Pet Project Backend: 최종 가이드

1. 프로젝트 개요 및 아키텍처

본 문서는 "Pet Project Backend"의 시스템 아키텍처, 설계 원칙, 그리고 표준 개발 워크플로우를 정의하는 기술 가이드입니다.

본 프로젝트는 Python Flask를 기반으로 하며, 애플리케이션 팩토리(Application Factory) 패턴과 **블루프린트(Blueprint)**를 활용하여 기능별 모듈화를 지향합니다. 핵심 설계 철학은 **'관심사의 분리(Separation of Concerns)'**로, 모든 코드는 명확한 역할과 책임을 가지는 계층으로 분리됩니다.

---

2. 로컬 개발 환경 설정

2.1. 사전 준비

- Git
- Anaconda or Miniconda

2.2. 초기 설정 절차

1. **리포지토리 복제 (Clone)**

```
git clone <repository_url>
cd pet_project_backend
```

2. **Conda 가상환경 생성**
   - ⚠ **중요: `environment.yml` 파일 수정** `conda env create` 명령어를 실행하기 전, 반드시 `environment.yml` 파일을 텍스트 편집기로 열어 맨 아래에 있는 `prefix:` 로 시작하는 줄을 찾아 **삭제**해주세요. 이 줄은 환경을 생성한 사람의 개인 컴퓨터 경로이므로, 삭제해야만 각 팀원의 환경에 맞게 설치됩니다.

```
conda env create -f environment.yml
```

3. **가상환경 활성화**

```
conda activate pet_project_backend
```

2.3. 비밀 파일 설정 (`.env` 및 `secrets`)

Git으로 공유되지 않는 민감한 파일들은 아래의 안내에 따라 설정해야 합니다.

1. **.env 파일 생성** 프로젝트 최상위 폴더의 `.env.example` 파일을 복사하여 `.env` 파일을 새로 만듭니다. `.env` 파일 안의 변수들을 자신의 로컬 환경에 맞게 수정합니다. `FLASK_ENV`, `JWT_SECRET_KEY` 등 팀 공용으로 사용하는 값은 팀 리드에게 문의하세요.
2. **secrets 폴더 내 키 파일 배치** 팀 리드로부터 보안 채널(슬랙 DM 등)을 통해 아래의 키 파일들을 전달받습니다.
    - `your-dev-firebase-key.json` (개발용 Firebase 키)
    - `your-test-firebase-key.json` (테스트용 Firebase 키)
    - `your_google_client_secret.json` (Google OAuth용 클라이언트 키)

    전달받은 파일들을 `pet_project_backend/secrets/` 폴더 안에 저장합니다. `.env` 파일에 작성된 경로와 파일명이 일치해야 합니다.

---

3. 의존성 관리: 라이브러리 추가 및 공유

개발 중 새로운 라이브러리를 설치한 경우, 반드시 다음 절차를 따라 팀원 전체에 공유해야 합니다.

1. **라이브러리 설치:** 현재 활성화된 가상환경에 필요한 라이브러리를 설치합니다.

```
conda install -c conda-forge <package_name>  최우선
conda install <package_name> 우선
# 또는 pip install <package_name> 쩔수
```

2. **environment.yml 파일 업데이트:** 아래 명령어를 실행하여 현재 환경의 패키지 목록을 `environment.yml` 파일에 덮어씁니다.

```
conda env export --no-builds > environment.yml
```

3. **커밋 및 푸시:** 변경된 `environment.yml` 파일을 커밋하고 푸시하여 팀원들에게 공유합니다. 다른 팀원들은 `conda env update --file environment.yml --prune` 명령으로 자신의 환경을 업데이트할 수 있습니다.

---

4. 프로젝트 구조 해설

```
/pet_project_backend/
|
|-- /app/                       # Flask 애플리케이션 코어
|   |-- /api/                   # 기능별 API (블루프린트)
|   |   |-- /auth/
|   |   `-- /pets/
|   |-- /core/                  # 핵심 공통 모듈 (설정, 보안)
|   |-- /models/                # 데이터 구조 정의 (데이터 클래스)
|   |-- /schemas/               # 데이터 유효성 검증 및 직렬화 (Marshmallow)
|   `-- __init__.py             # 애플리케이션 팩토리
```

```
|
|-- /ml_models/                 # 머신러닝 관련 코드 (분리된 영역)
|
|-- /secrets/                   # Git 추적 제외된 비밀 키 파일 저장소
|
|-- run.py                      # 앱 서버 실행 스크립트
|-- .env                        # Git 추적 제외된 환경 변수 파일
|-- .env.example                # .env 파일의 템플릿
|-- .gitignore                  # Git 추적 제외 목록
`-- environment.yml             # Conda 환경 설정 파일
```

- `run.py`: 애플리케이션 서버를 실행하는 유일한 진입점입니다.

```python
# run.py
import os
from app import create_app # app 패키지로부터 create_app 함수를 가져옵니다.

# 1. 환경 변수 'FLASK_ENV'를 읽어와 현재 실행 환경을 결정합니다.
#    변수가 없으면 기본값으로 'development'를 사용합니다.
env = os.getenv('FLASK_ENV', 'development')

# 2. 결정된 환경(예: 'development')에 맞는 앱 인스턴스를 생성합니다.
app = create_app(env)

# 3. 이 스크립트가 직접 실행될 때만 Flask 개발 서버를 구동합니다.
if __name__ == '__main__':
    # app.config에 저장된 호스트, 포트, 디버그 설정을 사용합니다.
    app.run(
        host=app.config.get('HOST'),
        port=app.config.get('PORT'),
        debug=app.config.get('DEBUG', False)
    )
```

- `app/__init__.py`: 애플리케이션 팩토리(`create_app`)가 위치하며, 앱의 생성과 설정을 총괄합니다.
- `app/core/`: `config.py`, `security.py` 등 프로젝트 전반에 영향을 미치는 핵심 로직을 담습니다.
- `app/api/`: 각 기능 도메인별 Blueprint가 위치합니다. 하위 폴더는 `routes.py`, `services.py` 등으로 구성됩니다.
- `app/models/`: Firestore에 저장될 데이터의 구조를 `@dataclass`를 이용해 정의합니다.
- `app/schemas/`: Marshmallow를 사용해 API 요청/응답 데이터의 유효성을 검증하고 형식을 변환(직렬화)합니다.

---

5. 기술 FAQ: 핵심 개념 상세 해설

1. 팩토리 패턴(Factory Pattern)이 무엇이며, 우리 프로젝트는 왜 `create_app()` 함수를 사용하나요?

- **개념:** 애플리케이션 객체의 생성 및 설정을 하나의 함수 안에 캡슐화하는 디자인 패턴입니다.
- **목적:** 순환 참조 방지, 테스트 용이성 향상, 설정의 유연한 주입을 위해 사용합니다.
- **단계별 동작 방식:**

1. `create_app` 함수가 호출되면, 비어있는 Flask 앱 객체를 생성합니다.
2. `config_by_name` 딕셔너리를 통해 전달받은 환경 이름(`development`, `testing` 등)에 맞는 설정 클래스를 로드합니다.
3. Firebase 등 공용 서비스를 초기화합니다.
4. `app/api` 폴더에 정의된 모든 블루프린트를 앱에 등록하여 API 엔드포인트를 활성화합니다.
5. 모든 설정이 완료된 앱 인스턴스를 반환합니다.

- **코드 예시 (`app/__init__.py`)**

```
from flask import Flask
from app.core.config import config_by_name # 환경별 설정 클래스 딕셔너리
import firebase_admin
from firebase_admin import credentials

def create_app(config_name: str = 'development'):
    """
    애플리케이션 팩토리 함수.
    환경 이름(config_name)을 받아 해당 환경에 맞는 앱 인스턴스를 생성하고 반환합
니다.
    """
    # --- 1단계: 뼈대 생성 ---
    # 기본적인 Flask 애플리케이션 객체를 생성합니다.
    app = Flask(__name__)

    # --- 2단계: 설정 주입 ---
    # config_name(예: 'development')에 해당하는 설정 클래스를 찾아 앱에 로드합니
다.
    app.config.from_object(config_by_name[config_name])

    # --- 3단계: 핵심 기능 초기화 ---
    # Firebase Admin SDK를 초기화합니다. 앱이 여러 번 로드되더라도 중복 초기화되
지 않도록 방지합니다.
    if not firebase_admin._apps:
        cred_path = app.config['FIREBASE_CREDENTIALS_PATH']
        cred = credentials.Certificate(cred_path)
        firebase_admin.initialize_app(cred)

    # --- 4단계: 기능 부품 조립 (블루프린트 등록) ---
    # 각 기능별로 정의된 블루프린트들을 앱에 등록합니다.
    # 이 시점에 블루프린트를 import하여 순환 참조를 방지합니다.
    from app.api.auth.routes import auth_bp
    from app.api.pets.routes import pets_bp

    # url_prefix를 지정하여 API 엔드포인트의 경로를 설정합니다.
    # 예: /api/auth, /api/pets
    app.register_blueprint(auth_bp, url_prefix='/api/auth')
    app.register_blueprint(pets_bp, url_prefix='/api/pets')

    # --- 5단계: 완성품 반환 ---
    # 모든 설정과 기능이 조립된 최종 app 객체를 반환합니다.
    return app
```

2. 애플리케이션의 '인스턴스화'는 무엇을 의미하나요?

- **개념:** 클래스(설계도)를 바탕으로, 메모리 상에서 실제 동작하는 객체(인스턴스)를 만드는 과정입니다.
- `app = create_app()` **실행 시 단계별 과정:**
    1. **객체 생성:** `app = Flask(__name__)`를 통해 기본 Flask 객체가 메모리에 생성됩니다.
    2. **상태 부여:** `app.config.from_object(...)`를 통해 설정값들이 객체의 속성으로 저장됩니다.
    3. **능력 부여:** Firebase 등 외부 서비스가 초기화되고, `app` 객체는 외부와 통신할 수 있는 능력을 갖게 됩니다.
    4. **기능 확장:** `app.register_blueprint(...)`를 통해 URL과 처리 함수가 매핑된 라우팅 테이블이 구축됩니다.

3. 순환 참조(Circular Import) 문제가 무엇이며, 팩토리 패턴이 어떻게 해결하나요?

- **개념:** 두 개 이상의 Python 모듈이 서로를 임포트하여 발생하는 무한 루프 문제입니다.
- **문제 발생 시나리오:** 만약 `app` 객체가 전역 변수라면, `routes.py`는 `app`을 임포트하고, `app`은 다시 `routes.py`의 블루프린트를 임포트해야 하므로 순환 참조가 발생합니다.
- **팩토리 패턴의 해결 방식:** 객체 생성 시점과 기능 등록 시점을 분리합니다. `create_app` 함수 내에서 `app` 객체를 먼저 생성한 뒤, 나중에 블루프린트를 임포트하여 등록합니다. 이로써 `routes.py`는 더 이상 `app` 객체를 직접 임포트할 필요가 없어지므로 의존성의 고리가 끊어집니다.

4. 동적 설정 주입(Dynamic Configuration)이 무엇인가요?

- **개념:** 애플리케이션 실행 시점에 환경(개발, 테스트 등)에 따라 다른 설정값을 적용하는 기법입니다.
- **우리 프로젝트 적용 방식:**
    1. `app/core/config.py`에 `Config`를 상속받는 `DevelopmentConfig`, `TestingConfig` 등 환경별 설정 클래스를 정의합니다.
    2. `create_app(config_name)` 함수는 인자로 받은 `config_name`에 맞는 클래스를 `config_by_name` 딕셔너리에서 찾아 설정을 로드합니다.
    3. `run.py`에서는 `os.getenv('FLASK_ENV')`를 통해 환경 이름을 결정하고, `create_app`에 전달하여 해당 환경에 맞는 앱을 실행합니다.
- **코드 예시 (`app/core/config.py`)**

```
import os
from dotenv import load_dotenv

# .env 파일의 변수들을 환경 변수로 로드
load_dotenv()

class Config:
    """기본 설정 (모든 환경에서 공유)"""
    HOST = os.getenv('FLASK_RUN_HOST', '0.0.0.0')
    PORT = int(os.getenv('FLASK_RUN_PORT', 5000))
    JWT_SECRET_KEY = os.getenv('JWT_SECRET_KEY')
    GOOGLE_CLIENT_SECRETS_PATH = os.getenv('GOOGLE_CLIENT_SECRETS_PATH')

class DevelopmentConfig(Config):
    """개발 환경 전용 설정"""
    DEBUG = True
    FIREBASE_CREDENTIALS_PATH = os.getenv('DEV_FIREBASE_CREDENTIALS_PATH')
```

```
class TestingConfig(Config):
    """테스트 환경 전용 설정"""
    TESTING = True
    DEBUG = False
    FIREBASE_CREDENTIALS_PATH = os.getenv('TEST_FIREBASE_CREDENTIALS_PATH')

# 문자열 이름을 실제 설정 클래스와 매핑
config_by_name = dict(
    development=DevelopmentConfig,
    testing=TestingConfig
)
```

5. 블루프린트(Blueprint)란 무엇이며, 어떻게 동작하나요?

- **개념:** 거대한 애플리케이션을 기능 단위로 나눈 **'기능별 미니 앱 설계도'**입니다.
- **단계별 동작 과정:**
  1. **설계도 작성 (`routes.py`):** `pets_bp = Blueprint(...)`로 설계도를 만들고, `@pets_bp.route(...)`로 URL 규칙을 명시합니다.
  2. **설계도 제출 (`__init__.py`):** `create_app` 함수 안에서 `from ... import pets_bp`로 설계도를 가져옵니다.
  3. **기능 조립 (`__init__.py`):** `app.register_blueprint(pets_bp, ...)`를 통해 설계도의 내용이 실제 앱의 라우팅 테이블에 등록되어 비로소 기능이 활성화됩니다.
- **코드 예시**
  - `app/api/pets/routes.py` **(설계도 작성)**

```
from flask import Blueprint

# 'pets'라는 이름으로 블루프린트 설계도를 생성합니다.
pets_bp = Blueprint('pets', __name__)

# '/' 경로에 대한 GET 요청 규칙을 설계도에 추가합니다.
@pets_bp.route('/', methods=['GET'])
def get_all_pets():
    return "List of all pets"
```

  - `app/__init__.py` **(기능 조립)**

```
# ... create_app 함수 내부 ...

# 'pets' 블루프린트 설계도를 가져옵니다.
from app.api.pets.routes import pets_bp

# 설계도를 실제 앱에 등록합니다.
# 이제 /api/pets/ 경로로 오는 요청은 pets_bp가 처리합니다.
app.register_blueprint(pets_bp, url_prefix='/api/pets')
```

6. '저수준 비즈니스 로직'이란 무엇이며, `services` 폴더의 역할은 무엇인가요?

- **개념:**
  - **저수준 로직:** **'어떻게'**에 집중하며, 기술적인 세부사항을 다룹니다. (예: DB에 데이터 쓰기)
  - **고수준 로직:** **'무엇을'**에 집중하며, 실제 비즈니스 정책을 다룹니다. (예: "사용자 등급에 따라 글쓰기 권한 부여")
- **`services` 폴더의 역할:** 우리 프로젝트에서 `app/services/`는 여러 기능에서 공통으로 사용하는 저수준의 공유 인프라 서비스를 정의합니다. 예를 들어, `firebase_service.py`는 'pets'나 'users' 도메인을 전혀 모른 채, 오직 "Firestore의 특정 컬렉션에 문서를 생성하라"는 기술적인 명령만 수행합니다.
- **코드 예시 (`app/services/firebase_service.py`)**

```
from firebase_admin import firestore

# Firestore 클라이언트 인스턴스를 가져옵니다.
db = firestore.client()

def create_document(collection_name: str, data: dict) -> str:
    """
    [저수준 함수] 특정 컬렉션에 데이터를 받아 문서를 생성합니다.
    - 이 함수는 '무엇을' 저장하는지(user, pet 등) 전혀 관심이 없습니다.
    - 오직 '어떻게' Firestore에 저장하는지에 대한 기술만 알고 있습니다.
    """
    # 1. 컬렉션 이름과 데이터라는 기술적인 파라미터를 받습니다.
    # 2. Firestore 클라이언트를 사용해 문서를 추가하는 기술적인 작업을 수행합니다.
    update_time, doc_ref = db.collection(collection_name).add(data)

    # 3. 생성된 문서의 ID(기술적 결과)를 반환합니다.
    return doc_ref.id
```

7. `/models` 디렉토리와 '데이터의 영속적인 구조'는 무엇을 의미하나요?

- **개념:**
  - **컬렉션/필드:** Firestore에서 '컬렉션'은 문서들의 그룹(테이블과 유사), '필드'는 문서 내의 데이터 항목(컬럼과 유사)을 의미합니다.
  - **'필드를 클래스로 정의':** 데이터베이스가 스키마를 강제하지 않더라도, 코드 수준에서 데이터 구조의 일관성을 유지하기 위한 약속입니다. 우리 프로젝트는 `@dataclass` 사용을 표준으로 합니다.
  - **영속적인 구조:** 애플리케이션이 꺼져도 데이터베이스에 계속 유지되는 데이터의 구조를 의미하며, `app/models`의 데이터 클래스가 이를 표현합니다.
- **코드 예시 (`app/models/pet.py`)**

```
from dataclasses import dataclass
from datetime import date

# @dataclass 데코레이터는 __init__, __repr__ 등 boilerplate 코드를 자동으로 생성해줍니다.
@dataclass
class Pet:
```

```
    """
    Firestore의 'pets' 컬렉션에 저장될 문서의 '영속적인 구조'를
    Python 코드로 명확하게 정의하는 클래스입니다.
    """
    # --- 필드 정의 ---
    # 이 클래스의 속성들은 Firestore 문서의 필드(key)에 해당합니다.
    id: str           # 문서의 고유 ID (Firestore에서 자동 생성)
    owner_id: str     # 반려동물 주인의 사용자 ID
    name: str         # 반려동물 이름
    breed: str        # 품종
    birth_date: date  # 생년월일
```

8. `/schemas`에서 요청/응답 데이터 구조를 어떻게 정의하나요?

- **개념:** API를 통해 클라이언트와 서버가 데이터를 주고받을 때의 **'공식적인 데이터 양식'**을 정의하고 유효성을 검사합니다.
- **단계별 적용 방식:**
  1. **요청(Request) 구조 정의:** `load_only=True` 옵션을 사용하여, 서버가 요청을 받을 때만 유효한 필드를 정의합니다. (예: `owner_id`)
  2. **응답(Response) 구조 정의:** `dump_only=True` 옵션을 사용하여, 서버가 응답을 보낼 때만 포함될 필드를 정의합니다. (예: 데이터베이스에서 생성된 `id`)
  3. **인스턴스화:** 스키마 객체는 모듈 레벨에서 한 번만 생성하여 재사용하는 것을 표준으로 합니다. (`pet_schema = PetSchema()`)
- **코드 예시 (`app/schemas/pet_schema.py`)**

```
from marshmallow import Schema, fields

class PetSchema(Schema):
    """
    Pet 데이터의 유효성 검증 및 직렬화를 위한 스키마.
    API의 요청/응답 데이터 구조를 정의합니다.
    """
    # --- 필드 정의 ---

    # dump_only=True: '직렬화(dump)' 시에만, 즉 서버가 클라이언트로 '응답'할 때
만 포함됩니다.
    # DB에서 자동 생성된 id를 클라이언트에게 보여주기 위해 사용합니다.
    id = fields.Str(dump_only=True)

    # required=True: '역직렬화(load)' 시에, 즉 서버가 클라이언트의 '요청'을 받을
때 필수입니다.
    # 이 필드가 없으면 유효성 검사 에러가 발생합니다.
    name = fields.Str(required=True)
    breed = fields.Str(required=True)
    birth_date = fields.Date(format='iso', required=True)

    # load_only=True: '역직렬화(load)' 시에만, 즉 서버가 '요청'을 받을 때만 허용
됩니다.
    # JWT 토큰 등 내부적으로 처리할 값을 받을 때 사용하며, '응답'에는 포함되지 않
```

```
아 정보 노출을 막습니다.
    owner_id = fields.Str(load_only=True)
```

9. Marshmallow 라이브러리와 데이터 직렬화/역직렬화는 무엇인가요?

- **개념:**
  - **Marshmallow:** 파이썬 객체와 JSON 같은 외부 데이터 형식 간의 변환 및 유효성 검증을 위한 라이브러리입니다.
  - **역직렬화 (Deserialization):** 외부 데이터(JSON) -> 내부 객체(Python Dict). 클라이언트 요청을 서버가 이해할 수 있는 형태로 바꾸고 유효성을 검증합니다. (`.load()` 메소드 사용)
  - **직렬화 (Serialization):** 내부 객체(Python Dict) -> 외부 데이터(JSON). 서버의 데이터를 클라이언트가 사용할 수 있는 형태로 가공하고 포맷팅합니다. (`.dump()` 메소드 사용)
- **예시:** API 응답 시, `.dump()` 메소드는 Firestore의 Timestamp 객체를 ISO 형식의 날짜 문자열로 변환하고, `load_only=True`로 설정된 필드(`owner_id` 등)는 결과에서 자동으로 제외하여 깨끗한 JSON을 만듭니다.
- **코드 예시 (요청 처리 및 응답 반환 과정)**

```
# app/api/pets/routes.py (일부)
from flask import request, jsonify, g
from marshmallow import ValidationError
from app.schemas.pet_schema import PetSchema

# --- 표준: 스키마 인스턴스를 모듈 레벨에서 생성 ---
pet_schema = PetSchema()

@pets_bp.route('/', methods=['POST'])
@jwt_required # JWT 인증이 필요한 라우트
def create_pet():
    # 1. 클라이언트가 보낸 JSON 요청 본문을 가져옵니다.
    json_data = request.get_json()

    try:
        # --- 2. 역직렬화 (Deserialization) ---
        # .load() 메소드로 JSON 데이터를 Python 딕셔너리로 변환하고 유효성을 검사합니다.
        # 'name', 'breed', 'birth_date'가 없으면 여기서 ValidationError 발생
        validated_data = pet_schema.load(json_data)

        # 3. 인증된 사용자의 ID를 가져옵니다. (@jwt_required가 g.user에 저장)
        owner_id = g.user['user_id']

        # 4. 서비스 계층에 비즈니스 로직 처리를 위임합니다.
        #    (실제로는 pet_services.register_pet(...) 와 같은 함수를 호출합니다.)
        new_pet_object = {"id": "new_db_id_123", "owner_id": owner_id, **validated_data}

        # --- 5. 직렬화 (Serialization) ---
        # .dump() 메소드로 Python 객체를 클라이언트에게 보낼 JSON 형식으로 변환합니다.
```

```
        # - 'id' 필드가 추가됩니다 (dump_only=True)
        # - 'owner_id' 필드가 제거됩니다 (load_only=True)
        # - 'birth_date' 객체가 "YYYY-MM-DD" 문자열로 변환됩니다.
        response_data = pet_schema.dump(new_pet_object)

        # 6. 최종 JSON 응답을 반환합니다.
        return jsonify(response_data), 201

    except ValidationError as err:
        # 유효성 검사 실패 시 에러 메시지를 반환합니다.
        return jsonify(err.messages), 400
```