# 📘 Vector Stores and Retrievers in LangChain

This tutorial will familiarize you with **LangChain Vector Store** and **Retriever abstractions**.

These abstractions are designed to support **retrieval of data** from vector databases (or other sources) for integration with **LLM workflows**.  
They are especially important for applications that need to **fetch data and reason over it** as part of **model inference** — such as **Retrieval-Augmented Generation (RAG)**.

## 🔹 Core Concepts

We will cover three important building blocks:

1. **Documents**
2. **Vector Store**
3. **Retrievers**

## 📄 Documents

LangChain implements a **Document abstraction**, which represents a **unit of text and associated metadata**.

Each Document object has two attributes:

* page\_content → the **text content** of the document.
* metadata → a **dictionary** that can store information about the source, relationships, or other context.

⚡ Note: A single Document usually represents a **chunk** of a larger file (e.g., a paragraph from a book or a section of an article).

### ✅ Example: Creating Documents

from langchain\_core.documents import Document  
  
documents = [  
 Document(  
 page\_content="Dogs are great companions, known for their loyalty and friendliness",  
 metadata={"source": "mammal-pet-doc"},  
 ),  
 Document(  
 page\_content="Cats are independent pets that often enjoy their own space",  
 metadata={"source": "mammal-pet-doc"},  
 ),  
 Document(  
 page\_content="Goldfish are popular pets for beginners, requiring relatively simple care",  
 metadata={"source": "fish-pet-doc"},  
 ),  
 Document(  
 page\_content="Parrots are intelligent birds capable of mimicking human speech",  
 metadata={"source": "bird-pet-doc"},  
 ),  
 Document(  
 page\_content="Rabbits are social animals that need plenty of space to hop around",  
 metadata={"source": "mammal-pet-doc"},  
 ),  
]

Here, we created **5 documents**, each containing content and metadata about pets.

## 🔑 Connecting LLM and Embeddings

Before working with vector stores, we need:

1. **LLM** (to generate answers from retrieved context)
2. **Embeddings model** (to convert text into vectors)

import os  
from dotenv import load\_dotenv  
from langchain\_groq import ChatGroq  
  
# Load API keys from environment file (.env)  
load\_dotenv()  
groq\_api\_key = os.getenv("Groq\_key")  
os.environ["HF\_token"] = os.getenv("HF\_token")  
  
# Initialize LLM (Groq with LLaMA 3.1)  
llm = ChatGroq(groq\_api\_key=groq\_api\_key, model="llama-3.1-8b-instant")  
llm

### 🔹 Embeddings

We use a **HuggingFace model** for embeddings:

from langchain\_huggingface import HuggingFaceEmbeddings  
  
embeddings = HuggingFaceEmbeddings(model\_name="all-MiniLM-L6-v2")

The model all-MiniLM-L6-v2 generates **sentence embeddings** (vector representations of text).

## 📦 Vector Store

A **Vector Store** is a database where documents are stored in **vector (numerical) form** for similarity search.  
We will use **ChromaDB** here.

### ✅ Creating a Vector Store from Documents

from langchain\_chroma import Chroma  
  
vectorestore = Chroma.from\_documents(documents, embedding=embeddings)  
vectorestore

This will: 1. Take each document. 2. Convert it into embeddings. 3. Store them in Chroma for efficient similarity search.

### 🔍 Searching in Vector Store

1. **Simple Similarity Search**

* vectorestore.similarity\_search("cat")
* Returns documents most similar to the query "cat".

1. **Asynchronous Search**

* await vectorestore.asimilarity\_search("cat")

1. **Similarity Search with Score**

* vectorestore.similarity\_search\_with\_score("cat")
* Returns (document, score) pairs, where the score indicates similarity (lower = more similar).

## 🎯 Retrievers

While vector stores are useful, they **cannot be directly used in LangChain Expression Language (LCEL) chains**, because they don’t subclass Runnable.

Retrievers solve this:  
- A **Retriever** wraps a vector store.  
- Exposes standardized methods (invoke, ainvoke, batch, etc.).  
- Allows seamless integration into **RAG pipelines**.

### ✅ Example: Manual Retriever

We can create a retriever from a similarity search function:

from typing import List  
from langchain\_core.runnables import RunnableLambda  
  
retriver = RunnableLambda(vectorestore.similarity\_search).bind(k=1)  
retriver.batch(["cat", "dog"])

Here: - RunnableLambda wraps the similarity search function.  
- bind(k=1) means only **1 most relevant document** is retrieved.

### ✅ Built-in Retriever from Vector Store

Instead of manually wrapping, we can use:

retriver = vectorestore.as\_retriever(  
 search\_type="similarity",  
 search\_kwargs={"k": 1}  
)  
retriver.batch(["cat", "dog"])

This uses VectorStoreRetriever with the given search\_type and parameters.

## 🔗 Retriever with Chain (RAG)

Now let’s implement **Retrieval-Augmented Generation (RAG)**:

1. Retrieve documents.
2. Pass them into an LLM with a **prompt template**.
3. Generate a contextualized answer.

### ✅ Code

from langchain\_core.prompts import ChatPromptTemplate  
from langchain\_core.runnables import RunnablePassthrough  
  
message = \"\"\"   
Answer this question using the provided context only.  
{question}  
  
Context:  
{context}  
\"\"\"  
  
# Prompt template  
prompt = ChatPromptTemplate.from\_messages([  
 ("human", message)  
])  
  
# Build RAG chain  
rag\_chain = {  
 "context": retriver, # fetches documents  
 "question": RunnablePassthrough() # passes user question directly  
} | prompt | llm  
  
# Invoke chain  
response = rag\_chain.invoke("tell me about dog")  
print(response.content)

### 🔹 Explanation of Flow

1. User asks → "tell me about dog".
2. Retriever fetches relevant document (dog description).
3. Prompt template inserts both the **question** and **retrieved context**.
4. LLM generates a **final response grounded in context**.

## 🚀 Key Takeaways

1. **Document** → basic text unit with metadata.
2. **Vector Store** → stores embeddings for similarity search (Chroma, Pinecone, Weaviate, etc.).
3. **Retriever** → bridges vector stores with LangChain pipelines.
4. **RAG** → retrieves context and augments LLM answers.
5. You can build retrievers manually or use as\_retriever() directly.