Claude Code

新 Repo 設定指南（Python）

補充篇：第八章 ～ 第十四章

測試規範 ｜ CI/CD ｜ 程式碼品質 ｜ 依賴管理 ｜ 安全性 ｜ 文件化 ｜ Git 工作流程

2026 年 4 月

目錄

第八章:測試規範與 Test Case 設定

完整的測試規範是高品質 Python 專案的核心。本章介紹如何以 pytest 為主軸，建立完整的測試架構、命名規範、覆蓋率目標，以及如何整合至 CLAUDE.md。

8.1 Python 測試框架選擇

建議以 pytest 作為主要測試框架，原因如下：

- 語法簡潔，不需繼承 TestCase 類別
- 強大的 fixture 系統，支援依賴注入
- 豐富的 plugin 生態系(pytest-cov、pytest-mock、pytest-asyncio)
- 與 CI/CD 工具整合良好
- 支援參數化測試(@pytest.mark.parametrize)

框架	適用情境	優點	缺點
pytest	所有 Python 專案(推薦)	簡潔、plugin 豐富	學習曲線稍高
unittest	標準函式庫內建	無需安裝、與 Java 相似	語法繁瑣
doctest	文件範例測試	直接嵌入 docstring	僅適合簡單測試

8.2 測試資料夾結構

建議採用以下資料夾結構，清楚區分不同層級的測試：

```
project-root/
├── src/
│   └── myapp/
│       ├── __init__.py
│       ├── models.py
│       └── services.py
├── tests/
│   ├── __init__.py
│   ├── conftest.py          # 全域 fixtures
│   ├── unit/                # 單元測試
│   │   ├── __init__.py
│   │   ├── conftest.py      # 單元測試專用 fixtures
│   │   ├── test_models.py
│   │   └── test_services.py
│   ├── integration/         # 整合測試
```

```
│   │   ├── __init__.py
│   │   ├── conftest.py
│   │   └── test_api.py
│   └── e2e/                    # 端對端測試
│       ├── __init__.py
│       └── test_workflows.py
├── pytest.ini                  # pytest 設定
└── pyproject.toml              # 統一設定檔（可替代 pytest.ini）
```

8.3 pytest.ini / pyproject.toml 測試設定範本

方案 A：使用 pytest.ini

```
# pytest.ini
[pytest]
testpaths = tests
python_files = test_*.py *_test.py
python_classes = Test*
python_functions = test_*
addopts =
    --strict-markers
    --tb=short
    --cov=src
    --cov-report=term-missing
    --cov-report=html:htmlcov
    --cov-fail-under=80
markers =
    unit: 單元測試
    integration: 整合測試（需要資料庫/外部服務）
    e2e: 端對端測試
    slow: 執行時間較長的測試
filterwarnings =
    error
    ignore::DeprecationWarning
```

方案 B：使用 pyproject.toml（推薦，統一設定）

```
# pyproject.toml
[tool.pytest.ini_options]
```

```
testpaths = ["tests"]
python_files = ["test_*.py", "*_test.py"]
python_classes = ["Test*"]
python_functions = ["test_*"]
addopts = [
    "--strict-markers",
    "--tb=short",
    "--cov=src",
    "--cov-report=term-missing",
    "--cov-report=html:htmlcov",
    "--cov-fail-under=80",
]
markers = [
    "unit: 單元測試",
    "integration: 整合測試",
    "e2e: 端對端測試",
    "slow: 執行時間較長的測試",
]
```

8.4 conftest.py 基本範本

conftest.py 定義共用的 fixtures，pytest 會自動載入。

```
# tests/conftest.py
import pytest
from unittest.mock import MagicMock, patch
from myapp.database import DatabaseClient
from myapp.config import Settings


@pytest.fixture(scope="session")
def app_settings():
    """提供測試用設定（不讀取真實 .env）"""
    return Settings(
        database_url="sqlite:///:memory:",
        api_key="test-api-key",
        debug=True,
```

```
    )


@pytest.fixture(scope="function")
def mock_db():
    """每個測試函式獨立的 mock 資料庫"""
    db = MagicMock(spec=DatabaseClient)
    db.is_connected.return_value = True
    return db


@pytest.fixture(scope="function")
def mock_external_api():
    """Mock 外部 API 呼叫"""
    with patch("myapp.services.requests.get") as mock_get:
        mock_get.return_value.status_code = 200
        mock_get.return_value.json.return_value = {"status": "ok"}
        yield mock_get


@pytest.fixture(autouse=True)
def reset_env(monkeypatch):
    """每次測試自動清除環境變數（避免測試間污染）"""
    monkeypatch.delenv("DATABASE_URL", raising=False)
    monkeypatch.delenv("API_KEY", raising=False)
```

8.5 測試命名規範

統一命名規範有助於快速理解測試目的。建議採用 Given-When-Then 風格：

```
# ✅ 推薦：清楚描述情境、動作、預期結果
def
test_calculate_total_given_valid_items_when_discount_applied_returns_correct_to
tal():
    ...

# ✅ 簡短版（適合簡單場景）
```

```
def test_user_login_with_valid_credentials_succeeds():
    ...

def test_user_login_with_invalid_password_raises_auth_error():
    ...

# ✅ 類別分組（相關測試放在同一個 class）
class TestUserService:
    def test_create_user_with_valid_data_returns_user_id(self):
        ...

    def test_create_user_with_duplicate_email_raises_value_error(self):
        ...

    def test_delete_user_when_user_not_found_raises_not_found_error(self):
        ...

# ✅ 參數化測試
@pytest.mark.parametrize("input,expected", [
    (0, "zero"),
    (1, "positive"),
    (-1, "negative"),
])
def test_classify_number_returns_correct_label(input, expected):
    assert classify_number(input) == expected
```

8.6 Coverage 設定

目標覆蓋率：80%（最低）、90%（推薦）。

使用 pyproject.toml 設定 coverage

```
# pyproject.toml
[tool.coverage.run]
source = ["src"]
branch = true                    # 分支覆蓋（更嚴格）
omit = [
    "*/tests/*",
```

```
    "*/migrations/*",
    "*/__init__.py",
    "*/conftest.py",
]

[tool.coverage.report]
show_missing = true
skip_covered = false
fail_under = 80                    # 低於 80% 時 CI 失敗
exclude_lines = [
    "pragma: no cover",
    "def __repr__",
    "if __name__ == .__main__.:",
    "raise NotImplementedError",
    "if TYPE_CHECKING:",
]

[tool.coverage.html]
directory = "htmlcov"
```

8.7 常用 pytest 指令

指令	說明
pytest	執行所有測試
pytest tests/unit/	只執行單元測試
pytest -v	顯示詳細輸出
pytest -k "test_login"	只執行名稱含 test_login 的測試
pytest -m unit	只執行標記為 unit 的測試
pytest --cov=src --cov-report=html	產生 HTML 覆蓋率報告
pytest --lf	只執行上次失敗的測試
pytest -x	遇到第一個失敗立即停止
pytest --tb=long	顯示完整 traceback
pytest -n auto	平行執行(需 pytest-xdist)

8.8 CLAUDE.md 測試規範片段

將以下規範加入 CLAUDE.md，指引 Claude Code 在修改程式碼時自動遵守測試要求：

```
## 測試規範

### 測試框架
- 使用 pytest 作為唯一測試框架
- 測試檔案放在 tests/ 目錄，命名以 test_ 開頭

### 撰寫規則
- 每個 public function/method 都必須有對應的單元測試
- 使用 Given-When-Then 命名風格：test_<功能>_<情境>_<預期結果>
- 不得在測試中使用真實的資料庫、API 或外部服務，一律使用 mock/fixture
- 每次新增功能或修 bug，同步新增/更新對應測試

### 覆蓋率要求
- 最低覆蓋率：80%
- 執行 pytest 前確認所有測試通過：pytest --tb=short

### 禁止事項
- 不得 commit 測試失敗的程式碼
- 不得使用 @pytest.mark.skip 跳過測試（除非附上 issue 連結）
```

第九章:CI/CD 設定(GitHub Actions)

CI/CD(持續整合/持續部署)確保每次 Push 或 PR 都自動執行測試、lint 和型別檢查,防止問題合併進主線。

9.1 完整 CI 工作流程範本

將以下檔案儲存至 .github/workflows/ci.yml:

```
# .github/workflows/ci.yml
name: CI

on:
  push:
    branches: [main, develop]
  pull_request:
    branches: [main, develop]

env:
  PYTHON_VERSION: "3.11"

jobs:
  test:
    name: Test & Lint
    runs-on: ubuntu-latest

    steps:
      - name: Checkout code
        uses: actions/checkout@v4

      - name: Set up Python
        uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
          cache: pip

      - name: Install uv
        run: pip install uv
```

```
      - name: Install dependencies
        run: uv sync --all-extras

      - name: Run ruff lint
        run: uv run ruff check .

      - name: Run ruff format check
        run: uv run ruff format --check .

      - name: Run mypy type check
        run: uv run mypy src/

      - name: Run pytest with coverage
        run: |
          uv run pytest --tb=short \
            --cov=src \
            --cov-report=xml \
            --cov-report=term-missing \
            --cov-fail-under=80

      - name: Upload coverage report
        uses: codecov/codecov-action@v4
        if: always()
        with:
          file: coverage.xml
          fail_ci_if_error: false

  security:
    name: Security Scan
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: ${{ env.PYTHON_VERSION }}
      - run: pip install bandit pip-audit
```

```
      - name: Run bandit
        run: bandit -r src/ -ll
      - name: Run pip-audit
        run: pip-audit
```

9.2 dependabot.yml 設定

自動更新依賴套件，降低安全漏洞風險：

```
# .github/dependabot.yml
version: 2
updates:
  # Python 依賴
  - package-ecosystem: pip
    directory: /
    schedule:
      interval: weekly
      day: monday
      time: "09:00"
    open-pull-requests-limit: 5
    labels:
      - dependencies
      - python
    commit-message:
      prefix: "chore(deps)"

  # GitHub Actions 版本
  - package-ecosystem: github-actions
    directory: /
    schedule:
      interval: weekly
    labels:
      - dependencies
      - github-actions
    commit-message:
      prefix: "chore(actions)"
```

9.3 Release 工作流程

建立 .github/workflows/release.yml 自動發布至 PyPI 或建立 GitHub Release：

```
# .github/workflows/release.yml
name: Release

on:
  push:
    tags:
      - "v*.*.*"

jobs:
  release:
    runs-on: ubuntu-latest
    permissions:
      contents: write
    steps:
      - uses: actions/checkout@v4
      - uses: actions/setup-python@v5
        with:
          python-version: "3.11"
      - run: pip install uv build
      - run: python -m build
      - name: Create GitHub Release
        uses: softprops/action-gh-release@v1
        with:
          files: dist/*
          generate_release_notes: true
```

📌 注意：PR 審查時，建議在 repo settings 中設定「Require status checks to pass before merging」，強制 CI 通過後才能合併。

第十章:程式碼品質工具設定

統一的程式碼風格和型別檢查,是維持大型 Python 專案可維護性的關鍵。以下介紹現代 Python 開發的完整品質工具鏈。

10.1 Ruff — 速度最快的 Python Linter

ruff 以 Rust 撰寫,速度比 flake8、isort、pylint 快 10-100 倍,並整合多種規則集。

```
# pyproject.toml
[tool.ruff]
target-version = "py311"
line-length = 88
src = ["src", "tests"]

[tool.ruff.lint]
select = [
    "E",    # pycodestyle errors
    "W",    # pycodestyle warnings
    "F",    # pyflakes
    "I",    # isort
    "B",    # flake8-bugbear
    "C4",   # flake8-comprehensions
    "UP",   # pyupgrade
    "N",    # pep8-naming
    "S",    # flake8-bandit (security)
    "ANN",  # flake8-annotations
]
ignore = [
    "E501",   # 行長度(由 formatter 處理)
    "ANN101", # self 不需型別標記
    "S101",   # assert 在測試中是允許的
]

[tool.ruff.lint.per-file-ignores]
"tests/**" = ["S", "ANN"]   # 測試不需要 security / 型別標記

[tool.ruff.format]
```

```
quote-style = "double"
indent-style = "space"
line-ending = "lf"
```

10.2 Mypy — 靜態型別檢查

mypy 在執行前偵測型別錯誤，減少 runtime bug：

```
# mypy.ini（或 pyproject.toml 的 [tool.mypy] 區塊）
[mypy]
python_version = 3.11
strict = true                        # 啟用所有嚴格檢查
warn_return_any = true
warn_unused_configs = true
disallow_untyped_defs = true
disallow_incomplete_defs = true
check_untyped_defs = true
no_implicit_optional = true
warn_redundant_casts = true
warn_unused_ignores = true
show_error_codes = true
exclude = [
    "tests/",
    "migrations/",
]

# 第三方套件（無型別定義時忽略）
[mypy-boto3.*]
ignore_missing_imports = true

[mypy-pandas.*]
ignore_missing_imports = true
```

10.3 Pre-commit Hooks — 本地端品質關卡

pre-commit 在 git commit 前自動執行所有品質檢查，防止不合規的程式碼進入 repo：

安裝 pre-commit

```
pip install pre-commit
pre-commit install          # 安裝 hooks（執行一次）
pre-commit run --all-files  # 手動對所有檔案執行
```

.pre-commit-config.yaml 完整範本

```
# .pre-commit-config.yaml
repos:
  # 基本檔案檢查
  - repo: https://github.com/pre-commit/pre-commit-hooks
    rev: v4.5.0
    hooks:
      - id: trailing-whitespace
      - id: end-of-file-fixer
      - id: check-yaml
      - id: check-json
      - id: check-toml
      - id: check-added-large-files
        args: ["--maxkb=1000"]
      - id: check-merge-conflict
      - id: detect-private-key
      - id: no-commit-to-branch
        args: ["--branch", "main", "--branch", "develop"]

  # Ruff lint + format
  - repo: https://github.com/astral-sh/ruff-pre-commit
    rev: v0.3.0
    hooks:
      - id: ruff
        args: [--fix]
      - id: ruff-format

  # Mypy 型別檢查
  - repo: https://github.com/pre-commit/mirrors-mypy
    rev: v1.8.0
    hooks:
      - id: mypy
```

```
        additional_dependencies: [types-requests, types-PyYAML]
        args: [--config-file=mypy.ini]

  # Commit message 規範 (Conventional Commits)
  - repo: https://github.com/compilerla/conventional-pre-commit
    rev: v3.1.0
    hooks:
      - id: conventional-pre-commit
        stages: [commit-msg]
```

10.4 加入 CLAUDE.md 的品質規範片段

```
## 程式碼品質規範

### Linting 與 Formatting
- 所有 Python 程式碼必須通過 ruff check 和 ruff format
- 執行：ruff check . --fix && ruff format .
- 不得使用 # noqa 忽略 lint 錯誤（除非有充分理由並附註說明）

### 型別標記
- 所有 public function 必須有完整的型別標記（含 return type）
- 禁止使用 Any 型別（除非是外部函式庫限制）
- 執行 mypy src/ 確保無型別錯誤

### 提交前
- 在 commit 前執行：pre-commit run --all-files
- 確保所有 hooks 通過後再 commit
```

第十一章:依賴管理與環境設定

現代 Python 專案建議使用 pyproject.toml 統一管理依賴，並以 uv 作為套件安裝工具，兼顧速度與可重現性。

11.1 pyproject.toml 完整範本

```
# pyproject.toml
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "my-project"
version = "0.1.0"
description = "專案描述"
readme = "README.md"
license = { text = "MIT" }
requires-python = ">=3.11"
authors = [{ name = "Your Name", email = "you@example.com" }]

# 正式依賴
dependencies = [
    "fastapi>=0.109.0",
    "pydantic>=2.5.0",
    "sqlalchemy>=2.0.0",
    "python-dotenv>=1.0.0",
    "httpx>=0.26.0",
]

[project.optional-dependencies]
# 開發依賴
dev = [
    "pytest>=7.4.0",
    "pytest-cov>=4.1.0",
    "pytest-asyncio>=0.23.0",
    "pytest-mock>=3.12.0",
    "ruff>=0.3.0",
```

```
    "mypy>=1.8.0",
    "pre-commit>=3.6.0",
    "bandit>=1.7.0",
]

[project.scripts]
my-app = "myapp.main:app"
```

11.2 requirements.txt vs pyproject.toml

比較項目	requirements.txt	pyproject.toml
標準化	非官方標準	PEP 517/518 官方標準
功能完整性	僅依賴清單	完整專案元數據
dev 依賴	需要另建檔案	optional-dependencies
工具整合	部分支援	完整支援(ruff、mypy、pytest)
建議用途	簡單腳本專案	正式套件或多人協作專案(推薦)

✅ 建議: 新專案一律使用 pyproject.toml;舊專案可以用 pip freeze > requirements.txt 生成鎖定版本供 CI 使用。

11.3 虛擬環境管理比較

工具	安裝	建立環境	特色
venv	內建	python -m venv .venv	最基本，無額外功能
uv	pip install uv	uv venv && uv sync	最快速(Rust)，推薦
poetry	pip install poetry	poetry install	依賴解析強，但較慢
conda	獨立安裝	conda create -n env	適合科學計算環境

11.4 uv 快速上手指令

uv 是目前速度最快的 Python 套件管理工具(Rust 實作)，比 pip 快 10-100 倍:

```
# 安裝 uv
```

```
pip install uv

# 建立虛擬環境
uv venv                          # 使用預設 Python 版本
uv venv --python 3.11            # 指定 Python 版本
source .venv/bin/activate        # macOS/Linux
.venv\Scripts\activate           # Windows

# 安裝依賴
uv sync                          # 安裝 pyproject.toml 所有依賴
uv sync --all-extras             # 含 dev 依賴
uv add fastapi                   # 新增正式依賴
uv add --dev pytest              # 新增開發依賴
uv remove fastapi                # 移除依賴

# 執行指令（不需先 activate）
uv run pytest
uv run python -m myapp

# 生成 lock file
uv lock                          # 生成 uv.lock
uv export > requirements.txt     # 匯出為 requirements.txt

# 升級依賴
uv lock --upgrade
```

11.5 .env / .env.example 設定規範

敏感設定(API key、密碼、連線字串)一律使用 .env 管理，並提供 .env.example 作為範本：

```
# .env.example（提交至 repo，不含真實值）
# ── 資料庫設定 ──────────────────────────
DATABASE_URL=postgresql://user:password@localhost:5432/mydb

# ── API 金鑰 ────────────────────────────
OPENAI_API_KEY=sk-xxxx
STRIPE_SECRET_KEY=sk_test_xxxx
```

```
# ── 應用程式設定 ──────────────────────────
DEBUG=false
SECRET_KEY=change-me-in-production
ALLOWED_HOSTS=localhost,127.0.0.1

# ── 第三方服務 ────────────────────────────
REDIS_URL=redis://localhost:6379/0
SENTRY_DSN=
```

```
# Python 中讀取 .env（使用 python-dotenv）
from dotenv import load_dotenv
import os

load_dotenv()  # 自動讀取 .env 檔案

DATABASE_URL = os.getenv("DATABASE_URL", "sqlite:///default.db")
DEBUG = os.getenv("DEBUG", "false").lower() == "true"

# 或使用 pydantic-settings（推薦用於 FastAPI 專案）
from pydantic_settings import BaseSettings

class Settings(BaseSettings):
    database_url: str
    debug: bool = False
    secret_key: str

    class Config:
        env_file = ".env"

settings = Settings()
```

📌 注意：.env 必須加入 .gitignore，只提交 .env.example。若不小心 commit 了 .env，立即使用 git-filter-repo 清除歷史，並輪換所有洩漏的金鑰。

第十二章:安全性設定

安全性設定是 Python 專案中經常被忽略的一環。本章介紹如何使用工具掃描安全漏洞，以及如何正確管理 Secrets。

12.1 bandit — 靜態安全掃描

bandit 自動偵測常見 Python 安全問題(SQL injection、hardcoded 密碼、不安全的 hash 等):

```
# 安裝
pip install bandit

# 基本掃描
bandit -r src/

# 只回報高嚴重性問題
bandit -r src/ -ll

# 輸出 JSON 報告
bandit -r src/ -f json -o bandit-report.json
```

pyproject.toml 中設定 bandit

```
# pyproject.toml
[tool.bandit]
targets = ["src"]
exclude_dirs = ["tests", "migrations"]
skips = [
    "B101",  # assert (測試用)
    "B601",  # paramiko (SSH)
]
severity = "medium"
confidence = "medium"
```

12.2 pip-audit — 依賴漏洞掃描

pip-audit 檢查已安裝套件是否有已知 CVE 漏洞:

```
# 安裝
pip install pip-audit

# 掃描所有依賴
pip-audit

# 掃描指定 requirements.txt
pip-audit -r requirements.txt

# 輸出 JSON 格式
pip-audit --format json

# 自動修復（謹慎使用）
pip-audit --fix
```

12.3 Secrets 管理規範

以下檔案和類型絕對不可以 commit 至 Git repo：

- API Keys（OpenAI、AWS、Stripe、GitHub 等）
- .env 檔案（含真實設定值）
- SSL/TLS 私鑰（*.pem, *.key, *.crt）
- 資料庫連線字串（含密碼）
- SSH 私鑰（id_rsa, id_ed25519 等）
- 服務帳號 JSON（Google Cloud, Firebase）
- JWT secret / cookie secret
- production 設定檔

完整 .gitignore 安全性區塊

```
# .gitignore — 安全性相關

# 環境變數
.env
.env.*
!.env.example

# SSL 憑證與私鑰
```

```
*.pem
*.key
*.crt
*.p12
*.pfx

# 服務帳號
*service-account*.json
*credentials*.json
gcp-*.json

# SSH 金鑰
id_rsa
id_ed25519
*.ssh

# 本地設定
local_settings.py
config.local.py
.secrets/
secrets/
```

12.4 GitHub Actions 中使用 Secrets

在 GitHub Actions 中，透過 Repository Settings > Secrets and Variables > Actions 設定敏感值：

```
# .github/workflows/ci.yml
jobs:
  deploy:
    runs-on: ubuntu-latest
    env:
      # 從 GitHub Secrets 注入（不會出現在 logs）
      DATABASE_URL: ${{ secrets.DATABASE_URL }}
      API_KEY: ${{ secrets.API_KEY }}
    steps:
      - name: Deploy
        run: |
```

```
          echo "Deploying..."
          # 不要用 echo $API_KEY！會洩漏到 logs

# 使用 Environment secrets（更細粒度的控制）
jobs:
  deploy-prod:
    environment: production  # 指定環境
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
```

12.5 CLAUDE.md 安全規則補充

```
## 安全性規範

### 絕對禁止
- 禁止在程式碼中 hardcode 任何 API key、密碼、token
- 禁止 commit .env 檔案（已在 .gitignore）
- 禁止使用 print() 或 logging 輸出敏感資訊

### 必須遵守
- 所有 secret 從環境變數或 Secrets Manager 讀取
- 新增依賴後執行 pip-audit 確認無漏洞
- SQL query 一律使用 parameterized query，禁止字串拼接
- 執行 bandit -r src/ 確保無高嚴重性安全問題

### 輸入驗證
- 所有 API 輸入使用 pydantic 驗證
- 不信任任何來自使用者的輸入
- 檔案上傳需驗證副檔名和 MIME type
```

第十三章:文件化規範

良好的文件是維護專案的基礎。本章涵蓋 docstring 規範、自動文件生成工具,以及 README 標準範本。

13.1 Docstring 規範

建議使用 Google style docstring,可讀性高且與多數工具相容:

```
# Google Style Docstring(推薦)
def calculate_discount(
    price: float,
    discount_rate: float,
    min_price: float = 0.0,
) -> float:
    """計算折扣後的價格。

    Args:
        price: 原始價格(必須大於 0)
        discount_rate: 折扣率(0.0 到 1.0 之間,例如 0.2 代表八折)
        min_price: 最低價格限制,預設為 0

    Returns:
        折扣後的最終價格

    Raises:
        ValueError: 當 price <= 0 或 discount_rate 不在合法範圍時

    Example:
        >>> calculate_discount(100.0, 0.2)
        80.0
        >>> calculate_discount(100.0, 0.2, min_price=90.0)
        90.0
    """
    if price <= 0:
        raise ValueError(f"price 必須大於 0,收到 {price}")
    if not 0 <= discount_rate <= 1:
        raise ValueError(f"discount_rate 必須在 0-1 之間,收到 {discount_rate}")
```

```
    return max(price * (1 - discount_rate), min_price)
```

13.2 MkDocs 文件生成

MkDocs + mkdocstrings 可從 docstring 自動生成 HTML 文件網站：

```
# 安裝
pip install mkdocs mkdocs-material mkdocstrings[python]

# 初始化
mkdocs new .

# 開發模式（hot reload）
mkdocs serve

# 建置靜態網站
mkdocs build
```

```
# mkdocs.yml
site_name: My Project Docs
site_url: https://myorg.github.io/myproject
repo_url: https://github.com/myorg/myproject

theme:
  name: material
  palette:
    - scheme: default
      primary: blue
  features:
    - navigation.tabs
    - search.suggest
    - content.code.copy

plugins:
  - search
  - mkdocstrings:
      handlers:
```

```
        python:
          options:
            docstring_style: google
            show_source: true
            show_root_heading: true

nav:
  - Home: index.md
  - API Reference:
    - Models: api/models.md
    - Services: api/services.md
  - Development Guide: development.md
```

13.3 README.md 標準範本

一份好的 README 應包含專案徽章、功能說明、快速開始，以及貢獻指引：

```
# README.md 標準範本

# 專案名稱

[![CI](https://github.com/org/repo/actions/workflows/ci.yml/badge.svg)](...)
[![Coverage](https://codecov.io/gh/org/repo/branch/main/graph/badge.svg)](...)
[![Python](https://img.shields.io/badge/python-3.11-blue)](...)
[![License](https://img.shields.io/badge/license-MIT-green)](...)

一句話描述專案的用途。

## 功能特色
- ✅ 功能 A
- ✅ 功能 B
- 🚧 功能 C（開發中）

## 快速開始

```bash
# 1. Clone repo
```
```

```
git clone https://github.com/org/repo.git
cd repo

# 2. 建立虛擬環境並安裝依賴
uv venv && source .venv/bin/activate
uv sync --all-extras

# 3. 複製並設定環境變數
cp .env.example .env

# 4. 啟動開發伺服器
uv run python -m myapp
```

```
## 開發指引

### 執行測試
```bash
uv run pytest
```

程式碼品質
```bash
uv run ruff check . --fix
uv run ruff format .
uv run mypy src/
```

License

MIT License — 詳見 LICENSE
```

13.4 API 文件化

FastAPI 會自動生成互動式 API 文件(Swagger UI + ReDoc)，只需確保所有 endpoint 都有完整的型別標記和 docstring：

```
from fastapi import FastAPI
from pydantic import BaseModel

app = FastAPI(
    title="My API",
    description="這是我的 API 說明",
    version="1.0.0",
    docs_url="/docs",      # Swagger UI
    redoc_url="/redoc",    # ReDoc
)

class UserCreate(BaseModel):
    email: str
    name: str

@app.post("/users", summary="建立使用者", tags=["Users"])
async def create_user(user: UserCreate) -> dict:
    """
    建立新的使用者帳號。

    - **email**: 使用者電子郵件（唯一）
    - **name**: 使用者顯示名稱
    """
    return {"id": 1, **user.model_dump()}
```

第十四章:Git 工作流程規範

統一的 Git 工作流程確保團隊協作順暢、歷史記錄清晰可讀，並支援自動化版本發布。

14.1 Branch 命名規範

前綴	用途	範例
feat/	新功能	feat/user-authentication
fix/	Bug 修復	fix/login-null-pointer
hotfix/	緊急 production 修復	hotfix/payment-crash
docs/	文件更新	docs/update-api-guide
chore/	維護、依賴更新	chore/upgrade-fastapi
refactor/	重構(不改變功能)	refactor/extract-user-service
test/	新增/修改測試	test/add-payment-tests
release/	版本發布準備	release/v1.2.0

```
# 建立並切換到新 branch
git checkout -b feat/user-authentication

# 完成後推送
git push -u origin feat/user-authentication

# 禁止直接 push 到 main/develop（透過 pre-commit hooks 或 branch protection 強制執行）
```

14.2 Commit Message 規範(Conventional Commits)

採用 Conventional Commits 格式，可自動生成 CHANGELOG 和 semantic versioning:

```
# 格式
<type>(<scope>): <description>

[optional body]

[optional footer(s)]
```

```
# ── 範例 ──────────────────────────────────
feat(auth): add JWT token refresh mechanism

Implements sliding session window for better UX.
Token TTL can be configured via TOKEN_EXPIRE_MINUTES env var.

Closes #123

# ── type 說明 ─────────────────────────────
feat:     新功能（觸發 minor version 升版）
fix:      Bug 修復（觸發 patch version 升版）
docs:     文件更新
style:    格式調整（不影響邏輯）
refactor: 重構
test:     測試相關
chore:    維護（更新依賴、CI 設定等）
perf:     效能優化
ci:       CI/CD 相關
BREAKING CHANGE: 不相容的 API 變更（觸發 major version 升版）
```

14.3 PR Template

建立 .github/PULL_REQUEST_TEMPLATE.md，統一 PR 描述格式：

```
# .github/PULL_REQUEST_TEMPLATE.md

## 🎯 變更摘要
<!-- 簡短說明這個 PR 做了什麼 -->

## 🔗 關聯 Issue
Closes #<!-- issue 號碼 -->

## 📋 變更類型
- [ ] 新功能 (feat)
- [ ] Bug 修復 (fix)
- [ ] 重構 (refactor)
```

```
- [ ] 文件更新 (docs)
- [ ] 測試 (test)
- [ ] 其他 (chore)

## 🧪 測試說明
<!-- 說明如何測試這個變更 -->
- [ ] 已新增/更新單元測試
- [ ] 已執行 pytest 全部通過
- [ ] 已執行 ruff check 無錯誤

## 📸 截圖（如適用）

## ⚠️ 注意事項
<!-- 審查者需要特別注意的地方 -->

## ✅ Checklist
- [ ] 程式碼已自我 review
- [ ] 文件已更新
- [ ] 沒有 TODO/FIXME 遺留
- [ ] 沒有 console.log / print debug 遺留
```

14.4 Issue Templates

建立 .github/ISSUE_TEMPLATE/ 資料夾，新增以下兩個範本：

Bug Report（bug_report.yml）

```
# .github/ISSUE_TEMPLATE/bug_report.yml
name: Bug Report
description: 回報錯誤或非預期行為
labels: ["bug", "needs-triage"]
body:
  - type: markdown
    attributes:
      value: 感謝您回報 bug！請填寫以下資訊協助我們重現問題。

  - type: textarea
    id: description
```

```
    attributes:
      label: 問題描述
      placeholder: 清楚描述發生了什麼問題
    validations:
      required: true

  - type: textarea
    id: steps
    attributes:
      label: 重現步驟
      value: |
        1. 前往 '...'
        2. 點擊 '...'
        3. 看到錯誤 '...'
    validations:
      required: true

  - type: textarea
    id: expected
    attributes:
      label: 預期行為
    validations:
      required: true

  - type: input
    id: version
    attributes:
      label: 版本
      placeholder: v1.2.3
    validations:
      required: true
```

Feature Request(feature_request.yml)

```
# .github/ISSUE_TEMPLATE/feature_request.yml
name: Feature Request
description: 建議新功能或改善現有功能
labels: ["enhancement"]
```

```
body:
  - type: textarea
    id: problem
    attributes:
      label: 問題描述
      placeholder: 您目前遇到什麼問題？
    validations:
      required: true

  - type: textarea
    id: solution
    attributes:
      label: 建議的解決方案
    validations:
      required: true

  - type: textarea
    id: alternatives
    attributes:
      label: 考慮過的替代方案
    validations:
      required: false
```

14.5 Semantic Versioning 版本規範

遵守 SemVer(語意化版本)格式:MAJOR.MINOR.PATCH

版本變化	觸發條件	範例
MAJOR (x.0.0)	BREAKING CHANGE(不相容 API 變更)	1.0.0 → 2.0.0
MINOR (0.x.0)	新增向下相容的功能(feat)	1.0.0 → 1.1.0
PATCH (0.0.x)	向下相容的 bug 修復(fix)	1.0.0 → 1.0.1

```
# 建立 tag 並推送（觸發 release workflow）
git tag v1.2.0
git push origin v1.2.0
```

```
# 或使用 npm 的 standard-version 自動化
npx standard-version --release-as minor
```

14.6 Branch Protection Rules 建議

在 GitHub Repository Settings > Branches 設定以下保護規則:

- require a pull request before merging(禁止直接 push)
- require approvals: 1(至少 1 人審查通過)
- require status checks to pass(CI 必須通過)
- require branches to be up to date(合併前需 rebase/merge main)
- restrict who can push to matching branches(限制可 push 的人員)
- do not allow bypassing the above settings(管理員也不能繞過)

附錄:補充設定完整 Checklist

新 Repo 建立後,依照以下清單確認所有補充設定均已完成:

測試基礎設定

1. 安裝 pytest, pytest-cov, pytest-mock, pytest-asyncio
2. 建立 tests/unit/, tests/integration/, tests/e2e/ 資料夾
3. 建立 tests/conftest.py(全域 fixtures)
4. 設定 pyproject.toml 中的 [tool.pytest.ini_options]
5. 設定 [tool.coverage.run] 覆蓋率,目標 80%+
6. 將測試規範加入 CLAUDE.md

CI/CD 設定

7. 建立 .github/workflows/ci.yml(lint + type check + test)
8. 建立 .github/dependabot.yml(自動更新依賴)
9. 在 GitHub repo 設定 Branch Protection Rules
10. 設定 Repository Secrets(API_KEY、DATABASE_URL 等)

程式碼品質

11. 安裝 ruff, mypy, pre-commit
12. 設定 pyproject.toml 中的 [tool.ruff] 和 [tool.ruff.lint]
13. 建立 mypy.ini 或設定 [tool.mypy]
14. 建立 .pre-commit-config.yaml
15. 執行 pre-commit install 安裝 hooks
16. 將品質規範加入 CLAUDE.md

依賴管理

17. 確認 pyproject.toml 含 [project.optional-dependencies] dev 區塊
18. 安裝 uv:pip install uv
19. 建立 .env.example 並確認 .env 在 .gitignore 中
20. 執行 uv lock 生成 lock file

安全性

21. 安裝 bandit, pip-audit
22. 執行 bandit -r src/ 確認無高嚴重性問題
23. 執行 pip-audit 確認無已知漏洞
24. 確認 .gitignore 含所有敏感檔案類型

25. 將安全規範加入 CLAUDE.md

文件化

26. 確認所有 public function 有 Google style docstring
27. 建立 README.md（含 badges、快速開始、開發指引）
28. （選用）設定 MkDocs 或 Sphinx 自動文件

Git 工作流程

29. 建立 .github/PULL_REQUEST_TEMPLATE.md
30. 建立 .github/ISSUE_TEMPLATE/bug_report.yml
31. 建立 .github/ISSUE_TEMPLATE/feature_request.yml
32. 安裝 conventional-pre-commit hook（commit message 規範）
33. 設定 GitHub Branch Protection Rules