pluralsight's GraphQL API, visit the Pluralsight Developer Portal at developer.pluralsight.com. For detailed information on the queries, mutations, and data entities available, please see the Schema Documentation on the Dev Portal.

Getting started

Before you can start using the API, you'll first need a Pluralsight GraphQL API token.

Once you have one, a best practice is to pass it into R via an environment variable as an entry in your user's `.Renviron` file. (somewhere like `~/.Renviron`). If this file doesn't exist, you can create it and then add the entry.

`~/.Renviron`

```
# add a new environment variable entry
PLURALSIGHT_GRAPHQL_TOKEN="<your_token>"
```

Save the file and restart your R session and you should have access to your token in R.

If you don't already have them installed, start by installing the R packages we'll be using. We'll use the ghql package as our GraphQL client and a few other packages to make things a little easier on ourselves.

```
> install.packages(c("ghql", "jsonlite", "dplyr", "readr"))
```

In our R script, we'll start by loading the packages we'll use:

```
library(ghql)
library(jsonlite)
library(dplyr)
```

Set up connection and initial query object

Next, let's set up the pieces we'll need to establish a connection and start querying data.

```
graphql_api_url <- "https://paas-api.pluralsight.com/graphql"

token <- Sys.getenv("PLURALSIGHT_GRAPHQL_TOKEN")

con <- GraphqlClient$new(
  url = graphql_api_url,
  headers = list(Authorization = paste0("Bearer ", token))
)

qry <- Query$new()
```

In a very basic form, our query to return skill assessments looks like this:

```
query {
 skillAssessmentCatalog {
   totalCount
   nodes {
     name
     domain
   }
 }
}
```

We can register this query with the `qry` object we just set up.

```
# register a new query named 'skill_assessments'
qry$query(
  "skill_assessments",
  "query {
     skillAssessmentCatalog {
       totalCount
       nodes {
         name
         domain
       }
     }
  }"
)
```

To keep things cleaner, we can save our query in a separate file `assessments.graphql` (in the same directory as our R script), and read it via the `readr` package's [`read_file`]() function.

```
qry$query("skill_assessments", readr::read_file("./assessments.graphql"))
```

Execute our GraphQL query

Now we can execute the 'skill_assessments' query using the connection (`con`) that we set up earlier.

```
response <- con$exec(qry$queries$skill_assessments) %>% fromJSON()
```

`response` should be a series of nested lists. If we then store the body of our query results like this:

```
catalog <- response$data$skillAssessmentCatalog
```

the dataframe containing skill assessments is available in `catalog$nodes`

If we notice though, this dataframe only contains 100 skill assessments. The total number of skill assessments in the catalog (`catalog$totalCount`) is considerably more than the 100 we just got back. What's going on here?

Querying with pagination

The Pluralsight GraphQL API uses pagination (see the [Using GraphQL]() documentation for more details) and the default is to return 100 results. As such, our starting query will only get us the first 100 skill assessments in the catalog. You can use the `first` parameter to specify more than 100. We recommend using 1,000 and paginating through any results beyond that. Pagination is a way to tell our query to "keep going" until we get all the results. Lucky for us, we can accomplish this by using the `pageInfo` data available with each query.

`pageInfo` gives us access to `endCursor` (this can be thought of as the last record in the page results) and `hasNextPage` (an indicator whether or not there is more data to be fetched).

Let's modify our query to make sure we're including these useful parts in our results, so that it looks like this:

`./assessments.graphql`

```
query {
  skillAssessmentCatalog {
    pageInfo {
      endCursor
      hasNextPage
    }
    nodes {
      name
      domain
    }
  }
}
```

and save the changes to `./assessments.graphql`. In R, if we re-execute this query now with:

```
qry <- Query$new()
qry$query("skill_assessments", readr::read_file("./assessments.graphql"))
response <- con$exec(qry$queries$skill_assessments) %>% fromJSON()
catalog <- response$data$skillAssessmentCatalog
```

From the R console, we should be able to see something like:

```
> catalog$pageInfo
$endCursor
[1] "M2I5MmJmYWItOWJiNS00MWYwLWI2NWYtZDk1M2IxODk2NWI3"

$hasNextPage
[1] TRUE
```

With one last modification to `./assessments.graphql` , we can parameterize our query to use a specific cursor as the starting point when we start pulling data. It should now look like:

`./assessments.graphql`

```
query($previousCursor: String) {
 skillAssessmentCatalog(after: $previousCursor) {
   pageInfo {
     endCursor
     hasNextPage
   }
   nodes {
     name
     domain
   }
  }
}
```

Write a function that collects the data

Now we've made it to the part where we can start collecting all the data. In theory, we'd like to write a function that queries the data for each page (starting with the first page) such that each time we run it, it:

1. collects the current page's data and appends it to a collection of overall data we're interested in
2. gets an id for the last record of the current page (to use as a reference point to start from)
3. continues on to the next page, starting our query from where we left off
4. stops when we make it to the end of the last page's results (there is no next page)
5. returns the overall set of data we've collected across all the pages

We can collect the full catalog of skill assessments by writing a recursive function `collect_assessments`.

```
# initialize container dataframe (tibble) that we'll add assessments to
skill_assessment_catalog <- tibble()

collect_assessments <- function(previous_cursor) {
  # stop collecting if we've made it to the last page
  if(is.null(previous_cursor)) {
    return()
  }

  # we'll pass these into our parameterized query
  vars <- list(previousCursor = previous_cursor)
  response <-
    con$exec(qry$queries$skill_assessments, vars) %>%
    fromJSON()

  catalog <- response$data$skillAssessmentCatalog

  # if there's an end cursor and the response has a next page,
  # start getting the next skill assessments after the 'previous_cursor'
  collect_assessments(catalog$pageInfo$endCursor)

  # this page's assessments
  assessments <- catalog$nodes %>% as_tibble()

  # add this page’s assessments to container with previously collected
  # (note the <<- to modify the container in the parent scope)
  skill_assessment_catalog <<-
    skill_assessment_catalog %>%
    bind_rows(assessments)
}

# the initial call has no previous_cursor, so it’ll start on first page

# each subsequent recursive call will be called with the previous page's
# endCursor as the starting point for the next query
collect_assessments(previous_cursor = NA)
```

Each time we call `collect_assessments`, it will add the current page’s assessments to our container dataframe `skill_assessment_catalog` until we’ve made it through all the pages.

As a final step, if we wrap this behavior in another function `get_skill_assessments`, we can keep `skill_assessment_catalog` in this parent function’s scope (and out of the global environment), and add some useful messaging as we collect all the data.

Putting it all together

Combining all the pieces, we should end up with:

```
library(ghql)
library(jsonlite)
library(dplyr)

# set up connection ----------------------------------------------------

graphql_api_url <- "https://paas-api.pluralsight.com/graphql"
```

```r
token <- Sys.getenv("PLURALSIGHT_GRAPHQL_TOKEN")

con <- GraphqlClient$new(
  url = graphql_api_url,
  headers = list(Authorization = paste0("Bearer ", token))
)

qry <- Query$new()
qry$query("skill_assessments", readr::read_file("./assessments.graphql")


# get the skill assessment catalog -------------------------------------

# recursively get the skill assessments until there is no endCursor
# returns a dataframe of Pluralsight’s skill assessments catalog
get_skill_assessments <- function(previous_cursor = NA) {

  # keep track of when we start
  start_time <- Sys.time()

  message("Collecting Pluralsight skill assessments...")

  # a container dataframe (tibble) that we'll add assessments to
  skill_assessment_catalog <- tibble()

  collect_assessments <- function(previous_cursor) {
    # stop collecting if we've made it to the last page
    if(is.null(previous_cursor)) {
      return()
    }

    # we'll pass these into our parameterized query
    vars <- list(previousCursor = previous_cursor)
    response <-
      con$exec(qry$queries$skill_assessments, vars) %>%
      fromJSON()
    catalog <- response$data$skillAssessmentCatalog

    # if there's an end cursor and the response has a next page,
    # start getting the next skill assessments after 'previous_cursor'
    collect_assessments(catalog$pageInfo$endCursor)

    # this page's assessments
    assessments <- catalog$nodes %>% as_tibble()

    # add this page’s assessments to container with previously collected
    # (note the <<- to modify the container in the parent scope)
    skill_assessment_catalog <<-
      skill_assessment_catalog %>%
      bind_rows(assessments)
  }

  # start collecting the full catalog
  collect_assessments(previous_cursor)

  # notify when complete
  duration <-
    as.numeric(Sys.time() - start_time, vars = "secs") %>%
    round(2)

  message(
    paste0(
      "Finished collecting",
      nrow(skill_assessment_catalog),
      " assessments in ",
      duration,
      "s."
    )
  )

  # return the full catalog sorted by assessment name
  skill_assessment_catalog %>% arrange(name)

}
```

Finally, you should be able to run the function we created and see something like:

```
> get_skill_assessments()
Collecting Pluralsight skill assessments...
Finished collecting 412 assessments in 2.65s.
# A tibble: 412 x 2
   name                                    domain
   <chr>                                   <chr>
 1 .NET Microservices                      Development
 2 3ds Max: Environment Modeling           Creative
 3 Active Directory Administration         IT Ops
 4 Administer and Monitor Couchbase        Data
 5 Administering Microsoft 365 Security    IT Ops
 6 Adobe Analytics for Developers          Creative
 7 Adobe Campaign v6.1                     Creative
 8 Adobe Experience Manager for Developers Development
 9 Adobe Target for Developers             Creative
10 Adobe Video and Audio Production        Creative
# … with 402 more rows
```

Conclusion

Data from the Pluralsight GraphQL API can be pulled into an R environment in a relatively straightforward way. By combining both Pluralsight's GraphQL data with internal customer data, customers can create unique analytical views that are tailored to their objectives.

Resources

- GraphQL Documentation hosted on our Developer portal
  - Pluralsight Developer Portal: Home
- Playground for exploring & converting query to basic python query
  - GraphQL Playground

FAQ

Q: Does the GraphQL API documentation cover data on SKILLS and FLOW plans?
A: Not at this time. It is only available for SKILLS data, though a plan that has both SKILLS and FLOW products can utilize the API to pull SKILLS data.

Q: Is there a limitation to the number of queries we create?
A: There is no limit to the number of queries. To minimize large data pulls, please use filters and pagination to create delta data pulls (and avoid all-time reports – especially for large datasets like CourseUsage, ChannelProgress, etc). This reduces load on our servers and provides a better experience for you and all customers who are using the API endpoint.

Q: How do I request an API key?
A: This can only be done by a Pluralsight Plan Admin on the Manage Keys tab of the Developer Portal. If you do not know who your Plan Admin(s) is/are, please reach out to your dedicated CSM or support resource to identify them.

Q: My company has more than 1 Pluralsight plan. Is there a way to pull data for more than 1 plan?How do I request a multi-plan API key?
A: Yes, a multi-plan API key is what you need. To request one, please reach out to us at professionalservices@pluralsight.com. As long as each plan has an Integrations or ProServ SKU, we will be able to create a multi-plan API key for you.

Q: How long does it take to generate an API token?

A: As a plan admin, you can create your own API Keys in the Pluralsight Developer Portal . Please contact support@pluralsight.com if you have any questions.

Q: Will you create and maintain the R script for our organization?
A: No.

Q: Does this cost extra?
A: API access is available as part of your subscription. Please see Plan Permissions for a listing of APIs available by subscription type.

Q: What data can I not see?
A: If it is not listed in the Developer Portal documentation schema, then it is not available. Keep an eye on the change log section of the Dev Portal for updates.

Q: I do not like using GraphQL APIs or am unfamiliar with them. Can I continue to use your REST APIs?
A: You may continue to use our SKILLS REST APIs currently, however they will be decommissioned in Q2 2021.. The data availability and manipulation through the REST APIs is not as complete. For example you can't get SkillIQ, RoleIQ, etc from REST APIs. We will be using GraphQL APIs as the standard for the SKILLS product. FLOW (formerly GitPrime) will continue to use REST. If you are using REST APIs currently for SKILLS data and want to know how to make the switch to GraphQL, please see the help documentation here.