# Retrievers

A retriever is an interface that returns documents given an unstructured query. It is more general than a vector store. A retriever does not need to be able to store documents, only to return (or retrieve) it. Vector stores can be used as the backbone of a retriever, but there are other types of retrievers as well.

## Get started

The public API of the `BaseRetriever` class in LangChain is as follows:

```
from abc import ABC, abstractmethod
from typing import Any, List
from langchain.schema import Document
from langchain.callbacks.manager import Callbacks

class BaseRetriever(ABC):
    ...
    def get_relevant_documents(
        self, query: str, *, callbacks: Callbacks = None, **kwargs: Any
    ) -> List[Document]:
        """Retrieve documents relevant to a query.
        Args:
            query: string to find relevant documents for
            callbacks: Callback manager or list of callbacks
        Returns:
```

```
            List of relevant documents
        """
        ...

    async def aget_relevant_documents(
        self, query: str, *, callbacks: Callbacks = None, **kwargs: Any
    ) -> List[Document]:
        """Asynchronously get documents relevant to a query.
        Args:
            query: string to find relevant documents for
            callbacks: Callback manager or list of callbacks
        Returns:
            List of relevant documents
        """
        ...
```

It's that simple! You can call `get_relevant_documents` or the async `get_relevant_documents` methods to retrieve documents relevant to a query, where "relevance" is defined by the specific retriever object you are calling.

Of course, we also help construct what we think useful Retrievers are. The main type of Retriever that we focus on is a Vectorstore retriever. We will focus on that for the rest of this guide.

In order to understand what a vectorstore retriever is, it's important to understand what a Vectorstore is. So let's look at that.

By default, LangChain uses Chroma as the vectorstore to index and search embeddings. To walk through this tutorial, we'll first need to install `chromadb`.

```
pip install chromadb
```

This example showcases question answering over documents. We have chosen this as the example for getting started because it nicely combines a lot of different elements (Text splitters, embeddings, vectorstores) and then also shows how to use them in a chain.

Question answering over documents consists of four steps:

1. Create an index
2. Create a Retriever from that index
3. Create a question answering chain
4. Ask questions!

Each of the steps has multiple sub steps and potential configurations. In this notebook we will primarily focus on (1). We will start by showing the one-liner for doing so, but then break down what is actually going on.

First, let's import some common classes we'll use no matter what.

```
from langchain.chains import RetrievalQA
from langchain.llms import OpenAI
```

Next in the generic setup, let's specify the document loader we want to use. You can download the `state_of_the_union.txt` file here

```
from langchain.document_loaders import TextLoader
loader = TextLoader('../state_of_the_union.txt', encoding='utf8')
```

# One Line Index Creation

To get started as quickly as possible, we can use the `VectorstoreIndexCreator`.

```
from langchain.indexes import VectorstoreIndexCreator
```

```
index = VectorstoreIndexCreator().from_loaders([loader])
```

```
    Running Chroma using direct local API.
    Using DuckDB in-memory for database. Data will be transient.
```

Now that the index is created, we can use it to ask questions of the data! Note that under the hood this is actually doing a few steps as well, which we will cover later in this guide.

```
query = "What did the president say about Ketanji Brown Jackson"
index.query(query)
```

```
    " The president said that Ketanji Brown Jackson is one of the nation's top legal minds, a former top litigator in private practice, a former federal public defender, and from a family of public school educators and police officers. He also said that she is a consensus builder and has received a broad range of support from the Fraternal Order of Police to former judges appointed by Democrats and Republicans."
```

```
query = "What did the president say about Ketanji Brown Jackson"
index.query_with_sources(query)
```

```
    {'question': 'What did the president say about Ketanji Brown Jackson',
     'answer': " The president said that he nominated Circuit Court of Appeals Judge Ketanji Brown Jackson, one of the nation's top legal minds, to continue Justice Breyer's legacy of excellence, and that she has received a broad range of support from the Fraternal Order of Police to former judges appointed by Democrats and Republicans.\n",
     'sources': '../state_of_the_union.txt'}
```

What is returned from the `VectorstoreIndexCreator` is `VectorStoreIndexWrapper`, which provides these nice `query` and `query_with_sources` functionality. If we just wanted to access the vectorstore directly, we can also do that.

```
index.vectorstore
```

```
    <langchain.vectorstores.chroma.Chroma at 0x119aa5940>
```

If we then want to access the VectorstoreRetriever, we can do that with:

```
index.vectorstore.as_retriever()
```

```
    VectorStoreRetriever(vectorstore=<langchain.vectorstores.chroma.Chroma object at 0x119aa5940>, search_kwargs={})
```

# Walkthrough

Okay, so what's actually going on? How is this index getting created?

A lot of the magic is being hid in this `VectorstoreIndexCreator`. What is this doing?

There are three main steps going on after the documents are loaded:

1. Splitting documents into chunks
2. Creating embeddings for each document
3. Storing documents and embeddings in a vectorstore

Let's walk through this in code

```
documents = loader.load()
```

Next, we will split the documents into chunks.

```
from langchain.text_splitter import CharacterTextSplitter
text_splitter = CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
texts = text_splitter.split_documents(documents)
```

We will then select which embeddings we want to use.

```
from langchain.embeddings import OpenAIEmbeddings
embeddings = OpenAIEmbeddings()
```

We now create the vectorstore to use as the index.

```
from langchain.vectorstores import Chroma
```

```
db = Chroma.from_documents(texts, embeddings)
```

```
    Running Chroma using direct local API.
    Using DuckDB in-memory for database. Data will be transient.
```

So that's creating the index. Then, we expose this index in a retriever interface.

```
retriever = db.as_retriever()
```

Then, as before, we create a chain and use it to answer questions!

```
qa = RetrievalQA.from_chain_type(llm=OpenAI(), chain_type="stuff", retriever=retriever)
```

```
query = "What did the president say about Ketanji Brown Jackson"
qa.run(query)
```

```
    " The President said that Judge Ketanji Brown Jackson is one of the nation's top legal minds, a former
top litigator in private practice, a former federal public defender, and from a family of public school
educators and police officers. He said she is a consensus builder and has received a broad range of support
from organizations such as the Fraternal Order of Police and former judges appointed by Democrats and
Republicans."
```

`VectorstoreIndexCreator` is just a wrapper around all this logic. It is configurable in the text splitter it uses, the embeddings it uses, and the vectorstore it uses. For example, you can configure it as below:

```
index_creator = VectorstoreIndexCreator(
    vectorstore_cls=Chroma,
    embedding=OpenAIEmbeddings(),
    text_splitter=CharacterTextSplitter(chunk_size=1000, chunk_overlap=0)
)
```

Hopefully this highlights what is going on under the hood of `VectorstoreIndexCreator`. While we think it's important to have a simple way to create indexes, we also think it's important to understand what's going on under the hood.