



# 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``：定義數據庫表的欄位。
- ``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` 的连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！



將 `main.py`、`database.py`、`MariaDB` build 在同一個 docker image，一啟用就能使用 `fastapi swagger` 測試 `crud`



要將 `main.py`、`database.py` 和 MariaDB 一起構建到一個 Docker 映像中，使其啟動後即可使用 FastAPI Swagger 測試 CRUD，可以按照以下步驟實現：

## 1. 資料夾結構

確保項目結構如下：

```
.
├── Dockerfile
├── requirements.txt
├── main.py
├── database.py
└── init.sql
```

## 2. 建立 `init.sql`

`init.sql` 用於 MariaDB 的初始化，包括創建數據庫和表：

```
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. 編寫 `Dockerfile`

```
# 使用 MariaDB 官方映像作為基礎
FROM mariadb:10.5

# 設置工作目錄
WORKDIR /app

# 複製初始化 SQL 文件到容器
COPY init.sql /docker-entrypoint-initdb.d/

# 安裝 Python 和 FastAPI 依賴
RUN apt-get update && apt-get install -y python3 python3-pip && \
    pip3 install fastapi uvicorn sqlalchemy pymysql

# 複製應用程式碼
COPY main.py database.py /app/

# 暴露服務端口
```



```
EXPOSE 3306 8000

# 啟動 MariaDB 和 FastAPI
CMD ["sh", "-c", "mysqld & sleep 10 && uvicorn main:app --host 0.0.0.0 --port 8000"]
```

#### 4. 編寫 `requirements.txt`

FastAPI 所需依賴：

```
fastapi
uvicorn
sqlalchemy
pymysql
```

#### 5. 編寫 `main.py`

```
from fastapi import FastAPI, HTTPException, Depends
from sqlalchemy.orm import Session
from sqlalchemy import Column, Integer, String, Float, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker

# FastAPI 應用
app = FastAPI()

# 數據庫設置
DATABASE_URL = "mysql+pymysql://root:@localhost:3306/fastapi_crud"
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()

# 定義數據表模型
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)

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

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

# CRUD 操作
@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

@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 not item:
        raise HTTPException(status_code=404, detail="Item not found")
    return item

@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 not item:
        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

@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 not item:
        raise HTTPException(status_code=404, detail="Item not found")
    db.delete(item)
    db.commit()
    return {"message": "Item deleted"}
```

---

## 6. 構建和運行容器

### 構建 Docker 映像

運行以下命令構建映像：

```
docker build -t fastapi-mariadb .
```

### 運行容器

啟動容器：

```
docker run -p 8000:8000 -p 3306:3306 --name fastapi-crud -d fastapi-mariadb
```

---

## 7. 測試 CRUD 操作

### 測試 API 文檔

訪問 <http://localhost:8000/docs>，可以看到 Swagger 自動生成的文檔，並直接測試 API。





## 測試 CRUD API

使用工具如 Postman 或 `curl`：

### 1. 新增資料：

```
curl -X POST "http://localhost:8000/items/" -H "Content-Type: application/json" -d '{"name": "Laptop", "description": "A laptop"}'
```

### 2. 查詢單筆資料：

```
curl -X GET "http://localhost:8000/items/1"
```

### 3. 更新資料：

```
curl -X PUT "http://localhost:8000/items/1" -H "Content-Type: application/json" -d '{"name": "Updated Laptop", "description": "An updated laptop"}'
```

### 4. 刪除資料：

```
curl -X DELETE "http://localhost:8000/items/1"
```

完成後，這個 Docker 容器即成為一個集成 FastAPI 和 MariaDB 的服務，啟動後即可進行 CRUD 操作測試。



這樣MariaDB用什麼帳密登錄，必須的嗎



在 Docker 中使用 MariaDB 時，默認情況下需要設置一個管理員帳號（`root`）的密碼。如果未提供密碼，MariaDB 無法正常啟動，因此設置帳密是必要的。

在上面提供的 `Dockerfile` 中，未明確設置 MariaDB 的 `root` 密碼，這將導致 MariaDB 啟動時失敗。我們需要在 `Dockerfile` 中加入環境變量來配置 MariaDB 的用戶名和密碼。

以下是改進後的解決方案：

## 改進的 `Dockerfile`

```
# 使用 MariaDB 官方映像作為基礎
FROM mariadb:10.5

# 設置 MariaDB 環境變量
ENV MARIADB_ROOT_PASSWORD=yourpassword
ENV MARIADB_DATABASE=fastapi_crud
ENV MARIADB_USER=fastapi_user
ENV MARIADB_PASSWORD=fastapi_password
```





```
# 設置工作目錄
WORKDIR /app

# 複製初始化 SQL 文件到容器
COPY init.sql /docker-entrypoint-initdb.d/

# 安裝 Python 和 FastAPI 依賴
RUN apt-get update && apt-get install -y python3 python3-pip && \
    pip3 install fastapi uvicorn sqlalchemy pymysql

# 複製應用程式碼
COPY main.py database.py /app/

# 暴露服務端口
EXPOSE 3306 8000

# 啟動 MariaDB 和 FastAPI
CMD ["sh", "-c", "mysqld & sleep 10 && uvicorn main:app --host 0.0.0.0 --port 8000"]
```

---

## 更改 `main.py` 的數據庫連接

確保 `main.py` 中使用了新創建的用户 (`fastapi\_user`) 和密碼 (`fastapi\_password`)：

```
DATABASE_URL = "mysql+pymysql://fastapi_user:fastapi_password@localhost:3306/fastapi_crud"
```

---

## 說明 MariaDB 用戶和密碼設置

1. `MARIADB\_ROOT\_PASSWORD`：
  - 設置 MariaDB 的 `root` 密碼，這是管理員帳戶的密碼，用於管理數據庫伺服器。
2. `MARIADB\_DATABASE`：
  - MariaDB 啟動時自動創建的數據庫名稱。
3. `MARIADB\_USER` 和 `MARIADB\_PASSWORD`：
  - 創建一個普通用戶帳戶 (`fastapi\_user`)，並為該用戶設置密碼 (`fastapi\_password`)。
  - 該用戶將被用來操作 `MARIADB\_DATABASE`。

---

## 為什麼需要設置 MariaDB 密碼？

1. 安全性：
  - MariaDB 默認會使用 `root` 用戶進行數據庫管理，設置密碼是確保數據安全的基本操作。
2. 自動化啟動：



- 如果沒有設置 `MARIADB\_ROOT\_PASSWORD`，MariaDB 容器在啟動時會因無法初始化而失敗。

### 3. 應用程序訪問：

- 通過設置普通用戶和密碼，應用程序可以用該帳戶安全地訪問數據庫。

完成上述改進後，MariaDB 和 FastAPI 的整合將變得更穩定可靠，並具備基本的安全保障。  
如果有其他疑問，歡迎隨時詢問！



FROM mariadb:10.5改為使用python3.9 slim



將基礎映像從 `mariadb:10.5` 改為 `python:3.9-slim`，並在其中安裝 MariaDB 和相關依賴，以下是完整改寫的 `Dockerfile` 和對應的說明：

### 改寫的 `Dockerfile`

```
# 使用 Python 3.9 slim 作為基礎映像
FROM python:3.9-slim

# 更新系統並安裝必要依賴和 MariaDB
RUN apt-get update && apt-get install -y \
    mariadb-server \
    mariadb-client \
    libmariadb-dev \
    gcc \
    libffi-dev \
    libssl-dev \
    && apt-get clean

# 設置 MariaDB 用戶、密碼和數據庫
ENV MARIADB_ROOT_PASSWORD=yourpassword
ENV MARIADB_DATABASE=fastapi_crud
ENV MARIADB_USER=fastapi_user
ENV MARIADB_PASSWORD=fastapi_password

# 複製初始化 SQL 文件到容器
COPY init.sql /docker-entrypoint-initdb.d/

# 安裝 FastAPI 和 Python 依賴
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# 複製應用程式碼
WORKDIR /app
COPY main.py database.py /app/

# 初始化 MariaDB 數據庫
```



```
RUN service mysql start && \
mysql -e "CREATE DATABASE ${MARIADB_DATABASE}" && \
mysql -e "CREATE USER '${MARIADB_USER}'@'%' IDENTIFIED BY '${MARIADB_PASSWORD}'" && \
mysql -e "GRANT ALL PRIVILEGES ON ${MARIADB_DATABASE}.* TO '${MARIADB_USER}'@'%' && \
mysql -e "FLUSH PRIVILEGES"

# 暴露服務端口
EXPOSE 3306 8000

# 啟動 MariaDB 和 FastAPI
CMD ["sh", "-c", "service mysql start && uvicorn main:app --host 0.0.0.0 --port 8000"]
```

---

## 說明

### 1. 基礎映像：

- 使用 `python:3.9-slim` 作為基礎映像，比 `mariadb:10.5` 更靈活，適合安裝 Python 相關依賴和 MariaDB。

### 2. 安裝 MariaDB 和依賴：

- `mariadb-server` 和 `mariadb-client`：安裝 MariaDB 伺服器 and 客戶端。
- `gcc`、`libffi-dev`、`libssl-dev`：編譯和運行 Python 模塊所需的依賴。

### 3. 環境變量：

- 使用 `ENV` 指令設置 MariaDB 的根密碼、數據庫名稱、用戶和密碼，便於初始化。

### 4. 初始化 SQL 文件：

- 將 `init.sql` 複製到 MariaDB 的初始化目錄 `/docker-entrypoint-initdb.d/`，MariaDB 啟動時自動執行。

### 5. 安裝 Python 依賴：

- 使用 `requirements.txt` 安裝 FastAPI 和其他 Python 依賴。

### 6. MariaDB 初始化命令：

- 手動啟動 MariaDB 並設置數據庫和用戶權限。

### 7. 啟動命令：

- 啟動 MariaDB 服務並同時運行 FastAPI。

---

## 相關文件

`requirements.txt`

```
fastapi
uvicorn
sqlalchemy
pymysql
```



`init.sql`

```
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  
);
```

## 構建和運行容器

### 1. 構建映像：

```
docker build -t fastapi-python-mariadb .
```

### 2. 運行容器：

```
docker run -p 8000:8000 -p 3306:3306 --name fastapi-container -d fastapi-python-mariadb
```

## 測試 API

### 測試 Swagger UI：

訪問 <http://localhost:8000/docs> 確保 API 可用。

### 測試 MariaDB：

使用 MySQL 客戶端或工具（如 DBeaver）連接 `localhost:3306`，用 `fastapi\_user` 和 `fastapi\_password` 登錄。

這樣即可實現一個基於 `python:3.9-slim` 的單容器應用，集成 MariaDB 和 FastAPI，啟動後即可完成測試 CRUD 功能！如果有其他問題，歡迎隨時詢問！



詳細說明程式碼



以下是改寫的 `Dockerfile` 及其相關內容的詳細說明，包括文件中的每一部分和執行的作用。



## Dockerfile 詳細說明

```
# 使用 Python 3.9 slim 作為基礎映像
FROM python:3.9-slim
```

### 1. 基礎映像：

- 使用 `python:3.9-slim`，這是一個輕量級 Python 映像，適合在容器中運行應用並安裝必要的依賴。

```
# 更新系統並安裝必要依賴和 MariaDB
RUN apt-get update && apt-get install -y \
    mariadb-server \
    mariadb-client \
    libmariadb-dev \
    gcc \
    libffi-dev \
    libssl-dev \
    && apt-get clean
```

### 2. 安裝系統依賴：

- `mariadb-server` 和 `mariadb-client``：安裝 MariaDB 伺服器 and 客戶端。
- `libmariadb-dev``：MariaDB 的開發庫，用於與 SQLAlchemy 配合使用。
- `gcc`、`libffi-dev`、`libssl-dev``：編譯和運行 Python 模塊所需的系統工具和庫。
- `apt-get clean``：清理暫存文件，減小映像大小。

```
# 設置 MariaDB 用戶、密碼和數據庫
ENV MARIADB_ROOT_PASSWORD=yourpassword
ENV MARIADB_DATABASE=fastapi_crud
ENV MARIADB_USER=fastapi_user
ENV MARIADB_PASSWORD=fastapi_password
```

### 3. 設置環境變量：

- `MARIADB_ROOT_PASSWORD``：設定 MariaDB `root`` 管理員的密碼。
- `MARIADB_DATABASE``：自動創建名為 `fastapi_crud`` 的數據庫。
- `MARIADB_USER` 和 `MARIADB_PASSWORD``：創建一個普通用戶，名稱為 `fastapi_user``，密碼為 `fastapi_password``，該用戶有訪問 `fastapi_crud`` 數據庫的權限。

```
# 複製初始化 SQL 文件到容器
COPY init.sql /docker-entrypoint-initdb.d/
```



#### 4. 初始化 SQL 文件：

- ``init.sql``：用於初始化數據庫，創建數據表和基礎結構。
- MariaDB 在啟動時會自動執行 ``docker-entrypoint-initdb.d/`` 目錄中的 SQL 文件。

```
# 安裝 FastAPI 和 Python 依賴
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt
```

#### 5. 安裝 Python 依賴：

- 將 ``requirements.txt`` 複製到容器中，並通過 ``pip`` 安裝 FastAPI 和其依賴的庫。

```
# 複製應用程式碼
WORKDIR /app
COPY main.py database.py /app/
```

#### 6. 應用程式碼：

- 設置容器內的工作目錄為 ``/app``。
- 複製 ``main.py`` 和 ``database.py`` 到容器的 ``/app`` 目錄。

```
# 初始化 MariaDB 數據庫
RUN service mysql start && \
  mysql -e "CREATE DATABASE ${MARIADB_DATABASE}" && \
  mysql -e "CREATE USER '${MARIADB_USER}'@'%' IDENTIFIED BY '${MARIADB_PASSWORD}'" && \
  mysql -e "GRANT ALL PRIVILEGES ON ${MARIADB_DATABASE}.* TO '${MARIADB_USER}'@'%' && \
  mysql -e "FLUSH PRIVILEGES"
```

#### 7. MariaDB 初始化：

- 手動啟動 MariaDB 服務。
- 創建指定的數據庫和用戶。
- 分配權限給用戶，確保應用可以訪問數據庫。

```
# 暴露服務端口
EXPOSE 3306 8000
```

#### 8. 暴露端口：

- ``3306``：MariaDB 的默認數據庫端口。





- `8000`：FastAPI 的默認服務端口。

```
# 啟動 MariaDB 和 FastAPI
CMD ["sh", "-c", "service mysql start && uvicorn main:app --host 0.0.0.0 --port 8000"]
```

#### 9. 啟動命令：

- 啟動 MariaDB 服務。
- 啟動 FastAPI，運行 `main.py`，並將應用綁定到 `0.0.0.0:8000`。

## `main.py` 和 `database.py` 的作用

#### 1. `database.py`：

- 配置數據庫連接，並定義 SQLAlchemy 的數據表模型。
- 提供會話管理的功能。

#### 2. `main.py`：

- 定義 FastAPI 應用，創建 CRUD 路由。
- 使用 SQLAlchemy 與數據庫交互。

## 運行步驟

### 1. 構建映像

執行以下命令構建 Docker 映像：

```
docker build -t fastapi-python-mariadb .
```

### 2. 運行容器

運行容器並綁定端口：

```
docker run -p 8000:8000 -p 3306:3306 --name fastapi-container -d fastapi-python-mariadb
```

### 3. 測試 API

訪問 <http://localhost:8000/docs> 測試 CRUD 操作。

### 4. 測試數據庫

通過 MySQL 客戶端或工具（如 DBeaver）連接 `localhost:3306`，登錄用戶名 `fastapi_user` 和密碼 `fastapi_password`，驗證數據庫連接是否成功。



如果還有其他疑問，歡迎隨時詢問！



```
docker run -p 8000:8000 -p 3306:3306 --name fastapi-container -d fastapi-python-mariadb?
```



命令 `docker run -p 8000:8000 -p 3306:3306 --name fastapi-container -d fastapi-python-mariadb` 的功能如下：

## 命令分解

1. `docker run`：
  - 啟動一個新的容器。
2. `-p 8000:8000`：
  - 將本地主機的 `8000` 端口映射到容器的 `8000` 端口。
  - 用於訪問 FastAPI 應用。
3. `-p 3306:3306`：
  - 將本地主機的 `3306` 端口映射到容器的 `3306` 端口。
  - 用於訪問容器內運行的 MariaDB。
4. `--name fastapi-container`：
  - 指定容器的名稱為 `fastapi-container`，方便後續操作。
5. `-d`：
  - 以分離模式運行容器（即在後台運行）。
6. `fastapi-python-mariadb`：
  - 要運行的映像名稱。該映像應該是之前通過 `docker build` 命令構建的映像。

## 如何使用該命令

1. **構建映像**：如果尚未構建映像，首先執行以下命令：

```
docker build -t fastapi-python-mariadb .
```

2. **啟動容器**：運行命令：

```
docker run -p 8000:8000 -p 3306:3306 --name fastapi-container -d fastapi-python-mariadb
```





3. **確認容器是否運行：** 使用以下命令檢查容器的運行狀態：

```
docker ps
```

輸出應該包含名為 `fastapi-container` 的容器，並顯示對應的端口映射。

4. **訪問應用：**

- 確保 FastAPI 應用在 `http://localhost:8000` 運行。
- 測試 Swagger 文檔：<http://localhost:8000/docs>。

5. **訪問數據庫：** 使用 MySQL 客戶端或工具（如 DBeaver）連接：

- 主機：`localhost`
- 端口：`3306`
- 用戶名：`fastapi_user`
- 密碼：`fastapi_password`
- 數據庫：`fastapi_crud`

## 常見問題

1. **MariaDB 無法啟動：**

- 確保 Dockerfile 中已正確設置環境變量，如 `MARIADB_ROOT_PASSWORD`。
- 確保初始化文件 `init.sql` 正確複製到 `/docker-entrypoint-initdb.d/`。

2. **無法訪問 FastAPI：**

- 確保容器啟動成功（`docker ps`）。
- 檢查容器日誌：

```
docker logs fastapi-container
```

3. **端口衝突：**

- 如果本地主機的 `8000` 或 `3306` 端口已被佔用，可以改用其他端口。例如：

```
docker run -p 8080:8000 -p 3307:3306 --name fastapi-container -d fastapi-python-mariadb
```

此命令成功執行後，您應該可以通過瀏覽器訪問 FastAPI Swagger 測試 CRUD，並使用客戶端工具操作 MariaDB。如果需要更深入的解釋，隨時告訴我！



詳細說明 `main.py`、`init.sql`



以下是對 `main.py` 和 `init.sql` 的詳細說明，分別涵蓋它們的功能和作用。

## 1. `main.py` 詳細說明

`main.py` 是應用的主要入口，基於 FastAPI 實現了與 MariaDB 的 CRUD 操作。

### 代碼解釋

```
from fastapi import FastAPI, HTTPException, Depends
from sqlalchemy.orm import Session
from sqlalchemy import Column, Integer, String, Float, create_engine
from sqlalchemy.ext.declarative import declarative_base
from sqlalchemy.orm import sessionmaker
```

#### 1. 導入依賴：

- `FastAPI`：用於創建應用和路由。
- `HTTPException`：用於拋出 HTTP 錯誤，如 404。
- `SQLAlchemy`：ORM，用於與 MariaDB 交互。
- `Session`：管理數據庫會話。
- `declarative\_base`：定義數據表模型。

```
app = FastAPI()
```

#### 2. 創建應用：

- 用 `FastAPI()` 創建應用實例，所有路由都將基於此應用定義。

```
DATABASE_URL = "mysql+pymysql://fastapi_user:fastapi_password@localhost:3306/fastapi_crud"
engine = create_engine(DATABASE_URL)
SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine)
Base = declarative_base()
```

#### 3. 數據庫連接設置：

- `DATABASE\_URL`：數據庫的連接 URL，包含：
  - 協議：`mysql+pymysql`
  - 用戶名：`fastapi\_user`
  - 密碼：`fastapi\_password`
  - 主機：`localhost`
  - 端口：`3306`
  - 數據庫名稱：`fastapi\_crud`



- ``create_engine``：創建與 MariaDB 的連接。
- ``SessionLocal``：會話工廠，用於創建數據庫會話。
- ``Base``：所有 ORM 模型的基類。

```
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)
```

#### 4. 定義數據表模型：

- 表名為 ``items``。
- 列定義：
  - ``id``：主鍵，自增。
  - ``name``：字符串，必填。
  - ``description``：字符串，可選。
  - ``price``：浮點數，必填。

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

#### 5. 初始化數據庫結構：

- 根據 ORM 模型創建數據庫表。

```
def get_db():
    db = SessionLocal()
    try:
        yield db
    finally:
        db.close()
```

#### 6. 數據庫會話依賴：

- 每個請求都將打開一個會話，請求結束後自動關閉。

```
@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
```

#### 7. 新增資料 (POST) :

- 接收 `name`、`description` 和 `price` 作為參數。
- 創建一個新項目，保存到數據庫。

```
@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 not item:
        raise HTTPException(status_code=404, detail="Item not found")
    return item
```

#### 8. 查詢單筆資料 (GET) :

- 根據 `item\_id` 查詢數據庫。
- 如果未找到，返回 404 錯誤。

```
@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 not item:
        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
```

#### 9. 更新資料 (PUT) :

- 根據 `item\_id` 查詢數據，若不存在返回 404。
- 更新數據並保存到數據庫。

```
@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 not item:
        raise HTTPException(status_code=404, detail="Item not found")
    db.delete(item)
    db.commit()
    return {"message": "Item deleted"}
```

#### 10. 刪除資料 (DELETE) :



- 根據 `item\_id` 查詢數據，若不存在返回 404。
- 刪除數據並提交事務。

---

## 2. `init.sql` 詳細說明

`init.sql` 是 MariaDB 初始化腳本，用於創建數據庫和基礎表結構。

### 內容解釋

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

#### 1. 創建數據庫：

- `CREATE DATABASE`：創建名為 `fastapi\_crud` 的數據庫。
- `USE`：選擇 `fastapi\_crud` 作為當前操作的數據庫。

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

#### 2. 創建數據表：

- 表名為 `items`。
- 列定義：
  - `id`：主鍵，自動增長。
  - `name`：字符串，必填。
  - `description`：文本類型，可選。
  - `price`：浮點數，必填。

---

`init.sql` 的作用是在 MariaDB 容器啟動時自動執行，確保數據庫和表的結構已準備就緒，便於應用直接使用。

---

## 整體流程

1. MariaDB 啟動時執行 `init.sql`，創建數據庫和表。
2. FastAPI 通過 `main.py` 中的路由與 MariaDB 交互，實現 CRUD 操作。
3. 用戶可通過 Swagger (`/docs`) 或 API 工具（如 Postman）測試功能。



如果有其他問題，歡迎隨時詢問！