Quickstart Tutorial

license BSD-3-Clause | stackoverflow #weaviate | github issues | release v1.20.1 | downloads 2,343,000 | go report A+

Overview

Welcome. Here, you'll get a quick taste of Weaviate in ~20 minutes.

You will:

- Build a vector database, and
- Query it with *semantic search*.

> OBJECT VECTORS
>
> With Weaviate, you have options to:
>
> - Have **Weaviate create vectors**, or
> - Specify **custom vectors**.
>
> This tutorial demonstrates both methods.

Source data

We will use a (tiny) dataset of quizzes.

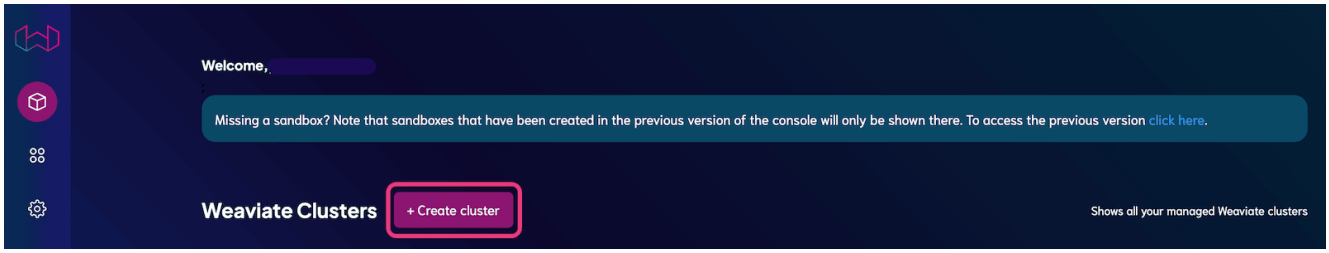
What data are we using?

Create an instance

First, create a Weaviate database.

1. Go to the WCS Console, and
    i. Click `Sign in with the Weaviate Cloud Services`.
    ii. If you don't have a WCS account, click on `Register`.
2. Sign in with your WCS username and password.
3. Click `Create cluster`.

TO CREATE A WCS INSTANCE:

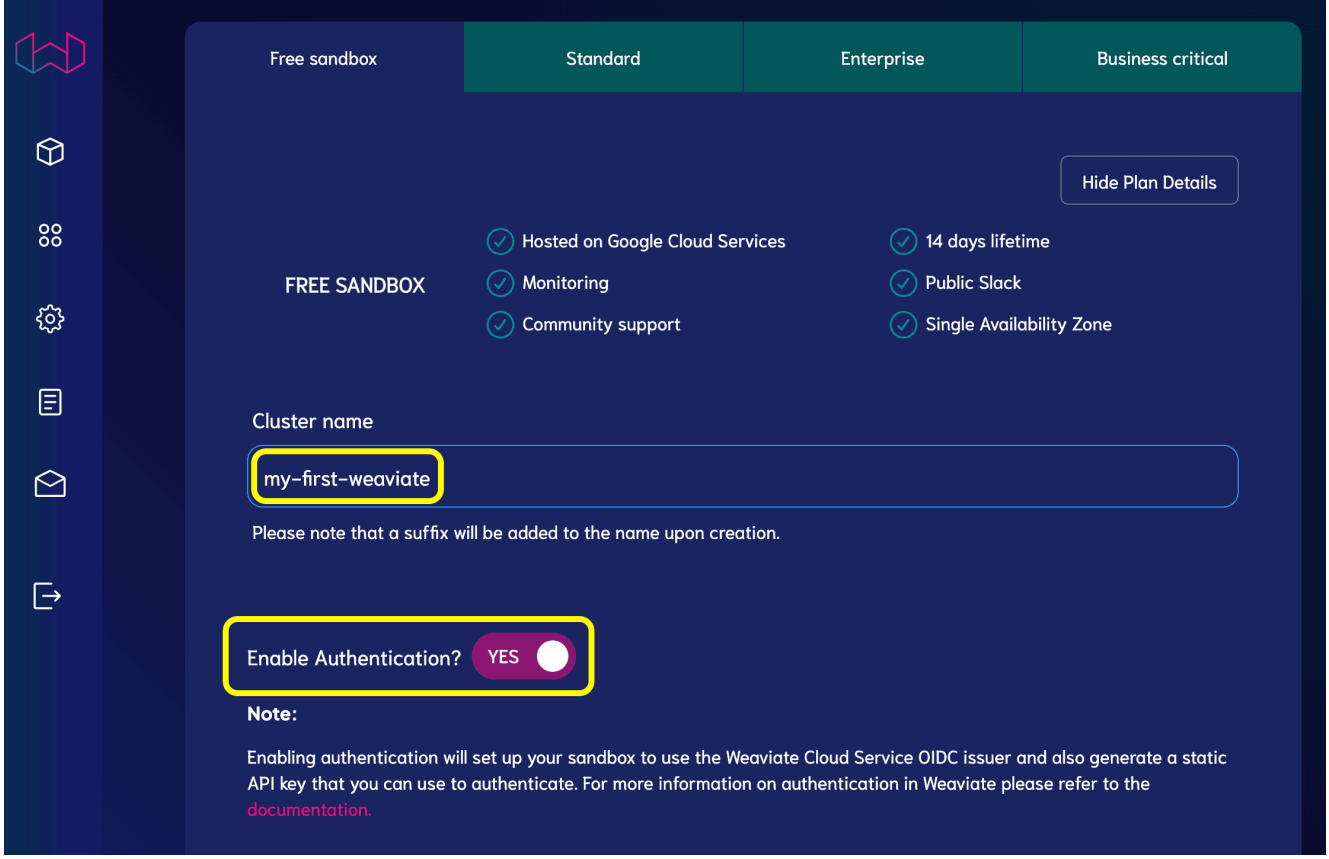


Can I use another method?

Then:

1. Select the `Free sandbox` tier.
2. Provide a *Cluster name*.
3. Set *Enable Authentication?* to `YES`.

YOUR SELECTIONS SHOULD LOOK LIKE THIS:



Click `Create`. This will take ~2 minutes and you'll see a tick ✔ when finished.

Note your cluster details

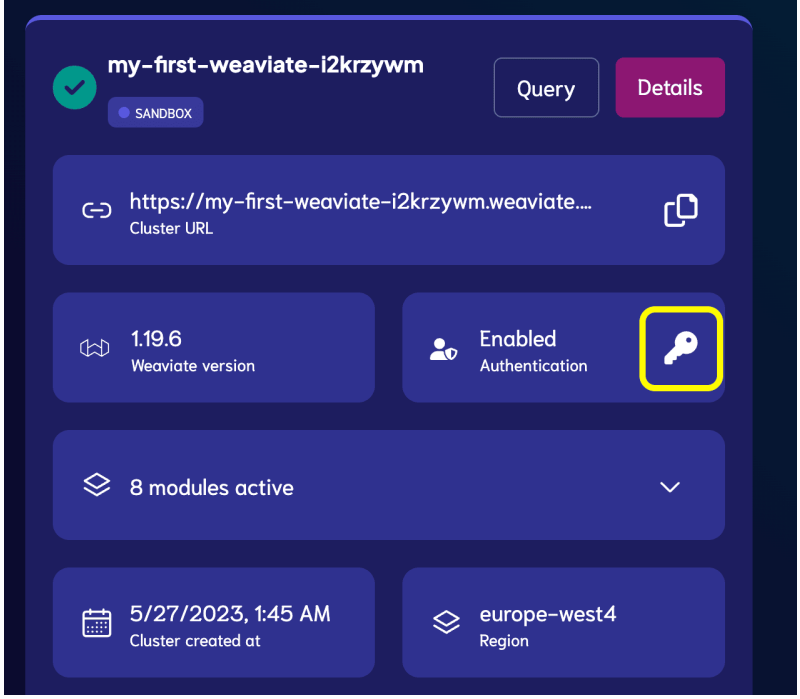
You will need:

- The Weaviate URL, and
- Authentication details (Weaviate API key).

Click `Details` to see them.

For the Weaviate API key, click on the 🔑 button.

YOUR WCS CLUSTER DETAILS SHOULD LOOK LIKE THIS:



Install a client library

We suggest useing a Weaviate client. To install your preferred client ⬇:

INSTALL CLIENT LIBRARIES

Python **TypeScript/JavaScript** **Go** **Java**

Add `weaviate-client` to your Python environment with `pip`:

```
$ pip install weaviate-client
```

Connect to Weaviate

From the `Details` tab in WCS, get:

- The Weaviate **API key**, and
- The Weaviate **URL**.

And because we will use the Hugging Face inference API to generate vectors, you need:

- A Hugging Face **inference API key**.

So, instantiate the client as follows:

Python TypeScript Go Curl

```
import weaviate
import json

client = weaviate.Client(
    url = "https://some-endpoint.weaviate.network",  # Replace with your endpoint
    auth_client_secret=weaviate.AuthApiKey(api_key="YOUR-WEAVIATE-API-KEY"),  # Replace w/ your Weaviate instance API key
    additional_headers = {
        "X-HuggingFace-Api-Key": "YOUR-HUGGINGFACE-API-KEY"  # Replace with your inference API key
    }
)
```

Now you are connected to your Weaviate instance!

Define a class

Next, we define a data collection (a "class" in Weaviate) to store objects in:

Python TypeScript Go Curl

```
class_obj = {
    "class": "Question",
    "vectorizer": "text2vec-huggingface",  # If set to "none" you must
```

```
always provide vectors yourself. Could be any other "text2vec-*" also.
    "moduleConfig": {
        "text2vec-huggingface": {
            "model": "sentence-transformers/all-MiniLM-L6-v2",  # Can be
any public or private Hugging Face model.
            "options": {
                "waitForModel": True
            }
        }
    }
}

client.schema.create_class(class_obj)
```

▸ What if I want to use a different vectorizer module?

This creates a class `Question`, tells Weaviate which `vectorizer` to use, and sets the `moduleConfig` for the vectorizer.

IS A `vectorizer` SETTING MANDATORY?

- No. You always have the option of providing vector embeddings yourself.
- Setting a `vectorizer` gives Weaviate the option of creating vector embeddings for you.
  - If you do not wish to, you can set this to `none`.

Now you are ready to add objects to Weaviate.

Add objects

We'll add objects to our Weaviate instance using a **batch import** process.

▸ Why use batch imports?

First, you will use the `vectorizer` to create object vectors.

Option 1: `vectorizer`

The code below imports object data without specifying a vector. This causes Weaviate to use the `vectorizer` defined for the class to create a vector embedding for each object.

Python TypeScript Go Curl

```python
# Load data
import requests
url = 'https://raw.githubusercontent.com/weaviate-tutorials/quickstart/main/data/jeopardy_tiny.json'
resp = requests.get(url)
data = json.loads(resp.text)

# Configure a batch process
with client.batch(
    batch_size=100
) as batch:
    # Batch import all Questions
    for i, d in enumerate(data):
        print(f"importing question: {i+1}")

        properties = {
            "answer": d["Answer"],
            "question": d["Question"],
            "category": d["Category"],
        }

        client.batch.add_data_object(
            properties,
            "Question",
        )
```

The above code:

- Loads objects,
- Initializes a batch process, and
- Adds objects to the target class (`Question`) one by one.

Option 2: Custom `vectors`

Alternatively, you can also provide your own vectors to Weaviate.

Regardless of whether a `vectorizer` is set, if a vector is specified, Weaviate will use it to represent the object.

Python TypeScript

```python
# Load data
import requests
fname = "jeopardy_tiny_with_vectors_all-MiniLM-L6-v2.json"  # This file includes vectors, created using `all-MiniLM-L6-v2`
url = f'https://raw.githubusercontent.com/weaviate-tutorials/quickstart/main/data/{fname}'
resp = requests.get(url)
data = json.loads(resp.text)

# Configure a batch process
with client.batch(
    batch_size=100
) as batch:
    # Batch import all Questions
    for i, d in enumerate(data):
        print(f"importing question: {i+1}")

        properties = {
            "answer": d["Answer"],
            "question": d["Question"],
            "category": d["Category"],
        }

        custom_vector = d["vector"]
        client.batch.add_data_object(
            properties,
            "Question",
            vector=custom_vector  # Add custom vector
        )
```

▸ Custom vectors with a `vectorizer`

VECTOR != OBJECT PROPERTY

Do *not* specify object vectors as an object property. This will cause Weaviate to treat it as a regular property, rather than as a vector embedding.

Putting it together

The following code puts the above steps together. You can run it yourself to import the data into your Weaviate instance.

End-to-end code

Queries

Now, we can run queries.

Semantic search

Let's try a similarity search. We'll use `nearText` search to look for quiz objects most similar to `biology`.

Python TypeScript Go Curl

```
import weaviate
import json

client = weaviate.Client(
    url = "https://some-endpoint.weaviate.network",  # Replace with your endpoint
    auth_client_secret=weaviate.AuthApiKey(api_key="YOUR-WEAVIATE-API-KEY"),  # Replace w/ your Weaviate instance API key
    additional_headers = {
        "X-HuggingFace-Api-Key": "YOUR-HUGGINGFACE-API-KEY"  # Replace with your inference API key
    }
)

nearText = {"concepts": ["biology"]}

response = (
    client.query
```

```
    .get("Question", ["question", "answer", "category"])
    .with_near_text(nearText)
    .with_limit(2)
    .do()
)

print(json.dumps(response, indent=4))
```

You should see a result like this (these may vary per module/model used):


```
{
    "data": {
        "Get": {
            "Question": [
                {
                    "answer": "DNA",
                    "category": "SCIENCE",
                    "question": "In 1953 Watson & Crick built a model of
the molecular structure of this, the gene-carrying substance"
                },
                {
                    "answer": "Liver",
                    "category": "SCIENCE",
                    "question": "This organ removes excess glucose from
the blood & stores it as glycogen"
                }
            ]
        }
    }
}
```


The response includes a list of top 2 (due to the `limit` set) objects whose vectors are most similar to the word `biology`.

> WHY IS THIS USEFUL?
>
> Notice that even though the word `biology` does not appear anywhere, Weaviate returns biology-related entries.
>
> This example shows why vector searches are powerful. Vectorized data objects allow for searches based on degrees of similarity, as shown here.

Semantic search with a filter

You can add a Boolean filter to your example. For example, let's run the same search, but only look in objects that have a "category" value of "ANIMALS".

Python TypeScript Go Curl

```python
nearText = {"concepts": ["biology"]}

response = (
    client.query
    .get("Question", ["question", "answer", "category"])
    .with_near_text(nearText)
    .with_where({
        "path": ["category"],
        "operator": "Equal",
        "valueText": "ANIMALS"
    })
    .with_limit(2)
    .do()
)

print(json.dumps(response, indent=4))
```

You should see a result like this (these may vary per module/model used):

```
{
    "data": {
        "Get": {
            "Question": [
                {
                    "answer": "Elephant",
                    "category": "ANIMALS",
                    "question": "It's the only living mammal in the
order Proboseidea"
                },
                {
                    "answer": "the nose or snout",
                    "category": "ANIMALS",
                    "question": "The gavial looks very much like a
crocodile except for this bodily feature"
                }
            ]
        }
    }
}
```

The response includes a list of top 2 (due to the `limit` set) objects whose vectors are most similar to the word `biology` - but only from the "ANIMALS" category.

> WHY IS THIS USEFUL?
>
> Using a Boolean filter allows you to combine the flexibility of vector search with the precision of `where` filters.

Recap

Well done! You have:

- Created your own cloud-based vector database with Weaviate,
- Populated it with data objects,
  - Using an inference API, or
  - Using custom vectors,
- Performed text similarity searches.

Where next is up to you. We include a few links below - or you can check out the sidebar.

Troubleshooting & FAQs

We provide answers to some common questions, or potential issues below.

How to confirm class creation

See answer

If you see `Error: Name 'Question' already used as a name for an Object class`

See answer

How to confirm data import

See answer

If the `nearText` search is not working

See answer

Will my sandbox be deleted?

Note: Sandbox expiry & options

Next

You can choose your direction from here. For example, you can:

- Go through our guided Tutorials, like how to
  - build schemas,
  - import data,
  - query data and more.
- Find out how to do specific things like:
  - searches
- Read about important concepts/theory about Weaviate
- Read our references for:
  - Configuration
  - API
  - Modules
  - Client libraries

More Resources

If you can't find the answer to your question here, please look at the:

1. Frequently Asked Questions. Or,
2. Knowledge base of old issues. Or,
3. For questions: Stackoverflow. Or,
4. For more involved discussion: Weaviate Community Forum. Or,
5. We also have a Slack channel.

1 reaction

1

0 comments – *powered by giscus*

Edit this page