FastAPI 與 Flask API

第一部分：基礎概念與兩大框架的對話

在深入探討如何使用 Flask 和 FastAPI 建構 API 之前，理解它們各自的設計哲學以及背後的技術基礎至關重要。這兩個框架不僅僅是工具，它們代表了兩種截然不同的軟體設計理念，這些理念源於它們所依賴的技術標準。

第一節：**Python API** 框架簡介

現代應用程式架構的核心，往往圍繞著 RESTful API（具象狀態傳輸應用程式介面）建構。無論是為網頁前端、行動應用程式還是微服務提供動力，API 都扮演著不可或缺的角色。在 Python 的世界裡，Flask 和 FastAPI 是建構這些 API 的兩大主流選擇。

Flask：備受推崇的微框架

[1]

Flask 自 2010 年問世以來，一直以其「微框架」的哲學聞名 [5]。這意味著它只提供一個最小的核心，主要包含路由（routing）和請求處理（request handling）功能，而將其他所有功能，如資料庫互動、表單驗證或使用者認證，都交由擴充套件（extensions）來實現 [1]。這種設計理念賦予開發者極大的靈活性和選擇權。

Flask 的核心依賴於兩個關鍵函式庫：Werkzeug，一個功能強大的 WSGI（Web Server Gateway Interface）工具集，負責處理底層的請求-回應週期；以及 Jinja2，一個現代化的模板引擎，主要用於渲染 HTML 頁面 [3]。這也揭示了 Flask 最初的設計目標不僅僅是 API，更是為了建構傳統的網頁應用程式。其悠久的歷史和龐大的社群是其主要優勢之一，為開發者提供了豐富的資源和支援 [5]。

FastAPI：為高效能而生的挑戰者

9

相較之下，FastAPI 是一個更現代化的框架，從一開始就是為了建構高效能 API 而設計的 [7]。它的架構基於兩個核心元件：Starlette，一個輕量級的 ASGI（Asynchronous Server Gateway Interface）框架，提供了非同步處理、WebSocket 支援和中介軟體等功能；以及 Pydantic，一個基於 Python 類型提示（type hints）的強大資料驗證函式庫 [1]。

FastAPI 的設計理念更偏向「內建電池」（batteries-included），但這些「電池」是專為 API 開發而設的。它承諾帶來三大核心優勢：極致的執行速度、顯著縮短的開發時間，以及因其現代化設計而減少的人為錯誤 [11]。

表 **1**：框架高層次比較

為了在深入探討前建立一個清晰的概覽，下表總結了兩個框架的核心差異。這不僅僅是功能的羅列，更是設計哲學上的權衡，幫助開發者理解選擇其中一個框架所隱含的意義。

特性	Flask	FastAPI
設計哲學	極簡主義、無主見、可擴充	API 優先、工具整合、效能驅動
核心技術	WSGI、同步 (Synchronous)	ASGI、非同步 (Asynchronous)
資料驗證	需外部函式庫 (如 Marshmallow, Flask-WTF)	內建 Pydantic 整合
API 文件	需手動或透過擴充套件 (如 Flasgger, Flask-RESTX)	自動產生 (基於 OpenAPI 的 Swagger UI & ReDoc)
理想使用情境	通用網站、中小型專案、快速原型開發	高效能 API、微服務、即時應用程式

第二節：架構的鴻溝：**WSGI vs. ASGI**

要真正理解 Flask 和 FastAPI 的本質區別，必須從它們的底層通訊協定——WSGI 和 ASGI——入手。這兩者並非伺服器本身，而是介於網頁伺服器（如 Gunicorn, Uvicorn）和 Python 應用程式（如 Flask, FastAPI）之間的一份「合約」或「規範」，定義了雙方如何溝通 [14]。

WSGI (Web Server Gateway Interface)：同步的穩定主力

[14]

WSGI 是 Python 網頁開發的長期標準，它採用同步模型運作。這意味著每個工作者行程（worker process）在同一時間只能處理一個請求。處理完成後，才能處理下一個請求 [16]。這種模式可以用一個簡單的類比來解釋：想像一位麵包師傅在烤蛋糕，他必須等待蛋糕完全烤好（完成第一個請求），才能開始製作糖霜（處理下一個請求）。

在 WSGI 的世界裡，要實現並行處理（concurrency），通常是透過啟動多個獨立的工作者行程或執行緒。每個行程獨立處理一個請求。雖然這種方式行之有效，但在處理大量 I/O 密集型（I/O-bound）任務時，會消耗較多的系統資源，因為當一個請求在等待資料庫回應或外部 API 回傳時，整個工作者行程都會被阻塞，無法處理其他事務 [1]。Flask 和早期版本的 Django 都是基於 WSGI 設計的 [14]。

ASGI (Asynchronous Server Gateway Interface)：非同步的後起之秀

[9]

ASGI 是 WSGI 的精神繼承者，專為解決非同步程式設計的需求而生。它基於 Python 的 asyncio 函式庫，允許在單一工作者行程內透過事件迴圈（event loop）來並行處理多個請求 [14]。

延續麵包師傅的類比：使用 ASGI，麵包師傅將蛋糕放入烤箱後（發起一個 I/O 等待操作），他不需要站在原地乾等，而是可以立即轉身去製作糖霜（處理另一個請求）。當烤箱發出提示音時（I/O 操作完成），他再回過頭來處理蛋糕。這種在等待期間切換任務的能力，極大地提高了效率，尤其是在處理網路請求、資料庫查詢等需要大量等待時間的場景中。

此外，ASGI 從設計之初就原生支援現代網路協定，如 WebSockets 和 HTTP/2，這些是 WSGI 難以優雅處理的 [15]。FastAPI 完全建立在 ASGI 之上，而 Django 近年來也增加了對 ASGI 的支援，使其能夠同時應對同步和非同步的場景 [14]。

表 2：**WSGI vs. ASGI** 技術細節對比

特性	WSGI (Web Server Gateway Interface)	ASGI (Asynchronous Server Gateway Interface)
模型	同步 (Synchronous)	非同步 (Asynchronous)
並行處理	多行程 / 多執行緒	單行程事件迴圈 (Event Loop)
協定支援	HTTP/1.1	HTTP/1.1, HTTP/2, WebSockets
適用情境	傳統的請求-回應模式	高並行、即時通訊應用
代表性伺服器	Gunicorn, uWSGI, mod_wsgi	Uvicorn, Hypercorn, Daphne

WSGI 與 ASGI 之間的選擇，是區分 Flask 和 FastAPI 最核心的架構決策。這個決策產生了一系列的連鎖效應，深刻地影響了框架的效能、程式設計範式乃至整個生態系統。

首先，這直接決定了效能的上限。FastAPI 的效能優勢主要體現在 I/O 密集型應用中。當應用程式需要頻繁地與資料庫、快取或外部服務互動時，ASGI 的非阻塞特性使其能夠用更少的資源處理更多的並行連線 [1]。相比之下，WSGI 應用需要啟動更多的行程來應對高並行，這會增加記憶體消耗和行程切換的開銷。然而，對於純 CPU 密集型任務，兩者的效能差距會縮小，一個配置得當的 WSGI 伺服器依然非常高效 [21]。

其次，它改變了開發者的程式設計範式。為了充分利用 ASGI 的優勢，開發者必須採用 async def 和 await 關鍵字來編寫非同步程式碼 [6]。這不僅是語法上的改變，更是一種心智模型的轉變，開發者需要理解事件迴圈和協程（coroutines）的工作原理，這對習慣於傳統同步程式設計的開發者來說是一個學習曲線。

最後，這個選擇影響了整個函式庫生態。在 async def 函式中，不能直接使用像 requests 這樣的同步函式庫，因為它會阻塞整個事件迴圈，使非同步的優勢蕩然無存。因此，開發者必須轉向一個全新的、專為 asyncio 設計的函式庫生態，例如使用 httpx 來取代 requests 進行 HTTP 請求，或使用 asyncpg 來非同步地操作 PostgreSQL 資料庫 [10]。這是採用 FastAPI 和非同步範式時一個需要考量的隱性成本。

第二部分：Flask 教學：使用成熟的微框架建構穩健的 Web 服務

本部分將提供一個完整且實用的教學，引導開發者從零開始使用 Flask 建構一個功能完備的 REST API，並深入探討其核心原則和常用模式。

第一節：準備工作

環境設定

24

在開始之前，建立一個乾淨的 Python 開發環境是至關重要的。這可以防止專案之間的依賴衝突。

1. 建立虛擬環境：打開終端機，導航到您的專案目錄，然後執行以下命令：
   Bash
   ```
   python -m venv venv
   ```
   這會在當前目錄下建立一個名為 venv 的虛擬環境。
2. 啟動虛擬環境：
   - 在 macOS/Linux 上：source venv/bin/activate
   - 在 Windows 上：venv\Scripts\activate
3. 安裝 **Flask**：在啟動的虛擬環境中，使用 pip 安裝 Flask：
   Bash
   ```
   pip install Flask
   ```

"Hello, World" API

24

現在，讓我們建立第一個簡單的 API。

1. 建立 **app.py** 檔案：在專案根目錄下建立一個名為 app.py 的檔案。
2. 編寫程式碼：

```python
from flask import Flask, jsonify

# 建立 Flask 應用程式實例
app = Flask(__name__)

# 定義根路由 (endpoint)
@app.route('/')
def hello_world():
    return jsonify({"message": "Hello, World!"})

# 執行應用程式
if __name__ == '__main__':
    app.run(debug=True)
```

   - app = Flask(__name__)：這是 Flask 應用程式的核心。__name__ 是 Python 的一個特殊變數，它代表當前模組的名稱。Flask 使用這個參數來確定應用程式的根路徑，以便找到模板和靜態檔案等資源。這種「顯式應用程式物件」的設計是 Flask 的一個核心理念，它使得應用程式的測試和使用「應用程式工廠」模式成為可能 [4]。
   - @app.route('/')：這是一個裝飾器（decorator），它將 URL 路徑 / 與下面的 hello_world 函式綁定。
   - jsonify({"message": "Hello, World!"})：jsonify 是 Flask 提供的一個輔助函式，它會將 Python 的字典序列化為 JSON 格式的回應，並設定正確的 Content-Type 標頭為 application/json。
   - app.run(debug=True)：這會啟動 Flask 內建的開發伺服器。debug=True 參數會啟用除錯模式，當程式碼發生錯誤時，瀏覽器會顯示詳細的追蹤資訊，並且在程式碼變更後會自動重載伺服器。請注意，這個內建伺服器僅供開發使用，不應在生產環境中部署 [2][4]。
3. 執行應用程式：在終端機中執行：

```bash
python app.py
```

   您將看到伺服器在 http://127.0.0.1:5000/ 上啟動。在瀏覽器或使用 curl 訪問該地址，您應該會看到 { "message": "Hello, World!" } 的 JSON 回應。

可擴充的專案結構

29

雖然單一檔案對於小型專案來說很方便，但隨著應用程式的成長，將所有程式碼放在一個檔案中會變得難以維護。一個更具擴充性的結構是將不同的功能模組化。一個常見的起點是：

```
/my_flask_api

|-- venv/
|-- app/
| |-- __init__.py      # 應用程式工廠和初始化
| |-- routes.py        # API 路由定義
| |-- models.py        # 資料庫模型
| |-- static/
| |-- templates/
|-- config.py          # 設定檔
|-- run.py             # 執行應用程式的腳本
```

我們將在本教學的後續部分逐步實現這種結構，特別是在介紹 Blueprints 和應用程式工廠時。

第二節：建構一個完整的 **CRUD API**

為了專注於 API 的核心機制，我們首先使用一個記憶體內的 Python 列表來模擬資料庫。我們將建立一個管理「任務（tasks）」的 API。

在 app.py 中，我們先定義一個模擬資料庫：

```
Python
```

```
from flask import Flask, jsonify, request

app = Flask(__name__)

# 模擬資料庫
tasks =
next_task_id = 3
```

GET 端點 (讀取資料)

4

- 獲取所有任務：
  Python
  ```
  @app.route('/tasks', methods=)
  def get_tasks():
      return jsonify({'tasks': tasks})
  ```

- 根據 ID 獲取單一任務：
  Python
  ```
  @app.route('/tasks/<int:task_id>', methods=)
  def get_task(task_id):
      task = next((task for task in tasks if task['id'] == task_id), None)
      if task is None:
          return jsonify({'error': 'Task not found'}), 404
      return jsonify({'task': task})
  ```

  這裡的 <int:task_id> 是一個路徑變數轉換器，它確保 task_id 是一個整數，並將其作為參數傳遞給 get_task 函式。如果找不到任務，我們回傳一個包含錯誤訊息的 JSON 和一個 404 Not Found 狀態碼。

POST 端點 (建立資料)

4

- 建立一個新任務：
  Python
  ```
  @app.route('/tasks', methods=)
  def create_task():
      global next_task_id
      if not request.json or not 'title' in request.json:
          return jsonify({'error': 'Missing title'}), 400

      new_task = {
          'id': next_task_id,
          'title': request.json['title'],
          'description': request.json.get('description', ""),
          'done': False
      }
      tasks.append(new_task)
      next_task_id += 1
      return jsonify({'task': new_task}), 201
  ```

  - request.json 或 request.get_json() 用於存取客戶端傳送的 JSON 請求主體 [26]。
  - 我們進行了簡單的驗證，確保請求中包含 title。
  - 成功建立後，我們回傳新建立的任務物件和一個 201 Created 狀態碼，這是 RESTful API 的標準實踐。

PUT 端點 (更新資料)

[4]

- 更新一個現有任務：
  Python
  ```
  @app.route('/tasks/<int:task_id>', methods=)
  def update_task(task_id):
      task = next((task for task in tasks if task['id'] == task_id), None)
      if task is None:
          return jsonify({'error': 'Task not found'}), 404
      if not request.json:
          return jsonify({'error': 'Invalid request'}), 400

      task['title'] = request.json.get('title', task['title'])
  ```

```
    task['description'] = request.json.get('description', task['description'])
    task['done'] = request.json.get('done', task['done'])

    return jsonify({'task': task})
```

DELETE 端點 (刪除資料)

4

- 刪除一個任務：
  Python
  ```
  @app.route('/tasks/<int:task_id>', methods=)
  def delete_task(task_id):
      global tasks
      task = next((task for task in tasks if task['id'] == task_id), None)
      if task is None:
          return jsonify({'error': 'Task not found'}), 404

      tasks = [t for t in tasks if t['id']!= task_id]
      return jsonify({'result': True}), 204
  ```

  成功刪除後，通常回傳一個 204 No Content 狀態碼，表示操作成功但回應主體中沒有內容。

在建構這些端點時，我們觀察到一個 Flask 的核心設計模式：request 物件。我們從 flask 模組中匯入 request，然後在視圖函式中直接使用它（例如 request.json），而無需將其作為參數傳遞 [26]。這之所以可行，是因為 Flask 的「應用程式上下文（application context）」和「請求上下文（request context）」機制。當一個請求進入時，Flask 會將這些上下文「推入」一個堆疊中，使得像

request 和 g（一個用於在請求生命週期內儲存臨時資料的全域物件）這樣的物件對於處理該特定請求的程式碼來說是可用的 [1]。

這種「上下文區域變數（context local）」或「偽全域變數（pseudo-global）」的設計，極大地簡化了函式簽名，提供了便利性。然而，這也帶來了一個隱含的後果：視圖函式（如 create_task）現在對 Flask 的請求上下文產生了隱藏的依賴。它無法在沒有設定這個上下文的情況下被獨立呼叫或測試，這使得程式碼與框架緊密耦合。正如我們將在第五部分看到的，這種設計是從 Flask 遷移到 FastAPI 時的主要痛點之一，因為所有依賴 request 或 g

的程式碼都必須被重構，以顯式地將所需資料作為函式參數傳遞下去[34]。Flask 的便利性在長期來看，可能會帶來架構上的挑戰。

第三節：與 **Flask-SQLAlchemy** 整合

在真實世界的應用中，資料需要被持久化。Flask-SQLAlchemy 是一個流行的擴充套件，它將強大的 SQLAlchemy ORM（物件關聯對應）整合到 Flask 中。

設定與配置

25

1. 安裝擴充套件：
   Bash
   ```
   pip install flask-sqlalchemy
   ```

2. 配置與初始化：修改 app.py。
   Python
   ```
   from flask import Flask, jsonify, request
   from flask_sqlalchemy import SQLAlchemy

   app = Flask(__name__)
   # 設定資料庫連接 URI，這裡我們使用 SQLite
   app.config = 'sqlite:///tasks.db'
   app.config = False # 關閉不必要的追蹤

   # 初始化 SQLAlchemy
   db = SQLAlchemy(app)
   ```

定義模型

25

我們需要定義一個 Python 類別來對應資料庫中的資料表。

Python

```python
class Task(db.Model):
    id = db.Column(db.Integer, primary_key=True)
    title = db.Column(db.String(100), nullable=False)
    description = db.Column(db.String(200), nullable=True)
    done = db.Column(db.Boolean, default=False)

    def to_dict(self):
        return {
            'id': self.id,
            'title': self.title,
            'description': self.description,
            'done': self.done
        }
```

Task 類別繼承自 db.Model，它的屬性使用 db.Column 來定義資料表的欄位。我們還添加了一個 to_dict 方法，方便將模型物件轉換為字典以進行 JSON 序列化。

建立資料庫資料表

37

在第一次執行應用程式之前，需要建立資料庫和資料表。

Python

```python
with app.app_context():
    db.create_all()
```

這段程式碼應該放在模型定義之後，應用程式執行之前。with app.app_context() 確保了資料庫操作在 Flask 的應用程式上下文中執行。

實現 CRUD 邏輯

36

現在，我們將之前的 API 端點重構為使用 db.session 來操作資料庫。

- **GET (Read)**:
  Python
  ```
  @app.route('/tasks', methods=)
  def get_tasks():
      tasks = Task.query.all()
      return jsonify({'tasks': [task.to_dict() for task in tasks]})

  @app.route('/tasks/<int:task_id>', methods=)
  def get_task(task_id):
      task = db.session.get(Task, task_id) # 建議使用 db.session.get
      if task is None:
          return jsonify({'error': 'Task not found'}), 404
      return jsonify({'task': task.to_dict()})
  ```

- **POST (Create)**:
  Python
  ```
  @app.route('/tasks', methods=)
  def create_task():
      data = request.get_json()
      if not data or not 'title' in data:
          return jsonify({'error': 'Missing title'}), 400

      new_task = Task(title=data['title'], description=data.get('description', ''))
      db.session.add(new_task)
      db.session.commit()

      return jsonify({'task': new_task.to_dict()}), 201
  ```

- **PUT (Update)**:
  Python

```python
@app.route('/tasks/<int:task_id>', methods=)
def update_task(task_id):
    task = db.session.get(Task, task_id)
    if task is None:
        return jsonify({'error': 'Task not found'}), 404

    data = request.get_json()
    task.title = data.get('title', task.title)
    task.description = data.get('description', task.description)
    task.done = data.get('done', task.done)

    db.session.commit()
    return jsonify({'task': task.to_dict()})
```

- **DELETE (Delete)**:
  Python
  ```python
  @app.route('/tasks/<int:task_id>', methods=)
  def delete_task(task_id):
      task = db.session.get(Task, task_id)
      if task is None:
          return jsonify({'error': 'Task not found'}), 404

      db.session.delete(task)
      db.session.commit()

      return '', 204 # 回傳空的回應主體和 204 狀態碼
  ```

第四節：使用 **Blueprints** 和應用程式工廠進行擴充

使用 **Blueprints** 進行模組化

4

當應用程式變得複雜時，Blueprints（藍圖）提供了一種將路由組織成模組化群組的方式 [41]。

1. 建立藍圖檔案 (app/tasks_bp.py)：
   Python
   ```
   from flask import Blueprint, jsonify, request
   from.models import db, Task # 假設模型在 models.py

   tasks_bp = Blueprint('tasks', __name__)

   # 將所有 /tasks 相關的路由從 app.py 移到這裡
   # 例如：@tasks_bp.route('/', methods=)
   # 注意路由路徑現在是相對於藍圖的 URL 前綴
   ```

2. 註冊藍圖：在主應用程式檔案中，註冊這個藍圖。
   Python
   ```
   from.tasks_bp import tasks_bp

   app.register_blueprint(tasks_bp, url_prefix='/tasks')
   ```

   現在，所有定義在 tasks_bp 中的路由都會自動加上 /tasks 的前綴。

應用程式工廠模式

27

應用程式工廠（Application Factory）是一個函式，它負責建立和配置 Flask 應用程式實例。這種模式是大型 Flask 應用的推薦實踐，因為它避免了全域應用程式物件，並使得為不同環境（開發、測試、生產）建立不同配置的應用程式變得容易。

1. 建立工廠函式 (app/__init__.py)：
   Python
   ```
   from flask import Flask
   from flask_sqlalchemy import SQLAlchemy
   from.config import Config # 假設設定在 config.py

   db = SQLAlchemy()

   def create_app(config_class=Config):
       app = Flask(__name__)
       app.config.from_object(config_class)
   ```

```
    db.init_app(app)

    from.tasks_bp import tasks_bp
    app.register_blueprint(tasks_bp, url_prefix='/tasks')

    return app
```

注意，擴充套件（如 db）在工廠外部被實例化，然後在工廠內部使用 init_app(app) 來綁定到特定的應用程式實例。這允許擴充套件物件在多個應用程式實例之間共用 [43]。

第三部分：FastAPI 教學：運用現代 Python 工程化高效能 API

本部分將提供一個與 Flask 教學平行的實踐指南，重點展示 FastAPI 的現代化特性，並與 Flask 的方法進行對比。

第一節：FastAPI 入門

環境設定

11

1. 建立並啟動虛擬環境（同 Flask）。
2. 安裝 **FastAPI** 和 **ASGI** 伺服器：FastAPI 建議安裝 fastapi 以及一個 ASGI 伺服器，如 uvicorn。安裝 fastapi[all] 會包含 uvicorn 和其他常用依賴。
   Bash
   ```
   pip install "fastapi[all]"
   ```

您的第一個端點

13

1. 建立 **main.py** 檔案。
2. 編寫程式碼：
   ```python
   from fastapi import FastAPI

   app = FastAPI()

   @app.get("/")
   async def root():
       return {"message": "Hello, World"}
   ```
   - app = FastAPI()：建立 FastAPI 應用程式實例。
   - @app.get("/")：這是一個「路徑操作裝飾器」。與 Flask 的 @app.route 不同，FastAPI 為每個 HTTP 方法（get, post, put, delete 等）提供了專門的裝飾器，這使得程式碼意圖更清晰 [1]。
   - async def root()：FastAPI 原生支援非同步函式。使用 async def 可以讓您在函式內部使用 await 來處理 I/O 密集型操作，而不會阻塞伺服器。

執行應用程式

[9]

FastAPI 沒有內建的開發伺服器，必須依賴一個 ASGI 伺服器來執行 [9]。

1. 在終端機中執行以下命令：
   ```bash
   uvicorn main:app --reload
   ```
   - main: 指的是 main.py 檔案。
   - app: 指的是在 main.py 中建立的 FastAPI 物件。
   - --reload: 這個旗標使伺服器在程式碼變更後自動重載，功能類似 Flask 的除錯模式。

伺服器將在 http://127.0.0.1:8000 上啟動。

第二節：**Pydantic** 與類型提示的威力

這是 FastAPI 與 Flask 最大的區別之一。FastAPI 利用現代 Python 的類型提示（Type Hints）和 Pydantic 函式庫，實現了自動化的資料驗證、序列化和文件產生。

定義資料結構 **(Schemas)**

1

我們使用 Pydantic 的 BaseModel 來定義 API 的資料結構。

1. 建立 **schemas.py** 檔案：

```python
from pydantic import BaseModel
from typing import Optional

class TaskBase(BaseModel):
    title: str
    description: Optional[str] = None

class TaskCreate(TaskBase):
    pass

class Task(TaskBase):
    id: int
    done: bool

    class Config:
        from_attributes = True # Pydantic V2+，舊版為 orm_mode = True
```

   - TaskBase：定義了任務共有的屬性。
   - TaskCreate：用於接收建立任務時的請求主體，它繼承自 TaskBase。
   - Task：用於 API 回應，包含了所有欄位。from_attributes = True 告訴 Pydantic 模型可以直接從 ORM 物件（或其他帶有屬性的物件）中讀取資料，這在與 SQLAlchemy 整合時非常有用。

自動請求驗證

9

現在，我們可以在路徑操作函式中使用這些 Pydantic 模型。

Python

```
# in main.py
from. import schemas

@app.post("/tasks")
def create_task(task: schemas.TaskCreate):
    # 'task' 是一個 Pydantic 模型實例，不是原始的字典
    return {"data": task}
```

- task: schemas.TaskCreate：這行程式碼是 FastAPI 魔法的核心。您只需用 Pydantic 模型來類型提示請求主體參數，FastAPI 就會：
  1. 讀取請求主體作為 JSON。
  2. 將其轉換為對應的 Python 類型。
  3. 使用 TaskCreate 模型對資料進行驗證。
  4. 如果資料無效（例如，title 缺失或類型錯誤），FastAPI 會自動回傳一個詳細的 422 Unprocessable Entity 錯誤回應，清楚地指出哪個欄位出了什麼問題 [1]。這極大地提升了開發體驗，省去了在 Flask 中需要手動編寫的大量驗證邏輯。

類型提示用於參數

13

FastAPI 也將類型提示應用於路徑和查詢參數。

Python

```python
@app.get("/items/{item_id}")
async def read_item(item_id: int, q: Optional[str] = None):
    # FastAPI 會自動驗證 item_id 是整數
    # q 是可選的字串查詢參數
    return {"item_id": item_id, "q": q}
```

FastAPI 會自動進行類型轉換和驗證。如果使用者傳入的 item_id 不是整數，將會收到一個 422 錯誤。

自動序列化

13

使用 response_model 參數，可以確保 API 的回應符合預期的結構。

Python

```python
@app.post("/tasks", response_model=schemas.Task)
def create_task(task: schemas.TaskCreate):
    #... 這裡的邏輯會建立一個包含 id 和 done 的任務物件...
    # 假設 created_task 是一個包含 id, title, description, done 的物件
    return created_task
```

FastAPI 會使用 response_model（這裡的 schemas.Task）來過濾回傳的資料。即使 created_task 物件包含了其他敏感欄位（如密碼雜湊），它們也不會出現在最終的 JSON 回應中。這提供了一層重要的安全保障。

第三節：建構一個完整的非同步 **CRUD API**

與 SQLAlchemy 整合

23

FastAPI 與 SQLAlchemy 的整合模式與 Flask 略有不同，它強調透過依賴注入來管理資料庫會話。

1. 安裝依賴：如果需要非同步支援，還需安裝非同步資料庫驅動，如 asyncpg (for PostgreSQL) 或 aiosqlite (for SQLite)。
   Bash
   ```
   pip install sqlalchemy "asyncpg" "psycopg2-binary"
   ```

2. 設定資料庫 (database.py)：
   Python
   ```
   from sqlalchemy.orm import sessionmaker
   from sqlalchemy.ext.asyncio import create_async_engine, AsyncSession
   from sqlalchemy.ext.declarative import declarative_base

   SQLALCHEMY_DATABASE_URL =
   "postgresql+asyncpg://user:password@host/dbname"

   engine = create_async_engine(SQLALCHEMY_DATABASE_URL)
   SessionLocal = sessionmaker(autocommit=False, autoflush=False, bind=engine,
   class_=AsyncSession)

   Base = declarative_base()
   ```

3. 定義模型 (models.py)：與 Flask-SQLAlchemy 類似，但繼承自我們在 database.py 中定義的 Base。
   Python
   ```
   from sqlalchemy import Column, Integer, String, Boolean
   from.database import Base

   class Task(Base):
     __tablename__ = "tasks"
     id = Column(Integer, primary_key=True, index=True)
     title = Column(String, index=True)
     description = Column(String, nullable=True)
     done = Column(Boolean, default=False)
   ```

4. 建立 **CRUD** 函式 (crud.py)：將所有資料庫操作邏輯封裝在此檔案中，以實現關注點分離。

```python
from sqlalchemy.orm import Session
from sqlalchemy.future import select
from. import models, schemas

async def get_task(db: Session, task_id: int):
    result = await db.execute(select(models.Task).filter(models.Task.id == task_id))
    return result.scalars().first()

async def create_task(db: Session, task: schemas.TaskCreate):
    db_task = models.Task(title=task.title, description=task.description)
    db.add(db_task)
    await db.commit()
    await db.refresh(db_task)
    return db_task
#... 其他 CRUD 函式...
```

依賴注入模式管理資料庫會話

23

這是 FastAPI 中管理資源（如資料庫連線）的標準模式。

1. 建立 **get_db** 依賴 (在 database.py 中)：

```python
async def get_db():
    async with SessionLocal() as session:
        yield session
```

這個 async def 產生器函式會建立一個資料庫會話，透過 yield 將其提供給路徑操作函式，並在請求處理完畢後（無論成功或失敗）自動關閉會話。

2. 注入資料庫會話 (在 main.py 中)：

```python
from fastapi import Depends
from sqlalchemy.orm import Session
```

```
from.database import get_db

@app.post("/tasks", response_model=schemas.Task)
async def create_task_endpoint(task: schemas.TaskCreate, db: Session =
Depends(get_db)):
    return await crud.create_task(db=db, task=task)
```

db: Session = Depends(get_db) 告訴 FastAPI，在執行 create_task_endpoint 之前，需要先執行 get_db 函式，並將其 yield 的值（即資料庫會話）作為 db 參數傳入。

第四節：掌握 **FastAPI** 的核心特性

依賴注入深度解析

46

依賴注入（Dependency Injection, DI）是 FastAPI 設計哲學的基石。它不僅僅用於資料庫會話。

- 共用邏輯：可以建立一個依賴來處理共用的查詢參數。
  Python
  ```
  async def common_parameters(q: Optional[str] = None, skip: int = 0, limit: int = 100):
      return {"q": q, "skip": skip, "limit": limit}

  @app.get("/items/")
  async def read_items(commons: dict = Depends(common_parameters)):
      return commons
  ```
- 認證與授權：DI 是實現安全機制的完美方式。可以建立一個 get_current_user 依賴，它負責驗證請求中的 token 並回傳使用者物件。需要保護的端點只需注入此依賴即可。
- 優勢：DI 將業務邏輯與框架本身解耦，提高了程式碼的重用性，並使得測試變得極為簡單，因為可以在測試中輕鬆地用模擬（mock）物件替換真實的依賴 [51]。

自動互動式 **API** 文件

9

這是 FastAPI 最受歡迎的功能之一。當您執行應用程式後：

1. 訪問 **/docs**：您會看到一個由 Swagger UI 提供的完整、互動式的 API 文件。
2. 訪問 **/redoc**：您會看到另一個由 ReDoc 提供的替代文件介面。

這份文件是 FastAPI 根據您的路徑操作、Pydantic 模型和參數類型提示自動產生的。您無需編寫任何額外的程式碼。開發者可以直接在 Swagger UI 介面中測試每個端點，輸入參數，並查看真實的回應 [12]。

這個功能的實現，源於一個緊密整合的系統：

1. 開發者在程式碼中加入 Python 類型提示 (task: schemas.TaskCreate)。
2. **Pydantic** 利用這個類型提示來進行自動驗證。
3. **FastAPI** 接著利用 Pydantic 模型來進行自動序列化。
4. 同時，FastAPI 檢查這些模型和類型提示，產生符合 **OpenAPI** 規範的 JSON Schema。
5. 最後，這個 OpenAPI Schema 被用來呈現出 Swagger UI 和 ReDoc 的自動文件。

這個「良性循環」意味著，開發者只需遵循現代 Python 的最佳實踐編寫程式碼，就能免費獲得驗證、序列化和文件三大核心功能。這與 Flask 的模式形成鮮明對比，在 Flask 中，這三者通常需要開發者分別使用不同的工具（如 Marshmallow、Flasgger）來手動整合。

使用 **APIRouter** 進行結構化

55

APIRouter 是 FastAPI 中對應 Flask Blueprints 的概念，用於將大型應用程式模組化。

1. 建立路由檔案 (routers/tasks.py)：
   ```Python
   from fastapi import APIRouter

   router = APIRouter()

   @router.get("/")
   async def read_tasks():
   ```

```
    #...
    pass
```

2. 在主應用中包含路由 (main.py)：
```
Python
from.routers import tasks

app = FastAPI()

app.include_router(
    tasks.router,
    prefix="/tasks",
    tags=["tasks"],
)
```

app.include_router 將 tasks 路由模組中的所有路徑操作都加入到主應用中，並可以統一設定路徑前綴和標籤（用於在文件中分組）。

第四部分：進階架構模式與策略決策

在掌握了兩個框架的基礎之後，本部分將從更高層次進行分析，幫助開發者根據專案需求做出明智的技術選型。

第一節：程式碼並排比較

1

為了直觀地比較開發體驗和程式碼的複雜度，下表展示了在兩個框架中實現同一個 POST /tasks 端點所需的程式碼。

表 3：**CRUD** 端點實現比較：**Flask vs. FastAPI**

功能	Flask (使用 Flask-SQLAlchemy)	FastAPI (使用 SQLAlchemy)
路由定義	@app.route('/tasks', methods=) def create_task():	@app.post("/tasks", response_model=schemas.Task, status_code=201) async def create_task(...):
資料驗證	data = request.get_json() if not data or 'title' not in data: abort(400)	(task: schemas.TaskCreate,...) (由 FastAPI 和 Pydantic 自動處理)
資料庫會話	from. import db db.session.add(...)	db: AsyncSession = Depends(get_db) db.add(...)
業務邏輯	new_task = Task(title=data['title']) db.session.add(new_task) db.session.commit()	return await crud.create_task(db=db, task=task) (邏輯封裝在 crud.py 中)
回應	return jsonify(new_task.to_dict()), 201	return created_task (由 response_model 自動序列化)

從上表可以清楚地看出，FastAPI 在程式碼的簡潔性、意圖的明確性（例如，顯式的依賴注入 vs. 隱藏的全域物件）以及關注點分離方面表現更佳。Flask 需要更多樣板程式碼來處理驗證和序列化，而 FastAPI 將這些工作內建於框架的核心流程中。

第二節：生態系統與擴充性

5

Flask 成熟的生態系統

Flask 的主要優勢之一是其龐大且成熟的擴充套件生態系統。幾乎所有可以想像到的功能，

從使用者登入（Flask-Login）、管理後台（Flask-Admin）到表單處理（Flask-WTF），都有對應的、經過社群長期考驗的擴充套件 [7]。這給予了開發者極大的自由度。然而，這也意味著開發者需要自己承擔選擇、整合和維護這些來自不同來源的元件的責任。

FastAPI 現代化的生態系統

FastAPI 的生態系統雖然較新，但正在快速成長。由於許多核心 API 功能（如驗證、文件）已經內建，對擴充套件的需求相對較少 [5]。其生態系統主要圍繞

asyncio 建立，這是一個重要的考量因素。開發者需要確保他們選擇的函式庫與非同步範式相容。

這兩種模式可以歸結為「自備所需」（Bring Your Own）與「整合套件」（Integrated）之間的權衡。Flask 提供了終極的靈活性，而 FastAPI 提供了一套為高效能 API 精心策劃的整合工具集。

第三節：選擇您的框架：決策矩陣

[6]

以下是一些具體的、可操作的建議，幫助您根據專案特性做出選擇。

選擇 **Flask** 的時機：

- 建構傳統網站：當您的主要目標是建構一個伺服器端渲染的傳統網站，而 API 只是其中的一部分功能時，Flask 及其模板引擎 Jinja2 是非常自然且強大的選擇 [7]。
- 團隊熟悉度：如果您的團隊對同步程式設計和 Flask 的生態系統有深厚的經驗，繼續使用 Flask 可以最大化開發效率 [6]。
- 特定擴充套件需求：如果您的專案依賴某個關鍵的 Flask 擴充套件，而這個擴充套件在 FastAPI 生態中沒有成熟的替代品 [45]。
- 快速原型開發：對於小型的、概念驗證性質的專案，Flask 的極簡主義使得啟動和執行變得非常迅速 [7]。

選擇 **FastAPI** 的時機：

- 高效能與高並行：當效能是專案的關鍵指標時，特別是對於 I/O 密集型應用，FastAPI 的非同步架構能帶來顯著的優勢 [1]。

- **API 優先的架構**：如果您正在為單頁應用（SPA）、行動應用或微服務架構建構後端 API，FastAPI 是專為此類場景設計的最佳選擇 [7]。
- **重視開發者體驗**：如果您的團隊重視自動資料驗證、互動式 API 文件和現代 Python 特性所帶來的開發效率提升，FastAPI 將會是一個令人愉悅的選擇 [11]。
- **擁抱非同步**：如果您的專案需要處理即時通訊（如 WebSockets）或可以從 async/await 的程式設計模型中獲益，FastAPI 提供了原生的、一流的支援 [6]。

第五部分：遷移之路：從 Flask 到 FastAPI

這最後一個實踐部分將探討一個常見的真實世界場景：如何將一個現有的 Flask 應用程式遷移到 FastAPI。

第一節：逐步遷移指南

64

遷移過程可以分解為一系列具體的步驟。

表 4：從 **Flask** 到 **FastAPI** 的關鍵遷移步驟

Flask 元件	FastAPI 等效元件與操作
from flask import Flask, request, jsonify	from fastapi import FastAPI, Request。操作：將直接使用 request 物件的程式碼重構為透過函式參數或依賴注入獲取資料。用直接回傳字典或 Pydantic 模型取代 jsonify。
app = Flask(__name__)	app = FastAPI()。操作：移除 __name__ 參數。
@app.route('/path', methods=)	@app.post('/path')。操作：為每個 HTTP 方法使用對應的 FastAPI 裝飾器。

路徑參數 <int:name>	路徑參數 {name}。操作：在函式簽名中用類型提示 name: int 來定義類型。
請求資料 request.args, request.form, request.get_json()	函式參數。操作：將查詢、表單或請求主體資料定義為帶有類型提示的函式參數。
模板 render_template('index.html',...)	templates.TemplateResponse('index.html', {"request": request,...})。操作：需要明確設定 Jinja2Templates 實例，並將 request 物件傳遞給回應。
靜態檔案 (隱式)	app.mount("/static", StaticFiles(directory="static"), name="static")。操作：需要明確掛載靜態檔案目錄。
Blueprint	APIRouter。操作：將藍圖重構為 API 路由器，並使用 app.include_router() 進行註冊。

第二節：常見挑戰與解決方案

[32]

上下文全域物件問題

這是遷移中最棘手的問題。Flask 程式碼庫中廣泛使用的 request 和 g 物件在 FastAPI 中沒有直接的對應物。

- 挑戰：深層巢狀的函式呼叫可能依賴於這些全域物件，使得追蹤資料來源變得困難 [34]。
- 解決方案：必須進行程式碼重構。從路由函式開始，追蹤 request 和 g 的使用情況，並將所需的資料（如使用者 ID、請求主體）作為顯式參數向下傳遞給被呼叫的函式。FastAPI 的依賴注入系統可以幫助管理這些傳遞的資料。

同步到非同步的轉換

- 挑戰：一個完全同步的 Flask 應用程式無法直接利用 FastAPI 的非同步優勢。
- 解決方案：FastAPI 提供了一個平滑的過渡路徑。它可以直接在執行緒池中執行同步的路徑操作函式（即沒有 async 關鍵字的函式）[1]。這意味著您可以先將路由和驗證邏輯遷移到 FastAPI，而暫時保留同步的業務邏輯。之後，再逐步識別出 I/O 密集的瓶頸，並將相關的函式庫（如資料庫驅動、HTTP 客戶端）替換為非同步版本，並將對應的程式碼路徑改為
  async/await。

依賴管理

- 挑戰：Flask 應用程式通常依賴大量擴充套件 [6][7]。
- 解決方案：在遷移時，需要評估每個擴充套件。對於某些擴充套件（如用於資料驗證或 API 文件的），其功能可能已經被 FastAPI 內建。對於其他擴充套件（如資料庫整合），則需要尋找 FastAPI 生態中對應的、通常是非同步優先的解決方案。

結論與最終建議

Flask 和 FastAPI 都是優秀的 Python 網頁框架，但它們服務於不同的目的，體現了不同的設計哲學。

- **Flask** 是一個成熟、靈活、高度可擴充的通用型微框架。它的優勢在於其龐大的社群、豐富的擴充套件生態系統以及極簡主義帶來的自由度。它非常適合建構傳統的伺服器渲染網站、快速原型開發，以及那些團隊對其生態系統有深厚積累的專案。
- **FastAPI** 則是一個現代、高效能、專為 **API** 開發而設計的框架。它的核心優勢在於其基於 ASGI 的非同步架構、與 Pydantic 深度整合所帶來的自動化資料驗證和文件產生，以及依賴注入系統提供的卓越開發體驗。

最終建議：

- 對於新的、以 **API** 為核心的專案，特別是當效能和高並行是重要考量時，強烈建議從 **FastAPI** 開始。它所提供的現代化功能，如類型安全、自動文件和非同步支援，將在專案的長期維護性、穩定性和開發者效率方面帶來巨大的回報。
- 對於那些主要目標是建構傳統網站，或者團隊已擁有豐富 Flask 經驗和成熟工具鏈的

專案，**Flask** 依然是一個非常強大且可靠的選擇。它的靈活性和成熟度不容小覷。

最終，最好的框架是那個最能滿足您專案需求和團隊風格的框架。理解它們各自的優勢和權衡，是做出正確技術決策的第一步。

引用的著作

1. Flask vs FastAPI: An In-Depth Framework Comparison | Better Stack Community, 檢索日期：7月 15, 2025, https://betterstack.com/community/guides/scaling-python/flask-vs-fastapi/
2. What is Flask and use cases of Flask? - DevOpsSchool.com, 檢索日期：7月 15, 2025, https://www.devopsschool.com/blog/what-is-flask-and-use-cases-of-flask/
3. What Is Flask? A Comprehensive Guide to the Python Backend Framework - MetaCTO, 檢索日期：7月 15, 2025, https://www.metacto.com/blogs/what-is-flask-a-comprehensive-guide-to-the-python-backend-framework
4. Welcome to Flask — Flask Documentation (3.1.x), 檢索日期：7月 15, 2025, https://flask.palletsprojects.com/
5. FastAPI vs Flask — A comparison for Python web development - UnfoldAI, 檢索日期：7月 15, 2025, https://unfoldai.com/fastapi-vs-flask/
6. Flask Vs. FastAPI: Which Framework is Right For You | Shakuro, 檢索日期：7月 15, 2025, https://shakuro.com/blog/fastapi-vs-flask
7. FastAPI vs. Flask: Python web frameworks comparison and tutorial - Contentful, 檢索日期：7月 15, 2025, https://www.contentful.com/blog/fastapi-vs-flask/
8. Mastering Flask: A Deep Dive - DEV Community, 檢索日期：7月 15, 2025, https://dev.to/leapcell/mastering-flask-a-deep-dive-56bo
9. FastAPI vs Flask: Comparison Guide to Making a Better Decision - Turing, 檢索日期：7月 15, 2025, https://www.turing.com/kb/fastapi-vs-flask-a-detailed-comparison
10. An Introduction to Using FastAPI - Refine dev, 檢索日期：7月 15, 2025, https://refine.dev/blog/introduction-to-fast-api/
11. FastAPI vs Flask: Key Differences, Performance, and Use Cases - Codecademy, 檢索日期：7月 15, 2025, https://www.codecademy.com/article/fastapi-vs-flask-key-differences-performance-and-use-cases
12. Flask vs FastAPI - Reddit, 檢索日期：7月 15, 2025, https://www.reddit.com/r/FastAPI/comments/1dl7fp2/flask_vs_fastapi/
13. FastAPI, 檢索日期：7月 15, 2025, https://fastapi.tiangolo.com/
14. WSGI vs ASGI: Understanding the Web Server Battle - Medium, 檢索日期：7月 15, 2025, https://medium.com/h7w/wsgi-vs-asgi-understanding-the-web-server-battle-d604622aaa6f
15. Difference Between ASGI and WSGI in Django - GeeksforGeeks, 檢索日期：7月 15, 2025,

https://www.geeksforgeeks.org/python/difference-between-asgi-and-wsgi-in-django/
16. How Python's WSGI vs. ASGI is Like Baking a Cake - Vonage, 検索日期：7月 15, 2025, https://developer.vonage.com/en/blog/how-wsgi-vs-asgi-is-like-baking-a-cake
17. What is WSGI & ASGI in Django & It's Key Features [2025] | Blogs | Free HRMS - Horilla, 検索日期：7月 15, 2025, https://www.horilla.com/blogs/what-is-wsgi-and-asgi-in-django-and-it-s-key-features/
18. How Python's WSGI vs. ASGI is Like Baking a Cake - Reddit, 検索日期：7月 15, 2025, https://www.reddit.com/r/Python/comments/rjyjra/how_pythons_wsgi_vs_asgi_is_like_baking_a_cake/
19. Introduction — ASGI 3.0 documentation, 検索日期：7月 15, 2025, https://asgi.readthedocs.io/en/latest/introduction.html
20. FastAPI vs Flask: what's better for Python app development? - Imaginary Cloud, 検索日期：7月 15, 2025, https://www.imaginarycloud.com/blog/flask-vs-fastapi
21. Benchmarks of FastAPI vs async Flask? - python - Stack Overflow, 検索日期：7月 15, 2025, https://stackoverflow.com/questions/76297879/benchmarks-of-fastapi-vs-async-flask/76298025
22. FastAPI vs Flask: The Showdown for Modern APIs | by Legitt AI | Medium, 検索日期：7月 15, 2025, https://medium.com/@legittai/fastapi-vs-flask-the-showdown-for-modern-apis-2b67975d09b5
23. Using FastAPI with SQLAlchemy. Integrating a Relational Database - Stackademic, 検索日期：7月 15, 2025, https://blog.stackademic.com/using-fastapi-with-sqlalchemy-5cd370473fe5
24. Developing RESTful APIs with Python and Flask - Auth0, 検索日期：7月 15, 2025, https://auth0.com/blog/developing-restful-apis-with-python-and-flask/
25. Flask SQLAlchemy Tutorial for Database - GeeksforGeeks, 検索日期：7月 15, 2025, https://www.geeksforgeeks.org/python/connect-flask-to-a-database-with-flask-sqlalchemy/
26. Python | Build a REST API using Flask - GeeksforGeeks, 検索日期：7月 15, 2025, https://www.geeksforgeeks.org/python/python-build-a-rest-api-using-flask/
27. Design Decisions in Flask — Flask Documentation (3.1.x), 検索日期：7月 15, 2025, https://flask.palletsprojects.com/en/stable/design/
28. Flask Tutorial - GeeksforGeeks, 検索日期：7月 15, 2025, https://www.geeksforgeeks.org/python/flask-tutorial/
29. Demystifying Flask's “Application Factory” - Hackers and Slackers, 検索日期：7月 15, 2025, https://hackersandslackers.com/flask-application-factory/
30. Use a Flask Blueprint to Architect Your Applications - Real Python, 検索日期：7月 15, 2025, https://realpython.com/flask-blueprint/
31. Flask: A Beginner-Friendly Web Framework and Its Real-World Use Cases -

Medium, 検索日期：7月 15, 2025, https://medium.com/@abhishekshaw020/flask-a-beginner-friendly-web-framework-and-its-real-world-use-cases-aca983123ec9
32. Migrating from Flask to FastAPI, Part 2 - Forethought AI Engineering, 検索日期：7月 15, 2025, https://engineering.forethought.ai/blog/2023/02/14/migrating-from-flask-to-fastapi-part-2/
33. Welcome to Flask — Flask Documentation (2.3.x), 検索日期：7月 15, 2025, https://flaskx.readthedocs.io/en/latest/
34. Migrating from Flask to FastAPI, Part 3 - Forethought AI Engineering, 検索日期：7月 15, 2025, https://engineering.forethought.ai/blog/2023/02/28/migrating-from-flask-to-fastapi-part-3/
35. Has anyone migrated to FastAPI? : r/flask - Reddit, 検索日期：7月 15, 2025, https://www.reddit.com/r/flask/comments/1d5xrwj/has_anyone_migrated_to_fastapi/
36. Flask SQLAlchemy (with Examples) - Python Tutorial, 検索日期：7月 15, 2025, https://pythonbasics.org/flask-sqlalchemy/
37. Quick Start — Flask-SQLAlchemy Documentation (3.1.x), 検索日期：7月 15, 2025, https://flask-sqlalchemy.readthedocs.io/en/stable/quickstart/
38. Flask SQLAlchemy Tutorial - Qodo, 検索日期：7月 15, 2025, https://www.qodo.ai/blog/flask-sqlalchemy-tutorial/
39. Flask SQLAlchemy - Tutorialspoint, 検索日期：7月 15, 2025, https://www.tutorialspoint.com/flask/flask_sqlalchemy.htm
40. Understanding Flask Blueprints: A Developer's Guide - Blog - App Generator, 検索日期：7月 15, 2025, https://app-generator.dev/blog/flask-blueprints-a-developers-guide/
41. Application Setup — Flask Documentation (3.1.x), 検索日期：7月 15, 2025, https://flask.palletsprojects.com/en/stable/tutorial/factory/
42. Modular Applications with Blueprints — Flask Documentation (3.1.x), 検索日期：7月 15, 2025, https://flask.palletsprojects.com/en/stable/blueprints/
43. Application Factories — Flask Documentation (3.1.x), 検索日期：7月 15, 2025, https://flask.palletsprojects.com/en/stable/patterns/appfactories/
44. Tutorial - User Guide - FastAPI, 検索日期：7月 15, 2025, https://fastapi.tiangolo.com/tutorial/
45. Moving from Flask to FastAPI - TestDriven.io, 検索日期：7月 15, 2025, https://testdriven.io/blog/moving-from-flask-to-fastapi/
46. Features - FastAPI, 検索日期：7月 15, 2025, https://fastapi.tiangolo.com/features/
47. FastAPI Pydantic - Tutorialspoint, 検索日期：7月 15, 2025, https://www.tutorialspoint.com/fastapi/fastapi_pydantic.htm
48. FastAPI - Pydantic - GeeksforGeeks, 検索日期：7月 15, 2025, https://www.geeksforgeeks.org/python/fastapi-pydantic/
49. Secure Authentication in FastAPI with JWT and PostgreSQL. | by Shubham Adhikari, 検索日期：7月 15, 2025, https://medium.com/@adhikarishubham419/secure-authentication-in-fastapi-wit

h-jwt-and-postgresql-fe08b12c8d77
50. Exploring FastAPI Dependency Injection: A Comprehensive Guide | by DZ | Medium, 検索日期：7月 15, 2025, https://medium.com/@melthaw/exploring-fastapi-dependency-injection-a-comprehensive-guide-103bc48d111f
51. Dependency Injection in FastAPI - GeeksforGeeks, 検索日期：7月 15, 2025, https://www.geeksforgeeks.org/python/dependency-injection-in-fastapi/
52. Mastering Dependency Injection in FastAPI: Clean, Scalable, and Testable APIs - Medium, 検索日期：7月 15, 2025, https://medium.com/@azizmarzouki/mastering-dependency-injection-in-fastapi-clean-scalable-and-testable-apis-5f78099c3362
53. Dependencies - FastAPI, 検索日期：7月 15, 2025, https://fastapi.tiangolo.com/tutorial/dependencies/
54. Fast API for Web Development: 2025 Detailed Review - Aloa, 検索日期：7月 15, 2025, https://aloa.co/blog/fast-api
55. FastAPI Core Features: From Project Structure to Middleware Mastery | by Aziz Marzouki, 検索日期：7月 15, 2025, https://medium.com/@azizmarzouki/fastapi-core-features-from-project-structure-to-middleware-mastery-b4a39b63d283
56. Bigger Applications - Multiple Files - FastAPI, 検索日期：7月 15, 2025, https://fastapi.tiangolo.com/tutorial/bigger-applications/
57. How to structure big FastAPI projects - DEV Community, 検索日期：7月 15, 2025, https://dev.to/timo_reusch/how-i-structure-big-fastapi-projects-260e
58. FastAPI vs. Flask: Comparing the Pros and Cons of Top Microframeworks for Building a REST API in Python - STX Next, 検索日期：7月 15, 2025, https://www.stxnext.com/blog/fastapi-vs-flask-comparison
59. Discussion of FastAPI vs Flask: Key Differences and Use Cases - DEV Community, 検索日期：7月 15, 2025, https://dev.to/atifwattoo/fastapi-vs-flask-key-differences-and-use-cases-4fce/comments
60. Why we choose FastAPI over Flask for building ML applications : r/Python - Reddit, 検索日期：7月 15, 2025, https://www.reddit.com/r/Python/comments/ravkys/why_we_choose_fastapi_over_flask_for_building_ml/
61. Which Is the Best Python Web Framework: Django, Flask, or FastAPI? | The PyCharm Blog, 検索日期：7月 15, 2025, https://blog.jetbrains.com/pycharm/2025/02/django-flask-fastapi/
62. Is FastAPI overtaking popularity from Django? : r/Python - Reddit, 検索日期：7月 15, 2025, https://www.reddit.com/r/Python/comments/16f9ou2/is_fastapi_overtaking_popularity_from_django/
63. Flask vs fastapi : r/flask - Reddit, 検索日期：7月 15, 2025, https://www.reddit.com/r/flask/comments/13pyxie/flask_vs_fastapi/
64. Migrating from Flask to FastAPI - Codegen, 検索日期：7月 15, 2025, https://graph-sitter.com/tutorials/flask-to-fastapi

65. Tutorial - Migrating a Web Application from Flask to FastAPI | Jessica Temporal, 検索日期：7月 15, 2025，https://jtemporal.com/flask-to-fastapi/
66. Tips on migrating from Flask to FastAPI and vice-versa | Jessica Temporal, 検索日期：7月 15, 2025，https://jtemporal.com/tips-on-migrating-from-flask-to-fastapi-and-vice-versa/
67. Migrating from Flask to FastAPI - python - Stack Overflow, 検索日期：7月 15, 2025，https://stackoverflow.com/questions/65795261/migrating-from-flask-to-fastapi
68. FastAPI Best Practices and Conventions we used at our startup - GitHub, 検索日期：7月 15, 2025，https://github.com/zhanymkanov/fastapi-best-practices