A Guide to the Labelbox Python SDK

Labelbox

Table of contents

00 The complete guide to the Labelbox Python SDK

While you can interact with Labelbox through our powerful GraphQL API, we also recognize that Python is the most common programming language used by data scientists and the machine learning community at large. In order to enable easier workflows for those most comfortable with Python, we set out to create a pip package that you can install and use out of the box. You'll be able to avoid writing and maintaining your own scripts, and writing queries and mutations that fit within our schema, without ever having to learn GraphQL.

With the Python SDK, you can:

1. Simplify your data import
We've simplified the data import process for Labelbox so that you can use bulk DataRow creation. Best yet, this process is asynchronous, meaning you can (but don't have to) wait for bulk creation to finish before continuing with other tasks (or even terminating the client).

2. Interact with the API in an object-oriented way
Create projects and datasets programmatically, export labels, and add metadata to your assets all in an object-oriented way—complete with all relationships between objects.

3. Set queue customization
With queue customization you can support active learning by adjusting your labeling queue to focus on improving confidence in specific classes. We also see customers use queue customization to support time-sensitive labeling, such as real-time labeling workflows that are part of human-in-the-loop AI products and services.

01 Getting started

We created this Python API so you can access all the functionalities of the Labelbox API without having to write GraphQL queries. This documentation will guide you through installation, setup, and some common use cases.

See the Python API repo in Github.

We released v2.4 on January 30, 2020. Run pip install --upgrade labelbox to upgrade (please consult the Changelog before upgrading). The Python API supports Python versions 3.6 and 3.7.

Python API features

- Read, create, and modify objects and object-to-object relationships
- Simplify your data import with bulk Data Row creation
- Download a JSON export of labels
- Prioritize, re-enqueue & adjust the count of labels in the labeling queue
- Use webhooks with limitations
- Support for Prediction and PredictionModel data types

Installation & authentication

Step 1:Create your API key
All users wishing to access the Python API must have an API key for authentication. For instructions on creating a key, see API keys.

Step 2:Install
Install Labelbox using pip.

```
user@machine:~$ pip install labelbox
```

Step 3: Authenticate
Pass your API key as an environment variable. Then, import and initialize the API Client.

```
user@machine:~$ export LABELBOX_API_KEY="<your_api_key_here>"
```

```
user@machine:~$ python3
```

```
from labelbox import Client
```

```
client = Client()
```

Passing in a custom endpoint is only applicable for on-premises use cases. If this applies to you, you may pass the API key and server endpoint explicitly when you initialize the Client object. Otherwise, refer to the step above.

```
from labelbox import Client
```

```
client = Client("<your_api_key_here>", "https://your-server.com/labelbox/")
```

Python script
Here is a Python snippet you can also use to connect to the API. Run this as a python script and make sure to include your API key within quotations since the expected data type for the API key is a string.

```
from labelbox import Client
```

```
if __name__ == '__main__':
```

```
API_KEY = "<your_api_key_here>"
```

```
client = Client(API_KEY)
```

Step 4:Create a project

02 Creating your first project

Before you start

To better understand the functionalities described from here on out, take a moment to read Overview & data types.

In order for the following methods to work, make sure the API client is initialized:

```
from labelbox import Client
```

```
client = Client()
```

Project setup

Step 1:Build your project's foundation
In the rough hierarchical structure of Labelbox's data objects, projects and datasets are considered "top-level" objects. They are the foundation upon which your labeling pipeline is structured. Because they are top-level, projects and datasets are created using the Labelbox Client directly.

Use the create_project method to create and name your project. You will be attaching your datasets to your project so name it accordingly.

```
project = client.create_project(name="<project_name>")
```

Within your project, use the create_dataset method to create a dataset, name it, and attach it to your project. The name of the dataset should reflect the nature of the data it contains.

```
dataset = client.create dataset
(name="<dataset_name>", projects=project)
```

Step 2:Add data to your project
There are two ways to create data rows within a dataset, in bulk and individually. For details on acceptable file types, see Data import overview.

The create_data_row method accepts files individually and is a *synchronous* operation.

```
dataset = client.get_dataset("dataset_id")

data_row = dataset.create_data_row(row_
data="http://my_site.com/photos/img_01.jpg")
```

You can also pass a string to a local file.

```
data_row = dataset.create_data_row(row_
data="path/to/file.jpg")
```

For instructions on how to bulk upload data rows using the create_data_rows method, see Data Rows.

Step 3:Connect an ontology and specify a label editor

Use this method below to finalize the setup for your newly created project.

```
project.setup(self, labeling_frontend,
labeling_frontend_options)
```

Use the labeling_frontend argument to specify which editor to use for the project. You can find all of the editors available within your organization by querying for client.get_labeling_frontends(), which returns a paginated collection.

The labeling_frontend_options argument takes in a stringified JSON or dict version of your ontology.

End-to-end python example

```
from labelbox import Client

import sys, os, json

if __name__ == '__main__':

client=Client(sys.argv[1])

project = client.create_project(name="Test
Project")

dataset = client.create_dataset(name="Test
Dataset", projects=project)

frontends = client.get_labeling_frontends()

for frontend in frontends:

print(frontend)

# option 1: copy ontology from existing
project

ontology = project.labeling_frontend_
options()

# option 2: copy + paste ontology directly as
stringified JSON

ontology = """

{

"tools": [],

"classifications": []

}

"""

# connecting interface and ontology to
finalize project set-up

project.setup(list(frontend_list)[0],
ontology)
```

03 Projects

These are some common use cases people have when working with Projects in the Python API. For a complete list of methods, see our API reference.

For the definition of a Project, see Overview & data types.

Before you start

Make sure the client is initialized.

```
from labelbox import Client

client = Client()
```

Create a Project

To learn how to create a Project, see Creating your first project.

Fetch a Project

Since a Project is a top-level object, you can get a specific Project by passing the unique ID of the Project to the client. The get_project method will only take a unique ID as an argument.

```
project = client.get_project("<project_id>")

print(project)
```

Fetch multiple projects

Here are three ways to fetch multiple Projects.

1.
Use the get_projects method to **get a list of all Projects** in your account. This sample code will return the name and uid for each Project.

```
for project in client.get_projects():

print(project.name, project.uid)
```

2.
Specify a Project by passing a comparison as a where parameter. You can use any of the standard comparison operators (==, !=, >, >=, <, <=) to specify which Projects you want. Because the get_projects method can return any number of results, it will give you a PaginatedCollection object over which you can iterate to get your specified Project.

```
from labelbox import Project

projects_x = client.get_
projects(where=Project.name ==
"MyProject")
```

Then, you can iterate over the PaginatedCollection object.

```
for x in projects_x:

print(x)
```

Or you can use this code to get your desired object which will raise an error if there is not at least one item in the collection.

```
item = next(iter(projects_x))

print(item)
```

For more information on pagination, see General concepts.

3.
Specify a Project by combining comparisons using logical expressions. Currently the where clause supports the logical AND operator.

```
from labelbox import Project

projects = client.get_projects(where=(Project.
name == "X") & (Project.description == "Y"))

for x in projects:

print(x)
```

Update a Project field

Use the update method to target and modify any updatable value in the object's type. The key value arguments define which fields should be updated with which values. For a complete list of updatable and non-updatable fields in the Project object, see the API reference.

```
project = client.get_project("<projectID>")

project.update(name="Project Name")
```

04 Datasets

These are some common use cases people have when working with Datasets in the Python API. For a complete list of methods, see our API reference.

For the definition of a Dataset, see Overview & data types.

Before you start

Make sure the client is initialized.

```
from labelbox import Client

client = Client()
```

Create a Dataset

To learn how to create a Dataset, see Creating your first project.

Fetch a Dataset

Since a Dataset is a top-level object, you can get a specific Dataset by passing the unique ID of the Dataset to the client. The get_dataset method will only take a unique ID as an argument.

```
dataset = client.get_dataset("<dataset_id>")

print(dataset)
```

Fetch multiple Datasets

Below are three examples for fetching multiple datasets using the get_datasets method.

1.
The code sample below demonstrates how to **fetch all Datasets in your account** and print the collection of Datasets by unique ID.

```
for dataset in client.get_datasets():

print(dataset.uid)
```

2.
Pass a where parameter with a comparison operator. Use the get_datasets method and any of the standard comparison operators (==, !=, >, >=, <, <=) to target a Dataset. Because the get_datasets method can

theoretically return any number of results, it will give you a PaginatedCollection object over which you can iterate to get your specified Dataset.

```
from labelbox import Dataset

datasets_x = client.get_datasets(where=Dataset.name == "X")for x in datasets_x:

print(x)
```

3.
You can also find Datasets by using the get_datasets method to **combine comparisons using logical expressions**. Currently the where clause supports the logical AND operator.

```
from labelbox import Project

datasets = client.get_datasets(where=(Project.name == "X") & (Project.description == "Y"))

for x in datasets_x:

print(x)
```

05 Data Rows

These are some common use cases people have when working with Data Rows in the Python API. For a complete list of methods, see our API reference.

For the definition of a Data Row, see Overview & data types.

Before you start

Make sure the client is initialized.

```
from labelbox import Client

client = Client()
```

Bulk add Data Rows

For **adding Data Rows individually**, see Creating your first project.

Dataset.create_data_rows() accepts a list of items and is an asynchronous bulk operation. This is typically faster and avoids API limit issues. Unless you specify otherwise, your code will continue before the Data Rows are fully created on the server side. Paths to local files must be passed as a string.

- external_id (*string*) - OPTIONAL
- row_data (*string*) - REQUIRED
- uid (*ID*) - not updatable
- updated_at (*DateTime*) - not updatable
- created_at (*DateTime*) - not updatable

DataRow.row_data should only be used when you are passing a path to an external URL. You must pass paths to external URLs as a dict.

```
dataset = client.get_dataset("<dataset_uid>")

task = dataset.create_data_rows([{DataRow.row_data:"http://my_site.com/photos/img_01.jpg"},{DataRow.row_data:"http://my_site.com/photos/img_02.jpg"}])
```

Pass paths to local files as strings.

```
task = dataset.create_data_rows(["path/to/file1.jpg", "path/to/file2.jpg"])
```

You may also include paths to local files and paths to URLs at same time.

```
task = dataset.create_data_rows([{DataRow.
row_data:"http://my_site.com/photos/
img_01.jpg"},"path/to/file2.jpg"])
```

You can (but don't have to) use task.wait_till_done() wait for the bulk creation to complete before continuing with other tasks. For more information on Tasks, see the API reference.

Add Data Rows from a JSON File

At a minimum, the JSON file you are using to import your Data Rows must include row_data as a key for each asset you wish to add. You have the option to include an external_id.

```
[
{
"row_data": "<IMG_URL>",
"external_id": "12345"
},
{
"row_data": "<IMG_URL_2>"
},
{
"row_data": "<path/to/file.jpg>"
}
]
```

Use Dataset.create_data_rows to import Data Rows from your JSON file.

```
dataset = client.get_dataset("<dataset_
uid>")
with open("file.json") as f: dataset.create_
data_rows(json.load(f))
```

This will return a task ID.

```
<Task ID: ck37lee3hyixi0725egxwooue>
```

See the API reference for more information

Fetch Data Row by external ID

The optional external_id field of the Data Row object is additional to the uid field. People have found this additional external_id field useful when they need to map the Data Row to an asset in their own system.

Use the data_row_for_external_id method for getting a single Data Row by external_id within a dataset.

```
dataset = client.get_dataset("<dataset_uid>")
data_row = dataset.data_row_for_external_
id("<external_id>")
```

If there are two Data Rows that have the same external ID, this method will raise a ResourceNotFoundError. See the API reference for more information.

Fetch all Data Rows in a Dataset

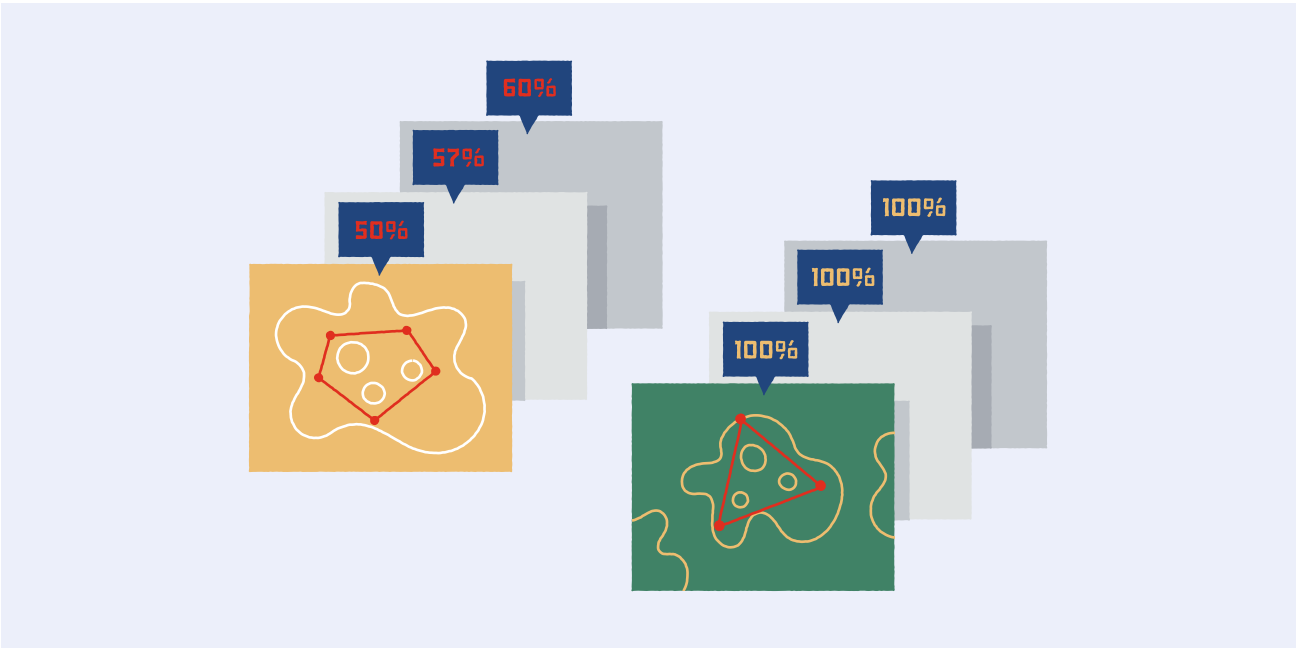
To fetch multiple Data Rows, use the data_rows method. This example demonstrates how to get a list of all Data Rows within a specified dataset.

```
dataset = client.get_dataset("<dataset_uid>")
for data_row in dataset.data_rows():
print(data_row)
```

Bulk delete Data Rows

To delete many Data Rows at a time, use the bulk_delete method. The code below demonstrates how to delete all Data Rows from a specified dataset if they were created after a certain date.

```
from labelbox import DataRow
dataset = client.get_dataset("<dataset_uid>")
data_rows = list(dataset.data_
rows(where=DataRow.created_at > some_
date))
DataRow.bulk_delete(data_rows)
```



06 Labels

These are some common use cases people have when working with Labels in the Python API. For a complete list of methods, see our API reference.

For the definition of a Label, see Overview & data types.

Before you start

Make sure the client is initialized.

```
from labelbox import Client

client = Client()
```

Add Labels

Use the create_label method for adding a label within a specified project. Only data_row is required.

```
project = client.get_project("<project_unique_id>")

data_row = dataset.create_data_row(row_data="http://my_site.com/photos/img_01.jpg")

label = project.create_label(data_row=data_row, label="<label_data_here>")
```

Export Labels

The sample below uses the export_labels method to print a URL to a JSON file containing the labels of a project.

```
project = client.get_project("<project_unique_id>")

url = project.export_labels()

print(url)
```

The response will be a URL of the label data file.

```
'https://storage.googleapis.com/labelbox-exports/cjnywra4rytzd079735j0hfnt/ck22dy2gmnbw08111o6y2ycs9/export-2019-10-29T22:59:08.592Z.json'
```

Bulk delete Labels

Use the bulk_delete method for deleting multiple labels at a time. Users usually seek to do this when they realize there was an error in the way that the labels were initially created. This method allows you to remove the existing annotations.

Here are two ways for bulk deleting labels within a specified project.

1.
Specify a project and delete use the bulk_delete method to **delete all labels** from that project.

```
project = client.get_project("<project_unique_id>")

Label.bulk_delete(project.labels())
```

2.
Do a **filtered relationship expansion** to specify which labels to delete within a project.

```
project = client.get_project("<project_unique_id>")

Label.bulk_delete(project.labels(where=Label.name=="x"))
```

07 General concepts

Here are some general concepts that are helpful to be aware of as you work with the Labelbox Python API.

Fields (Get/Update)

Get
Access a field as an attribute of the object.

Example:

```
project.name # Get name of project
```

Update
To update a field, use the update method to target and modify an updatable value in the object's type. Pass the field and the new value.

Example:

```
project.update(name="Project Name") # Update project name
```

Relationships (Get/Add/Update)

Get
To access related objects, call the relationship as a method. Example: To get all datasets for a project, define project and call datasets as a method to access the datasets related to project.

Example:

```
project.datasets() # Get all datasets for project
```

Add
To add a relationship between two objects, call the connect method directly from the relationship.

Example:

```
project.datasets.connect(dataset_1) # Connect dataset_1 to project
```

Update
To update a relationship, use the disconnect

method and then the connect method. It is important to note that update() does not work for updating relationships.

Example:

```
project.datasets.disconnect(dataset_1) # Disconnect dataset_1

project.datasets.connect(dataset_2) # Connect dataset_2
```

Pagination

Sometimes, a call to the server may result in a very large number of objects being returned. To prevent too many objects being returned at once, the Labelbox server API limits the number of returned objects. The Python API respects that limit and automatically paginates fetches. This is done transparently for you, but it has some implications.

```
projects = client.get_projects()

type(projects)

# PaginatedCollection

projects = list(projects)

type(projects)

# listproject = projects[0]

datasets = project.datasets()

type(datasets)

# PaginatedCollection

for dataset in datasets:

Dataset.name
```

There are several points of interest in the code above.

1. For both the top-level object fetch, client.get_projects(), and the relationship call, project.datasets(), a PaginatedCollection object is returned. This PaginatedCollection object takes care of the paginated fetching.

2. Note that nothing is fetched immediately when the PaginatedCollection object is created.

3. Round-trips to the server are made only as you iterate through a PaginatedCollection. In the code above that happens when a list is initialized with a PaginatedCollection, and when a PaginatedCollection is iterated over in a for loop.

4. You cannot get a count of objects in the relationship from a PaginatedCollection nor can you access objects within it like you would a list (using squared-bracket indexing). You can only iterate over it.

Be careful about converting a PaginatedCollection into a list. This will cause all objects in that collection to be fetched from the server. In cases when you need only some objects (let's say the first 10 objects), it is much faster to iterate over the PaginatedCollection and simply stop once you're done.

The following code demonstrates how to do this.

```
data_rows = dataset.data_rows()

first_ten = []

for data_row in data_rows:

first_ten.append(data_row)

if len(first_ten) >= 10:

Break
```

Immediate updates on the server side

Each data update using object.update() on the client side immediately performs the same update on the server side. If the client side update does not raise an exception, you can assume that the update successfully passed on the server side.

Field caching

When you fetch an object from the server, the client obtains all field values for that

object. When you access that obtained field value, the cached value is returned. There is no round-trip to the server to get the field value you have already fetched. Server-side updates that happen after the client-side fetch are not auto-propagated, meaning the values returned will still be the cached values.

Relationship fetching

Unlike fields, relationships are not cached. Relationships are fetched every time you call them. This is made explicit by defining relationships as callable methods on objects (refer to section above).

```
project.name # Fields are accessed as attributes
project.datasets() # Relationships are called as methods
```

In many cases you may not be concerned with relationship data freshness because only you will only be modifying your data during small timeframes. In those situations, it is completely fine to keep references to related objects.

```
project_datasets = list(project.datasets())
```

08 Python API SDK

These are some common questions users have when they are learning the capabilities of our Python API.

What should I do when I hit my API rate limit?
When the user attempts too many requests to the API within a short timeframe, the API will raise an ApiLimitError exception. Our support engineers recommend that you implement your own retry logic if this happens. However, if you have hit the 500 API calls per limit, consider making less calls to the API.

Can I customize my queue/set prioritization?
Yes. To learn how to customize the order of the data rows in a queue, see Project.set_labeling_parameter_overrides(self, data) in our API reference. For a detailed explanation of how queue customization works in Labelbox, see Queue customization.

Can I re-enqueue Labels?
This can be done with Label.bulk_delete(). It will automatically re-enqueue the labels. For instructions on how to do this with the GraphQL API, see Re-enqueue labels programmatically.

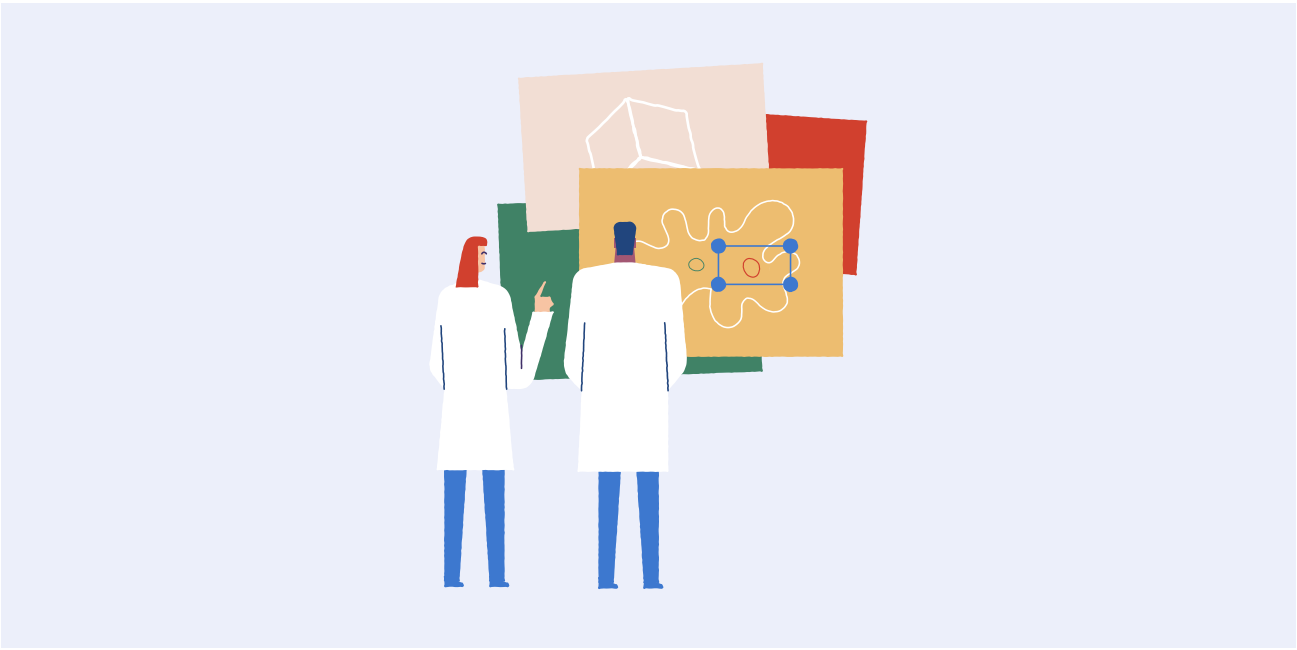
How can I sort the Labelbox objects I get from the API?
One-to-many relationship fetches can be sorted by calling a relationship and providing passing a value for order_by. You can only sort one field at a time in ascending or descending order. Currently, it's not possible to automatically sort top-level collections (such as Client.get_projects()) in a fetch. The example below fetches and alphabetically sorts in ascending order the datasets from a specified project.

```
project = client.get_project("<project_id>")
sorted_datasets = project.datasets(order_by=Dataset.name.asc)
```

If your Python API does not have the method I'm looking for, can I write my own custom GraphQL queries?
You can write your own custom GraphQL queries using Client.execute().

Can I add users with the Python API?
A method for adding users (in bulk or individually) to a project or an organization does not exist in version 2.2 of the Python API.

If you are looking for a particular functionality that has not been mentioned in this page or previous pages in this section, please take a look at our comprehensive **Python API reference.**



09 API reference

General Classes

- Client

Data Classes

- Project
- Dataset
- DataRow
- Label
- AssetMetadata
- LabelingFrontend
- Task
- Webhook
- User
- Organization
- Review
- Prediction
- PredictionModel
- LabelerPerformance

Error Classes

- LabelboxError
- AuthenticationError
- AuthorizationError
- ResourceNotFoundError
- ValidationFailedError
- InvalidQueryError
- NetworkError
- TimeoutError
- InvalidAttributeError
- ApiLimitError

General classes

Client

Class labelbox.client.Client.

A Labelbox client. Contains info necessary for connecting to a Labelbox server (URL, authentication key). Provides functions for querying and creating top-level data objects (**Projects**, **Datasets**).

Object Methods

__init__(self, api_key, endpoint='https://api.labelbox.com/graphql')

Creates and initializes a Labelbox Client

- **Args:**
  - *api_key (str)*: API key. If None, the key is obtained from the "LABELBOX_API_KEY" environment variable.
  - *endpoint (str):* URL of the Labelbox server to connect to.

- **Raises:**
  - *labelbox.exceptions.AuthenticationError*: If no `api_key` is provided as an argument or via the environment variable.

create_dataset(self, **kwargs)

Creates a **Dataset** object on the server. Attribute values are passed as keyword arguments:

```
>>> project = client.get_project("uid_of_my_project")
```

```
>>> dataset = client.create_dataset(name="MyDataset",
```

```
>>> projects=project)
```

- **Kwargs**: Keyword arguments with new **Dataset** attribute values. Keys are attribute names (in Python, snake-case convention) and values are desired attribute values.
- **Returns**: A new **Dataset** object.
- **Raises**:
  - *InvalidAttributeError*: If the **Dataset** type does not contain any of the attribute names given in kwargs.

create_project(self, **kwargs)

Creates a **Project** object on the server. Attribute values are passed as keyword arguments:

```
>>> project = client.create_project(name="MyProject")
```

- **Kwargs**: Keyword arguments with new **Project** attribute values. Keys are attribute names (in Python, snake-case convention) and values are desired attribute values.
- **Returns**: A new **Project** object.
- **Raises**:
  - *InvalidAttributeError*: If the **Project** type does not contain any of the attribute names given in kwargs.

execute(self, query, params, timeout=10.0)

Sends a request to the server for the execution of the given query. Checks the response for errors and wraps errors in appropriate **labelbox.exceptions.LabelboxError** subtypes.

- **Args**:
  - *query (str)*: The query to execute.
  - *params (dict)*: Query parameters referenced within the query.
  - *timeout (float)*: Max allowed time for query execution, in seconds.
- **Returns**: dict, parsed JSON response.
- **Raises**:
  - *labelbox.exceptions.AuthenticationError:* If authentication failed.
  - *labelbox.exceptions.InvalidQueryError:* If `query` is not syntactically or semantically valid (checked server-side).
  - *labelbox.exceptions.ApiLimitError*: If the server API limit was exceeded. See "How to import data" in the online documentation to see API limits.
  - *labelbox.exceptions.TimeoutError*: If response was not received in `timeout` seconds.
  - *labelbox.exceptions.NetworkError:* If an unknown error occurred most likely due to connection issues.
  - *labelbox.exceptions.LabelboxError:* If an unknown error of any kind occurred.

get_dataset(self, dataset_id)

Gets a single **Dataset** with the given ID.

- **Args**:
  - *dataset_id (str)*: Unique ID of the **Dataset**.
- **Returns**: The sought **Dataset**.
- **Raises**:
  - *labelbox.exceptions Resource NotFoundError:* If there is no **Dataset** with the given ID.

get_datasets(self, where)

Fetches all the datasets the user has access to.

- **Args:**
  - *where (Comparison, LogicalOperation or None):* The `where` clause for filtering.
- **Returns**: An iterable of **Datasets** (typically a **PaginatedCollection**).

get_labeling_frontends(self, where)

Fetches all the labeling frontends.

- **Args:**
  - *where (Comparison, LogicalOperation or None):* The `where` clause for filtering.
- **Returns**: An iterable of **LabelingFrontends** (typically a **PaginatedCollection**).

get_organization(self)

Gets the **Organization** DB object of the current user.

get_project(self, project_id)

Gets a single **Project** with the given ID.

- **Args**:
  - *project_id (str)*: Unique ID of the **Project**.
- **Returns**: The sought **Project**.
- **Raises**:
  - l*abelbox.exceptions. ResourceNotFoundError*: If there is no **Project** with the given ID.

get_projects(self, where)

Fetches all the projects the user has access to.

- **Args:**
  - *where (Comparison, LogicalOperation or None):* The `where` clause for filtering.
- **Returns**: An iterable of **Projects** (typically a **PaginatedCollection**).

get_user(self)

Gets the current **User** database object.

upload_data(self, data)

Uploads the given data (bytes) to Labelbox.

- **Args**:
  - *data (bytes)*: The data to upload.
- **Returns**: str, the URL of uploaded data.
- **Raises**:
  - *labelbox.exceptions.LabelboxError:* If upload failed.

Data classes

Project

Class labelbox.schema.project.Project (Updateable, Deletable).

A **Project** is a container that includes a labeling frontend, an ontology, datasets and labels.

Fields

- auto_audit_number_of_labels (*Int*)
- auto_audit_percentage (*Float*)
- created_at (*DateTime*)
- description (*String*)
- last_activity_time (*DateTime*)
- name (*String*)
- setup_complete (*DateTime*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- labels (*Label, ToMany*)
- active_prediction_model (*PredictionModel ToOne*)
- benchmarks (*Benchmark ToMany*)
- created_by (*User ToOne*)
- datasets (*Dataset ToMany*)
- labeling_frontend (*LabelingFrontend ToOne*)
- labeling_frontend_options (*LabelingFrontendOptions ToMany*)
- labeling_parameter_overrides (*LabelingParameterOverride ToMany*)
- organization (*Organization ToOne*)
- predictions (*Prediction ToMany*)
- reviews (*Review ToMany*)
- webhooks (*Webhook ToMany*)

Object methods

create_label(self, **kwargs)

Creates a label on this **Project**.

- **Kwargs**: **Label** attributes. At the minimum the label `**DataRow**`.

create_prediction(self, label, data_row, prediction_model)

Creates a **Prediction** within this **Project**.

- **Args**:
  - *label (str)*: The `label` field of the new **Prediction**>Prediction>Prediction.
  - *data_row (DataRow> DataRow>DataRow> DataRow>DataRow)*: The DataRow for which the **The DataRow** for which the **Prediction**>Prediction is created
  - *prediction_model (PredictionModel> PredictionModel> PredictionModel> PredictionModel> PredictionModel or None):* The **PredictionModel** within which the new **Prediction** is created. If None then this **Project**'s active_prediction_model is used.
- **Returns**: A newly created **Prediction**.
- **Raises**:
  - *labelbox.excepions.InvalidQueryError:* if given `prediction_model` is None and this **Project**'s active_prediction_model is also None.

create_prediction_model(self, name, version)

Creates a **PredictionModel** connected to this **Project**.

- **Args**:
  - *name (str)*: The new **PredictionModel**>PredictionModel's name.
  - *version (int):* The new **PredictionModel**'s version.
- **Returns**: A newly created **PredictionModel**.

delete(self)

Deletes this DB object from the DB (server side). After a call to this you should not use this DB object anymore.

export_labels(self, timeout_seconds=60)

Calls the server-side ***Label*** exporting that generates a JSON payload, and returns the URL to that payload. Will only generate a new URL at a max frequency of 30 min.

- **Args**:
  - *timeout_seconds (float)*: Max waiting time, in seconds.
- **Returns**: URL of the data file with this **Project**'s labels. If the server didn't generate during the `timeout_seconds` period, None is returned.

extend_reservations(self, queue_type)

Extends all the current reservations for the current user on the given queue type.

- **Args**:
  - *queue_type (str)*: Either "LabelingQueue" or "ReviewQueue"
- **Returns:** int, the number of reservations that were extended.

labeler_performance(self)

Returns the labeler performances for this **Project**.

- Returns: A **PaginatedCollection** of **LabelerPerformance** objects.

labels(self, datasets, order_by)

Custom relationship expansion method to support limited filtering.

- **Args**:
  - *datasets (iterable of Datasets> Dataset> Datasets>Datasets> Dataset)*: Optional collection of **Datasets** whose **Labels**>Labels>Labels>**Labels**>Labels are sought. If not provided, all **Labels** in this **Project** are returned.
  - *order_by (None or (Field, Field.Order))*: Ordering clause.

review_metrics(self, net_score)

Returns this **Project**'s review metrics.

- **Args**:
  - *net_score (None or Review.NetScore)*: Indicates desired metric.
- **Returns**: int, aggregation count of reviews for given net_score.

set_labeling_parameter_overrides(self, data)

Adds labeling parameter overrides to this project. Example:

```
>>> project.set_labeling_parameter_overrides([
>>>   (data_row_1, 2, 3), (data_row_2, 1, 4)])
```

- **Args**:
  - *data (iterable)*: An iterable of tuples. Each tuple must contain (**DataRow**, priority, numberOfLabels) for the new override.
- **Returns**: bool, indicates if the operation was a success.

setup(self, labeling_frontend, labeling_frontend_options)

Finalizes the **Project** setup.

- **Args**:
  - *labeling_frontend (LabelingFrontend)*: Which UI to use to label the data.
  - *labeling_frontend_options (dict or str)*: Labeling frontend options, a.k.a. project ontology. If given a `dict` it will be converted to `str` using `json.dumps`.

unset_labeling_parameter_overrides(self, data_rows)

Removes labeling parameter overrides to this project.

- **Args**:
  - *data_rows (iterable)*: An iterable of **DataRows**.
- **Returns:** bool, indicates if the operation was a success.

*update(self, **kwargs)*

Updates this DB object with new values. Values should be passed as key-value arguments with field names as keys:

```
>>> db_object.update(name="New name", title="A title")
```

- **Kwargs**: Key-value arguments defining which fields should be updated for which values. Keys must be field names in this DB object's type.
- **Raises**:
  - *InvalidAttributeError*: if there exists a key in `kwargs` that's not a field in this object type.

upsert_review_queue(self, quota_factor)

Reinitiates the review queue for this project.

- **Args:**
  - *quota_factor (float)*: Which part (percentage) of the queue to reinitiate. Between 0 and 1.

Dataset

Class labelbox.schema.dataset.Dataset (Updateable, Deletable).

A dataset is a collection of **DataRows**>DataRows. For example, if you have a CSV with 100 rows, you will have 1 **Dataset** and 100 **DataRows**.

Fields

- created_at (*DateTime*)
- description (*String*)
- name (*String*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- created_by (*User ToOne*)
- data_rows (*DataRow ToMany*)
- organization (*Organization ToOne*)
- projects (*Project ToMany*)

Object methods

create_data_row(self, **kwargs)

Creates a single **DataRow** belonging to this dataset.

```
>>> dataset.create_data_row(row_data="http://my_site.com/photos/img_01.jpg")
```

- **Kwargs**: Key-value arguments containing new `**DataRow**` data. At a minimum `kwargs` must contain `row_data`. The value for `row_data` is a string. If it is a path to an existing local file then it is uploaded to Labelbox's server. Otherwise it is treated as an external URL.
- **Raises**:
  - *InvalidQueryError*: If `**DataRow**.row_data` field value is not provided in `kwargs`.
  - *InvalidAttributeError*: in case the DB object type does not contain any of the field names given in `kwargs`.

create_data_rows(self, items)

Creates multiple **DataRow**>DataRow> DataRow>DataRow objects based on the given `items`. Each element in `items` can be either a `str` or a `dict`. If it is a `str`, then it is interpreted as a local file path. The file is uploaded to Labelbox and a **DataRow**> DataRow>DataRow referencing it is created. If an item is a `dict`, then it should map `**DataRow**>DataRow` fields (or their names) to values. At the minimum an `item` passed as a `dict` must contain a `**DataRow**.row_data` key and value.

```
>>> dataset.create_data_rows([
>>> {DataRow.row_data:"http://my_site.com/photos/img_01.jpg"},
>>> "path/to/file2.jpg"
>>> ])
```

- **Args**:
  - *items (iterable of (dict or str))*: See above for details.
- **Returns**: **Task>**Task representing the data import on the server side. The **Task** can be used for inspecting task progress and waiting until it's done.
- **Raises**:
  - *InvalidQueryError*: If the `items` parameter does not conform to the specification above or if the server did not accept the **DataRow**>DataRow creation request (unknown reason).
  - *ResourceNotFoundError*: If unable to retrieve the **Task** for the import process. This could imply that the import failed.
  - ***InvalidAttributeError***: If there are fields in `items` not valid for a **DataRow**.

data_row_for_external_id(self, external_id)

Convenience method for getting a single `**DataRow**` belonging to this `**Dataset**` that has the given `external_id`.

- **Args**:
  - *external_id (str)*: External ID of the sought `**DataRow**`.
- **Returns**: A single `**DataRow**` with the given ID.
- **Raises**:
  - *labelbox.exceptions.ResourceNotFoundError:* If there is no `**DataRows**>DataRow>DataRows>*DataRows*>DataRow` in this `DataSet` with the given external ID, or if there are multiple `**DataRows**` for it.

delete(self)

Deletes this DB object from the DB (server side). After a call to this you should not use this DB object anymore.

update(self, **kwargs)

Updates this DB object with new values. Values should be passed as key-value arguments with field names as keys:

>>> db_object.update(name="New name", title="A title")

- **Kwargs**: Key-value arguments defining which fields should be updated for which values. Keys must be field names in this DB object's type.
- **Raises**:
  - *InvalidAttributeError*: if there exists a key in `kwargs` that's not a field in this object type.

DataRow

Class labelbox.schema.data_row.DataRow (Updateable, BulkDeletable).

A **DataRows**>DataRow represents a single piece of data. For example, if you have a CSV with 100 rows, you will have 1 **Dataset** and 100 **DataRows**.

Fields

- created_at (*DateTime*)
- external_id (*String*)
- row_data (*String*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- created_by (*User ToOne*)
- dataset (*Dataset ToOne*)
- labels (*Label ToMany*)
- metadata (*AssetMetadata ToMany*)
- organization (*Organization ToOne*)
- predictions (*Prediction ToMany*)

Static methods

bulk_delete(data_rows)

Deletes all the given **DataRows.**

- Args:
  - *data_rows* (list of DataRows>DataRow >DataRows>DataRows >DataRow): The **DataRows** to delete.

Object methods

create_metadata(self, meta_type, meta_value)

Creates an asset metadata for this **DataRow**.

- **Args:**
  - *meta_type (str)*: Asset metadata type, must be one of: VIDEO, IMAGE, TEXT.
  - *meta_value (str)*: Asset metadata value.
- **Returns: AssetMetadata** DB object.

delete(self)

Deletes this DB object from the DB (server side). After a call to this you should not use this DB object anymore.

*update(self, **kwargs)*

Updates this DB object with new values. Values should be passed as key-value arguments with field names as keys:

>>> db_object.update(name="New name", title="A title")

- **Kwargs:** Key-value arguments defining which fields should be updated for which values. Keys must be field names in this DB object's type.
- **Raises**:
  - *InvalidAttributeError*: if there exists a key in `kwargs` that's not a field in this object type.

Label

Class labelbox.schema.label.Label (Updateable, BulkDeletable).

Label represents an assessment on a **DataRow.** For example one label could contain 100 bounding boxes (annotations)

Fields

- agreement (Float)
- benchmark_agreement (Float)
- is_benchmark_reference (Boolean)
- label (String)
- seconds_to_label (Float)
- uid (ID)

Relationships

- created_by (User ToOne)
- data_row (DataRow ToOne)
- project (Project ToOne)
- reviews (Review ToMany)

Static methods

bulk_delete(labels)

Deletes all the given **Labels**.

- **Args**:
  - *labels (list of Labels>Label> Labels>Labels>Label):* The **Labels** to delete.

Object methods

create_benchmark(self)

Creates a Benchmark for this **Label**.

- **Returns**: The newly created Benchmark.

*create_review(self, **kwargs)*

Creates a **Review** for this label.

- **Kwargs**: **Review**>Review attributes. At a

minimum a `**Review**.score` field value must be provided.

delete(self)

Deletes this DB object from the DB (server side). After a call to this you should not use this DB object anymore.

*update(self, **kwargs)*

Updates this DB object with new values. Values should be passed as key-value arguments with field names as keys:

>>> db_object.update(name="New name", title="A title")

- **Kwargs**: Key-value arguments defining which fields should be updated for which values. Keys must be field names in this DB object's type.
- **Raises**:
  - *InvalidAttributeError*: if there exists a key in `kwargs` that's not a field in this object type.

AssetMetaData

Class labelbox.schema.asset_metadata.AssetMetadata.

AssetMetadata is a datatype to provide extra context about an asset while labeling.

Constants

- VIDEO (str)
- IMAGE (str)
- TEXT (str)

Fields

- meta_type (String)
- meta_value (String)
- uid (ID)

Relationships

LabelingFrontEnd

Class labelbox.schema.labeling_frontend.LabelingFrontend.

Is a type representing an HTML / JavaScript UI that is used to generate labels. "Image Labeling" is the default Labeling Frontend that comes in every organization. You can create new labeling frontends for an organization.

Fields

- description (*String*)
- iframe_url_path (*String*)
- name (*String*)
- uid (*ID*)

Relationships

- projects (Project ToMany)

Task

Class labelbox.schema.task.Task.

Represents a server-side process that might take a longer time to process. Allows the **Task** state to be updated and checked on the client side.

Fields

- completion_percentage (*Float*)
- created_at (*DateTime*)
- name (*String*)
- status (*String*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- created_by (User ToOne)
- organization (Organization ToOne)

Object methods

refresh(self)

Refreshes **Task** data from the server.

wait_till_done(self, timeout_seconds=60)

Waits until the task is completed. Periodically queries the server to update the task attributes.

- **Args:**
  - *timeout_seconds (float)*: Maximum time this method can block, in seconds. Defaults to one minute.

Webhook

Class labelbox.schema.webhook.Webhook (Updateable).

Represents a server-side rule for sending notifications to a web-server whenever one of several predefined actions happens within a context of a **Project** or an **Organization**.

Constants

- ACTIVE (*str*)
- INACTIVE (*str*)
- REVOKED (*str*)
- LABEL_CREATED (*str*)
- LABEL_UPDATED (*str*)
- LABEL_DELETED (*str*)

Fields

- created_at (*DateTime*)
- status (*String*)
- topics (*String*)
- uid (*ID*)
- updated_at (*DateTime*)
- url (*String*)

Relationships

- created_by (User ToOne)
- organization (Organization ToOne)
- project (Project () ToOne)

Static methods

create(client, topics, url, secret, project)

Creates a **Webhook**.

- **Args**:
  - *client (Client)*: The Labelbox client used to connect to the server.
  - *topics (list of str)*: A list of topics this **Webhook** should get notifications for.
  - *url (str)*: The URL to which notifications should be sent by the Labelbox server.
  - *secret (str)*: A secret key used for signing notifications.
  - *project (Project or None)*: The project for which notifications should be sent. If None notifications are sent for all events in your organization.
- **Returns**: A newly created **Webhook**.

Object methods

update(self, topics, url, status)

Updates this **Webhook**.

- **Args**:
  - *topics (list of str)*: The new topics value, optional.
  - *url (str)*: The new URL value, optional.
  - *status (str)*: The new status value, optional.

User

Class labelbox.schema.user.User.

A **User** is a registered Labelbox user (for example you) associated with data they create or import and an **Organization** they belong to.

Fields

- created_at (*DateTime*)
- email (*String*)
- intercom_hash (*String*)

- is_external_user (*Boolean*)
- is_viewer (*Boolean*)
- nickname (*String*)
- name (*String*)
- picture (*String*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- created_tasks (Task ToMany)
- organization (Organization ToOne)
- projects (Project ToMany)

Organization

Class labelbox.schema.organization.Organization.

An **Organization**>Organization >Organization >Organization is a group of **User**>Users >Users> Users associated with data created by **User**>Users >Users within that **Organization**>Organization >Organization. Typically all **User**>Users within an **Organization**>Organization have access to data created by any **User** in the same **Organization**.

Fields

- created_at (*DateTime*)
- name (*String*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- projects (*Project ToMany*)
- users (*User ToMany*)
- webhooks (*Webhook ToMany*)

Review

Class labelbox.schema.review.Review (Updateable, Deletable).

Reviewing labeled data is a collaborative quality assurance technique. A **Review** object indicates the quality of the assigned **Label**. The aggregated review numbers can be obtained on a **Project** object.

Constants

- Enumeration NetScore
  - Negative
  - Zero
  - Positive

Fields

- created_at (*DateTime*)
- score (*Float*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- created_by (User ToOne)
- label (Label ToOne)
- organization (Organization ToOne)
- project (Project ToOne)

Object methods

delete(self)

Deletes this DB object from the DB (server side). After a call to this you should not use this DB object anymore.

update(self, **kwargs)

Updates this DB object with new values. Values should be passed as key-value arguments with field names as keys:

>>> db_object.update(name="New name", title="A title")

- **Kwargs**: Key-value arguments defining which fields should be updated for which values. Keys must be field names in this DB object's type.
- **Raises**:
  - *InvalidAttributeError*: if there exists a key

in `kwargs` that's not a field in this object type.

Prediction

Class labelbox.schema.prediction.Prediction.

A prediction created by a **PredictionModel**.

Fields

- agreement (*Float*)
- created_at (*DateTime*)
- label (*String*)
- uid (*ID*)
- updated_at (*DateTime*)

Relationships

- data_row (*DataRow ToOne*)
- organization (*Organization ToOne*)
- prediction_model (*PredictionModel ToOne*)
- project (*Project ToOne*)

PredictionModel

Class labelbox.schema.prediction.PredictionModel.

A prediction model represents a specific version of a model.

Fields

- created_at (*DateTime*)
- name (*String*)
- slug (*String*)
- uid (*ID*)
- updated_at (*DateTime*)
- version (*Int*)

Relationships

- created_by (*User ToOne*)
- created_predictions (*Prediction ToMany*)
- organization (*Organization ToOne*)

LabelerPerformance

Class labelbox.schema.project.LabelerPerformance.

Named tuple containing info about a labeler's performance.

10 Error classes

LabelboxError

Class labelbox.exceptions.LabelboxError.

Base class for exceptions.

AuthenticationError

Class labelbox.exceptions.AuthenticationError.

Raised when an API key fails authentication.

AuthorizationError

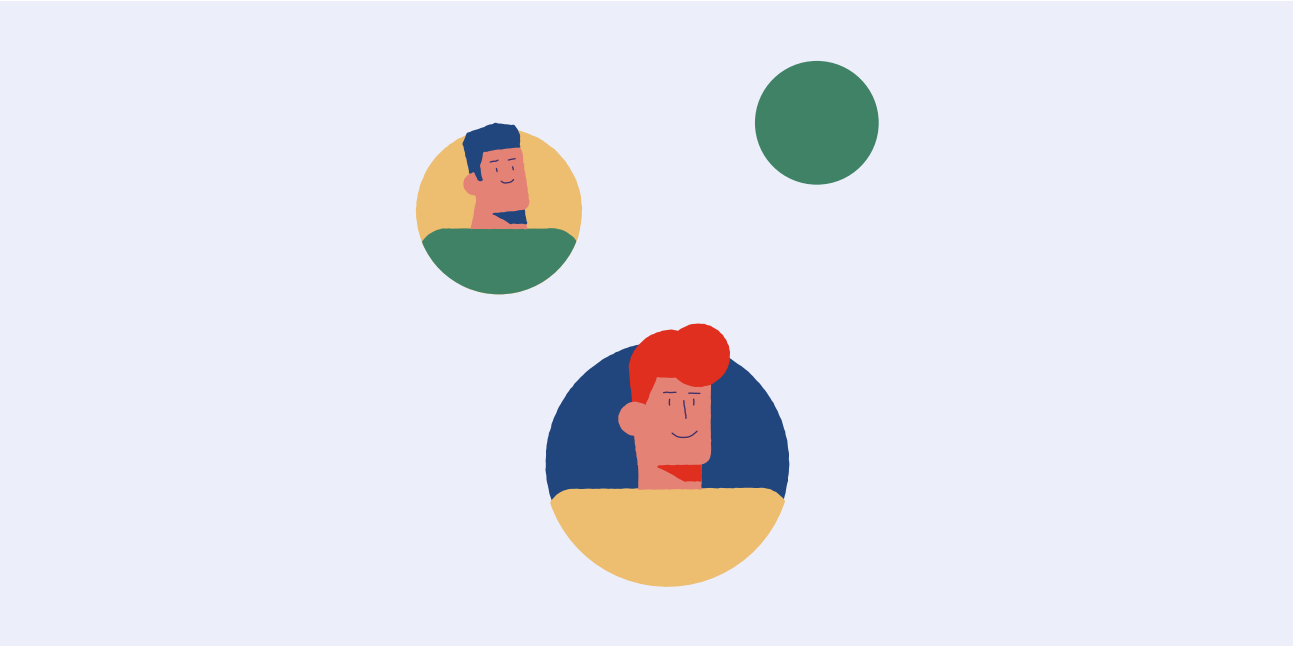
Class labelbox.exceptions.AuthorizationError.

Raised when a user is unauthorized to perform the given request.

ResourceNotFoundError

Class labelbox.exceptions.ResourceNotFoundError.

Exception raised when a given resource is not found.



ValidationFailedError

Class labelbox.exceptions.
ValidationFailedError.

Exception raised for when a GraphQL query fails validation (query cost, etc.) E.g. a query that is too expensive, or depth is too deep.

InvalidQueryError

Class labelbox.exceptions.InvalidQueryError.

Indicates a malconstructed or unsupported query (either by GraphQL in general or by Labelbox specifically). This can be the result of either client or server side query validation.

NetworkError

Class labelbox.exceptions.NetworkError.

Raised when an HTTPError occurs.

TimeoutError

Class labelbox.exceptions.TimeoutError.

Raised when a request times-out.

InvalidAttributeError

Class labelbox.exceptions.
InvalidAttributeError.

Raised when a field (name or Field instance) is not valid or found for a specific DB object type.

ApiLimitError

Class labelbox.exceptions.ApiLimitError.

Raised when the user performs too many requests in a short period of time.