**Part 7 - Serving LLM-based Web Applications**

Hi everyone, welcome to Part 7 of the course, “Serving LLM-based Web Applications”

In this part, we will explore how to serve LLM-based web applications using a combination of powerful tools and frameworks.

We will leverage LangChain for seamless integration with language models, the OpenAI API for accessing cutting-edge AI capabilities, and Gradio for creating user-friendly interfaces. FastAPI will be our framework of choice for building high-performance APIs, and Docker will facilitate containerization and deployment.

By the end of this part, you will have practical knowledge on building and deploying web applications, ensuring seamless integration and communication between different parts of the application using modern development tools and frameworks.

---

**What we will learn in this part**

* **Create the Frontend and Backend as two separate services**
  + This involves setting up and developing both the client-side and server-side components of the web application.
* **Communicate between frontend and backend using a REST API**
  + Implementing REST API endpoints to facilitate interaction between the frontend (user interface) and backend (server logic and database).
* **Serve the application with Docker**
  + Learn how to containerize the application using Docker.
  + Install, run, and enable communication between the frontend and backend within a single Docker container.
* **Use-case**
  + Build an LLM-based song recommendation app as a practical example to apply the concepts learned.

---

Let’s first explore the architecture and tools used for serving LLM-based web applications.

The Backend service is powered by Python, LangChain, OpenAI API, FAISS.

The Frontend service is built with Python and Gradio

Then we leverage leverages FastAPI to create a robust REST API, enabling the frontend to interact with the backend.

Docker ensures that the entire application can be containerized, making it easier to deploy and manage.

---

In addition to the software prerequisites for the Python project that we discussed in part 1, this part for serving our LLM-based web application requires two additional pieces of software:

* **FastAPI:**
  + A web framework for building APIs with Python 3.7+.
  + Installation command: pip install fastapi==0.104.1
* **Docker Desktop:**
  + Containerization software that includes Docker Engine, Docker CLI, Docker Compose, etc.
  + Necessary for creating, deploying, and managing containers.
  + Available for Windows, Mac (Intel and Apple Silicon chips), and Linux.
  + You can download Docker Desktop via this link.

---

This setup ensures you have the necessary tools to create and serve the backend of your LLM-based web application using FastAPI and Docker.

FastAPI will handle the API creation, while Docker will manage the application’s containerization.

---

Let me introduce our use case: Serving the LLM-based Song Recommendation App.

The LLM-based Song Recommendation App provides personalized song recommendations based on user inputs, such as the number of songs, desired song topic, and mood.

Powered by OpenAI's GPT-3.5-Turbo API, Langchain, FastAPI, Gradio, and Docker, users can easily generate song recommendations, by simply clicking the generate button.

This app demonstrates how to build an application with separate frontend and backend services, communicating between them via REST API, and serving everything within a Docker container.

---

I am going to show you what is the source code structure

---

This slide introduces the source code structure for the application. Here is the summarized explanation:

**Source Code Structure**

* **Folder Structure**:
  + **.ipynb\_checkpoints**: Contains checkpoints for Jupyter notebooks.
  + **backend**: Directory for the backend service code.
  + **frontend**: Directory for the frontend service code.
  + **image**: Directory for any image-related assets or Docker images.
  + **docker-compose.yml**: The Docker Compose file for orchestrating multi-container Docker applications.
  + **README.md**: The README file with documentation and instructions.

This organization helps in separating the concerns of the backend and frontend, making the development, deployment, and maintenance of the application more manageable. The use of Docker Compose ensures that all services can be easily started and managed together.

---

This slide provides an overview of the README file for the AI Song Recommendation App project. It outlines the project's structure and the essential steps to set up and deploy the application.

This slide effectively summarizes the project's key aspects and provides a concise guide to setting up and deploying the AI Song Recommendation App.

---

This slide provides an overview of the docker-compose.yml file used in the AI Song Recommendation App project. Here's a high-level introduction to the slide:

**docker-compose.yml Overview**

* **Purpose**: The docker-compose.yml file defines and manages multi-container Docker applications, specifying how to build and run the backend and frontend services.
* **Key Components**:
  + **Version**: Specifies the version of the Docker Compose file format.
  + **Services**:
    - **Backend**:
      * **Restart Policy**: Always restart the container if it stops.
      * **Build Context**: Build the backend service from the ./backend directory.
      * **Ports**: Maps port 8000 of the host to port 8000 of the container.
      * **Volumes**: Mounts the ./backend directory to /app inside the container.
    - **Frontend**:
      * **Restart Policy**: Always restart the container if it stops.
      * **Build Context**: Build the frontend service from the ./frontend directory.
      * **Ports**: Maps port 7860 of the host to port 7860 of the container.
      * **Volumes**: Mounts the ./frontend directory to /app inside the container.

This slide effectively summarizes the structure and configuration of the docker-compose.yml file, highlighting how it facilitates the deployment and management of the backend and frontend services for the AI Song Recommendation App.

---

Now let’s look at Backend source-code

This slide provides an overview of the backend source code structure for the AI Song Recommendation App. Here's a high-level introduction to the slide:

**Backend Source Code Overview**

* **Directory Structure**:
  + **.env**: Contains environment variables used for configuration.
  + **Dockerfile**: Defines the instructions for building the Docker image for the backend service.
  + **main.py**: The main Python script for the backend application. It likely includes the FastAPI setup and endpoints.
  + **requirements.txt**: Lists the Python dependencies required for the backend application.

This slide effectively summarizes the key components and files in the backend directory, highlighting their purposes and how they contribute to the functionality and deployment of the backend service for the AI Song Recommendation App.

---

**Backend's Requirements**

* **Directory Structure**:
  + **.env**: Contains environment variables for configuration.
  + **Dockerfile**: Instructions for building the Docker image for the backend service.
  + **main.py**: Main Python script for the backend application, including the FastAPI setup and endpoints.
  + **requirements.txt**: Lists the Python dependencies required for the backend application.
* **Dependencies**:
  + **python-dotenv==1.0.0**: For loading environment variables from a .env file.
  + **langchain==0.0.345**: For implementing language models and chains.
  + **openai==1.3.7**: OpenAI's Python client library for accessing the OpenAI API.
  + **fastapi==0.104.1**: A modern, fast (high-performance) web framework for building APIs with Python 3.6+.
  + **requests==2.31.0**: A simple, yet elegant, HTTP library for Python.
  + **uvicorn==0.24.0**: A lightning-fast ASGI server implementation, using uvloop and httptools.

This slide effectively summarizes the key components and files in the backend directory, highlighting their purposes and the required dependencies for setting up the backend service for the AI Song Recommendation App.

---

**Backend's .env**

This slide explains the role and content of the .env file in the backend directory for our AI Song Recommendation App.

**Overview**:

* The .env file stores environment variables used by the application.
* These variables often include sensitive information such as API keys, which should not be hard-coded into the source code for security reasons.

**Key Points**:

* The .env file is located in the backend directory.
* It contains the OPENAI\_API\_KEY, which is necessary for accessing the OpenAI API.
* The format of the .env file is straightforward: KEY="value"

**Example**:

plaintext

Copier le code

OPENAI\_API\_KEY="sk-SXP80...."

**Importance**:

* Using an .env file allows for easy management and modification of configuration settings without changing the source code.
* It enhances security by keeping sensitive information out of the source code.

This slide emphasizes the critical role of the .env file in securely managing configuration settings for the backend service.

---

Now let’s look at main.py in backend service

**Backend's main.py**

This slide provides an overview of the main.py file for the backend service, which is a crucial component for the AI Song Recommendation App.

**Components**:

1. **Importing Libraries**:
   * The file begins by importing necessary libraries and modules:

**Loading Environment Variables**:

* Environment variables are loaded to securely access API keys:

**Defining the Prompt Template**:

* A prompt template for the chatbot is defined to generate song recommendations:

**Initializing FastAPI**:

* An instance of the FastAPI application is created

**Defining an Endpoint**:

* An endpoint for getting song recommendations is defined

**Running the Application**:

* The application is run using uvicorn:

**Summary**:

* This main.py file sets up the backend service using FastAPI, defines a prompt for generating song recommendations, and runs the application with uvicorn. The prompt template ensures the recommendations are based on specified criteria (number of songs, topic, mood), making the service flexible and user-centric.

---

**Backend's Dockerfile**

The Dockerfile for the backend service defines the steps to build a Docker image that will run the FastAPI application. Here's a high-level overview of what each section does:

**Base Image and Working Directory**:

* FROM python:3.10: Use the official Python 3.10 image as the base image.
* WORKDIR /code: Set the working directory inside the container to /code.

**Copying and Installing Dependencies**:

* COPY ./requirements.txt /code/requirements.txt: Copy the requirements file into the container.
* RUN pip install --no-cache-dir --upgrade -r /code/requirements.txt: Install the Python dependencies listed in requirements.txt.

**User Configuration**:

* RUN useradd -m -u 1000 user: Create a new user named "user" with user ID 1000.
* USER user: Switch to the newly created user for security purposes.
* ENV HOME=/home/user PATH=/home/user/.local/bin:$PATH: Set environment variables for the user's home directory and update the PATH.

**Setting Up the Application**:

* WORKDIR $HOME/app: Set the working directory to the user's home directory.
* COPY --chown=user . $HOME/app: Copy the current directory contents into the container at $HOME/app, setting the owner to the user.

**Starting the Application**:

* CMD exec uvicorn main:app --host 0.0.0.0 --port 8000: Run the application using uvicorn, specifying the module (main:app), the host (0.0.0.0), and the port (8000).

**Summary:**

* **Base Image**: Uses Python 3.10.
* **Dependencies**: Installs dependencies from requirements.txt.
* **User Setup**: Creates and switches to a non-root user for running the application.
* **Application Setup**: Copies application code into the container and sets the working directory.
* **Command**: Runs the FastAPI application using uvicorn.

This Dockerfile ensures that the backend service runs securely and efficiently in a containerized environment.

---

Let’s see frontend source code

**Frontend Source Code**

This slide provides an overview of the frontend components of the application:

* **image**: Contains static assets.
* **Dockerfile**: Instructions to build the frontend Docker image.
* **gr\_main.py**: Main script for the Gradio interface.
* **nb\_access\_endpoint\_FastAPI.ipynb**: Jupyter notebook for testing the backend connection.
* **requirements.txt**: Lists dependencies for the frontend.

This setup allows for a containerized and seamless user interaction with the backend.

---

**Frontend's requirements.txt**

This slide lists the dependencies required for the frontend of the application:

* **gradio==4.7.1**: For creating the web interface.
* **streamlit==1.29.0**: An alternative for creating the web interface.
* **requests==2.31.0**: For making HTTP requests to the backend.

These dependencies ensure the frontend operates smoothly, providing an interactive interface for users.

---

**Frontend's main.py**

This slide shows a function from the frontend's main.py script. The function, get\_recommended\_song, takes three parameters: no\_song (an integer), topic (a string), and mood (a string). The function performs the following steps:

1. **HTTP GET Request**:
   * Sends a GET request to the backend's /song\_chain endpoint.
   * Includes the parameters (no\_song, topic, mood) in the request.
2. **Error Handling**:
   * Raises an HTTP error if the status code is 4xx or 5xx.
   * Catches exceptions and prints the error message if the request fails.
3. **Response Handling**:
   * Prints a success message if the server is running and the endpoint exists.
   * Returns the response in JSON format if successful.

This function ensures the frontend can effectively communicate with the backend to fetch recommended songs based on user inputs.

---

**Frontend's Dockerfile**

This slide details the Dockerfile for the frontend component of the application. The Dockerfile contains the instructions to create a Docker image for the frontend service. Here is a breakdown of the steps:

1. **Base Image**:
   * Uses python:3.10 as the base image.
2. **Working Directory**:
   * Sets the working directory to /code.
3. **Copy Requirements**:
   * Copies the requirements.txt file to the /code directory.
4. **Install Dependencies**:
   * Installs the dependencies listed in requirements.txt using pip.
5. **Create a User**:
   * Adds a new user named user with user ID 1000.
6. **Switch User**:
   * Switches to the newly created user.
7. **Set Environment Variables**:
   * Sets the home directory for the user and updates the PATH.
8. **Working Directory for App**:
   * Sets the working directory to the user's home directory at $HOME/app.
9. **Copy Application Code**:
   * Copies the current directory contents to the container at $HOME/app and sets the owner to user.
10. **Command to Run**:
    * Sets the default command to run the Python script gr\_main.py.

This Dockerfile ensures that the frontend service is containerized and ready to be deployed, with all dependencies installed and the application code properly copied into the container.

---

Let’s playground with notebook file to test the rest api communication between the BE and FE services

---

**Launching Backend Service Locally**

This slide demonstrates how to launch the backend service locally. Here is a step-by-step guide:

1. **Navigate to Backend Directory**:
   * Change the directory to the backend folder:

bash

Copier le code

cd backend

1. **Run the Backend Service**:
   * Execute the main.py file to start the backend service:

bash

Copier le code

python main.py

1. **Backend Service Startup**:
   * Upon successful execution, the console will display the following messages indicating the service has started:
     + INFO: Started server process [16036]
     + INFO: Waiting for application startup.
     + INFO: Application startup complete.
     + INFO: Uvicorn running on http://localhost:8000 (Press CTRL+C to quit)

These steps ensure that the backend service is running locally and is accessible at http://localhost:8000.

---

**Notebook Playground to Test Backend & Frontend Services**

This slide shows how to use a notebook to test both the backend and frontend services. Here's a brief walkthrough:

* **Playground with Frontend Notebook**:
  + **Python Code to Get Recommended Songs**:

**Example Output**:

* The function get\_recommended\_song is tested with parameters: no\_song = 3, topic = 'New Year', and mood = 'Happy'.
* The backend service returns a list of recommended songs:
  + - The message from the backend indicates a successful GET request and status code 200 OK.

Using this notebook, you can easily test the interaction between your backend and frontend services by simulating API calls and verifying the responses.

---

**Launching App with Docker-Compose**

This slide introduces how to launch the application using docker-compose. Docker-Compose is a tool for defining and running multi-container Docker applications. It allows you to manage the whole application lifecycle with a single command, simplifying deployment and orchestration.

**Key Points:**

* **Docker-Compose File**: The configuration file (docker-compose.yml) specifies the services, networks, and volumes required for the application.
* **Single Command Deployment**: By running docker-compose up --build, you can build, create, start, and attach to containers for the services defined in the Compose file.

This approach makes it easier to manage complex applications with multiple services, ensuring consistent environments across development, testing, and production.

---

**Launching App with Docker-Compose**

In this slide, we demonstrate how to launch the application using docker-compose. Docker-Compose is a powerful tool for defining and running multi-container Docker applications, simplifying the management of complex applications.

**Key Steps:**

1. **Navigate to the Project Directory**:
   * Ensure you are in the project directory where the docker-compose.yml file is located.
2. **Run Docker-Compose Command**:
   * Use the command docker-compose up -d --build to build and start the application in detached mode.
   * The --build flag ensures that the images are built before starting the containers.

**Output:**

* The command initializes the network and starts the defined services (backend and frontend containers).
* The output shows the creation and startup status of the containers.

By using Docker-Compose, we can easily manage and deploy our multi-service application, ensuring a consistent environment across different stages of development and deployment.

---

**Docker Desktop Interface**

This slide shows the Docker Desktop interface, which provides a graphical view of all your Docker containers, images, and other resources.

**Key Highlights:**

1. **Containers View**:
   * Lists all the running and stopped containers.
   * Provides details such as container name, status, ports, and actions.
2. **Project-Specific Containers**:
   * The highlighted section shows the containers related to our app-serving-docker-fastapi-gradio project.
   * It includes the frontend and backend containers, both of which are currently running.
3. **Port Information**:
   * Displays the ports on which the containers are running. For instance, the frontend is running on port 7860 and the backend on port 8000.
4. **Actions**:
   * Options to start, stop, restart, and delete containers.

Docker Desktop provides an easy-to-use interface for managing Docker containers, making it convenient to monitor the status and logs of running applications.

---

**Test Backend**

This slide demonstrates how to test the backend of our application using Docker Desktop.

**Key Points:**

1. **Docker Desktop Interface**:
   * Displays the list of running containers.
   * Highlights our specific project containers for easy identification.
2. **Backend Container**:
   * Shows the backend container named backend-1 within the app-serving-docker-fastapi-gradio project.
   * Indicates the backend container is running and accessible.
3. **Port Information**:
   * The backend service is running on port 8000.
   * This port can be clicked to directly access and test the backend API endpoints.

**How to Test:**

1. **Open Docker Desktop**:
   * Navigate to the Containers section.
2. **Locate the Backend Container**:
   * Find backend-1 under the project name app-serving-docker-fastapi-gradio.
3. **Access the Backend**:
   * Click on the port 8000:8000 link to open the backend service in your web browser.
   * You can test the API endpoints directly from your browser or using tools like Postman.

By following these steps, you can verify that the backend service is running correctly and responding to requests as expected.

---

**Test Backend**

To ensure the backend of our application is functioning properly, we will use the FastAPI documentation interface.

**Steps to Test the Backend:**

1. **Access the Documentation**:
   * Open your web browser.
   * Navigate to http://localhost:8000/docs/. This URL brings up the interactive API documentation provided by FastAPI.
2. **Interactive API Documentation**:
   * This interface allows you to test the various API endpoints.
   * The available endpoint for our song recommendation app is /song\_chain.
3. **Testing the /song\_chain Endpoint**:
   * You can input the parameters required by the endpoint directly in the documentation interface.
   * For example, you can set:
     + no\_song (number of songs to recommend): 3
     + topic (song topic): New Year
     + mood (song mood): Happy
4. **Execute the Request**:
   * Click the "Try it out" button.
   * The API will return a list of recommended songs based on the provided parameters.
5. **Verify the Response**:
   * Ensure that the response includes a list of song recommendations.
   * The response should match the expected output based on the input parameters.

This interactive documentation provided by FastAPI simplifies the process of testing and verifying that the backend API endpoints are working correctly.

---

**Test Backend**

We can see the backend test interface where we interact with the FastAPI documentation page. Here's a brief overview of how the test is performed:

1. **Parameter Input**:
   * The parameters for the API request are input in the form provided.
   * In this example:
     + no\_song: 3
     + topic: New Year
     + mood: Happy
2. **Execution**:
   * Click on the "Execute" button to send the API request.
3. **Response Verification**:
   * The API returns a list of recommended songs based on the input parameters.
   * The response body shows:
     + "Auld Lang Syne" by Mariah Carey
     + "New Year's Day" by Taylor Swift
     + "Happy New Year" by ABBA

This verifies that the backend is working as expected and returning the correct recommendations.

---

**Test Frontend**

In this slide, we demonstrate how to test the frontend service using Docker Desktop.

1. **Access Frontend Container**:
   * The frontend container can be accessed through Docker Desktop. Ensure that it is running.
   * The container named frontend-1 should be visible and running.
2. **Port Configuration**:
   * The frontend is configured to run on port 7860.
   * Clicking on the link 7860:7860 will open the frontend interface in the web browser.

By following these steps, you can verify that the frontend service is up and running correctly, allowing you to interact with the application's user interface.

---

**Test Frontend**

To verify that the frontend is working correctly, follow these steps:

1. **Access the Frontend Interface**:
   * Open your web browser and navigate to http://localhost:7860/.
   * This URL will open the frontend interface for the LLM-based Song Recommendation app.
2. **User Interface Overview**:
   * The interface allows you to select the number of song recommendations.
   * You can specify the song topic and mood.
   * Clicking on the "Generate" button will provide a list of recommended songs based on the input criteria.
3. **Verification**:
   * Ensure that the recommended songs are displayed correctly as per the inputs provided.
   * This confirms that the frontend is successfully communicating with the backend service to fetch and display song recommendations.

By following these steps, you can ensure that the frontend service is up and running, providing the intended functionality of the application.

---

**Stopping the App with Docker Compose**

To properly stop the running application, follow these steps:

1. **Open Terminal/Command Prompt**:
   * Navigate to the directory containing your Docker Compose file (docker-compose.yml).
2. **Run the Docker Compose Down Command**:
   * Execute the command:

sh

Copier le code

docker-compose down

* + This command stops and removes the containers, networks, and volumes created by docker-compose up.

1. **Verify the Shutdown**:
   * You will see confirmation messages indicating that the containers and network have been removed successfully.

By following these steps, you ensure that all resources used by your Docker containers are properly cleaned up. This helps in maintaining a clean and efficient development environment.

---

In summary, we have successfully learned how to serve a demo LLM-based web application by leveraging two separate services: the backend and the frontend. The backend is built with Python, LangChain, and OpenAI API, while the frontend utilizes Python, Gradio, and Requests. The communication between these services is facilitated through a REST API using FastAPI. Additionally, we utilized Docker for deploying the application, employing tools such as Docker Desktop, Dockerfile, and Docker-compose. This comprehensive approach ensures an efficient and scalable solution for serving LLM-based applications.