



# Ubuntu 安裝 MariaDB



ubuntu安裝mariadb



以下是在 Ubuntu 上安裝 MariaDB 的步驟：

## 1. 更新系統套件

執行以下命令來更新系統的套件索引：

```
sudo apt update
sudo apt upgrade -y
```

## 2. 安裝 MariaDB

在 Ubuntu 的默認套件庫中可以直接安裝 MariaDB，執行以下命令：

```
sudo apt install mariadb-server mariadb-client -y
```

## 3. 啟動 MariaDB 服務

安裝完成後，啟動 MariaDB 服務並將其設置為開機啟動：

```
sudo systemctl start mariadb
sudo systemctl enable mariadb
```

## 4. 安全性設置

MariaDB 提供了一個安全設置腳本，執行以下命令進行配置：





```
sudo mysql_secure_installation
```

在此過程中，您將被提示以下幾個問題：

- 設定 root 密碼（根據需要進行設置）。
- 刪除匿名用戶。
- 禁止 root 遠程登錄。
- 刪除測試資料庫。
- 重新加載權限表。

根據需求選擇 "Yes" 或 "No"。

---

## 5. 驗證安裝

確保 MariaDB 服務已正確運行：

```
sudo systemctl status mariadb
```

進入 MariaDB 以確認其工作正常：

```
sudo mysql
```

輸入以下命令檢查版本：

```
SELECT VERSION();
```

輸入以下命令退出：

```
EXIT;
```

---

## 6. 配置 MariaDB（可選）

如果需要進一步配置 MariaDB，可以修改配置文件：

```
sudo nano /etc/mysql/mariadb.conf.d/50-server.cnf
```

完成修改後，重啟 MariaDB 服務以應用更改：



```
sudo systemctl restart mariadb
```

完成以上步驟後，MariaDB 已成功安裝並可以使用。若有任何問題，請隨時詢問！



用fast api 寫c/r/u/d with mariadb



以下是一個使用 FastAPI 和 MariaDB 實現基本 CRUD 操作的範例。使用的工具包括 **FastAPI** 作為框架，**SQLAlchemy** 作為 ORM，**MariaDB** 作為數據庫。

## 1. 環境設置

確保已安裝以下工具：

- Python 3.8+
- MariaDB 數據庫
- 相關 Python 套件

安裝必要的 Python 套件：

```
pip install fastapi uvicorn sqlalchemy pymysql
```

## 2. 創建數據庫和表

使用 MariaDB CLI 或 GUI 建立數據庫和表：

```
CREATE DATABASE fastapi_crud;
USE fastapi_crud;

CREATE TABLE items (
    id INT AUTO_INCREMENT PRIMARY KEY,
    name VARCHAR(255) NOT NULL,
    description TEXT,
```



```
price FLOAT NOT NULL
);
```

### 3. 實現 CRUD API

創建以下檔案結構：

```
.
├── main.py
└── database.py
```

``database.py``

此文件用於配置 MariaDB 連接和 SQLAlchemy 的數據庫引擎。

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "mysql+pymysql://username:password@localhost/fastapi_crud"

engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

``main.py``

包含所有 API 邏輯。

```
from fastapi import FastAPI, Depends, HTTPException
from sqlalchemy.orm import Session
from sqlalchemy import Column, Integer, String, Float
from database import Base, engine, SessionLocal

# 初始化數據庫
Base.metadata.create_all(bind=engine)

app = FastAPI()

# 定義數據庫模型
class Item(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(255), nullable=False)
    description = Column(String(255), nullable=True)
    price = Column(Float, nullable=False)
```



```
# 數據庫會話依賴
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()

# 創建資料
@app.post("/items/")
def create_item(name: str, description: str, price: float, db: Session = Depends(get_db)):
    item = Item(name=name, description=description, price=price)
    db.add(item)
    db.commit()
    db.refresh(item)
    return item

# 獲取資料 (單筆)
@app.get("/items/{item_id}")
def read_item(item_id: int, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    return item

# 獲取所有資料
@app.get("/items/")
def read_items(skip: int = 0, limit: int = 10, db: Session = Depends(get_db)):
    items = db.query(Item).offset(skip).limit(limit).all()
    return items

# 更新資料
@app.put("/items/{item_id}")
def update_item(item_id: int, name: str, description: str, price: float, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    item.name = name
    item.description = description
    item.price = price
    db.commit()
    db.refresh(item)
    return item

# 刪除資料
@app.delete("/items/{item_id}")
def delete_item(item_id: int, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    db.delete(item)
```



```
db.commit()
return {"detail": "Item deleted successfully"}
```

## 4. 啟動伺服器

在終端運行以下命令：

```
uvicorn main:app --reload
```

## 5. 測試 API

使用工具如 **Postman** 或 **cURL** 測試以下路徑：

- `POST /items/``：新增資料
- `GET /items/``：查詢所有資料
- `GET /items/{item_id}``：查詢單筆資料
- `PUT /items/{item_id}``：更新資料
- `DELETE /items/{item_id}``：刪除資料

範例請求：

```
POST /items/
{
  "name": "Example Item",
  "description": "An example description",
  "price": 19.99
}
```

這樣就完成了基於 FastAPI 和 MariaDB 的 CRUD 實現！如果有進一步需求，請告訴我。



詳細說明程式碼



以下是關於 `main.py` 和 `database.py` 的詳細說明，逐步解析每段程式碼：



---

## `database.py`

這個文件主要負責配置 MariaDB 的連接以及 ORM 基礎設置。

```
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

- ``create_engine``：創建一個 SQLAlchemy 引擎，用於連接到資料庫。
- ``declarative_base``：用於定義 ORM 模型的基礎類。
- ``sessionmaker``：用於創建資料庫會話的工廠函數。

---

```
DATABASE_URL = "mysql+pymysql://username:password@localhost/fastapi_crud"
```

- ``DATABASE_URL``：定義資料庫連接字串，格式為 ``mysql+pymysql://username:password@host/database_name``。
  - ``mysql+pymysql``：使用 ``pymysql`` 驅動程式連接到 MySQL/MariaDB。
  - 替換 ``username`` 和 ``password`` 為您的 MariaDB 資訊。

---

```
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

- ``engine``：建立資料庫的連接引擎。
- ``SessionLocal``：創建一個資料庫會話工廠，``autocommit`` 和 ``autoflush`` 用於控制會話行為。
- ``Base``：作為所有 ORM 模型的基類，供我們後續定義表結構使用。

---

## `main.py`

這個文件包含主要的 CRUD 業務邏輯和 FastAPI 的端點設置。



## 初始化資料庫

```
from sqlalchemy import Column, Integer, String, Float
from database import Base, engine, SessionLocal
```

- ``Column``：定義資料庫表的欄位。
- ``Integer``、``String``、``Float``：SQLAlchemy 提供的數據類型。

```
Base.metadata.create_all(bind=engine)
```

- ``create_all``：根據 ORM 模型定義，創建資料庫中的表結構。如果表已存在，則不會重複創建。

## FastAPI 應用和 ORM 模型

```
app = FastAPI()

class Item(Base):
    __tablename__ = "items"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String(255), nullable=False)
    description = Column(String(255), nullable=True)
    price = Column(Float, nullable=False)
```

- ``FastAPI()``：創建一個 FastAPI 應用實例。
- ``Item``：定義 ORM 模型，對應資料庫中的 ``items`` 表。
  - ``__tablename__``：指定表名。
  - ``id``：主鍵，為整數型且自動增長。
  - ``name``：商品名稱，為必填。
  - ``description``：商品描述，可為空。
  - ``price``：商品價格，為必填。

## 數據庫會話依賴

```
def get_db():
    db = SessionLocal()
```





```
try:
    yield db
finally:
    db.close()
```

- `get_db`：用於生成資料庫會話的依賴函數。FastAPI 會自動處理會話的開啟和關閉。

## CRUD API

### 新增資料

```
@app.post("/items/")
def create_item(name: str, description: str, price: float, db: Session = Depends(get_db)):
    item = Item(name=name, description=description, price=price)
    db.add(item)
    db.commit()
    db.refresh(item)
    return item
```

- `POST /items/`：新增一筆資料。
  - 接收 `name`、`description` 和 `price` 作為請求參數。
  - 創建一個新的 `Item` 對象，並將其新增到資料庫。
  - `commit`：提交事務以保存更改。
  - `refresh`：更新 `item` 以獲取其自動生成的 ID。
  - 返回新增的資料。

### 查詢單筆資料

```
@app.get("/items/{item_id}")
def read_item(item_id: int, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    return item
```

- `GET /items/{item_id}`：根據 ID 查詢一筆資料。
  - 使用 SQLAlchemy 查詢資料庫，返回第一筆匹配的記錄。
  - 如果未找到，拋出 404 錯誤。



---

## 查詢所有資料

```
@app.get("/items/")
def read_items(skip: int = 0, limit: int = 10, db: Session = Depends(get_db)):
    items = db.query(Item).offset(skip).limit(limit).all()
    return items
```

- `GET /items/`：查詢多筆資料。
  - 支援分頁功能，使用 `skip` 和 `limit` 控制返回的資料範圍。
  - 返回查詢結果的列表。

---

## 更新資料

```
@app.put("/items/{item_id}")
def update_item(item_id: int, name: str, description: str, price: float, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    item.name = name
    item.description = description
    item.price = price
    db.commit()
    db.refresh(item)
    return item
```

- `PUT /items/{item_id}`：更新一筆資料。
  - 查詢目標資料，若不存在，拋出 404。
  - 更新目標資料的屬性並保存更改。

---

## 刪除資料

```
@app.delete("/items/{item_id}")
def delete_item(item_id: int, db: Session = Depends(get_db)):
    item = db.query(Item).filter(Item.id == item_id).first()
    if item is None:
        raise HTTPException(status_code=404, detail="Item not found")
    db.delete(item)
```



```
db.commit()
return {"detail": "Item deleted successfully"}
```

- ``DELETE /items/{item_id}``：刪除一筆資料。
  - 查詢目標資料，若不存在，拋出 404。
  - 刪除資料並提交事務。

這樣，每個 CRUD 操作的實現邏輯就清楚了！如果還有疑問，歡迎詢問。



包成docker image使用



將上述 FastAPI 和 MariaDB 應用包裝成 Docker Image，並使用 Docker Compose 管理多容器部署。以下是完整步驟：

## 1. 新增 `Dockerfile`

在專案根目錄下建立一個 `Dockerfile`：

```
# 使用官方 Python 基礎映像
FROM python:3.10-slim

# 設置工作目錄
WORKDIR /app

# 複製需求檔案並安裝依賴
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 複製應用程式碼
COPY . .

# 暴露 FastAPI 預設的埠
EXPOSE 8000
```



```
# 啟動 FastAPI 服務
CMD ["uvicorn", "main:app", "--host", "0.0.0.0", "--port", "8000"]
```

## 2. 建立 `requirements.txt`

將應用程式需要的依賴包列入 `requirements.txt`：

```
fastapi
uvicorn
sqlalchemy
pymysql
```

## 3. 新增 `docker-compose.yml`

在專案根目錄下建立 `docker-compose.yml`，同時包含 FastAPI 和 MariaDB 容器：

```
version: "3.8"

services:
  app:
    build:
      context: .
      dockerfile: Dockerfile
    ports:
      - "8000:8000"
    environment:
      - DATABASE_URL=mysql+pymysql://root:password@mariadb:3306/fastapi_crud
    depends_on:
      - mariadb
    volumes:
      - ./app

  mariadb:
    image: mariadb:10.5
    restart: always
    environment:
      MARIADB_ROOT_PASSWORD: password
      MARIADB_DATABASE: fastapi_crud
    ports:
      - "3306:3306"
    volumes:
      - mariadb_data:/var/lib/mysql
```



```
volumes:
  mariadb_data:
```

## 4. 更新 `database.py`

確保 `DATABASE\_URL` 的連接字串可以透過環境變數設定：

```
import os
from sqlalchemy import create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

DATABASE_URL = os.getenv("DATABASE_URL", "mysql+pymysql://root:password@localhost/fastapi_crud")

engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

## 5. 構建和啟動容器

建構 **Docker** 映像和啟動容器

運行以下命令：

```
docker-compose up --build
```

- `--build`：確保重建 Docker 映像。
- 啟動後，`FastAPI` 將運行在 `http://localhost:8000`。

## 6. 測試服務

確認容器狀態

檢查容器是否正確運行：

```
docker-compose ps
```

測試 API

您可以使用工具如 `Postman` 或直接訪問：



- ``http://localhost:8000/items/``：使用 CRUD API。
- ``http://localhost:8000/docs``：自動生成的 Swagger 文檔。

---

## 7. 儲存和分享 Image（可選）

將 Docker Image 上傳到 Docker Hub 或其他容器倉庫：

### 登入 Docker Hub

```
docker login
```

### 建立並標籤 Image

```
docker build -t yourusername/fastapi-mariadb-app .  
docker tag yourusername/fastapi-mariadb-app:latest
```

### 推送到 Docker Hub

```
docker push yourusername/fastapi-mariadb-app
```

現在，您可以在其他環境下載並運行該容器。

---

這樣就完成了將 FastAPI 和 MariaDB 應用封裝成 Docker Image 並使用 Docker Compose 部署的流程。如果有疑問，歡迎隨時詢問！



教我fast api



## 什麼是 FastAPI？

FastAPI 是一個現代化的、快速（高性能）的 Python 網頁框架，用於構建 API。它基於 **Starlette** 和 **Pydantic**，提供了開箱即用的功能如自動生成 API 文檔、高性能、支持異步處理等。

以下是一個簡單的入門教學，讓你快速掌握 FastAPI 的基本用法。



## 1. 安裝 FastAPI

### 安裝 FastAPI 和 Uvicorn

Uvicorn 是一個 ASGI 服務器，用於運行 FastAPI 應用：

```
pip install fastapi uvicorn
```

## 2. Hello World 範例

建立主程式 `main.py`：

```
from fastapi import FastAPI

# 創建一個 FastAPI 應用
app = FastAPI()

# 定義一個 GET 路由
@app.get("/")
def read_root():
    return {"message": "Hello, FastAPI!"}

# 定義一個帶參數的路由
@app.get("/hello/{name}")
def read_item(name: str):
    return {"message": f"Hello, {name}!"}
```

### 啟動服務器

執行以下命令啟動服務：

```
uvicorn main:app --reload
```

### 測試 API

訪問以下網址：

- <http://127.0.0.1:8000/>：返回 `{"message": "Hello, FastAPI!"}`
- <http://127.0.0.1:8000/hello/John>：返回 `{"message": "Hello, John!"}`

## 3. 路由和方法

FastAPI 支持常見的 HTTP 方法如 `GET`、`POST`、`PUT` 和 `DELETE`。



範例：

```
@app.post("/create")
def create_item(item: dict):
    return {"status": "success", "item": item}

@app.put("/update/{item_id}")
def update_item(item_id: int, item: dict):
    return {"status": "updated", "id": item_id, "item": item}

@app.delete("/delete/{item_id}")
def delete_item(item_id: int):
    return {"status": "deleted", "id": item_id}
```

## 4. 使用 Pydantic 模型

Pydantic 用於數據驗證和處理。

定義模型：

```
from pydantic import BaseModel

# 定義資料模型
class Item(BaseModel):
    name: str
    description: str = None
    price: float
    is_offer: bool = False
```

使用模型處理請求：

```
@app.post("/items/")
def create_item(item: Item):
    return {"item_name": item.name, "item_price": item.price}
```

請求範例（使用 Postman 或 cURL）：

```
POST /items/
{
  "name": "Apple",
  "description": "A tasty fruit",
  "price": 1.2,
  "is_offer": true
}
```





## 5. 自動生成 API 文檔

FastAPI 自動生成交互式文檔：

- **Swagger UI** : <http://127.0.0.1:8000/docs>
- **ReDoc** : <http://127.0.0.1:8000/redoc>

## 6. 支援異步處理

FastAPI 原生支持異步函數：

```
import asyncio

@app.get("/async")
async def async_endpoint():
    await asyncio.sleep(2)
    return {"message": "This was an async response!"}
```

## 7. 查詢參數與表單數據

查詢參數：

```
@app.get("/search/")
def search_items(q: str, limit: int = 10):
    return {"query": q, "limit": limit}
```

請求範例：

```
GET /search/?q=FastAPI&limit=5
```

表單數據：

安裝 **python-multipart**：

```
pip install python-multipart
```

接著使用 `Form` 來處理表單數據：

```
from fastapi import Form

@app.post("/form/")
```



```
def handle_form(username: str = Form(...), password: str = Form(...)):
    return {"username": username, "password": password}
```

## 8. 錯誤處理與異常

使用 HTTP 異常：

```
from fastapi import HTTPException

@app.get("/error/{item_id}")
def read_item(item_id: int):
    if item_id != 1:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id}
```

## 9. 整合資料庫

FastAPI 通常與 **SQLAlchemy** 一起使用來操作資料庫，具體範例參見先前的 CRUD + MariaDB 部分。

## 10. 部署到生產環境

在生產環境中，建議使用 **Gunicorn** 配合 **Uvicorn**：

```
pip install gunicorn
gunicorn -w 4 -k uvicorn.workers.UvicornWorker main:app
```

這是 FastAPI 的基本介紹。如果需要進一步深入了解特定功能，請隨時告訴我！



詳細說明fast api





FastAPI 是一個現代化的、快速的 Python 網頁框架，用於構建高效的 API，提供了自動生成文檔、數據驗證和支持異步處理的功能。以下是對 FastAPI 的詳細說明，包括其核心功能和應用方法。

---

## 核心功能特性

### 1. 高性能：

- 基於 ASGI（異步服務網關接口），使用 `Starlette` 作為底層框架和 `Pydantic` 作為數據驗證工具。
- 性能與 Node.js 和 Go 相媲美。

### 2. 簡單易用：

- 提供類似 Flask 的簡單語法，支持自動生成 OpenAPI 規範和文檔。

### 3. 數據驗證和類型支持：

- 使用 `Pydantic` 提供強大的數據驗證功能。
- 支持 Python 3.6+ 的類型提示，提升開發效率和可靠性。

### 4. 支持同步和異步：

- 完全支持 Python 的 `async`/`await`，非常適合處理需要高並發的場景。

### 5. 自動生成 API 文檔：

- 提供交互式 Swagger UI 和 ReDoc 文檔，方便測試和查看 API。

---

## FastAPI 核心概念

### 1. 應用物件：

- 使用 `FastAPI` 類創建應用。

```
from fastapi import FastAPI
app = FastAPI()
```

### 2. 路由：

- 定義路由處理不同的 HTTP 方法（如 `GET`、`POST`）。



```
@app.get("/")
def read_root():
    return {"message": "Hello, FastAPI!"}
```

### 3. 參數：

- 支持路徑參數、查詢參數、表單參數、標頭參數等。

```
@app.get("/items/{item_id}")
def read_item(item_id: int, q: str = None):
    return {"item_id": item_id, "q": q}
```

### 4. 數據模型：

- 使用 `Pydantic` 定義數據模型，並用於請求和響應的驗證。

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    price: float
    is_offer: bool = False
```

### 5. 異步處理：

- 原生支持異步函數，方便處理需要高效並發的場景。

```
import asyncio

@app.get("/async")
async def async_endpoint():
    await asyncio.sleep(1)
    return {"message": "This is an async response!"}
```

---

## 詳細功能與示例

### 1. 路徑參數

FastAPI 支持在路由中傳遞參數，並自動進行類型驗證。

```
@app.get("/users/{user_id}")
def get_user(user_id: int):
    return {"user_id": user_id}
```



- 輸入 `/users/123`，`user\_id` 自動解析為整數。

---

## 2. 查詢參數

查詢參數從 URL 中的 `?` 開始，例如 `/items/?name=apple&price=10`。

```
@app.get("/items/")
def get_items(name: str, price: float = None):
    return {"name": name, "price": price}
```

- 查詢參數可以設置預設值，未提供時返回 `None`。

---

## 3. 數據模型驗證

`Pydantic` 提供強大的數據驗證和結構化功能。

```
from pydantic import BaseModel

class Item(BaseModel):
    name: str
    description: str = None
    price: float
    is_offer: bool = False

@app.post("/items/")
def create_item(item: Item):
    return item
```

- 請求示例：

```
{
  "name": "Laptop",
  "description": "A powerful machine",
  "price": 999.99,
  "is_offer": true
}
```

- 自動驗證數據類型，若格式錯誤，返回詳細的錯誤訊息。



## 4. 自動生成文檔

FastAPI 自動生成兩種文檔：

- **Swagger UI** : <http://127.0.0.1:8000/docs>
  - **ReDoc** : <http://127.0.0.1:8000/redoc>
- 

## 5. 錯誤處理

使用 `HTTPException` 提供自定義錯誤響應。

```
from fastapi import HTTPException

@app.get("/items/{item_id}")
def read_item(item_id: int):
    if item_id == 0:
        raise HTTPException(status_code=404, detail="Item not found")
    return {"item_id": item_id}
```

---

## 6. 異步支持

使用 Python 的異步語法處理耗時操作。

```
@app.get("/delayed/")
async def delayed_response():
    import asyncio
    await asyncio.sleep(2)
    return {"message": "This response was delayed by 2 seconds"}
```

---

## 7. 中間件

中間件允許在每個請求和響應之前執行自定義邏輯。

```
from fastapi import Request
from fastapi.middleware.cors import CORSMiddleware

app.add_middleware(
    CORSMiddleware,
    allow_origins=["*"],
```



```
allow_methods=["*"],
allow_headers=["*"],
)

@app.middleware("http")
async def add_custom_header(request: Request, call_next):
    response = await call_next(request)
    response.headers["X-Custom-Header"] = "Custom Value"
    return response
```

---

## 8. 整合資料庫

FastAPI 通常與 SQLAlchemy 一起使用進行數據庫操作。

```
from sqlalchemy import create_engine, Column, Integer, String
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

DATABASE_URL = "sqlite:///./test.db"
engine = create_engine(DATABASE_URL, connect_args={"check_same_thread": False})
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

class User(Base):
    __tablename__ = "users"
    id = Column(Integer, primary_key=True, index=True)
    name = Column(String, index=True)

Base.metadata.create_all(bind=engine)
```

---

## 9. 測試

FastAPI 提供測試支持，使用 `TestClient` 模擬請求。

```
from fastapi.testclient import TestClient

client = TestClient(app)

def test_read_root():
    response = client.get("/")
    assert response.status_code == 200
    assert response.json() == {"message": "Hello, FastAPI!"}
```



## 10. 部署

在生產環境中，建議使用 **Gunicorn** 與 **Uvicorn** 部署：

```
pip install gunicorn
gunicorn -w 4 -k uvicorn.workers.UvicornWorker main:app
```

這是 FastAPI 的詳細介紹和應用示例。如果您有特定的需求或問題，可以進一步討論！



```
CREATE DATABASE fastapi_crud;
USE fastapi_crud;
```

```
CREATE TABLE items (
  id INT AUTO_INCREMENT PRIMARY KEY,
  name VARCHAR(255) NOT NULL,
  description TEXT,
  price FLOAT NOT NULL
);?
```



以下是對您提供的 SQL 腳本的詳細解釋，以及如何在 MariaDB/MySQL 中正確執行該腳本：

## 腳本功能

### 1. 創建資料庫

```
CREATE DATABASE fastapi_crud;
```

- `CREATE DATABASE`：用於創建新的資料庫。
- `fastapi_crud`：資料庫名稱，可根據需求修改。
- 執行後，會在數據庫伺服器中創建一個名為 `fastapi_crud` 的新資料庫。

### 2. 使用資料庫





```
USE fastapi_crud;
```

- `USE`：選擇指定的資料庫，後續的操作都會應用於該資料庫。

### 3. 創建表

```
CREATE TABLE items (  
    id INT AUTO_INCREMENT PRIMARY KEY,  
    name VARCHAR(255) NOT NULL,  
    description TEXT,  
    price FLOAT NOT NULL  
);
```

這段 SQL 創建了一個名為 `items` 的表，包含以下列：

#### 1. `id`：

- 資料類型：`INT`（整數型）。
- `AUTO_INCREMENT`：該列值會自動遞增，用於唯一標識每條記錄。
- `PRIMARY KEY`：設置 `id` 為主鍵，確保每條記錄唯一。

#### 2. `name`：

- 資料類型：`VARCHAR(255)`，表示最大可存儲 255 個字符的字符串。
- `NOT NULL`：此列必須有值，不能為空。

#### 3. `description`：

- 資料類型：`TEXT`，用於存儲較長的文本內容。
- 沒有指定 `NOT NULL`，表示該列可以為空。

#### 4. `price`：

- 資料類型：`FLOAT`，用於存儲商品價格。
- `NOT NULL`：此列必須有值。

## 如何執行該腳本

1. 進入 **MariaDB 或 MySQL CLI**：打開終端並輸入：



```
mysql -u username -p
```

替換 `username` 為您的數據庫用戶名，並輸入密碼。

2. **執行腳本**：將腳本複製到 MariaDB/MySQL CLI，逐行執行，或保存為一個文件（如 `init.sql`），然後運行：

```
source /path/to/init.sql;
```

3. **檢查結果**：驗證資料庫和表是否正確創建：

```
SHOW DATABASES;  
SHOW TABLES;  
DESCRIBE items;
```

## 示例操作（插入和查詢數據）

插入數據：

```
INSERT INTO items (name, description, price) VALUES  
( 'Laptop', 'A powerful laptop', 999.99),  
( 'Phone', 'A smartphone with high resolution', 599.99);
```

查詢數據：

```
SELECT * FROM items;
```

執行後應返回類似以下的結果：

id	name	description	price
1	Laptop	A powerful laptop	999.99
2	Phone	A smartphone with high resolution	599.99

如果有任何問題，歡迎隨時詢問！