External GraphQL API Documentation

Overview

The Vector LMS EDU Edition API provides read and write access to user rosters and read access to learner progress and course data.

It is a GraphQL API. More information on GraphQL can be found here.

API location: `https://[customer specific subdomain/URL]/graphql`

Authentication

The required OAUTH2 client credentials (`client_id` and `client_secret`) can be generated and disabled via the administrator section of the platform. `client_secret` will only ever be shown once. Multiple client/secrets can be active for a given customer.

The Vector LMS EDU Edition API requires every request to provide an authorization token, provided in a header property called `Authorization`. When developing your API integration, you will need to implement a way to first authenticate your API, retrieve the `access_token` and `token_type` from a successful response to the URL below, and then pass the `access_token` and `token type` to each subsequent request in the `Authorization` header property. If the request returns a `401 Unauthorized`, verify you have the correct values for `access_token` and `token type`, and ensure the parameter names are in the correct case. You may need to regenerate your `client_secret` in the platform; if you do, be sure any other callers using the same credentials get the new value.

```
https://[customer specific subdomain/URL]/oauth/token?
grant_type=client_credentials&client_id=&client_secret=
```

All other requests in the API require you to provide the value of the `access_type` and `token_type` in a header property called `Authorization`. If the request does not include the `Authorization` header property, then the API will reject the unauthenticated request with a `401 Unauthorized` HTTP status code. Below is an example of an API request with the `Authorization` header.

The `expires_in` property in the response indicates the number of seconds that remain before the token expires.

Step 1 - Get a Token

```
curl --location --request POST 'https://[customer specific
subdomain/URL]/oauth/token?grant_type=client_credentials&client_id=xxx
xxxx&client_secret=yyyyyyy' \
--header 'Accept: application/json' \
--header 'Content-Type: application/json'
```

Step 2 - Retrieve Token Value from Response

With a `200 OK` response, value of `access_token` and `token_type` can be retrieved from the response and needs to be saved for use in later requests.

Sample response:

```
{
    "access_token": "d9a252cb-d159-11ec-b07b-0a4e64b4609b",
    "token_type": "Bearer",
    "expires_in": 7200,
    "created_at": 1556024563
}
```

Step 3 - Send an authenticated request to the GraphQL endpoint

When requests are prepared for the GraphQL endpoint, a header property called `Authorization` must be included. The `Authorization` value is constructed by concatenating the value of `token_type`, a space, and then the value of `access_token`. For example:

```
curl --request POST --location 'https://[customer specific
subdomain/URL]/graphql' \
--header 'Content-Type: application/json' \
--header 'Accept: application/json' \
--header 'Authorization: Bearer d9a252cb-d159-11ec-b07b-0a4e64b4609b'
\
--data '{ "query": "{ ... }" }'
```

API Rate Limit

The rate limit for the API is up to 150 requests per rolling 300 seconds and may depend on the complexity of the queries requested. If this limit is exceeded, the API will respond with an error with HTTP status code `429` (too many requests). The limit is lower than one would expect from a traditional RESTful API, however it should still be comparable because more work can be completed in a single call to the GraphQL endpoint than could be done via a single call to a REST API.

In some rare conditions, the system may be too busy to process the request. In that case an HTTP status `503` (service unavailable) is returned.

No payload is guaranteed to be returned if `429` or `503` is returned. In either case, it is strongly recommended the client use an exponential backoff approach until normal responses resume.

Vector Solutions reserves the right to add additional usage limits without notice.

Request Paging

Many types return an internal `Page` type. When included, the `page` field acts as a cursor (or "Connection") to represent the count and start of the total results returned. The actual results are then present in the `nodes` field. When querying, the `before`, `after`, `first`, and `last` can select the desired portion of results. The number of records returned may differ from the request due to number of records available and will not exceed 100 per page.

See https://graphql.org/learn/pagination/ for more details on pagination in GraphQL.

Errors

For most error conditions, except for the rate limit errors noted above, an HTTP `200` is returned with an `errors` array field in the JSON structure is returned:

```
{
  "data": // Optional -- input query
  "errors":
  [
    {
      "code": "{ERROR CODE}",
      "message": "{Human-Readable Error message}",
      "path": [ "key", "key", "key" ]  // path is present on some non-
parsing errors.
      "locations":
      [
        { "line": ..., "column": ... }
      ] // locations is present only for parsing errors.
    }
    ...  // At least one error, but any number are possible.
  ],
}
```

The `code` and `message` will exist on all errors. `paths` will depend on the query. `locations` only appear on parser errors.

See http://spec.graphql.org/October2021/#sec-Errors for more details on general GraphQL usage of the above.

Versioning

The API is versionless. It will evolve over time and most changes to it will be backward-compatible. Planned backwards-incompatible changes will be advertised in advance using the `@deprecated` directive.

Vector Solutions will follow the above process whenever possible. Critical security or performance issues may require us to make immediate breaking changes to the API without notice.

Usage

User Roster Management

Person

Represents a record of an individual known to the system. All persons are trainees, some also have elevated permissions allowing for administration of the system. The management of those permissions is outside of the scope of this API and can be done via the product interface.

If an external ID is provided, this should match any other methods of identifying persons via external systems (such as SAML or LDAP). Management of those external integrations are also outside of the scope of this API.

Position and Location

A position represents a role or classification known to the system to collect persons within a hierarchy, such as matriculation year, department, or skillset, etc.

A location similarly represents a place which collect persons within a separately hierarchy, such as a campus, dorm, etc. Please note that the Location does not have to represent a physical location and can be used as an additional grouping mechanism.

Job

A job associates a position with a location for a period of time for a particular person with the purpose of assigning courses to that Job. An example of a Job is a **Teacher of the Arts in South Bend Campus, starting March 5, 2020**. A person can have any number of jobs, whose overlapping positions and locations provide a complete picture of their roles. Please note that if a person has multiple Jobs, they will show as multiple records in reports.

Course Completion Data

Progress

Progress represents a trainee's state of training for a specific course, whether and when they completed it, and the best assessment score percentage they achieved.

A person may have multiple progress records that reflect different trainings of the same course. Progress records only reflect in progress or completed trainings.

CourseInfo

Contains basic information about a course under progress.

GraphQL Schema

```
schema {
query: QueryRoot
mutation: MutationRoot
}

type QueryRoot {
Location(locationId: ID!): Location
Locations(
code: String
name: String
parentId: ID
after: ID
before: ID
first: Int
last: Int
): PagedLocation
Completions(
locationId: ID
endDate: DateTime!
```

positionId: ID
startDate: DateTime!
after: ID
before: ID
first: Int
last: Int
): PagedProgress
CourseInfo(courseInfoId: ID!): CourseInfo
Job(jobId: ID!): Job
Progress(progressId: ID!): Progress
People(
locationId: ID
positionId: ID
active: Boolean
after: ID
before: ID
first: Int
last: Int
): PagedPerson
Person(personId: ID!): Person
Position(positionId: ID!): Position
Positions(
code: String
name: String
parentId: ID
after: ID
before: ID
first: Int
last: Int
): PagedPosition
}

type MutationRoot {
Location(locationId: ID!): LocationMutation
Position(positionId: ID!): PositionMutation
Person(personId: ID!): PersonMutation
Job(jobId: ID!): JobMutation
addLocation(code: String, name: String!, parentId: ID): Location
addPosition(code: String, name: String!, parentId: ID): Position
addPerson(
address1: String
address2: String
address3: String
beginDate: String
locationId: ID
city: String
externalUniqueId: String
email: String
first: String!

middle: String
last: String!
password: String
phone: String
positionId: ID
postalCode: String
state: String
username: String!
): Person
}

 # Roster Types

type Person {
address1: String
address2: String
address3: String
city: String
country: String
email: String
externalUniqueId: String
first: String
jobs: [Job]
last: String
middle: String
progress: [Progress]
personId: ID!
phone: String
postalCode: String
state: String
username: String
}

type PersonMutation {
update(
address1: String
address2: String
address3: String
city: String
country: String
email: String
first: String
last: String
phone: String
postalCode: String
state: String
username: String
): Person
changePassword(password: String!): Person

```
deactivate: Person
addJob(
locationId: ID!
positionId: ID!
title: String
beginDate: Date
endDate: Date
): Job
}

type JobMutation {
update(beginDate: Date, endDate: Date, title: String): Job
deactivate: Job
}

type Job {
beginDate: Date
location: Location!
endDate: Date
jobId: ID!
person: Person!
position: Position!
title: String
}

type Location {
locationId: ID!
children: [Location]
code: String
name: String
parent: Location
}

type LocationMutation {
remove: Location
update(code: String, name: String): Location
}

type Position {
children: [Position]
code: String
name: String
parent: Position
positionId: ID!
}

type PositionMutation {
remove: Position
update(code: String, name: String): Position
```

```
}

# Course Progress Types

type Progress {
completed: Boolean
completeTime: DateTime
courseInfo: CourseInfo!
progressId: ID!
maxQuizScore: Float
person: Person!
}

type CourseInfo {
courseInfoId: ID!
title: String
}

# Paging Types

type PageInfo {
count: Int
endCursor: ID
hasNextPage: Boolean!
hasPreviousPage: Boolean!
startCursor: ID
totalCount: Int
}

type PagedLocation {
nodes: [Location]
pageInfo: PageInfo
}

type PagedProgress {
nodes: [Progress]
pageInfo: PageInfo
}

type PagedPerson {
nodes: [Person]
pageInfo: PageInfo
}

type PagedPosition {
nodes: [Position]
pageInfo: PageInfo
}
```