第五章　LangChainの実戦

各コンポーネントの詳細な解説に入る前に、まずは一から実用的で意義のあるプロジェクトを完成させましょう。目的は、LangChainが大規模言語モデルを基盤としたアプリケーション開発フレームワークとしてどれほど強力であるかを直感的に理解することです。

Ø　プロジェクトおよび実装フレームワーク

プロジェクト： 社内従業員ナレッジベースQ&Aシステム

プロジェクト紹介 ：「社内従業員ナレッジベースQ&Aシステム」は、社内の情報共有と効率的な業務サポートを目的としたシステムです。このシステムは、従業員が業務に関する質問を入力すると、ナレッジベースから適切な回答を自動的に検索し、迅速に提供します。これにより、情報の検索時間を短縮し、社内の知識共有を促進します。

上記のニーズに基づき、さまざまな内部知識マニュアルに基づいた「Doc-QA」システムを開発します。このシステムはLangChainフレームワークを最大限に活用し、従業員マニュアルから発生するさまざまな質問に対応します。このQ&Aシステムは、従業員の質問を理解し、最新の従業員マニュアルに基づいて正確な回答を提供します。

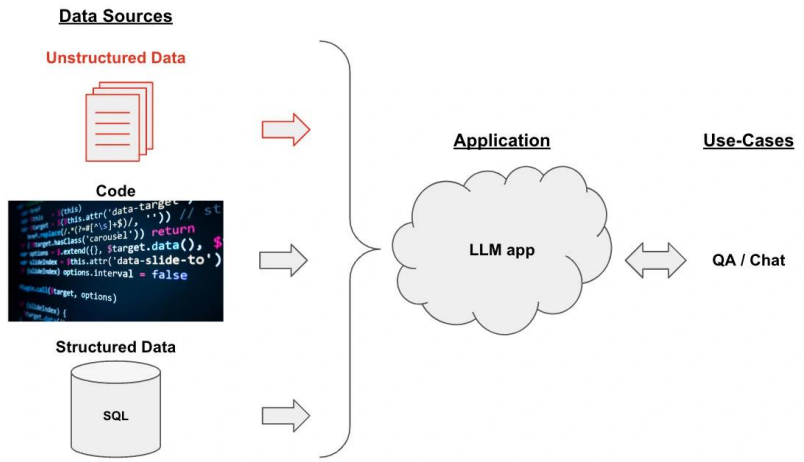
第五章　LangChainの実戦

Ø　プロジェクトおよび実装フレームワーク

開発フレームワーク：以下の画像は、LangChainフレームワークを使用してナレッジベース文書システムを実現するための全体的なフレームワークを示しています。

全体フレームワークは、以下の三つの部分に分かれています。

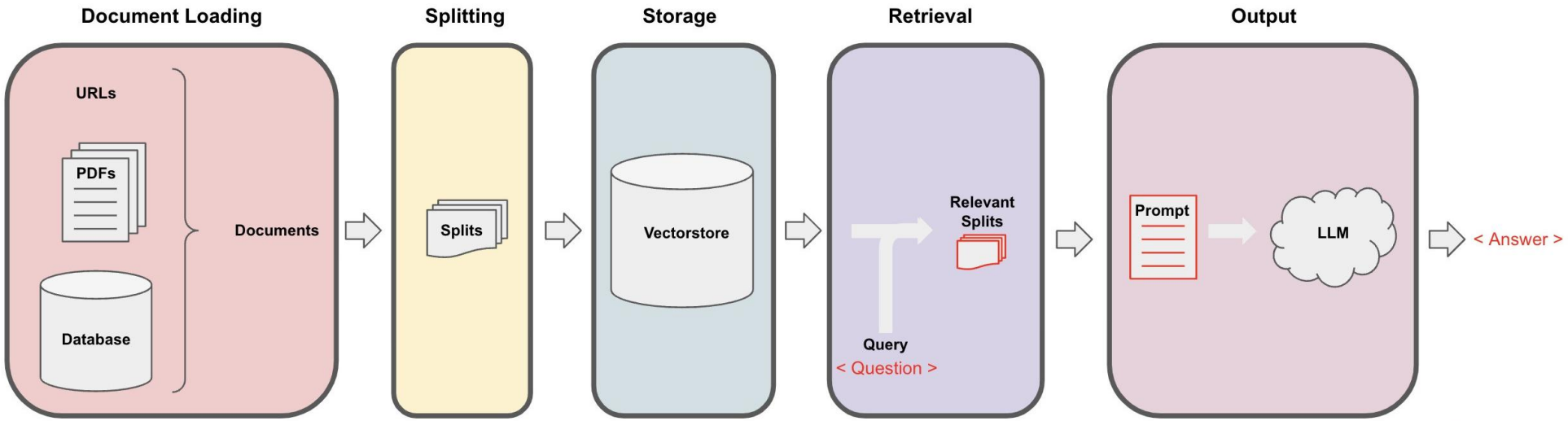
1. **データソース（Data Sources）**：データには多くの種類があり、この例では、非構造化データの処理に焦点を当てています。
2. **モデルアプリケーション（Application、つまりLLM App）**：大規模なモデルを論理エンジンとして利用し、必要な回答を生成します。
3. **ユースケース**（Use-Cases）：大規模モデルが生成する回答を基に、QAシステムやチャットボットなどのシステムを構築します



第五章　LangChainの実戦

Ø　プロジェクトおよび実装フレームワーク

核心実装メカニズム：核心実装メカニズムは、以下の図に示すデータ処理パイプライン（Pipeline）です。



このパイプラインの各ステップにおいて、LangChainは文書に基づくQ&A機能を実現するための関連ツールを提供しています。

第五章　LangChainの実戦

Ø　プロジェクトおよび実装フレームワーク

具体的な実装手順は以下の五つのステップに基づいています：

Loading: ドキュメントローダーがDocumentsをLangChainが読み取れる形式でロードします。

Splitting: テキストスプリッターがDocumentsを指定されたサイズに分割し、それらを「ドキュメントブロック」または「ドキュメントスライス」と呼びます。

Storage: 前のステップで分割した「ドキュメントブロック」を「埋め込み（Embedding）」形式でベクトルデータベース（Vector DB）に保存し、「埋め込みスライス」として形成します。

Retrieval: アプリケーションがストレージから分割されたドキュメントを検索します（例えば、コサイン類似度を比較して入力質問に類似した埋め込みスライスを見つけます）。

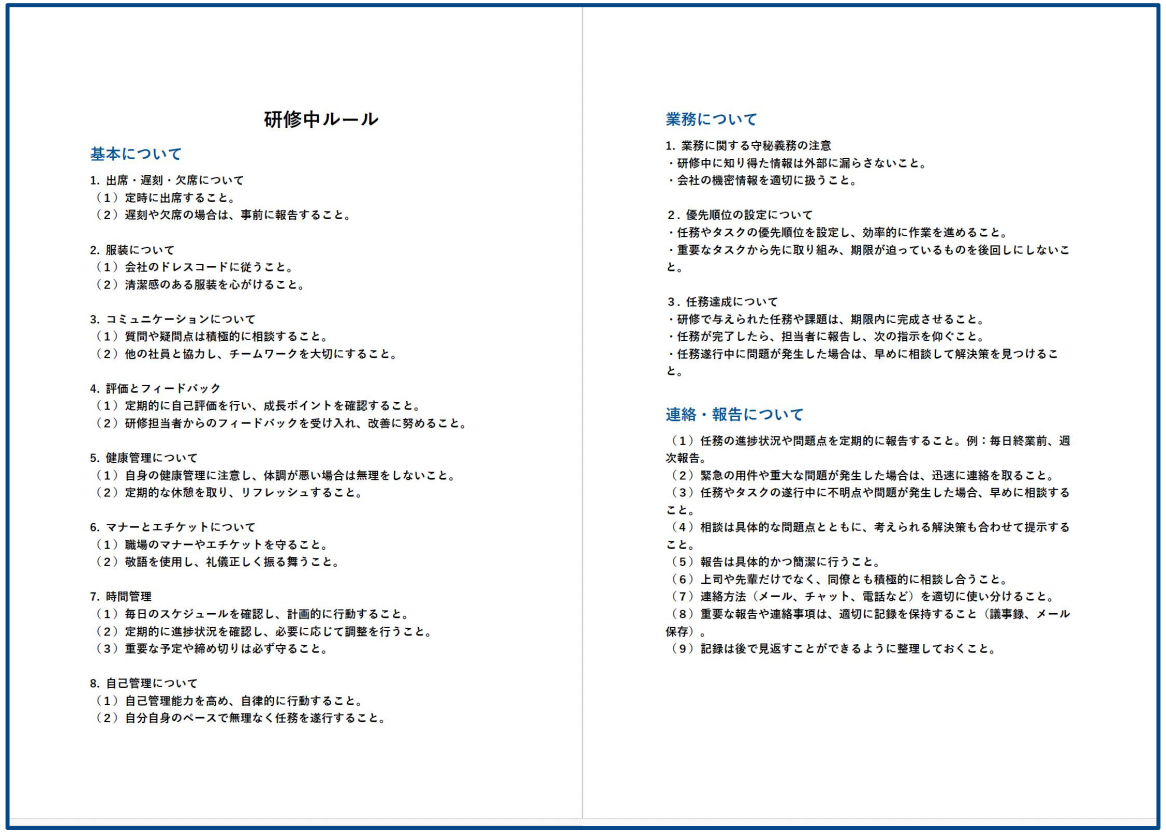
Output: 質問と類似する埋め込みスライスを言語モデル（LLM）に渡し、質問と検索された分割を含むプロンプトを使用して回答を生成します。

上記の5つのステップの紹介は非常に簡単で、いくつかの概念（例えば、埋め込みやベクトルストレージ）は初めて登場するため、理解には背景知識が必要です。焦らずに、これからこの5つのステップを具体的に説明します。

第五章　LangChainの実戦

Ø　データの準備とロード

会社の内部資料は、PDF、Word、TXT形式のさまざまなファイルである可能性があります。ここでは、研修の規定を例として取り上げます。文書の例は以下の通りです。



研修中ルール

基本について

1. 出席・遅刻・欠席について
（1）定時に出席すること。
（2）遅刻や欠席の場合は、事前に報告すること。

2. 服装について
（1）会社のドレスコードに従うこと。
（2）清潔感のある服装を心がけること。

3. コミュニケーションについて
（1）質問や疑問点は積極的に相談すること。
（2）他の社員と協力し、チームワークを大切にすること。

4. 評価とフィードバック
（1）定期的に自己評価を行い、成長ポイントを確認すること。
（2）研修担当者からのフィードバックを受け入れ、改善に努めること。

5. 健康管理について
（1）自身の健康管理に注意し、体調が悪い場合は無理をしないこと。
（2）定期的な休憩を取り、リフレッシュすること。

6. マナーとエチケットについて
（1）職場のマナーやエチケットを守ること。
（2）敬語を使用し、礼儀正しく振る舞うこと。

7. 時間管理
（1）毎日のスケジュールを確認し、計画的に行動すること。
（2）定期的に進捗状況を確認し、必要に応じて調整を行うこと。
（3）重要な予定や締め切りは必ず守ること。

8. 自己管理について
（1）自己管理能力を高め、自律的に行動すること。
（2）自分自身のペースで無理なく任務を遂行すること。

業務について

1. 業務に関する守秘義務の注意
・研修中に知り得た情報は外部に漏らさないこと。
・会社の機密情報を適切に扱うこと。

2. 優先順位の設定について
・任務やタスクの優先順位を設定し、効率的に作業を進めること。
・重要なタスクから先に取り組み、期限が迫っているものを後回しにしないこと。

3. 任務達成について
・研修で与えられた任務や課題は、期限内に完成させること。
・任務が完了したら、担当者に報告し、次の指示を仰ぐこと。
・任務遂行中に問題が発生した場合は、早めに相談して解決策を見つけること。

連絡・報告について

（1）任務の進捗状況や問題点を定期的に報告すること。例：毎日終業前、週次報告。
（2）緊急の用件や重大な問題が発生した場合は、迅速に連絡を取ること。
（3）任務やタスクの遂行中に不明点や問題が発生した場合、早めに相談すること。
（4）相談は具体的な問題点とともに、考えられる解決策も合わせて提示すること。
（5）報告は具体的かつ簡潔に行うこと。
（6）上司や先輩だけでなく、同僚とも積極的に相談し合うこと。
（7）連絡方法（メール、チャット、電話など）を適切に使い分けること。
（8）重要な報告や連絡事項は、適切に記録を保持すること（議事録、メール保存）。
（9）記録は後で見返すことができるように整理しておくこと。

第五章　LangChainの実戦

Ø　データの準備とロード

まず、LangChainのdocument_loadersを使用して、さまざまな形式のテキストファイルをロードします。（これらのファイルは「companydoc」ディレクトリに置いていますが、自分でフォルダを作成する場合は、コード内のディレクトリを調整する必要があります。）

このステップでは、PDF、Word、TXTファイルからテキストをロードし、それらのテキストをリストに保存します。（注意：PyPDF、Docx2txtなどのライブラリをインストールする必要があるかもしれません。

```
import os
os.environ["OPENAI_API_KEY"] = ‘あなたのOpenAI API key’

from langchain.document_loaders import PyPDFLoader
from langchain.document_loaders import Docx2txtLoader
from langchain.document_loaders import TextLoader

base_dir = ‘.\companydoc'
documents = []
```

第五章　LangChainの実戦

Ø　データの準備とロード

```
# 前ページからの続き
for file in os.listdir(base_dir):
    file_path = os.path.join(base_dir, file)
    if file.endswith('.pdf'):
        loader = PyPDFLoader(file_path)
        documents.extend(loader.load())
    elif file.endswith('.docx'):
        loader = Docx2txtLoader(file_path)
        documents.extend(loader.load())
    elif file.endswith('.txt'):
        loader = TextLoader(file_path)
        documents.extend(loader.load())
```

ここではまず、OpenAIのAPIキーをインポートします。後でOpenAIのモデルを使用して、文書のEmbeddingとQ&Aシステムでの回答生成を行う必要があるためです。

もちろん、LangChainがサポートするLLMはOpenAIだけではありません。このフレームワークに従いながら、Embeddingモデルと回答生成する言語モデルを他のオープンソースモデルに置き換えることも可能です。

第五章　LangChainの実戦

Ø　テキストの分割

次に、ロードしたテキストをより小さなblockに分割して、Embeddingとベクトルストレージを行えるようにします。このステップでは、LangChainの`RecursiveCharacterTextSplitter`を使用してテキストを分割する必要があります。

```
from langchain.text_splitter import RecursiveCharacterTextSplitter

text_splitter = RecursiveCharacterTextSplitter(chunk_size=200,
chunk_overlap=10)

chunked_documents = text_splitter.split_documents(documents)
```

現在、文書は約200文字の文書ブロックに分割されました。このステップは、これらを次に示すベクトルデータベースに保存する準備のためのものです。

第五章　LangChainの実戦

Ø　ベクトルデータベースへの保存

次に、これらの分割されたテキストを埋め込み形式に変換し、それをベクトルデータベースに保存する必要があります。この例では、OpenAIEmbeddingsを使用してEmbeddingを生成し、Qdrantというベクトルデータベースを使用してEmbeddingを保存します（ここでは **pip install qdrant-client** が必要です）。

具体的な実装コードは以下の通りです。

```
from langchain.vectorstores import Qdrant
from langchain.embeddings import OpenAIEmbeddings

vectorstore = Qdrant.from_documents(
    documents=chunked_documents,
    embedding=OpenAIEmbeddings(),
    location=":memory:",
    collection_name="my_documents”
)
```

現在、すべての内部文書は「文書ブロックのEmbedding」形式でベクトルデータベースに保存されています。これで、ベクトルデータベースをクエリすることで、大まかに関連する情報を見つけることができます。

第五章　LangChainの実戦

Ø　関連情報の取得

内部文書がベクトルデータベースに保存された後、質問やタスクに応じて最も関連する情報を抽出する必要があります。この時、情報抽出の基本的な方法は、質問もベクトルに変換し、それをベクトルデータベース内の各ベクトルと比較して、最も近い情報を抽出することです。ベクトル間の比較は通常、ベクトルの距離または類似度に基づきます。高次元空間では、一般的なベクトル距離や類似度の計算方法には、ユークリッド距離とコサイン類似度があります。

ユークリッド距離: これは最も直接的な距離測定方法で、2次元平面上で2点間の直線距離を測定するのと同じです。高次元空間では、2つのベクトルのユークリッド距離は、対応する各次元の差の平方和の平方根です。

コサイン類似度: 多くの場合、ベクトルの大きさよりもその方向に関心があります。たとえば、テキスト処理では、単語のベクトルはテキストの長さによって大きさに大きな違いがあるかもしれませんが、方向の方が意味を反映します。コサイン類似度は、ベクトル間の方向の類似性を測定し、その値は-1から1の範囲です。値が1に近いほど、2つのベクトルの方向が似ていることを示します。。

第五章　LangChainの実戦

Ø　関連情報の取得

内部文書がベクトルデータベースに保存された後、質問やタスクに応じて最も関連する情報を抽出する必要があります。この時、情報抽出の基本的な方法は、質問もベクトルに変換し、それをベクトルデータベース内の各ベクトルと比較して、最も近い情報を抽出することです。ベクトル間の比較は通常、ベクトルの距離または類似度に基づきます。高次元空間では、一般的なベクトル距離や類似度の計算方法には、ユークリッド距離とコサイン類似度があります。

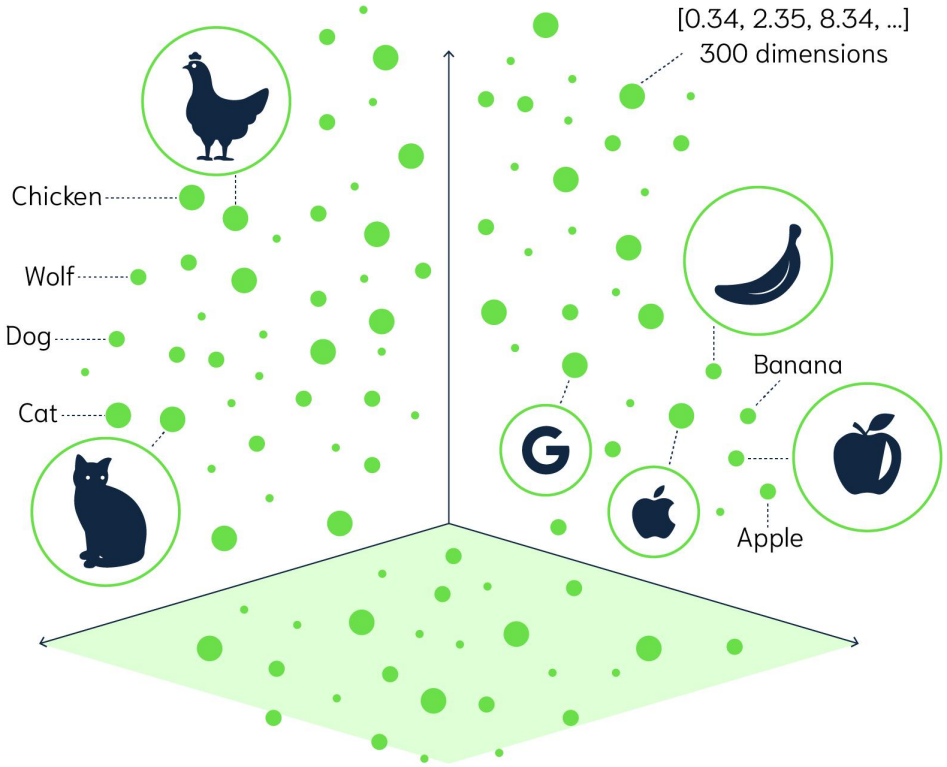
ユークリッド距離: これは最も直接的な距離測定方法で、2次元平面上で2点間の直線距離を測定するのと同じです。高次元空間では、2つのベクトルのユークリッド距離は、対応する各次元の差の平方和の平方根です。

コサイン類似度: 多くの場合、ベクトルの大きさよりもその方向に関心があります。たとえば、テキスト処理では、単語のベクトルはテキストの長さによって大きさに大きな違いがあるかもしれませんが、方向の方が意味を反映します。コサイン類似度は、ベクトル間の方向の類似性を測定し、その値は-1から1の範囲です。値が1に近いほど、2つのベクトルの方向が似ていることを示します。

第五章　LangChainの実戦

Ø　関連情報の取得

これらの2つの方法は、さまざまな機械学習のタスクで広く使用されています。どちらの方法を選択するかは、具体的なアプリケーションシナリオに依存します。一般的には、数量などの大きさの違いが重要な場合はユークリッド距離を使用し、テキストなどの意味的な違いが重要な場合はコサイン類似度を使用します。



第五章　LangChainの実戦

Ø　関連情報の取得

具体的には、ユークリッド距離は絶対的な距離を測定し、ベクトルの絶対的な違いを反映するのに優れています。データの絶対的な大きさが重要な場合、例えばアイテム推薦システムでは、ユーザーの購入量が好みの強さを反映する可能性があるため、ユークリッド距離を使用することが考えられます。また、データセット内の各ベクトルの大きさが似ていて、データの分布が大体均等な場合にも、ユークリッド距離が適しています。

一方、コサイン類似度は方向の類似性を測定し、ベクトルの角度の違いを重視し、大きさの違いにはあまり関心がありません。テキストデータやその他の高次元の疎データを扱う際に、特に有用です。たとえば、情報検索やテキスト分類のタスクでは、テキストデータは高次元の単語ベクトルとして表現されることが多く、単語ベクトルの方向が意味的な類似性を反映するため、コサイン類似度を使用することが適しています。

ここでは、テキストデータを処理しており、目標はQ&Aシステムを構築することです。したがって、意味的に問題の可能な回答を理解し比較する必要があります。このため、コサイン類似度を尺度として使用することが推奨されます。質問と回答のベクトルが意味空間での方向を比較することで、提示された質問に最も一致する回答を見つけることができます。

第五章　LangChainの実戦

Ø　関連情報の取得

このステップのコード部分では、チャットモデルを作成します。その後、RetrievalQAチェーンを作成する必要があります。これは質問の回答を生成するための検索型Q&Aモデルです。

RetrievalQAチェーンには、以下の2つの重要な構成要素があります。

LLM：質問に対する回答を担当します。

retriever：質問に基づいて関連する文書を検索し、具体的なEmbeddingを見つけます。これらのEmbeddingに対応する「文書ブロック」が知識情報として使用され、質問と一緒に大規模モデルに渡されます。ローカル文書から得られる知識は非常に重要です。なぜなら、インターネット情報から訓練された大規模モデルは企業の内部知識を持っていないからです。

具体的な実装コードは以下の通りです。

第五章　LangChainの実戦

Ø　関連情報の取得

```
import logging
from langchain.chat_models import ChatOpenAI
from langchain.retrievers.multi_query import MultiQueryRetriever
from langchain.chains import RetrievalQA

logging.basicConfig()
logging.getLogger('langchain.retrievers.multi_query').setLevel(logging.INF
O)

llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0)

retriever_from_llm =
MultiQueryRetriever.from_llm(retriever=vectorstore.as_retriever(), llm=llm)

qa_chain = RetrievalQA.from_chain_type(llm,retriever=retriever_from_llm)
```

これで、次のステップの準備が整いました。次に、システムユーザーからの具体的な質問を受け取り、その質問に基づいて情報を検索し、回答を生成します。

第五章　LangChainの実戦

Ø　回答の生成と表示

このステップは、Q&Aシステムアプリケーションの主要なUIインタラクション部分です。ここでは、ユーザーの質問を受け取り、対応する回答を生成するためにFlaskアプリケーションを作成し、最終的に`index.html`を通じて回答をレンダリングし表示します。

このステップでは、以前に作成したRetrievalQAチェーンを使用して関連する文書を取得し、回答を生成します。その後、これらの情報をユーザーに返し、ウェブページ上に表示します。

具体的な実装コードは以下の通りです。

第五章　LangChainの実戦

Ø　回答の生成と表示

```
# 前ページからの続き
from flask import Flask, request, render_template

app = Flask(__name__)

@app.route("/", methods=["GET", "POST"])
def home():
if request.method == "POST":

question = request.form.get("question")

result = qa_chain({"query": question})

return render_template("index.html", result=result)

return render_template("index.html")


if __name__ == "__main__":
app.run(host="0.0.0.0", debug=True, port=3000)
```

第五章　LangChainの実戦

Ø　回答の生成と表示

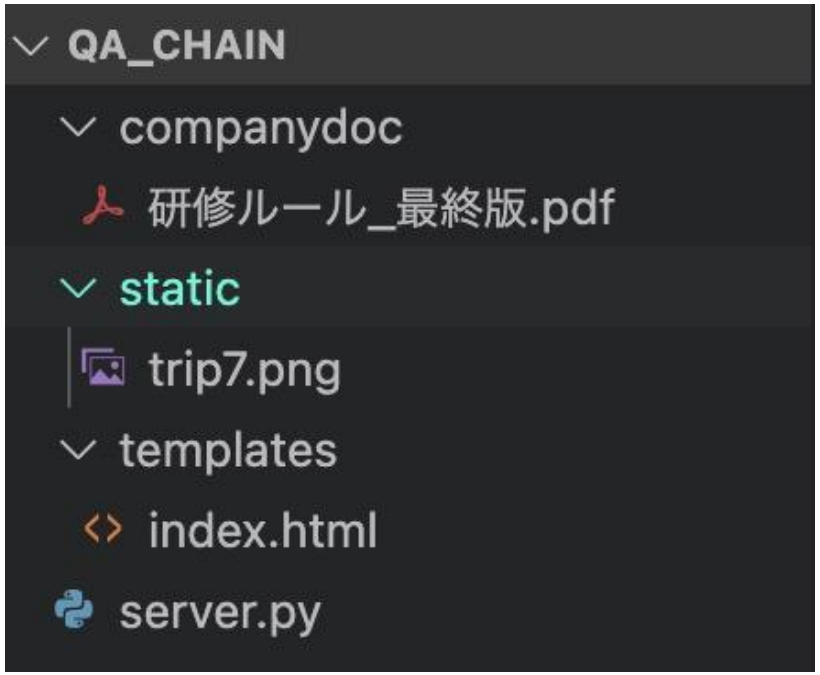
関連するHTMLページの主要なコードは以下の通りです：

```
<body>
        <div class="container">
                <div class="header">
                <h1>Trip7社内従業員ナレッジベースQ&Aシステム</h1>
                <img src="{{ url_for('static', filename='trip7.png') }}" alt="" width="200">
        </div>
        <form method="POST">
                <label for="question">Enter your question:</label><br>
                <input type="text" id="question" name="question"><br>
                <input type="submit" value="Submit">
        </form>
        {% if result is defined %}
        <h2>Answer</h2>
                <p>{{ result.result }}</p>
        {% endif %}
        </div>
</body>
```

第五章　LangChainの実戦

Ø　回答の生成と表示

プロジェクトのディレクトリ構造は以下の通りです。



第五章　LangChainの実戦

Ø　回答の生成と表示

プログラムを実行した後、http://127.0.0.1:3000/ のウェブページを開きます。ウェブページとやり取りすると、Q&Aシステムが会社の内部資料に特化した回答を正常に生成していることが確認できます。

第五章　LangChainの実戦

Ø　まとめ

上記のプロセスを振り返ってみましょう。以下の図に示されているように、まずローカルの知識を分割してEmbeddingを行い、それをベクトルデータベースに保存します。その後、ユーザーの入力とベクトルデータベースから検索されたローカルの知識を大規模モデルに渡し、最終的に望ましい回答を生成します。