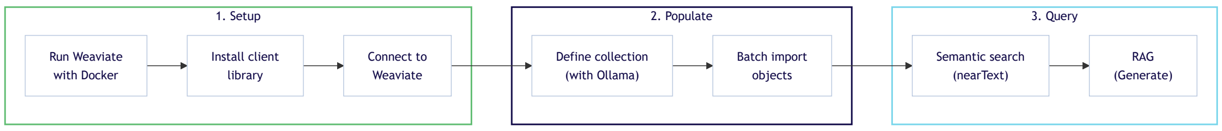
Quickstart: Locally hosted

Expected time: 30 minutes Prerequisites: None

WHAT YOU WILL LEARN

This quickstart shows you how to combine open-source Weaviate and Ollama to:

1. Set up a Weaviate instance. (10 minutes)
2. Add and vectorize your data. (10 minutes)
3. Perform a semantic search and retrieval augmented generation (RAG). (10 minutes)



Notes:

- The code examples here are self-contained. You can copy and paste them into your own environment to try them out.
- If you prefer to use cloud-based resources, see Quickstart: with cloud resources.

Prerequisites

Before we get started, install Docker and Ollama on your machine.

Then, download the `nomic-embed-text` and `llama3.2` models by running the following command:

```
ollama pull nomic-embed-text
ollama pull llama3.2
```

We will be running Weaviate and language models locally. We recommend that you use a modern computer with at least 8GB or RAM, preferably 16GB or more.

Step 1: Set up Weaviate

1.1 Create a Weaviate database

Save the following code to a file named `docker-compose.yml` in your project directory.

```
---
services:
  weaviate:
    command:
    - --host
    - 0.0.0.0
    - --port
    - '8080'
    - --scheme
    - http
    image: cr.weaviate.io/semitechnologies/weaviate:1.32.2
    ports:
    - 8080:8080
    - 50051:50051
    volumes:
    - weaviate_data:/var/lib/weaviate
    restart: on-failure:0
    environment:
      QUERY_DEFAULTS_LIMIT: 25
      AUTHENTICATION_ANONYMOUS_ACCESS_ENABLED: 'true'
      PERSISTENCE_DATA_PATH: '/var/lib/weaviate'
      ENABLE_API_BASED_MODULES: 'true'
      ENABLE_MODULES: 'text2vec-ollama,generative-ollama'
      CLUSTER_HOSTNAME: 'node1'
volumes:
  weaviate_data:
...
```

Run the following command to start a Weaviate instance using Docker:

```
docker-compose up -d
```

1.2 Install a client library

We recommend using a client library to work with Weaviate. Follow the instructions below to install one of the official client libraries, available in Python, JavaScript/TypeScript, Go, and Java.

Python JS/TS Go Java

Install the latest, Python client `v4`, by adding `weaviate-client` to your Python environment with `pip`:

```
pip install -U weaviate-client
```

1.3: Connect to Weaviate

Now you can connect to your Weaviate instance.

The example below shows how to connect to Weaviate and perform a basic operation, like checking the cluster status.

Python JS/TS Go Java Curl

quickstart_check_readiness.py

```python
import weaviate

client = weaviate.connect_to_local()

print(client.is_ready())  # Should print: `True`

client.close()  # Free up resources
```

API docs

If you did not see any errors, you are ready to proceed. We will replace the simple cluster status check with more meaningful operations in the next steps.

Step 2: Populate the database

Now, we can populate our database by first defining a collection then adding data.

2.1 Define a collection

WHAT IS A COLLECTION?

A collection is a set of objects that share the same data structure, like a table in relational databases or a collection in NoSQL databases. A collection also includes additional configurations that define how the data objects are stored and indexed.

The following example creates a *collection* called `Question` with:

- Ollama embedding model integration to create vectors during ingestion & queries, using the `nomic-embed-text` model, and
- Ollama generative AI integrations for retrieval augmented generation (RAG), using the `llama3.2` model.

Python JS/TS Go Java Curl

.VECTORS.TEXT2VEC_XXX WITH AUTOSCHEMA

Defining a collection with `Configure.Vectors.text2vec_xxx()` with Python client library `4.16.0`-`4.16.3` will throw an error if no properties are defined and `vectorize_collection_name` is not set to `True`.

This is addressed in `4.16.4` of the Weaviate Python client. See this FAQ entry for more details: Invalid properties error in Python client versions 4.16.0 to 4.16.3.

quickstart_create_collection.py

```python
import weaviate
from weaviate.classes.config import Configure

client = weaviate.connect_to_local()

questions = client.collections.create(
    name="Question",
    vector_config=Configure.Vectors.text2vec_ollama(  # Configure the Ollama
embedding integration
        api_endpoint="http://ollama:11434",  # If using Docker you might need:
http://host.docker.internal:11434
        model="nomic-embed-text",  # The model to use
    ),
    generative_config=Configure.Generative.ollama(  # Configure the Ollama
generative integration
        api_endpoint="http://ollama:11434",  # If using Docker you might need:
http://host.docker.internal:11434
        model="llama3.2",  # The model to use
    ),
)

client.close()  # Free up resources
```

API docs

Run this code to create the collection to which you can add data.

▸ Do you prefer a different setup?

2.2 Add objects

We can now add data to our collection.

The following example:

- Loads objects, and
- Adds objects to the target collection (`Question`) using a batch process.

BATCH IMPORTS

(Batch imports) are the most efficient way to add large amounts of data, as it sends multiple objects in a single request. See the How-to: Batch import guide for more information.

Python JS/TS Go Java Curl

quickstart_import.py

```python
import weaviate
import requests, json

client = weaviate.connect_to_local()

resp = requests.get(
    "https://raw.githubusercontent.com/weaviate-tutorials/quickstart/main/data/jeopardy_tiny.json"
)
data = json.loads(resp.text)

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

with questions.batch.fixed_size(batch_size=200) as batch:
    for d in data:
        batch.add_object(
            {
                "answer": d["Answer"],
                "question": d["Question"],
                "category": d["Category"],
            }
        )
        if batch.number_errors > 10:
            print("Batch import stopped due to excessive errors.")
            break

failed_objects = questions.batch.failed_objects
if failed_objects:
    print(f"Number of failed imports: {len(failed_objects)}")
    print(f"First failed object: {failed_objects[0]}")

client.close()  # Free up resources
```

API docs

During a batch import, any failed objects can be obtained through `batch.failed_objects`. Additionally, a running count of failed objects is maintained and can be accessed through `batch.number_errors` within the context manager. This counter can be used to stop the import process in order to investigate the failed objects or references. Find out more about error handling on the Python client reference page.

Run this code to add the demo data.

Step 3: Queries

Weaviate provides a wide range of query tools to help you find the right data. We will try a few searches here.

3.1 Semantic search

Semantic search finds results based on meaning. This is called `nearText` in Weaviate.

The following example searches for 2 objects whose meaning is most similar to that of `biology`.

Python JS/TS Go Java Curl

quickstart_neartext_query.py

```python
import weaviate
import json

client = weaviate.connect_to_local()

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

response = questions.query.near_text(
    query="biology",
    limit=2
)

for obj in response.objects:
    print(json.dumps(obj.properties, indent=2))

client.close()  # Free up resources
```

API docs

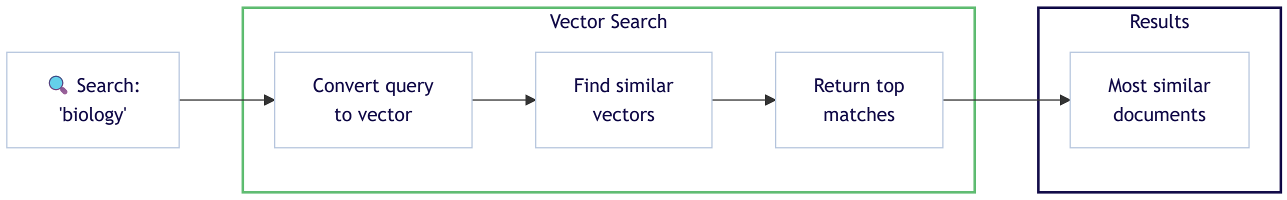
Run this code to perform the query. Our query found entries for `DNA` and `species`.

▸ Example full response in JSON format

If you inspect the full response, you will see that the word `biology` does not appear anywhere.

Even so, Weaviate was able to return biology-related entries. This is made possible by *vector embeddings* that capture meaning. Under the hood, semantic search is powered by vectors, or vector embeddings.

Here is a diagram showing the workflow in Weaviate.



WHERE DID THE VECTORS COME FROM?

Weaviate used the locally hosted Ollama model to generate a vector embedding for each object during import. During the query, Weaviate similarly converted the query (`biology`) into a vector.

As we mentioned above, this is optional. See Starter Guide: Bring Your Own Vectors if you would prefer to provide your own vectors.

MORE SEARCH TYPES AVAILABLE

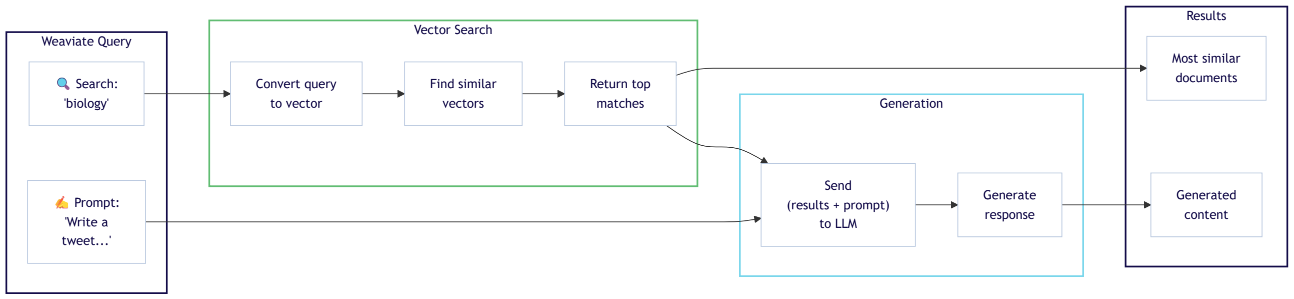
Weaviate is capable of many types of searches. See, for example, our how-to guides on similarity searches, keyword searches, hybrid searches, and filtered searches.

3.2 Retrieval augmented generation

Retrieval augmented generation (RAG), also called generative search, combines the power of generative AI models such as large language models (LLMs) with the up-to-date truthfulness of a database.

RAG works by prompting a large language model (LLM) with a combination of a *user query* and *data retrieved from a database*.

This diagram shows the RAG workflow in Weaviate.



The following example combines the same search (for `biology`) with a prompt to generate a tweet.

Python JS/TS Go Java Curl

quickstart_rag.py

```python
import weaviate

client = weaviate.connect_to_local()

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

response = questions.generate.near_text(
    query="biology",
    limit=2,
    grouped_task="Write a tweet with emojis about these facts."
)

print(response.generative.text)  # Inspect the generated text

client.close()  # Free up resources
```

API docs

Run this code to perform the query. Here is one possible response (your response will likely be different).

```
🧬 In 1953 Watson & Crick built a model of the molecular structure of DNA, the
gene-carrying substance! 🧬 🔬

🦢 2000 news: the Gunnison sage grouse isn't just another northern sage
grouse, but a new species! 🦢 🌿 #ScienceFacts #DNA #SpeciesClassification
```

The response should be new, yet familiar. This because you have seen the entries above for `DNA` and `species` in the semantic search section.

The power of RAG comes from the ability to transform your own data. Weaviate helps you in this journey by making it easy to perform a combined search & generation in just a few lines of code.

Recap

In this quickstart guide, you:

- Created a Serverless Weaviate sandbox instance on Weaviate Cloud.
- Defined a collection and added data.
- Performed queries, including:
  - Semantic search, and
  - Retrieval augmented generation.

Where to go next is up to you. We include some suggested steps and resources below.

Next

Try these additional resources to learn more about Weaviate:

More on search

See how to perform searches, such as keyword, similarity, hybrid, image, filtered and reranked searches.

Manage data

See how to manage data, such as manage collections, create objects, batch import data and use multi-tenancy.

RAG

Check out the Starter guide: retrieval augmented generation, and the Weaviate Academy unit on chunking.

Workshops and office hours

We hold in-person and online workshops, office hours and events for different experience levels. Join us!

FAQs & Troubleshooting

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

Questions

Can I use different integrations?

▸ See answer

Troubleshooting

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

▸ See answer

How to confirm collection creation

▸ See answer

How to confirm data import

▸ See answer

If the `nearText` search is not working

See answer

Questions and feedback

If you have any questions or feedback, let us know in the user forum.

Technical questions

If you have questions feel free to post on our **Community forum**.

Documentation feedback

Leave feedback by opening a GitHub issue.

Edit this page