Python

Overview

This page broadly covers the Weaviate Python client (v4 release). For usage information not specific to the Python client, such as code examples, see the relevant pages in the Weaviate documentation. Some frequently used sections are listed here for convenience.

Installation

MIGRATING FROM v3 TO v4

If you are migrating from the v3 client to the v4, see this dedicated guide.

The Python client library is developed and tested using Python 3.8+. It is available on PyPI.org, and can be installed with:

```
pip install -U weaviate-client  # For beta versions: `pip install --pre -U "weaviate-client==4.*"`
```

Requirements

gRPC

The v4 client uses remote procedure calls (RPCs) under-the-hood. Accordingly, a port for gRPC must be open to your Weaviate server.
docker-compose.yml example

WCS compatibility

The free (sandbox) tier of WCS is compatible with the v4 client as of 31 January, 2024. Sandboxes created before this date will not be compatible with the v4 client.

Weaviate server version

The v4 client requires Weaviate `1.23.7` or higher. Generally, we encourage you to use the latest version of the Python client *and* the Weaviate server.

High-level ideas

Helper classes

The client library provides numerous additional Python classes to provide IDE assistance and typing help. You can import them individually, like so:
from weaviate.classes.config import Property, ConfigFactory
from weaviate.classes.data import DataObject
from weaviate.classes.query import Filter
But it may be convenient to import the whole set of classes like this. You will see both usage styles in our documentation.
import weaviate.classes as wvc
For discoverability, the classes are arranged into submodules.
See the list of submodules

Connection termination

You must ensure your client connections are closed. You can use `client.close()`, or use a context manager to close client connections for you.

`client.close()` with `try` / `finally`

This will close the client connection when the `try` block is complete (or if an exception is raised).

```
import weaviate

client = weaviate.connect_to_local()  # Connect with default parameters

try:
  pass  # Do something with the client

finally:
  client.close()  # Ensure the connection is closed
```

Context manager

This will close the client connection when you leave the `with` block.

```
import weaviate

with weaviate.connect_to_local() as client:
  # Do something with the client
  pass
  # The connection is closed automatically when the context manager exits
```

Instantiate a client

There are multiple ways to connect to your Weaviate instance. To instantiate a client, use one of these styles:

- Python client v4 helper methods
- Python client v4 explicit connection
- Python client v3 style connection

Python client v4 helper functions

- `weaviate.connect_to_wcs()`
- `weaviate.connect_to_local()`
- `weaviate.connect_to_embedded()`
- `weaviate.connect_to_custom()`
- WCS
- Local
- Embedded
- Custom

```
import weaviate
import os

client = weaviate.connect_to_wcs(
  cluster_url=os.getenv("WCS_DEMO_URL"),  # Replace with your Weaviate Cloud URL
  auth_credentials=weaviate.auth.AuthApiKey(os.getenv("WCS_DEMO_RO_KEY")),  # Replace with your Weaviate Cloud key
  headers={'X-OpenAI-Api-key': os.getenv("OPENAI_APIKEY")}  # Replace with your OpenAI API key
)
```

The `v4` client helper functions provide some optional parameters to customize your client.

- Specify external API keys
- Specify connection timeout values
- Specify authentication details

External API keys

To add API keys for services such as Cohere or OpenAI, use the `headers` parameter.

```
import weaviate
import os

client = weaviate.connect_to_local(
```

```
  headers={"X-OpenAI-Api-key": os.getenv("OPENAI_APIKEY")}
)
```

Timeout values

You can set timeout values, in seconds, for the client. Use the `Timeout` class to configure the timeout values for initialization checks as well as query and insert operations.

```
import weaviate
from weaviate.classes.init import AdditionalConfig, Timeout

client = weaviate.connect_to_local(
  port=8080,
  grpc_port=50051,
  additional_config=AdditionalConfig(
    timeout=Timeout(init=2, query=45, insert=120)  # Values in seconds
  )
)
```

TIMEOUTS ON `generate` (RAG) QUERIES

If you see errors while using the `generate` submodule, try increasing the query timeout values (`Timeout(query=60)`).

The `generate` submodule uses a large language model to generate text. The submodule is dependent on the speed of the language model and any API that serves the language model.

Increase the timeout values to allow the client to wait longer for the language model to respond.

Authentication

Some of the `connect` helper functions take authentication credentials. For example, `connect_to_wcs` accepts a WCS API key or OIDC authentication credentials.

- API Key
- OIDC Credentials

```
import weaviate
import os

client = weaviate.connect_to_wcs(
  cluster_url=os.getenv("WCS_DEMO_URL"),  # Replace with your Weaviate Cloud URL
  auth_credentials=weaviate.auth.AuthApiKey(os.getenv("WCS_DEMO_RO_KEY")),  # Replace with your Weaviate Cloud key
```

```
  headers={'X-OpenAI-Api-key': os.getenv("OPENAI_APIKEY")}  # Replace with your
OpenAI API key
)
```

For OIDC authentication with the Client Credentials flow, use the `AuthClientCredentials` class.

For OIDC authentication with the Refresh Token flow, use the `AuthBearerToken` class.

If the helper functions do not provide the customization you need, use the `WeaviateClient` class to instantiate the client.

Python client v4 explicit connection

If you need to pass custom parameters, use the `weaviate.WeaviateClient` class to instantiate a client. This is the most flexible way to instantiate the client object.

When you instantiate a connection directly, you have to call the `.connect()` method to connect to the server.

```
import weaviate
from weaviate.connect import ConnectionParams
from weaviate.classes.init import AdditionalConfig, Timeout
import os

client = weaviate.WeaviateClient(
  connection_params=ConnectionParams.from_params(
      http_host="localhost",
      http_port="8099",
      http_secure=False,
      grpc_host="localhost",
      grpc_port="50052",
      grpc_secure=False,
  ),
  auth_client_secret=weaviate.auth.AuthApiKey("secr3tk3y"),
  additional_headers={
      "X-OpenAI-Api-Key": os.getenv("OPENAI_APIKEY")
  },
  additional_config=AdditionalConfig(
      timeout=Timeout(init=2, query=45, insert=120),  # Values in seconds
  ),
)

client.connect()  # When directly instantiating, you need to connect manually
```

Python client v3 API

To create an older, `v3` style `Client` object, use the `weaviate.Client` class. This method available for backwards compatibility. Where possible, use a client v4 connection.
To create a `v3` style client, refer to the v3 client documentation.

Initial connection checks

When establishing a connection to the Weaviate server, the client performs a series of checks. These includes checks for the server version, and to make sure that the REST and gRPC ports are available.
You can set `skip_init_checks` to `True` to skip these checks.

```
import weaviate

client = weaviate.connect_to_local(
  skip_init_checks=True
)
```

In most cases, you should use the default `False` setting for `skip_init_checks`.
However, setting `skip_init_checks=True` may be a useful temporary measure if you have connection issues.
For additional connection configuration, see Timeout values.

Batch imports

The `v4` client offers two ways to perform batch imports. From the client object directly, or from the collection object.
We recommend using the collection object to perform batch imports of single collections or tenants. If you are importing objects across many collections, such as in a multi-tenancy configuration, using `client.batch` may be more convenient.

Batch sizing

There are three methods to configure the batching behavior. They are `dynamic`, `fixed_size` and `rate_limit`.

Method	Description	When to use
`dynamic`	The batch size and the number of concurrent requests are dynamically adjusted on-the-fly during import, depending on the server load.	Recommended starting point.

`fixed_size`	The batch size and number of concurrent requests are fixed to sizes specified by the user.	When you want to specify fixed parameters.
`rate_limit`	The number of objects sent to Weaviate is rate limited (specified as n_objects per minute).	When you want to avoid hitting third-party vectorization API rate limits.

Usage

We recommend using a context manager as shown below.
These methods return completely localized context managers. Accordingly, attributes of one batch such as `failed_objects` and `failed_references` will not be included in any subsequent calls.

- Dynamic
- Fixed Size
- Rate limited

```
import weaviate

client = weaviate.connect_to_local()

try:
  with client.batch.dynamic() as batch:  # or <collection>.batch.dynamic()
    # Batch import objects/references - e.g.:
    batch.add_object(properties={"title": "Multitenancy"}, collection="WikiArticle",
uuid=src_uuid)
    batch.add_object(properties={"title": "Database schema"}, collection="WikiArticle",
uuid=tgt_uuid)
    batch.add_reference(from_collection="WikiArticle", from_uuid=src_uuid,
from_property="linkedArticle", to=tgt_uuid)

finally:
  client.close()
```

If the background thread that is responsible for sending the batches raises an exception during batch processing, the error is raised to the main thread.

Error handling

During a batch import, any failed objects or references will be stored for retrieval. Additionally, a running count of failed objects and references is maintained.

The counter can be accessed through `batch.number_errors` within the context manager.
A list of failed objects can be obtained through `batch.failed_objects` and a list of failed references can be obtained through `batch.failed_references`.
Note that these lists are reset when a batching process is initialized. So make sure to retrieve them before starting a new batch import block.

```
import weaviate
import weaviate.classes as wvc

client = weaviate.connect_to_local()

try:
  # ===== First batch import block =====
  with client.batch.rate_limit(requests_per_minute=600) as batch:  # or <collection>.batch.rate_limit()
      # Batch import objects/references
      for i in source_iterable:  # Some insertion loop
          if batch.number_errors > 10:  # Monitor errors during insertion
              # Break or raise an exception
              pass
  # Note these are outside the `with` block - they are populated after the context manager exits
  failed_objs_a = client.batch.failed_objects  # Get failed objects from the first batch import
  failed_refs_a = client.batch.failed_references  # Get failed references from the first batch import

  # ===== Second batch import block =====
  # This will clear the failed objects/references
  with client.batch.rate_limit(requests_per_minute=600) as batch:  # or <collection>.batch.rate_limit()
      # Batch import objects/references
      for i in source_iterable:  # Some insertion loop
          if batch.number_errors > 10:  # Monitor errors during insertion
              # Break or raise an exception
              pass
  # Note these are outside the `with` block - they are populated after the context manager exits
  failed_objs_b = client.batch.failed_objects  # Get failed objects from the second batch import
  failed_refs_b = client.batch.failed_references  # Get failed references from the second batch import
```

```
finally:
  client.close()
```

Batch vectorization

ADDED IN `v1.25`.

Some model providers provide batch vectorization APIs, where each request can include multiple objects.
From Weaviate `v1.25.0`, a batch import automatically makes use of the model providers' batch vectorization APIs where available. This reduces the number of requests to the model provider, improving throughput.
The client automatically handles vectorization if you set the vectorizer when you create the client connection for your batch import.

- Create a client

To add or modify the vectorization settings, update the client connection. This example adds multiple vectorizers:

- Cohere. Set the service API key. Set the request rate.
- OpenAI. Set the service API key. Set the base URL.
- VoyageAI. Set the service API key.
    - Modify the client
- from weaviate.classes.config import Integrations
-

```
integrations = [
    # Each model provider may expose different parameters
    Integrations.cohere(
        api_key=cohere_key,
        requests_per_minute_embeddings=rpm_embeddings,
    ),
    Integrations.openai(
        api_key=openai_key,
        requests_per_minute_embeddings=rpm_embeddings,
        tokens_per_minute_embeddings=tpm_embeddings,   # e.g. OpenAI also exposes tokens per minute for embeddings
    ),
]
client.integrations.configure(integrations)
```

Working with collections

Instantiate a collection

You can instantiate a collection object by creating a collection, or by retrieving an existing collection.

- Create a collection
- With cross-references
- Get a collection

```
import weaviate
import weaviate.classes.config as wvcc

client = weaviate.connect_to_local()

try:
  # Note that you can use `client.collections.create_from_dict()` to create a collection from a v3-client-style JSON object
  collection = client.collections.create(
    name="TestArticle",
    vectorizer_config=wvcc.Configure.Vectorizer.text2vec_cohere(),
    generative_config=wvcc.Configure.Generative.cohere(),
    properties=[
      wvcc.Property(
        name="title",
        data_type=wvcc.DataType.TEXT
      )
    ]
  )

finally:
  client.close()
```

Collection submodules

Operations in the `v4` client are grouped into submodules. The key submodules for interacting with objects are:

- `data`: CUD operations (read operations are in `query`)
- `batch`: Batch import operations
- `query`: Search operations

- `generate`: Retrieval augmented generation operations
    - Build on top of `query` operations
- `aggregate`: Aggregation operations

`data`

The `data` submodule contains all object-level CUD operations, including:

- `insert` for creating objects.
    - This function takes the object properties as a dictionary.
- `insert_many` for batch creating multiple objects.
    - This function takes the object properties as a dictionary or as a `DataObject` instance.
- `update` for updating objects (for `PATCH` operations).
- `replace` for replacing objects (for `PUT` operations).
- `delete_by_id` for deleting objects by ID.
- `delete_many` for batch deletion.
- `reference_xxx` for reference operations, including `reference_add`, `reference_add_many`, `reference_update` and `reference_delete`.

See some examples below. Note that each function will return varying types of objects.
`insert_many` SENDS ONE REQUEST

As of `4.4b1`, `insert_many` sends one request for the entire function call. A future release may send multiple requests as batches.

- Insert
- Insert many
- Delete by id
- Delete many

```
questions = client.collections.get("JeopardyQuestion")

new_uuid = questions.data.insert(
    properties={
        "question": "This is the capital of Australia."
    },
    references={  # For adding cross-references
        "hasCategory": target_uuid
    }
)
```

`insert_many` with DataObjects

The `insert_many` function takes a list of `DataObject` instances or a list of dictionaries. This is useful if you want to specify additional information to the properties, such as cross-references, object uuid, or a custom vector.

```
from weaviate.util import generate_uuid5

questions = client.collections.get("JeopardyQuestion")

data_objects = list()
for i in range(5):
    properties = {"question": f"Test Question {i+1}"}
    data_object = wvc.data.DataObject(
        properties=properties,
        uuid=generate_uuid5(properties)
    )
    data_objects.append(data_object)

response = questions.data.insert_many(data_objects)
```

Cross-reference creation

Cross-references should be added under a `references` parameter in the relevant function/method, with a structure like:

```
{
  "<REFERENCE_PROPERTY_NAME>": "<TARGET_UUID>"
}
```

For example:

```
from weaviate.util import generate_uuid5

questions = client.collections.get("JeopardyQuestion")

data_objects = list()
for i in range(5):
    properties = {"question": f"Test Question {i+1}"}
    data_object = wvc.data.DataObject(
        properties=properties,
        references={
            "hasCategory": target_uuid
        },
        uuid=generate_uuid5(properties)
    )
    data_objects.append(data_object)
```

```
response = questions.data.insert_many(data_objects)
```

Using the `properties` parameter to add references is deprecated and will be removed in the future.

`query`

The `query` submodule contains all object-level query operations, including `fetch_objects` for retrieving objects without additional search parameters, `bm25` for keyword search, `near_<xxx>` for vector search operators, `hybrid` for hybrid search and so on.

These queries return a `_QueryReturn` object, which contains a list of `_Object` objects.

- BM25
- Near text

```
questions = client.collections.get("JeopardyQuestion")
response = questions.query.bm25(
    query="animal",
    limit=2
)

for o in response.objects:
    print(o.properties)  # Object properties
```

Queries with custom returns

You can further specify:

- Whether to include the object vector (via `include_vector`)
    - Default is `False`
- Which properties to include (via `return_properties`)
    - All properties are returned by default
- Which references to include (via `return_references`)
- Which metadata to include
    - No metadata is returned by default

Each object includes its UUID as well as all properties by default.

For example:

- Default
- Customized returns

```
questions = client.collections.get("JeopardyQuestion")
response = questions.query.bm25(
```

```
    query="animal",
    limit=2
)

for o in response.objects:
    print(o.properties)  # All properties by default
    print(o.references)  # References not returned by default
    print(o.uuid)  # UUID included by default
    print(o.vector)  # No vector
    print(o.metadata)  # No metadata
```

`query` + group by

Results of a query can be grouped by a property as shown here.
The results are organized by both their individual objects as well as the group.

- The `objects` attribute is a list of objects, each containing a `belongs_to_group` property to indicate which group it belongs to.
- The `group` attribute is a dictionary with each key indicating the value of the group, and the value being a list of objects belonging to that group.

```
questions = client.collections.get("JeopardyQuestion")
response = questions.query.near_text(
    query="animal",
    distance=0.2,
    group_by=wvc.query.GroupBy(
        prop="points",
        number_of_groups=3,
        objects_per_group=5
    )
)

for k, v in response.groups.items():  # View by group
    print(k, v)

for o in response.objects:  # View by object
    print(o)
```

`generate`

The RAG / generative search functionality is a two-step process involving a search followed by prompting a large language model. Therefore, function names are shared across the

`query` and `generate` submodules, with additional parameters available in the `generate` submodule.

- Generate
- Query

```
questions = client.collections.get("JeopardyQuestion")
response = questions.generate.bm25(
    query="animal",
    limit=2,
    grouped_task="What do these animals have in common?",
    single_prompt="Translate the following into French: {answer}"
)

print(response.generated)  # Generated text from grouped task
for o in response.objects:
    print(o.generated)  # Generated text from single prompt
    print(o.properties)  # Object properties
```

Outputs of the `generate` submodule queries include `generate` attributes at the top level for the `grouped_task` tasks, while `generate` attributes attached with each object contain results from `single_prompt` tasks.

`aggregate`

To use the `aggregate` submodule, supply one or more ways to aggregate the data. For example, they could be by a count of objects matching the criteria, or by a metric aggregating the objects' properties.

- Count
- Metric

```
from weaviate.classes.query import Filter

questions = client.collections.get("JeopardyQuestion")
response = questions.aggregate.over_all(
    filters=Filter.by_property(name="question").like("*animal*"),
    total_count=True
)

print(response.total_count)
```

`aggregate` + group by

Results of a query can be grouped and aggregated as shown here.

The results are organized the group, returning a list of groups.

```
from weaviate.classes.aggregate import GroupByAggregate

questions = client.collections.get("JeopardyQuestion")
response = questions.aggregate.near_text(
    query="animal",
    distance=0.2,
    group_by=GroupByAggregate(prop="points"),
    return_metrics=wvc.query.Metrics("points").integer(mean=True)
)

for o in response.groups:
    print(o)
```

Collection iterator (`cursor` API)

The `v4` client adds a Pythonic iterator method for each collection. This wraps the `cursor` API and allows you to iterate over all objects in a collection.

This example fetches all the objects, and their properties, from the `questions` collection.

```
all_objects = [question for question in questions.iterator()]
```

You can specify which properties to retrieve. This example fetches the `answer` property.

```
all_object_answers = [question for question in questions.iterator(return_properties=["answer"])]
```

You can also specify which metadata to retrieve. This example fetches the `creation_time` metadata.

```
all_object_ids = [question for question in questions.iterator(return_metadata=wvc.query.MetadataQuery(creation_time=True))]  # Get selected metadata
```

Since the `cursor` API requires the object UUID for indexing, the `uuid` metadata is always retrieved.

You can also get the size of the collection by using the built-in `len` function.

```
articles = client.collections.get("Article")
print(len(articles))
```

Data model and generics

You can choose to provide a generic type to a query or data operation. This can be beneficial as the generic class is used to extract the return properties and statically type the response.

```
from typing import TypedDict
```

```
questions = client.collections.get("JeopardyQuestion")

class Question(TypedDict):
    question: str
    answer: str
    points: int

response = questions.query.fetch_objects(
    limit=2,
    return_properties=Question,  # Your generic class is used to extract the return properties and statically type the response
    return_metadata=wvc.query.MetadataQuery(creation_time=True)  # MetaDataQuery object is used to specify the metadata to be returned in the response
)
```

Migration guides

MIGRATING FROM `v3` TO `v4`

If you are migrating from the `v3` client to the `v4`, see this dedicated guide.

Beta releases

Migration guides - beta releases

Best practices and notes

Exception handling

The client library raises exceptions for various error conditions. These include, for example:

- `weaviate.exceptions.WeaviateConnectionError` for failed connections.
- `weaviate.exceptions.WeaviateQueryError` for failed queries.
- `weaviate.exceptions.WeaviateBatchError` for failed batch operations.
- `weaviate.exceptions.WeaviateClosedClientError` for operations on a closed client.

Each of these exceptions inherit from `weaviate.exceptions.WeaviateBaseError`, and can be caught using this base class, as shown below.

```
try:
    collection = client.collections.get("NonExistentCollection")
```

```
    collection.query.fetch_objects(limit=2)
except weaviate.exceptions.WeaviateBaseError as e:
    print(f"Caught a Weaviate error: {e.message}")
```

You can review this module which defines the exceptions that can be raised by the client library.

The client library doc strings also provide information on the exceptions that can be raised by each method. You can view these by using the `help` function in Python, by using the `?` operator in Jupyter notebooks, or by using an IDE, such as hover-over tooltips in VSCode.

Thread-safety

While the Python client is fundamentally designed to be thread-safe, it's important to note that due to its dependency on the `requests` library, complete thread safety isn't guaranteed.

This is an area that we are looking to improve in the future.

THREAD SAFETY

The batching algorithm in our client is not thread-safe. Keep this in mind to help ensure smoother, more predictable operations when using our Python client in multi-threaded environments.

If you are performing batching in a multi-threaded scenario, ensure that only one of the threads is performing the batching workflow at any given time. No two threads can use the same `client.batch` object at one time.

Response object structure

Each query response object typically include multiple attributes. Consider this query.

```
questions = client.collections.get("JeopardyQuestion")
response = questions.generate.near_text(
    query="history",
    limit=2,
    single_prompt="Translate this into French {question}",
    grouped_task="Summarize this into a sentence",
    return_metadata=wvc.query.MetadataQuery(
        distance=True,
        creation_time=True
    )
)

print("Grouped Task generated outputs:")
print(response.generated)
for o in response.objects:
```

```
    print(f"Outputs for object {o.uuid}")
    print(f"Generated text:")
    print(o.generated)
    print(f"Properties:")
    print(o.properties)
    print(f"Metadata")
    print(o.metadata)
```

Each response includes attributes such as `objects` and `generated`. Then, each object in `objects` include multiple attributes such as `uuid`, `vector`, `properties`, `references`, `metadata` and `generated`.

_GenerativeReturn(objects=[_GenerativeObject(uuid=UUID('61e29275-8f53-5e28-a355-347d45a847b3'), metadata=_MetadataReturn(creation_time=datetime.datetime(2024, 1, 2, 18, 3, 7, 475000, tzinfo=datetime.timezone.utc), last_update_time=None, distance=0.19253945350646973, certainty=None, score=None, explain_score=None, is_consistent=None, rerank_score=None), properties={'points': 1000.0, 'answer': 'Daniel Boorstein', 'air_date': datetime.datetime(1990, 3, 26, 0, 0, tzinfo=datetime.timezone.utc), 'round': 'Double Jeopardy!', 'question': 'This historian & former Librarian of Congress was teaching history at Harvard while studying law at Yale'}, references=None, vector=None, generated="Cet historien et ancien bibliothécaire du Congrès enseignait l'histoire à Harvard tout en étudiant le droit à Yale."), _GenerativeObject(uuid=UUID('e987d1a1-2599-5dd8-bd22-4f3b0338539a'), metadata=_MetadataReturn(creation_time=datetime.datetime(2024, 1, 2, 18, 3, 8, 185000, tzinfo=datetime.timezone.utc), last_update_time=None, distance=0.193121075630188, certainty=None, score=None, explain_score=None, is_consistent=None, rerank_score=None), properties={'points': 400.0, 'air_date': datetime.datetime(2007, 5, 11, 0, 0, tzinfo=datetime.timezone.utc), 'answer': 'an opinion', 'round': 'Jeopardy!', 'question': 'This, a personal view or belief, comes from the Old French for "to think"'}, references=None, vector=None, generated='Ceci, une opinion personnelle ou une croyance, provient du vieux français signifiant "penser".')], generated='Daniel Boorstein, a historian and former Librarian of Congress, taught history at Harvard while studying law at Yale, and an opinion is a personal view or belief derived from the Old French word for "to think".')

To limit the response payload, you can specify which properties and metadata to return.

Input argument validation

The client library performs input argument validation by default to make sure that the input types match the expected types.

You can disable this validation to improve performance. You can do this by setting the `skip_argument_validation` parameter to `True` when you instantiate a collection object, with `collections.get`, or with `collections.create` for example.

```
# Configure the `performant_articles` to skip argument validation on its methods
performant_articles = client.collections.get("Article", skip_argument_validation=True)
```

This may be useful in cases where you are using the client library in a production environment, where you can be confident that the input arguments are typed correctly.

Tab completion in Jupyter notebooks

If you use a browser to run the Python client with a Jupyter notebook, press `Tab` for code completion while you edit. If you use VSCode to run your Jupyter notebook, press `control` + `space` for code completion.

Raw GraphQL queries

To provide raw GraphQL queries, you can use the `client.graphql_raw_query` method (previously `client.query.raw` in the `v3` client). This method takes a string as input.

Code examples & resources

Usage information for various operations and features can be found throughout the Weaviate documentation.
Some frequently used sections are the how-to guides for Managing data and Queries. The how-to guides include concise examples for common operations.
In particular, check out the pages for:

- Client instantiation,
- Manage collections,
- Batch import
- Cross-reference
- Basic search
- Similarity search
- Filters

The Weaviate API reference pages for search and REST may also be useful starting points.