

▼ 3.1 FastAPI(1)

▼ 1. 백엔드 프로그래밍

1.1 Server 구성 Use Case

- 앱/웹 서비스의 서버가 존재
- 머신러닝 서비스의 서버가 존재
- 서비스 서버에서 머신러닝 서버에 예측 요청하며 통신함(혹은 서비스 서버의 한 프로세스로 실행)

1.2 Server의 형태

모놀리식 아키텍처(Monolithic Architecture)

- 하나의 큰 서버 = 종업원, 요리사, 서랍, 계산 등을 모두 하나에서 처리하는 경우(큰 범주로 모두 서버)

마이크로서비스 아키텍처(Microservice Architecture - MSA)

- 종업원, 요리사, 서랍, 계산 등을 각각의 개별 서버로 구성하고 서로 통신하도록 하는 경우

1.3 REST API

REST API : 정보를 주고 받을 때 널리 사용되는 형식

- Representational State Transfer의 약자
- Resource, Method, Representation of Resource로 구성
 - Resource : Unique한 ID를 가지는 리소스, URI
 - Method : 서버에 요청을 보내기 위한 방식 : GET, POST, PUT, PATCH, DELETE

REST란 형식의 API

- 각 요청이 어떤 동작이나 정보를 위한 것을 요청 모습 자체로 추론할 수 있음
- 기본적인 데이터 처리 : 조회 작업, 새로 추가, 수정, 삭제
- CRUD : Create, Read, Update, Delete

▼ 1.4 HTTP Method

HTTP(Hyper Text Transfer Protocol) : 정보를 주고 받을 때 지켜야 하는 통신 프로토콜(규약), 약속 - HTTP는 기본적으로 80번 포트를 사용하고 있으며, 서버에서 80번 포트를 열어주지 않으면 HTTP 통신이 불가능

클라이언트 : 요청을 하는 플랫폼. 브라우저 같은 웹일 수 있고, 앱일수도 있음. 우리가 Python을 사용해 요청하는 것도 클라이언트

URI와 URL

- URL : Uniform Resource Locator로 인터넷 상 자원의 위치
- URI : Uniform Resource Identifier로 인터넷 상의 자원을 식별하기 위한 문자열의 구성
- URI는 URL을 포함하게 되며, URI가 더 포괄적인 범위

HTTP Method 종류

- GET : 정보를 요청하기 위해 사용(Read)
- POST : 정보를 입력하기 위해 사용(Create)
- PUT : 정보를 업데이트하기 위해 사용(Update)
- PATCH : 정보를 업데이트하기 위해 사용(Update)
- DELETE : 정보를 삭제하기 위해 사용>Delete)

GET, POST의 차이

GET

- 어떤 정보를 가져와서 조회하기 위해 사용되는 방식
- URL에 변수(데이터)를 포함시켜 요청함
- 데이터를 Header(헤더)에 포함하여 전송함
- URL에 데이터가 노출되어 보안에 취약
- 캐싱할 수 있음

POST

- 데이터를 서버로 제출해 추가 또는 수정하기 위해 사용하는 방식

- URL에 변수(데이터)를 노출하지 않고 요청
- 데이터를 Body(바디)에 포함
- URL에 데이터가 노출되지 않아 기본 보안은 되어 있음
 - 바디에 암호화 필요
- 캐싱할 수 없음(다만 그 안에 아키텍처로 캐싱할 수 있음)

처리 방식	GET	POST
URL에 데이터 노출 여부	O	X
URL 예시	localhost:8080/login?id=kyle	localhost:8080/login
데이터의 위치	Header	Body
캐싱 가능 여부	O	X

1.5 Header와 Body

Header와 Body

- Http 통신은 Request 하고, Response를 받을 때 정보를 패킷(Packet)에 저장
- Packet 구조 : Header / Body
- Header : 보내는 주소, 받는 주소, 시간
- Body : 실제 전달하려는 내용

1.6 Status Code

Status Code : 클라이언트 요청에 따라 서버가 어떻게 반응하는지를 알려주는 Code

- 1xx(정보) : 요청을 받았고, 프로세스를 계속 진행함
- 2xx(성공) : 요청을 성공적으로 받았고, 실행함
- 3xx(리다이렉션) : 요청 완료를 위한 추가 작업이 필요
- 4xx(클라이언트 오류) : 요청 문법이 잘못되었거나 요청을 처리할 수 없음
- 5xx(서버 오류) : 서버가 요청에 대해 실패함

1.7 동기과 비동기

- 동기(Sync) : 서버에서 요청을 보냈을 때, 응답이 돌아와야 다음 동작을 수행할 수 있음. A 작업이 모두 완료될 때까지 B 작업은 대기해야 함
 - 예: 주피터 노트북 실행 방식
- 비동기(Async) : 요청을 보낼 때 응답 상태와 상관없이 다음 동작을 수행함. A작업과 B 작업이 동시에 실행됨
 - <https://www.daleseo.com/python-asyncio/>
 - <https://wikidocs.net/21046>

1.8 IP

IP 정의

- 네트워크에 연결된 특정 PC의 주소를 나타내는 체계
- Internet Protocol의 줄임말, 인터넷상에서 사용하는 주소체계
- 4덩이의 숫자로 구성된 IP 주소 체계를 IPv4라고 함
- 각 덩어리마다 0~255로 나타낼 수 있음. $2^8 = 256$ 이므로 4덩이로 256의 4승인 43억개의 IP 주소를 표현할 수 있음
- 몇가지는 용도가 정해짐
 - localhost, 127.0.0.1 : 현재 사용 중인 Local PC
 - 0.0.0.0, 255.255.255.255 : broadcast address, 로컬 네트워크에 접속된 모든 장치와 소통하는 주소
 - 개인 PC 보급으로 누구나 PC를 사용해 IPv4로 할당할 수 있는 한계점 진입, IPv6이 나옴

1.9 Port

Port

- IP 주소 뒤에 나오는 숫자
- PC에 접속할 수 있는 통로(채널)
- 사용 중인 포트는 중복할 수 없음
- Jupyter Notebook은 8888
- Port는 0 ~ 65535까지 존재
- 그 중 0~1024는 통신을 위한 규약에 정해짐

- 22 : SSH
- 80 : HTTP
- 443 : HTTPS

▼ 2. FastAPI

2.1 FastAPI 소개 & 특징

소개, JetBrains Python Developer Survey 기준

- 2021: FastAPI (14%), Flask (46%), Django (45%)

특징

- Node.js, go와 대등한 성능
- Flask와 비슷한 구조 Microservice에 적합
- Swagger 자동 생성 Pydantic을 이용한 Serialization

▼ 2.2 FastAPI vs Flask

장점

- Flask보다 간결한 Router 문법
- Asynchronous(비동기) 지원
- Built-in API Documentation (Swagger)
- Pydantic을 이용한 Serialization 및 Validation

아쉬운 점

- 아직까지는 Flask의 유저가 더 많음
- ORM 등 Database와 관련된 라이브러리가 적음

Flask	FastAPI
<pre>@app.route("/", methods=["GET"]) @app.route("/", methods=["POST"])</pre>	<pre>@app.get("/") @app.post("/")</pre>
Flask	FastAPI
<pre>@app.route("/books", methods=["GET"]) def books_table_update(): Title = request.args.get('title', None) Author = request.args.get('author', None)</pre>	<pre>@app.get("/books_title/{book_title}/author/{author}") async def books_table_update(book_title: str, author: str):</pre>

프로젝트 구조 (v1)

- 프로젝트의 코드가 들어갈 모듈 설정(app). 대안 : 프로젝트 이름, src 등
- __main__.py는 간단하게 애플리케이션을 실행할 수 있는 Entrypoint 역할 (참고)
 - Entrypoint : 프로그래밍 언어에서 최상위 코드가 실행되는 시작점 또는 프로그램 진입점
- main.py 또는 app.py : FastAPI의 애플리케이션과 Router 설정
- model.py는 ML model에 대한 클래스와 함수 정의

▼ 2.3 Poetry

Poetry

- Dependency Resolver로 복잡한 의존성들의 버전 충돌을 방지
- Virtualenv를 생성해서 격리된 환경에서 빠르게 개발이 가능해짐
- 기존 파이썬 패키지 관리 도구에서 지원하지 않는 Build, Publish가 가능
- pyproject.toml을 기준으로 여러 툴들의 config를 명시적으로 관리
- 새로 만든 프로젝트라면 poetry를 사용해보고, virtualenv 등과 비교하는 것을 추천

Poetry 사용 흐름

1. 프로젝트 init

```
poetry init
```

- 사용할 라이브러리 지정
- pyproject.toml에 설정 저장됨

```
cat pyproject.toml
```

대화 형식으로 패키지 설치 가능

- 패키지 이름 검색 및 선택
- 패키지 버전 명시
- Dependency(프로덕션용)
- Development Dependency(Dev용)
- 개발 환경마다 필요한 패키지 분리

2. Poetry Shell 활성화

```
poetry shell
```

3. Poetry Install : pyproject.toml에 저장된 내용에 기반해 라이브러리 설치

```
poetry install
```

4. Poetry Add : 필요한 패키지를 추가하고 싶은 경우 사용

```
poetry add
```

5. poetry.lock

- Writing lock file에서 생성되는 파일
- 이 파일이 존재하면 작성하고 있는 프로젝트 의존성과 동일한 의존성을 가질 수 있음
- Github Repository에 꼭 커밋!

6. 01_simple_webserver.py의 app()을 uvicorn으로 실행하고 수정되면 reload

```
uvicorn 01_simple_webserver:app --reload
```

▼ 2.4 Swagger

- REST API 설계 및 문서화할 때 사용
- 다른 개발팀과 협업하는 경우
- 구축된 프로젝트를 유지보수하는 경우

2.5 Simple Web Server

- localhost:8000/docs로 이동하면 Swagger를 확인할 수 있음
- localhost:8000/redoc

▼ 3.1 FastAPI(2)

▼ 1. FastAPI 기본 지식

- 웹에서 GET Method를 사용해 데이터를 전송할 수 있음
- ID가 402인 사용자 정보를 가져오고 싶은 경우 방식

</users/402>

Path Parameter 방식

- 서버에 402라는 값을 전달하고 변수로 사용
- 웹에서 GET Method를 사용해 데이터를 전송할 수 있음
- ID가 402인 사용자 정보를 가져오고 싶은 경우 방식

/users?id=402

Query Parameter 방식

- Query String
- API 뒤에 입력 데이터를 함께 제공하는 방식으로 사용
- Query String은 Key, Value의 쌍으로 이루어지며 &로 연결해 여러 데이터를 넘길 수 있음

</users/kyle> : Path

</users?name=kyle> : Query

- Path Parameter : 저 경로에 존재하는 내용이 없으므로 404 Error 발생
- Query Parameter : 데이터가 없는 경우 빈 리스트가 나옴 => 추가로 Error Handling이 필요
- Resource를 식별해야 하는 경우 : Path Parameter가 더 적합
- 정렬, 필터링을 해야 하는 경우 : Query Parameter가 더 적합

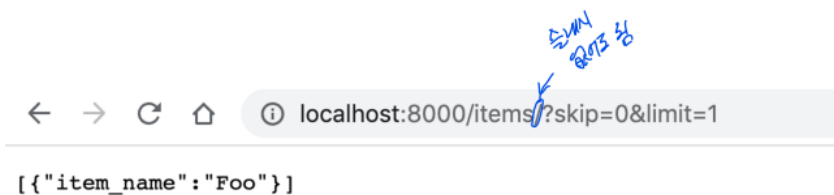
1.1 Path Parameter

- FastAPI는 데코레이터로 GET, POST를 표시
- @app.get @app.post

터미널에서 Request 로그가 남음

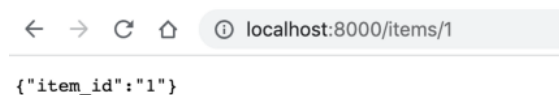
▼ 1.2 Query Parameter

- Query Parameter를 적용(URL 뒤에 ?를 붙이고 Key, Value 형태로 연결)



▼ 1.3 Optional Parameter

- 특정 파라미터는 Optional(선택적)으로 하고 싶은 경우
- typing 모듈의 Optional을 사용
- Optional을 사용해 이 파라미터는 Optional임을 명시(기본 값은 None)



1.4 Request Body

- 클라이언트에서 API에 데이터를 보낼 때, Request Body를 사용함
- 클라이언트 => API : Request Body
- API의 Response => 클라이언트 : Response Body
 - Request Body에 데이터가 항상 포함되어야 하는 것은 아님
 - Request Body에 데이터를 보내고 싶다면 POST Method를 사용

- (참고) GET Method는 URL, Request Header로 데이터 전달

POST Method는 Request Body에 데이터를 넣어 보냄

- Body의 데이터를 설명하는 Content-Type이란 Header 필드가 존재하고, 어떤 데이터 타입인지 명시해야 함

대표적인 콘텐츠 타입

- application/x-www-form-urlencoded : BODY에 Key, Value 사용. & 구분자 사용
- text/plain : 단순 txt 파일
- multipartform-data : 데이터를 바이너리 데이터로 전송

1.5 Response Body

- API의 Response => 클라이언트 : Response Body
- Decorator의 response_model 인자로 주입 가능 역할
- Output Data를 해당 정의에 맞게 변형
- 데이터 Validation
- Response에 대한 Json Schema 추가
- 자동으로 문서화

1.6 Form, File

- Form(입력) 형태로 데이터를 받고 싶은 경우
- Form을 사용하려면 python-multipart를 설치해야 함

```
pip install python-multipart
```

프론트도 간단히 만들기 위해 Jinja2 설치

```
pip install Jinja2
```

- 파이썬에서 사용할 수 있는 템플릿 엔진 : Jinja Template => 프론트엔드 구성
- templates.TemplateResponse로 해당 HTML로 데이터를 보냄
- Form(...): ...는 무엇일까?
 - Python ellipsis : Required(꼭 필수 요소)를 의미

File

- File 업로드하고 싶은 경우
- File을 사용할 때도 python-multipart를 설치해야 함
- UploadFile Import
- "/"로 접근할 때 보여줄 HTML 코드
- HTML에서 action으로 넘김

▼ 2. Pydantic

2.1 Pydantic

- FastAPI에서 Class 사용할 때 보이던 Pydantic
- Data Validation / Settings Management 라이브러리
- Type Hint를 런타임에서 강제해 안전하게 데이터 핸들링
- 파이썬 기본 타입(String, Int 등) + List, Dict, Tuple에 대한 Validation 지원
- 기존 Validation 라이브러리보다 빠름, [Benchmark](#)
- Config를 효과적으로 관리하도록 도와줌
- 머신러닝 Feature Data Validation으로도 활용 가능

두 가지 기능

1. Validation
2. Config 관리

2.2 Pydantic Validation

예시

- Machine Learning Model Input Validation

- Online Serving에서 Input 데이터를 Validation하는 Case

Validation Check Logic

- 조건 1: 올바른 url을 입력 받음 (url)
- 조건 2: 1-10 사이의 정수 입력 받음 (rate)
- 조건 3: 올바른 폴더 이름을 입력 받음(target_dir)

사용할 수 있는 방법

1. 일반 Python Class를 활용한 Input Definition 및 Validation
 - 너무 길고 복잡해질 수 있음
2. Dataclass를(python 3.7 이상 필요) 활용한 Input Definition 및 Validation
 - Validation 로직을 따로 작성하지 않으면, 런타임에서 type checking을 지원하지 않음
3. Pydantic을 활용한 Input Definition 및 Validation
 - 훨씬 간결해진 코드 (6라인)(vs 52라인 Python Class, vs 50라인 dataclass)
 - 주로 쓰이는 타입들(http url, db url, enum 등)에 대한 Validation이 만들어져 있음
 - 런타임에서 Type Hint에 따라서 Validation Error 발생
 - 어디서 에러가 발생했는지 location, type, message 등을 알려줌
 - Custom Type에 대한 Validation도 쉽게 사용 가능, 참고: <https://pydantic-docs.helpmanual.io/usage/types/>

2.3 Pydantic Config

- 애플리케이션은 종종 설정을 상수로 코드에 저장함
- 이것은 Twelve-Factor를 위반
- Twelve-Factor는 설정을 코드에서 엄격하게 분리하는 것을 요구함
- Twelve-Factor App은 설정을 환경 변수(envvars나 env라고도 불림)에 저장함
- 환경 변수는 코드 변경 없이 쉽게 배포 때마다 쉽게 변경할 수 있음
- The Twelve-Factor App이라는 SaaS(Software as a Service)를 만들기 위한 방법론을 정리한 규칙들에 따르면, 환경 설정은 애플리케이션 코드에서 분리되어 관리되어야 함

참고 글: <https://12factor.net/ko/config>

Config 작성법

1. .ini, .yaml 파일 등으로 config 설정하기 : 하드코딩
2. flask-style config.py : 코드량 증가
3. pydantic base settings : 필수는 아님
 - Validation처럼 Pydantic은 BaseSettings를 상속한 클래스에서 Type Hint로 주입된 설정 데이터를 검증할 수 있음
 - Field 클래스의 env 인자로, 환경 변수로 부터 해당 필드를 오버라이딩 할 수 있음
 - yaml, ini 파일들을 추가적으로 만들지 않고, .env 파일들을 환경별로 만들어 두거나, 실행 환경에서 유연하게 오버라이딩 할 수 있음

▼ 3.1 FastAPI(3)

▼ 1. FastAPI 익숙해지기

1.1 Event Handler

- 이벤트가 발생했을 때, 그 처리를 담당하는 함수
- FastAPI에선 Application이 실행할 때, 종료될 때 특정 함수를 실행할 수 있음

```
@app.on_event("startup")
@app.on_event("shutdown")
```

예시

- startup 할 때 머신러닝 모델 Load
- shutdown 할 때 로그 저장

▼ 1.2 API Router

- API Router는 더 큰 애플리케이션들에서 많이 사용되는 기능
- API Endpoint를 정의
- Python Subpackage
- APIRouter는 Mini FastAPI로 여러 API를 연결해서 활용

- 기존에 사용하던 @app.get, @app.post을 사용하지 않고, router 파일을 따로 설정하고 app에 import해서 사용

예제 프로젝트 구조

```

.
└─ app : 메인 App
    ├── __init__.py : App, Python Package 생성을 위한 파일
    ├── main.py : main Module. import app.main
    ├── dependencies.py : 의존성 Module, import app.dependencies
    └─ routers : Sub FastAPI(Python Subpackage)
        ├── __init__.py : Python Subpackage routers 생성
        ├── items.py : items Submodule, import app.routers.items
        └─ users.py : users Submodule, import app.routers.users
    └─ internal : internal Python subpackage
        ├── __init__.py
        └─ admin.py

```

▼ 1.3 Error Handling

- Error Handling은 웹 서버를 안정적으로 운영하기 위해 반드시 필요한 주제
- 서버에서 Error가 발생한 경우, 어떤 Error가 발생했는지 알아야 하고 요청한 클라이언트에 해당 정보를 전달해 대응할 수 있어야 함
- 서버 개발자는 모니터링 도구를 사용해 Error Log를 수집해야 함
- 발생하고 있는 오류를 빠르게 수정할 수 있도록 예외 처리를 잘 만들 필요가 있음

item_id가 5일 경우 Internal Server Error 500 Return

- 이렇게 되면 클라이언트는 어떤 에러가 난 것인지 정보를 얻을 수 없고, 자세한 에러를 보려면 서버에 직접 접근해서 로그를 확인해야 함
- 에러 핸들링을 더 잘 하려면 에러 메시지와 에러의 이유 등을 클라이언트에 전달하도록 코드를 작성해야 함

- FastAPI의 HTTPException은 Error Response를 더 쉽게 보낼 수 있도록 하는 Class
- HTTPException을 이용해서 클라이언트에게 더 자세한 에러 메시지를 보내는 코드 작성

1.4 Background Task

- FastAPI는 Starlett이라는 비동기 프레임워크를 래핑해서 사용
 - FastAPI의 기능 중 Background Tasks 기능은 오래 걸리는 작업들을 background에서 실행함
 - Online Serving에서 CPU 사용이 많은 작업들을 Background Task로 사용하면,
 - 클라이언트는 작업 완료를 기다리지 않고 즉시 Response를 받아볼 수 있음
 - 특정 작업 후, 이메일 전송하는 Task 등
- 작업 결과물을 조회할 때는 Task를 어딘가에 저장해두고, GET 요청을 통해 Task가 완료됐는지 확인

▼ 2. FastAPI가 어렵다면

▼ 2.1 프로젝트 구조 - Cookiecutter

백엔드 어려운 이유

- 프로젝트 구조를 어떻게 잡아야 할지 모르겠다
- 객체 지향이 낯설다
- 백엔드 프로그래밍 자체가 처음
- 목표의 부재

Cookiecutter

- 많은 사람들이 프로젝트 구조에 대한 고민이 많아 템플릿을 서로 공유
 - <https://github.com/cookiecutter/cookiecutter>
- CLI 형태로 프로젝트 생성 과정을 도와줌
- 회사에서 공통의 프로젝트 구조가 필요하면 쿠키 커터로 설정
- 개인용 쿠키 커터 템플릿을 만드는 것도 좋은 방법
- Cookiecutter Data Science : <https://github.com/drivendata/cookiecutter-data-science>

- cookiecutter-fastapi
 - <https://github.com/arthurhenrique/cookiecutter-fastapi>

2.2 객체 지향

- 절차형 프로그래밍 vs 객체 지향 프로그래밍의 차이 이해해보기
- 객체 지향 프로그래밍은 코드의 중복을 최소화해서 재사용성을 증가시킴
- 복잡한 로직이 들어갈수록 점점 빛을 발휘함
- 현재 가지고 있는 코드를 Class로 변경해보기
- Pydantic Use Case 탐색하기

2.3 Try & Error

- 목표를 설정하고(무엇을 만들겠다) => 기능을 정의 => 하나씩 구현

▼ FastAPI 코드 실습

```

in [ ]:
    # -*- coding: utf-8 -*-
    import torch
    import streamlit as st

    from transformers import QuestionAnsweringPipeline, RobertaForQuestionAnswering, AutoTokenizer

    while True:
        time.sleep(20)
        @st.cache(allow_output_mutation=True)
        def get_pipeline():
            tokenizer = AutoTokenizer.from_pretrained("xlm-roberta-large")
            myQAModel = RobertaForQuestionAnswering.from_pretrained("CodeNinja1126/xlm-roberta-large-kor-mrc")
            QAPipeline = QuestionAnsweringPipeline(model = myQAModel, tokenizer = tokenizer)
            return QAPipeline

        context = st.text_area("Context Paragraph", "")
        question = st.text_input("Question", "")
        model=get_pipeline()
        if context:
            if question:
                outputs = model(question = question, context = context, topk = 3, max_seq_len = 512)
                answer = outputs[0]["answer"]
                output_answer = st.text_area("Answer", answer)
  
```

```

from fastapi import FastAPI, UploadFile, File
from fastapi.param_functions import Depends
from pydantic import BaseModel, Field
from uuid import UUID, uuid4
from typing import List, Union, Optional, Dict, Any

from datetime import datetime

from app.model import MyEfficientNet, get_model, get_config, predict_from_image_byte

class Product(BaseModel):
    id: UUID = Field(default_factory=uuid4)
    name: str
    price: float

# Field : 모델 스키마 또는 복잡한 Validation 검사를 위해 필드에 대한 추가 정보를 제공할 때 사용
# UUID : 고유 식별자
# default_factory : Proudcut Class가 처음 만들어질 때 호출되는 함수를 uuid4로 정한다.
# => Proudcut Class를 생성하면 uuid4를 만들어서 id에 저장

class Order(BaseModel):
    id: UUID = Field(default_factory=uuid4)
    # 최초의 빈 리스트를 만들어서 저장한다.
    products: List[Product] = Field(default_factory=list)
    created_at: datetime = Field(default_factory=datetime.now)
    updated_at: datetime = Field(default_factory=datetime.now)
  
```

```
def get_order_by_id(order_id: UUID) -> Optional[Order]:
    # 제네레이터 사용으로 메모리 절약 사용
    # iter는 반복 가능한 객체에서 이터레이터 반환
    # next는 이터레이터에서 값을 차례대로 꺼냄
    return next((order for order in orders if order.id == order_id), None)

@app.post("/order", description="주문을 요청합니다")
async def make_order(files: List[UploadFile] = File(...),
                     model: MyEfficientNet = Depends(get_model),
                     config: Dict[str, Any] = Depends(get_config)):
    # Depends : 의존성 주입
    # 반복적이고 공통적인 로직이 필요할 때 사용할 수 있음
    # 모델 Load, Config Load
    # async, Depends 검색해서 학습하기
    products = []
    for file in files:
        image_bytes = await file.read()
        inference_result = predict_from_image_byte(model=model, image_bytes=image_bytes, config=config)
        product = InferenceImageProduct(result=inference_result)
        products.append(product)

    new_order = Order(products=products)
    orders.append(new_order)
    return new_order
```

Makefile

```
INFO: Stopping reloader process [59160]
byeon@byeon_Macbook ~/Dropbox/06-Lecture/Boostcamp-AI-Tech-Product-Serving/part3/01-fastapi main ➡ ls
Makefile app examples pyproject.toml
README.md assignments poetry.lock requirements.txt
byeon@byeon_Macbook ~/Dropbox/06-Lecture/Boostcamp-AI-Tech-Product-Serving/part3/01-fastapi main ➡ cat Makefile
file
run_black:
    python3 -m black . -l 119

run_server:
    python3 -m app

run_client:
    python3 -m streamlit run app/frontend.py

run_app: run_server run_client
byeon@byeon_Macbook ~/Dropbox/06-Lecture/Boostcamp-AI-Tech-Product-Serving/part3/01-fastapi main ➡ make -j 2
run_app
python3 -m app
python3 -m streamlit run app/frontend.py
INFO: Will watch for changes in these directories: ['/Users/byeon/Dropbox/06-Lecture/Boostcamp-AI-Tech-Product-Serving/part3/01-fastapi']
INFO: Started reloader process [61353] using statreload
```

3.2 Docker

1. Docker 소개

1.1 가상화란?

개발을 진행한 Local 환경과 Production 서버 환경이 다른 경우

- Local 환경은 윈도우
- 서버 환경은 Linux

OS가 다르기 때문에 라이브러리, 파이썬 등 설치할 때 다르게 진행해야 함

Local 환경과 서버가 같은 OS를 사용해도, 서버에서 올바르게 작동하지 않을 수 있음

- Local의 환경 변수
- Production 서버의 환경 변수(Env)
- Production 서버의 사용자 그룹, Permission

다양한 설정을 README 등에 기록하고, 항상 실행하도록 하는 방법

- 사람이 진행하는 일이라 Human Error 발생
- 매번 이런 작업을 해야 하는 과정이 귀찮음
 - 예 : Jupyter Notebook 서버를 만들기 위해 클라우드에서 클릭 - 이름 - 클릭 만들기 => 인스턴스로 접속해서 필요한 패키지 설치하는 과정

운영하고 있는 Server가 100대라면?

- 특정 서버 업데이트가 진행되었다면(윈도우, 스마트폰 OS 자동 업데이트 실행) => 나머지 서버에도 모두 접속해 업데이트 필요

서버 환경까지도 모두 한번에 소프트웨어화 할 수 있을까?

▼ 1.2 Docker 등장하기 전

가상화 기술로 주로 VM(Virtual Machine)을 사용

- VM은 호스트 머신이라고 하는 실제 물리적인 컴퓨터 위에, OS를 포함한 가상화 소프트웨어를 두는 방식

GCP의 Compute Engine 또는 AWS EC2가 이런 개념을 활용

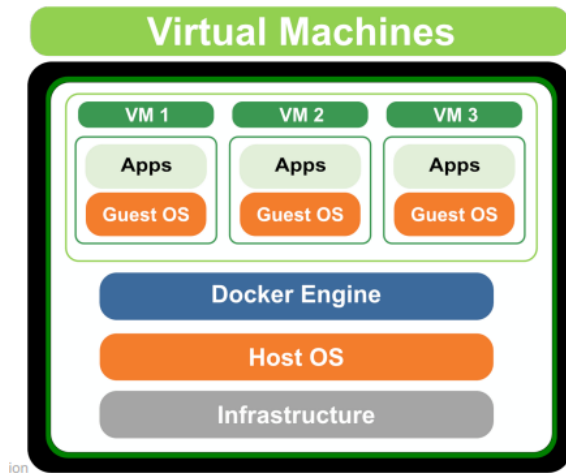
- 클라우드 회사에서 미리 만든 이미지를 바탕으로, Computing 서비스를 통해 사용자에게 동일한 컴퓨팅 환경을 제공

그러나 OS 위에 OS를 하나 더 실행시키는 점에서 VM은 굉장히 리소스를 많이 사용

- 이런 경우를 “무겁다” 라고 표현

Container : VM의 무거움을 크게 덜어주면서, 가상화를 좀 더 경량화된 프로세스의 개념으로 만든 기술

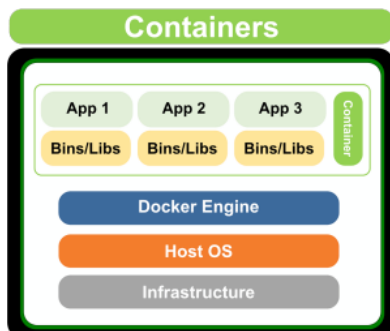
- 이 기술의 등장으로 이전보다 빠르고 가볍게 가상화를 구현할 수 있음



▼ 1.3 Docker 소개

Container 기술을 쉽게 사용할 수 있도록 나온 도구가 바로 Docker

- 2013년에 오픈소스로 등장
- 컨테이너에 기반한 개발과 운영을 매우 빠르게 확장

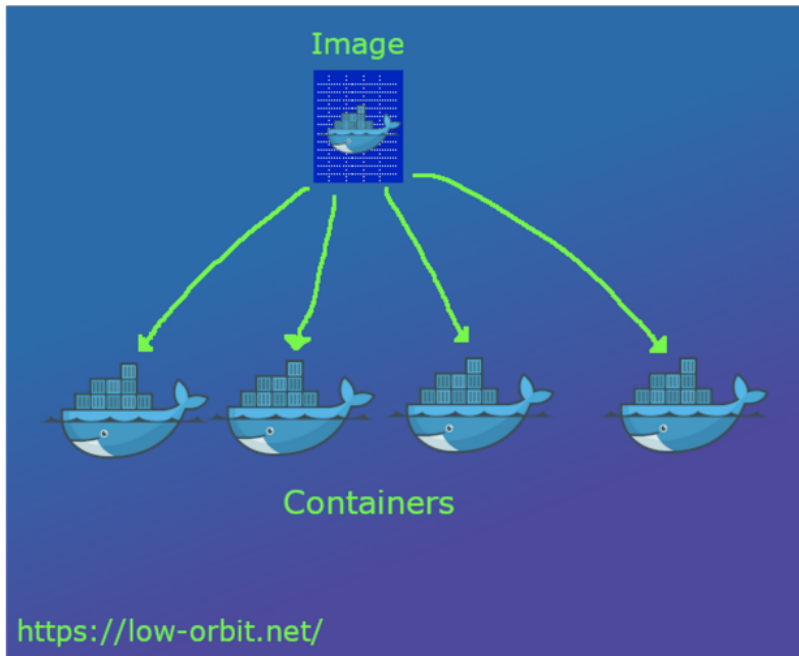


Docker Image (예:이미지)

- 컨테이너를 실행할 때 사용할 수 있는 “템플릿”
- Read Only

Docker Container (예:공시디)

- Docker Image를 활용해 실행된 인스턴스
- Write 가능



1.4 Docker로 할 수 있는 일

1. 다른 사람이 만든 소프트웨어를 가져와서 바로 사용할 수 있음
 - 예) MySQL을 Docker로 실행
 - 예) Jupyter Notebook을 Docker로 실행
2. 다른 사람이 만든 소프트웨어 : Docker Image
 - OS, 설정을 포함한 실행 환경
 - Linux, Window, Mac 어디에서나 동일하게 실행할 수 있음
3. 자신만의 이미지를 만들면 다른 사람에게 공유할 수 있음
 - 원격 저장소에 저장하면 어디서나 사용할 수 있음
4. 원격 저장소 : Container Registry
 - 회사에서 서비스를 배포할 때는 원격 저장소에 이미지를 업로드하고, 서버에서 받아서 실행하는 식으로 진행
 - 예: 도커허브, GCR, ECR

▼ 2. Docker 실습하며 배워보기

2.1 설치하고 실행하기

docker pull “이미지 이름:태그”

- docker pull mysql:8로 mysql 8 버전의 이미지를 다

다운받은 이미지 확인

- docker images

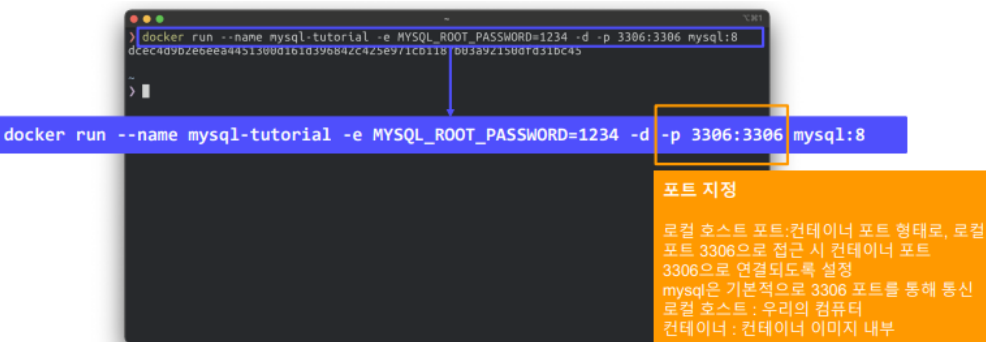
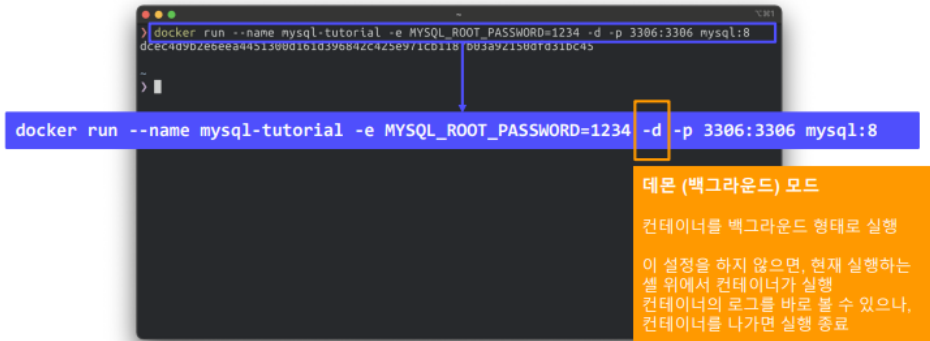
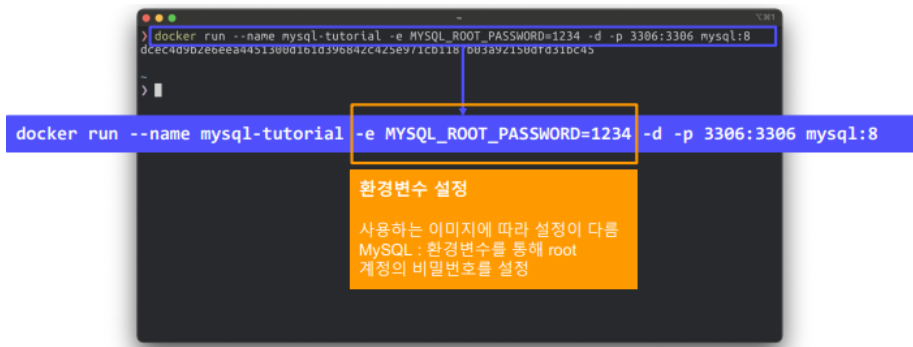
docker run “이미지 이름:태그”

- 다운받은 MySQL 이미지 기반으로 Docker Container를 만들고 실행

```
docker run --name mysql-tutorial -e MYSQL_ROOT_PASSWORD=1234 -d -p 3306:3306 mysql:8
```

컨테이너 이름

지정하지 않으면 랜덤으로 생성



docker ps

- 실행한 컨테이너는 docker ps 명령어로 확인할 수 있음

docker exec -it "컨테이너 이름(혹은 ID)" /bin/bash

- MySQL이 실행되고 있는지 확인하기 위해 컨테이너에 진입
- Compute Engine에서 SSH와 접속하는 것과 유사
- it : 인터랙티브 실행
- `/bin/bash` : 로 실행하겠다

mysql -u root -p

- MySQL 프로세스로 들어가면 MySQL 쉘 화면이 보임
- 도커 컨테이너 안의 mysql 실행

docker ps -a

- 작동을 멈춘 컨테이너는 docker ps -a 명령어로만 확인할 수 있음
- (docker ps는 실행중인 컨테이너 목록만 보여줌)

docker rm "컨테이너 이름(ID)"

- 멈춘 컨테이너를 삭제
- (멈춘 컨테이너만 삭제할 수 있지만 docker rm "컨테이너 이름(ID)" -f 로 실행중인 컨테이너도 삭제 가능)
 - f : force

기본 명령어 정리

- docker pull "이미지 이름:태그" : 필요한 이미지 다운
- docker images :다운받은 이미지 목록 확인

- docker run "이미지 이름:태그": 이미지를 기반으로 컨테이너 생성
- docker ps : 실행중인 컨테이너 목록 확인
- docker exec -it "컨테이너 이름(ID)" [/bin/bash](#) : 컨테이너에 진입
- docker stop "컨테이너 이름(ID)" : 실행중인 컨테이너를 중지
- docker rm "컨테이너 이름(ID)" : 중지된 컨테이너 삭제

docker run 할 때 파일을 공유하는 방법

- Volume Mount
 - Docker Container 내부는 특별한 설정이 없으면 컨테이너를 삭제할 때 파일이 사라짐
 - (=Host와 Container와 파일 공유가 되지 않음)
 - 만약 파일을 유지하고 싶다면 Host(우리의 컴퓨터)와 Container의 저장소를 공유해야 함
- Volume Mount를 진행하면 Host와 Container의 폴더가 공유됨
- -v 옵션을 사용하며, -p(Port)처럼 사용함. -v Host_Folder:Container_Folder

예 : docker run -it -p 8888:8888 -v [/some/host/folder/for/work](#):/home/jovyan/workspace jupyter/minimal-notebook

Dockerhub에 공개된 모든 이미지를 다운받을 수 있음

- MySQL도 Dockerhub에서 다운로드
- Dockerhub에 웬만한 오픈소스들이 공개되어 있고, 우리는 필요한 이미지를 찾아 실행!

▶ 2.2 Docker Image 만들기

↳ 숨겨진 셀 15개

▶ 2.3 Registry에 Docker Image Push

로컬에서 GCR로 이미지를 Push

1. 먼저 gcloud를 로그인하고, 프로젝트 세팅
 - gcloud : 구글 클라우드 플랫폼 제품을 CLI로 쉽게 사용할 수 있도록 만든 도구
2. [Cloud SDK Install](#) 에서 OS를 확인한 후 다운로드 및 실행

↳ 숨겨진 셀 6개

▼ 3. Docker 이미지로 배포하기

▶ 3.1 Serverless Cloud 서비스(Cloud Run)

도커 이미지를 서버에 배포하는 가장 간단한 방법 : Cloud 서비스 활용

- GCP : Cloud Run
- AWS : ECS

↳ 숨겨진 셀 3개

▶ 3.2 Compute Engine에 Docker Image 배포하기(Streamlit)

Streamlit과 GCR 호환이 안되서 다른 방식으로 Docker Image 배포

- Compute Engine을 띄우고, 해당 인스턴스 실행될 때 Docker Image를 가지고 실행하도록 설정
- Github Action을 사용해 Docker Image Push 자동화!
- Part 2 - CI/CD에서 진행한 Streamlit 파일을 기반으로 실행

↳ 숨겨진 셀 30개

▶ 3.3 Docker Compose

↳ 숨겨진 셀 6개

▼ 3.3 Logging

▼ 1. Logging Basics

1.1 로그란?

로그의 어원

- 통나무
- 과거 선박의 속도를 측정하기 위해 칩 로그라는 것을 사용
- 배의 앞에서 통나무를 띄워서 배의 선미까지 도달하는 시간을 재는 방식에 로그를 사용
- 요즘엔 컴퓨터에 접속한 기록, 특정 행동을 한 경우 남는 것을 로그라고 부름

머신러닝에서 로깅

- 머신러닝 인퍼런스 요청 로그, 인퍼런스 결과를 저장

▼ 1.2 데이터 적재 방식

데이터의 종류

- 데이터베이스 데이터(서비스 로그, Database에 저장)
 - 서비스가 운영되기 위해 필요한 데이터
 - 예) 고객이 언제 가입했는지, 어떤 물건을 구입했는지 등
- 사용자 행동 데이터(유저 행동 로그, 주로 Object Storage, 데이터 웨어하우스에 저장)
 - 유저 로그라고 지칭하면 사용자 행동 데이터를 의미
 - 서비스에 반드시 필요한 내용은 아니고, 더 좋은 제품을 만들기 위해 또는 데이터 분석시 필요한 데이터
 - 앱이나 웹에서 유저가 어떤 행동을 하는지를 나타내는 데이터
 - UX와 관련해서 인터랙션이 이루어지는 관점에서 발생하는 데이터
 - 예) Click, View, 스와이프 등
- 인프라 데이터(Metric)
 - 백엔드 웹 서버가 제대로 동작하고 있는지 확인하는 데이터
 - Request 수, Response 수
 - DB 부하 등

Metric

- 값을 측정할 때 사용
- CPU, Memory 사용량, % 등

Log

- 운영 관점에서 알아야 하는 데이터를 남길 때 사용
- 함수가 호출되었다. 예외 처리가 되었다 등

Trace

- 개발 관점에서 알아야 하는 것
- 예외 Trace

Database(RDB)에 저장하는 방식

- 다시 웹, 앱 서비스에서 사용되는 경우 활용
- 실제 서비스용 Database

Database(NoSQL)에 저장하는 방식

- Elasticsearch, Logstash or Fluent, Kibana에서 활용하려는 경우
- 스키마에 덜 민감

Object Storage에 저장하는 방식

- S3, Cloud Storage에 파일 형태로 저장
- csv, parquet, json 등
- 별도로 Database나 Data Warehouse로 옮기는 작업이 필요

Data Warehouse에 저장하는 방식

- 데이터 분석시 활용하는 데이터 웨어하우스로 바로 저장

RDBMS

- 관계형 데이터베이스(Relational Database)
- 행과 열로 구성
- 데이터의 관계를 정의하고, 데이터 모델링 진행

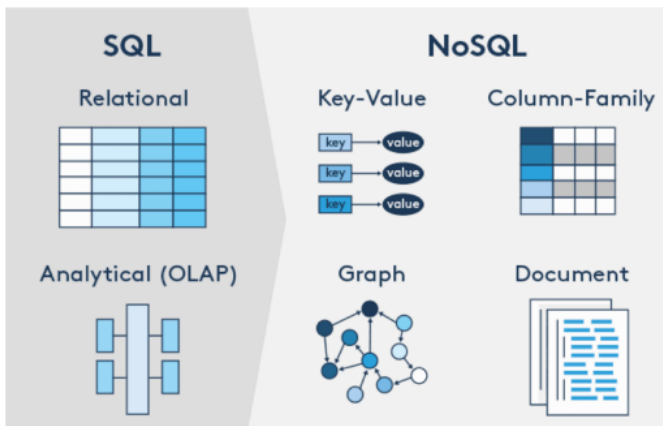
- 비즈니스와 연관된 중요한 정보
 - 예) 고객 정보, 주문 요청
- 영구적으로 저장해야 하는 것은 데이터베이스에 저장
- 데이터 추출시 SQL 사용
- MySQL, PostgreSQL 등

NoSQL

- 스키마가 Strict한 RDBMS와 다르게 스키마가 없거나 느슨한 스키마만 적용
- Not Only SQL
- 데이터가 많아지며 RDBMS로 트래픽을 감당하기 어려워서 개발됨
- 일반적으로 RDBMS에 비해 쓰기과 읽기 성능이 빠름
- Key Value Store, Document, Column Family, Graph 등
- JSON 형태와 비슷하며 XML(nested 구조) 등도 활용됨

```
{
  key : value,
  key2 : {key2-1:value21, key2-2:value22}
}
```

- MongoDB



Object Storage

- 어떤 형태의 파일이여도 저장할 수 있는 저장소
- AWS S3, GCP Cloud Storage 등
- 특정 시스템에서 발생하는 로그를 xxx.log에 저장한 후, Object Storage에 저장하는 형태
- 비즈니스에서 사용되지 않는 분석을 위한 데이터
- 이미지, 음성 등을 저장

Data Warehouse

- 여러 공간에 저장된 데이터를 한 곳으로 저장
- 데이터 창고
- RDBMS, NoSQL, Object Storage 등에 저장된 데이터를 한 곳으로 옮겨서 처리
- RDBMS와 같은 SQL을 사용하지만 성능이 더 좋은 편
- AWS Redshift, GCP BigQuery, Snowflake 등

프로젝트 상황 : Serving 과정에서 데이터 기록

- 데이터 분석은 BigQuery에서 진행
- 해당 로그는 애플리케이션과 상관없음

사용 가능 대안

- 파이션 로깅 모듈을 사용해 CSV로 저장해서 Cloud Storage에 업로드
- BigQuery로 바로 데이터 추가

1.3 저장된 데이터 활용 방식

1. "image.jpg"로 마스크 분류 모델로 요청했다
 - image.jpg를 중간에 Object Storage에 저장하면 실제로 우리가 볼 때의 실제 Label과 예측 Label을 파악할 수 있음
2. "image.jpg" 같은 이름의 이미지로 10번 요청했다

- 같은 이미지로 예측한다고 하면 중간에 저장해서 기존에 예측한 결과를 바로 Return할 수 있겠다(Redis 등을 사용해 캐싱)

3. Feature = [[2, 5, 10, 4]] 으로 수요 예측 모델로 요청했다

- 어떤 Feature가 들어오는지 알 수 있고, Feature를 사용할 때 모델이 어떻게 예측하는지 알 수 있음

4. 현재 시스템이 잘 동작하는지 알 수 있음

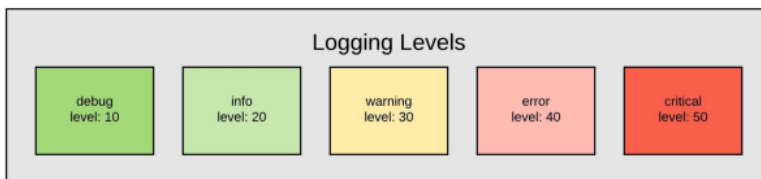
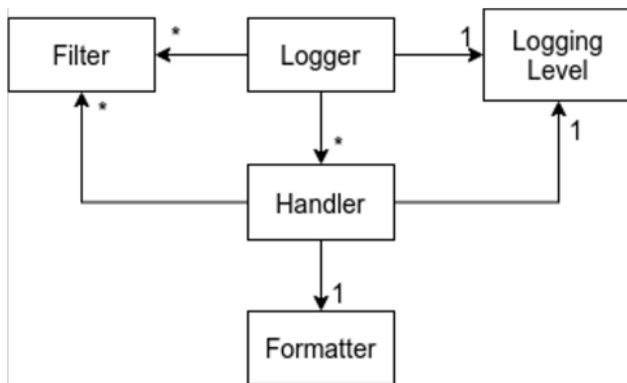
▼ 2. Logging in Python

2.1 Python Logging Module

Log Level

- 기본 Logging Level은 WARNING
- 설정하지 않으면 WARNING보다 심각한 레벨의 로그만 보여줌

Level	Value	설명	API
DEBUG	10	(문제 해결에 필요한) 자세한 정보를 공유	logging.debug()
INFO	20	작업이 정상적으로 작동하고 있는 경우 사용	logging.info()
WARNING	30	예상하지 못한 일이거나 발생 가능한 문제일 경우. 작업은 정상적으로 수행	logging.warning()
ERROR	40	프로그램이 함수를 실행할 수 없는 심각한 상황	logging.error()
CRITICAL	50	프로그램이 동작할 수 없는 심각한 문제	logging.critical()



logging vs print

- console에만 output을 출력하는 print
- logging은 file, websocket 등 파이썬이 다룰 수 있는 모든 포맷으로 output을 출력할 수 있음
 - 언제 어디서(파일 이름과 코드 상의 몇번째 줄인지) 해당 output이 발생했는지 알 수 있음
 - output을 심각도에 따라 분류할 수 있음
 - 예) Dev 환경에서는 debug 로그까지, Prod(운영) 환경에서는 info 로그만 보기 등)
 - 다만 print보다 알아야 하는 지식이 존재

2.2 Logger

- 로그를 생성하는 Method 제공(logger.info() 등)
- 로그 Level과 Logger에 적용된 Filter를 기반으로 처리해야 하는 로그인지 판단
- Handler에게 LogRecord 인스턴스 전달

logging.getLogger(name)으로 Logger Object 사용

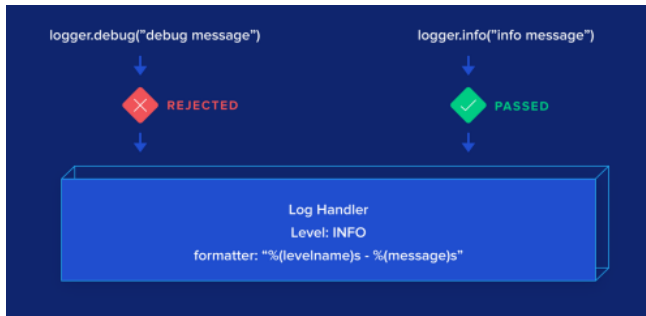
- name이 주어지면 해당 name의 logger 사용하고, name이 없으면 root logger 사용

- 마침표로 구분되는 계층 구조
- `logging.getLogger('foo.bar')` => `logging.getLogger('foo')`의 자식 logger 반환

`logging.setLevel()` : Logger에서 사용할 Level 지정

▼ 2.3 Handler

- Logger에서 만들어진 Log를 적절한 위치로 전송(파일 저장 또는 Console 출력 등)
- Level과 Formatter를 각각 설정해서 필터링 할 수 있음
- StreamHandler, FileHandler, HTTPHandler 등

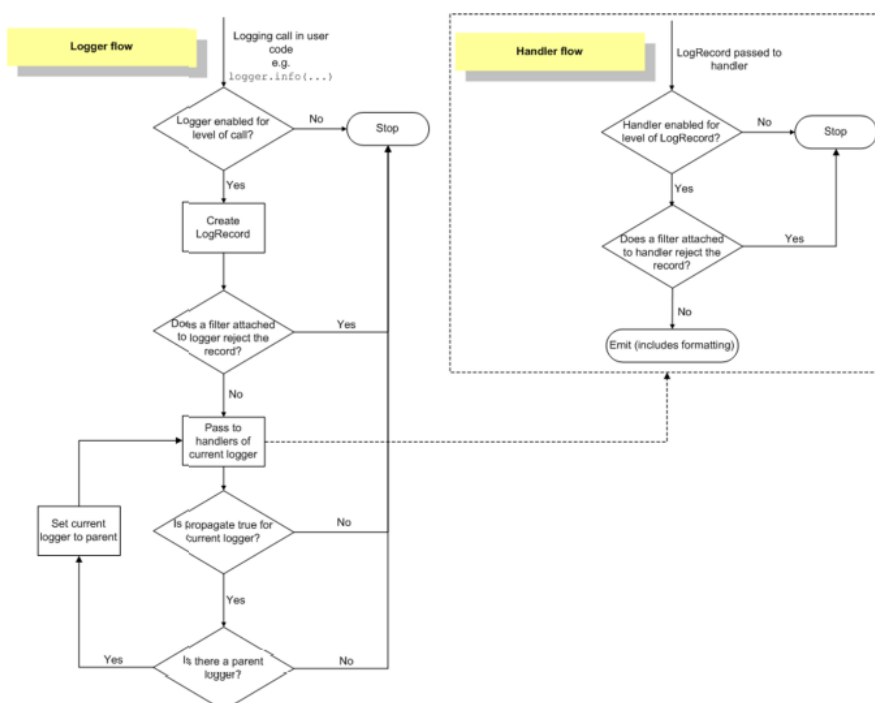


▼ 2.4 Formatter

- 최종적으로 Log에 출력될 Formatting 설정
- 시간, Logger 이름, 심각도, Output, 함수 이름, Line 정보, 메시지 등 다양한 정보 제공



▼ 2.5 Logging Flow



▼ 3. Online Serving Logging(BigQuery)

BigQuery에 Online Serving Input과 Output 로그 적재

1. 빅쿼리 테이블을 세팅합니다
2. 빅쿼리에 적재하기 쉽게 JSON 형태로 로그를 정제 -> pythonjsonlogger를 사용
3. python logging 모듈을 사용해서 빅쿼리에(실시간) 로그 적재(file과 console에도 남을 수 있도록 handler를 지정)

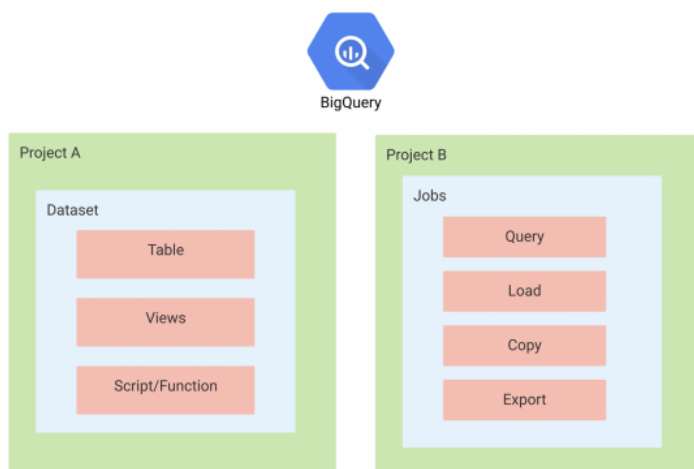
BigQuery

- Google Cloud Platform의 데이터 웨어하우스
- 데이터 분석을 위한 도구로 Apache Spark의 대용으로 활용 가능
- Firebase, Google Analytics 4와 연동되어 많은 회사에서 사용 중

▼ 3.1 BigQuery 데이터 구조

GCP의 Project 내부에 BigQuery의 리소스가 존재

- Dataset, Table, View 등



3.2 BigQuery 데이터세트 만들기

GCP Console에서 BigQuery 접근 - API 사용 - 데이터 세트 만들기 클릭

3.3 BigQuery 테이블 만들기

파티션 설정 하는 이유

- 빅쿼리는 데이터를 조회할 때 모든 데이터를 조회하지 않고 일부를 조회하기 때문에 비용을 줄이기 위한 방법

▼ 3.4 BigQuery로 실시간 로그 데이터 수집하기

config.yaml 파일에 formatter에 대한 구성 설정

- simple : 일반 포매팅
- json : json으로 로그를 변환 () 는 logger config에서 사용할 클래스를 지정
- pythonjsonlogger의 JsonFormatter 클래스를 사용하면 일반 텍스트를 json으로 변환.
- handlers와 loggers, root logger를 정의
- console handler는 기본 StreamHandler를 사용해서 콘솔/터미널에 로그를 출력
- logfile handler는 파일에 로그 저장
- maxbytes, backupCount 등도 설정 가능

BigQueryHandlerConfig 클래스

- Pydantic 커스텀 클래스를 사용하기 위해 arbitrary_types_allowed True

- BigqueryHandler는 StreamHandler를 상속받아 사용
- emit Method : BigQueryLogSchema 로그 형태에 맞게 파싱해서 데이터 가공
- bigquery_client.insert_rows_json Method를 사용해 데이터를 실시간으로 저장

▼ 3.4 MLFlow

▼ 1. MLflow 개념 잡기

MLflow가 없던 시절

- 사람들이 각자 자신의 코드를 Jupyter Notebook에서 작성
- 머신러닝 모델 학습시 사용한 Parameter, Metric을 따로 기록
- 개인 컴퓨터, 연구실 서버를 사용하다가 메모리 초과로 Memory Exceed 오류 발생
- 학습하며 생긴 Weight File을 저장해 다른 동료들에게 공유
- Weight File 이름으로 Model Versioning을 하거나 아예 Versioning을 하지 않음

MLflow가 해결하려는 문제들

1. 실험을 추적하기 어렵다
2. 코드를 재현하기 어렵다
3. 모델을 패키징하고 배포하는 방법이 어렵다
4. 모델을 관리하기 위한 중앙 저장소가 없다

▼ 1.1 MLflow란

머신러닝 실험, 배포를 쉽게 관리할 수 있는 오픈소스

MLflow의 핵심 기능

1. Experiment Management & Tracking

- 머신러닝 관련 “실험”들을 관리하고, 각 실험의 내용들을 기록할 수 있음
 - 예를 들어, 여러 사람이 하나의 MLflow 서버 위에서 각자 자기 실험을 만들고 공유할 수 있음
- 실험을 정의하고, 실험을 실행할 수 있음. 이 실행은 머신러닝 훈련 코드를 실행한 기록
 - 각 실행에 사용한 소스 코드, 하이퍼 파라미터, Metric, 부산물(모델 Artifact, Chart Image) 등을 저장

2. Model Registry

- MLflow로 실행한 머신러닝 모델을 Model Registry(모델 저장소)에 등록할 수 있음
- 모델 저장소에 모델이 저장될 때마다 해당 모델에 버전이 자동으로 올라감(Version 1 -> 2 -> 3..)
- Model Registry에 등록된 모델은 다른 사람들에게 쉽게 공유 가능하고, 쉽게 활용할 수 있음

3. Model Serving

- Model Registry에 등록된 모델을 REST API 형태의 서버로 Serving 할 수 있음
- Input = Model의 Input
- Output = Model의 Output
- 직접 Docker Image 만들지 않아도 생성할 수 있음

MLflow Component

1. MLflow Tracking

- 머신러닝 코드 실행, 로깅을 위한 API, UI
- MLflow Tracking을 사용해 결과를 Local, Server에 기록해 여러 실행과 비교할 수 있음
- 팀에선 다른 사용자의 결과와 비교하며 협업할 수 있음

2. MLflow Project

- 머신러닝 프로젝트 코드를 패키징하기 위한 표준
- Project
 - 간단하겐 소스 코드가 저장된 폴더
 - Git Repo
 - 의존성과 어떻게 실행해야 하는지 저장

- MLflow Tracking API를 사용하면 MLflow는 프로젝트 버전을 모든 파라미터와 자동으로 로깅

3. MLflow Model

- 모델은 모델 파일과 코드로 저장, 재현을 위한 도구
- 다양한 플랫폼에 배포할 수 있는 여러 도구 제공
- MLflow Tracking API를 사용하면 MLflow는 자동으로 해당 프로젝트에 대한 내용을 사용함

4. MLflow Registry

- MLflow Model의 전체 Lifecycle에서 사용할 수 있는 중앙 모델 저장소

▼ 1.2 MLflow 실습하며 알아보기

Experiment(실험)

- MLflow에서 제일 먼저 Experiment를 생성
- 하나의 Experiment는 진행하고 있는 머신러닝 프로젝트 단위로 구성
 - 예) “개/고양이 이미지 분류 실험”, “택시 수요량 예측 분류 실험”
- 정해진 Metric으로 모델을 평가
 - 예) RMSE, MSE, MAE, Accuracy
- 하나의 Experiment는 여러 Run(실행)을 가짐

```
pip install mlflow

pip install numpy sklearn matplotlib

# Experiment 생성
mlflow experiments create --experiment-name my-first-experiment

# mlrns 폴더 생성 확인
# mlrns는 실험 기록이 저장되는 폴더
ls

# Experiment 리스트 확인
mlflow experiments list
```

프로젝트(MLProject)

- MLflow를 사용한 코드의 프로젝트 메타 정보 저장
- 프로젝트를 어떤 환경에서 어떻게 실행시킬지 정의
- 패키지 모듈의 상단에 위치
- 고정된 이름 사용

Run(실행)

- 하나의 Run은 코드를 1번 실행한 것을 의미
- 보통 Run은 모델 학습 코드를 실행
- 즉, 한번의 코드 실행 = 하나의 Run 생성
- Run을 하면 여러가지 내용이 기록됨

Run에서 로깅하는 것들

- Source : 실행한 Project의 이름
- Version : 실행 Hash
- Start & end time
- Parameters : 모델 파라미터
- Metrics : 모델의 평가 지표, Metric을 시각화할 수 있음
- Tags : 관련된 Tag
- Artifacts : 실행 과정에서 생기는 다양한 파일들(이미지, 모델 Pickle 등)

```
# Run으로 실행, conda 없이
mlflow run logistic_regression --experiment-name Default --no-conda

# web ui 열기
mlflow ui
```

▶ 1.3 MLflow 코드 실습

↳ 숨겨진 셀 11개

▼ 2. MLflow 서버로 배포하기

▼ 2.0 MLflow 구조

1. 파이썬 코드 (with MLflow package)
 - 모델을 만들고 학습하는 코드
 - mlflow run 으로 실행
2. Tracking Server
 - 파이썬 코드가 실행되는 동안 Parameter, Metric, Model 등 메타 정보 저장
 - 파일 혹은 DB에 저장
3. Artifact Store
 - 파이썬 코드가 실행되는 동안 생기는 Model File, Image 등의 아티팩트를 저장
 - 파일 혹은 스토리지에 저장

```
tree mlruns
```

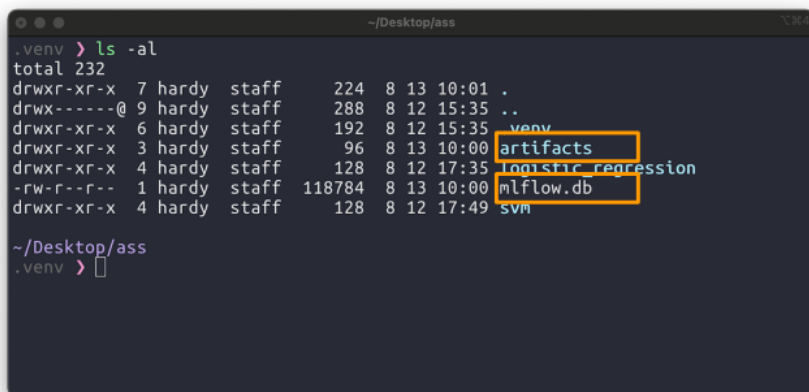
▼ 2.1 Tracking Server와 외부 스토리지 사용하기

```
# mlflow server 명령어로 Backend Store URI 지정
mlflow server --backend-store-uri sqlite:///mlflow.db --default-artifact-root $(pwd)/artifacts

# 환경변수 지정
export MLFLOW_TRACKING_URI = "http://127.0.0.1:5000"

# Experiments 생성
# experiments create --experiment-name my-experiment

# Run
# mlflow run svm --experiment-name 02 --no-conda
```



```
~/Desktop/ass
.venv > ls -al
total 232
drwxr-xr-x 7 hardy staff 224 8 13 10:01 .
drwx-----@ 9 hardy staff 288 8 12 15:35 ..
drwxr-xr-x 6 hardy staff 192 8 12 15:35 .venv
drwxr-xr-x 3 hardy staff 96 8 13 10:00 artifacts
drwxr-xr-x 4 hardy staff 128 8 12 17:35 logistic_regression
-rw-r--r-- 1 hardy staff 118784 8 13 10:00 mlflow.db
drwxr-xr-x 4 hardy staff 128 8 12 17:49 svm

~/Desktop/ass
.venv >
```

2.2 MLflow 실제 활용 사례

MLflow Tracking Server는 하나로 통합 운영

- Tracking Server를 하나 배포하고, 팀 내 모든 Researcher가 이 Tracking Server에 실험 기록
 - 배포할 때는 Docker Image, Kubernetes 등에 진행(회사의 인프라에 따라 다름)
- 로그나 모델이 한 곳에 저장되므로, 팀 내 모든 실험을 공유할 수 있음
- Artifact Storage와 DB 역시 하나로 운영

- Artifact Storage는 GCS나 S3 같은 스토리지 이용
- DB는 CloudSQL이나 Aurora RDS 같은 DB 이용
- 이 두 저장소는 Tracking Server에 의해 관리

▼ 4.1 BentoML

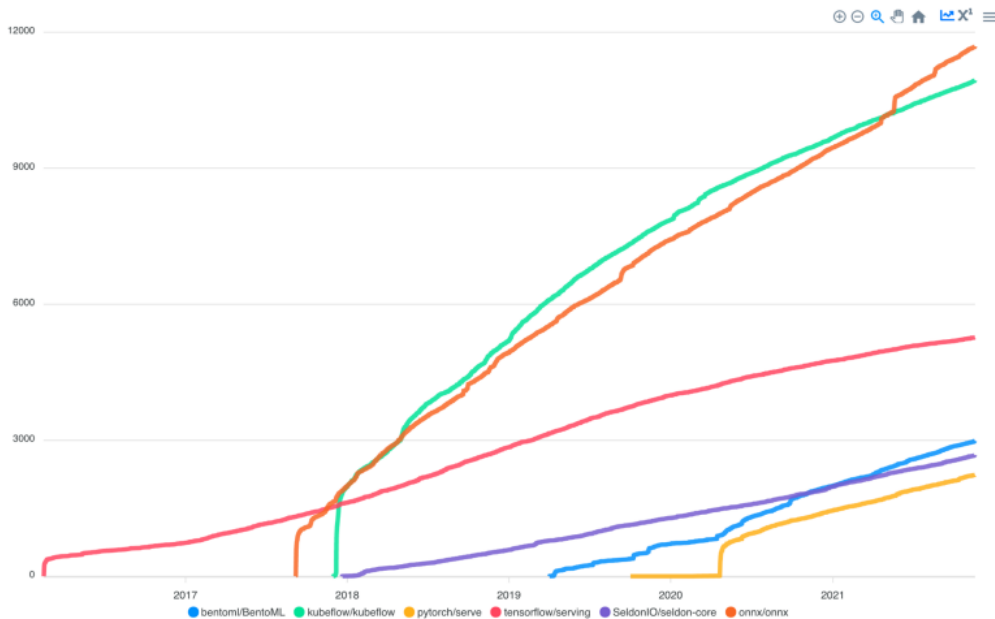
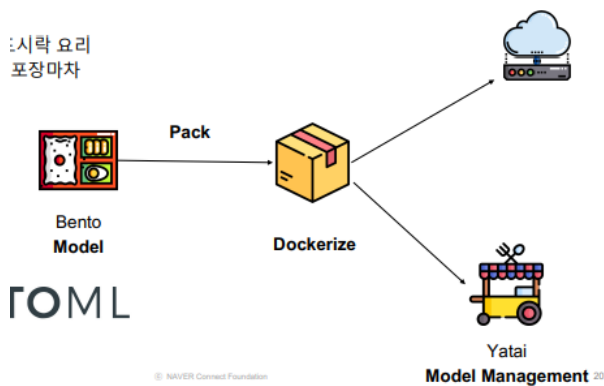
▼ 1. BentoML

1.1 Introduction

만약 30개~50개의 모델을 만들어야 한다면?

- 많은 모델을 만들다보니 반복되는 작업이 존재(Config, FastAPI 설정 등)
- 여전히 Serving은 어렵다
 - 더 빠르게 간단하게 하고 싶다. 더 쉽게 만들고 싶다. 추상화 불가능할까?

▼ 1.2 BentoML 소개



1.3 BentoML 특징

- 쉬운 사용성
- Online / Offline Serving 지원
- Tensorflow, PyTorch, Keras, XGBoost 등 Major 프레임워크 지원
- Docker, Kubernetes, AWS, Azure 등의 배포 환경 지원 및 가이드 제공
- Flask 대비 100배의 처리량
- 모델 저장소(Yatai) 웹 대시보드 제공

- 데이터 사이언스와 DevOps 사이의 간격을 이어주며 높은 성능의 Serving이 가능하게 함

1.4 BentoML이 해결하는 문제

문제 1: Model Serving Infra의 어려움

- Serving을 위해 다양한 라이브러리, Artifact, Asset 등 사이즈가 큰 파일을 패키징
- Cloud Service에 지속적인 배포를 위한 많은 작업이 필요
- BentoML은 CLI로 이 문제의 복잡도를 낮춤(CLI 명령어로 모두 진행 가능하도록)

문제 2: Online Serving의 Monitoring 및 Error Handling

- Online Serving으로 API 형태로 생성
- Error 처리, Logging을 추가로 구현해야 함
- BentoML은 Python Logging Module을 사용해 Access Log, Prediction Log를 기본으로 제공
- Config를 수정해 Logging도 커스텀할 수 있고, Prometheus 같은 Metric 수집 서버에 전송할 수 있음

문제 3: Online Serving 퍼포먼스 튜닝의 어려움

- BentoML은 [Adaptive Micro Batch](#) 방식을 채택해 동시에 많은 요청이 들어와도 높은 처리량을 보여줌

▼ 2. BentoML 시작하기

2.1 BentoML 설치하기

```
# python 3.6 이상 지원
pip install bentoml
```

▼ 2.2 BentoML 사용 Flow

- 모델 학습 코드 생성
- Prediction Service Class 생성
- Prediction Service에 모델 저장(Pack)
- (Local) Serving
- Docker Image Build(컨테이너화)
- Serving 배포

(기존에) Serving하기 위해 해야하는 작업

- FastAPI Web Server 생성
- Input, Output 정의
- 의존성 작업(requirements.txt, Docker 등)

Prediction Service Class 생성

- BentoService를 활용해 Prediction Service Class 생성, 예측할 때 사용하는 API를 위한 Class
- @env : 파이썬 패키지, install script 등 서비스에 필요한 의존성을 정의
- @artifacts : 서비스에서 사용할 Artifact 정의 => Sklearn
- BentoService를 상속하면, 해당 서비스를 Yatai(모델 이미지 레지스트리)에 저장
- @api : API 생성
 - Input과 Output을 원하는 형태(Dataframe, Tensor, JSON 등)으로 선택할 수 있음
 - Doc String으로 Swagger에 들어갈 내용을 추가할 수 있음
- @artifacts에 사용한 이름을 토대로 self.artifacts.model로 접근
 - API에 접근할 때 해당 Method 호출
- .pakc : 학습한 모델을 Prediction Service에 Pack
- Model Artifact를 주입
 - BentoML Bundle : Prediction Service를 실행할 때 필요한 모든 코드, 구성이 포함된 폴더, 모델 제공을 위한 바이너리

bentoml list : BentoML에 저장된 Prediction Service 확인

bentoml serve IrisClassifier:latest : 웹서버 실행

로그는 ~/bentoml/logs에 저장됨

터미널에서 curl로 Request해도 정상적으로 실행됨

```
curl -i W
--header "Content-Type: application/json" W
--request POST W
--data '[[5.1, 3.5, 1.4, 0.2]]' W
http://localhost:5000/predict
```

야타이, Web UI : localhost:3000

```
bentoml yatai-service-start
```

도커 이미지 빌드

```
bentoml containerize IrisClassifier:latest -t iris-classifier
```

docker 명령어나 FastAPI를 사용하지 않고 웹 서버를 띄우고, 이미지 빌드!

- 예전보다 더 적은 리소스로 개발 가능

▼ 3. BentoML Component

3.1 BentoService

- bentoml.BentoService는 예측 서비스를 만들기 위한 베이스 클래스
- @bentoml.artifacts : 여러 머신러닝 모델 포함할 수 있음
- @bentoml.api : Input/Output 정의
- API 함수 코드에서 self.artifacts.{ARTIFACT_NAME}으로 접근할 수 있음
- 파이썬 코드와 관련된 종속성 저장

3.2 Service Environment

- 파이썬 관련 환경, Docker 등을 설정할 수 있음
- @bentoml.env(infer_pip_packages=True) : import를 기반으로 필요한 라이브러리 추론
- requirements_txt_file을 명시할 수도 있음
- pip_packages=[]를 사용해 버전을 명시할 수 있음
- docker_base_image를 사용해 Base Image를 지정할 수 있음
- setup_sh를 지정해 Docker Build 과정을 커스텀할 수 있음
- @bentoml.ver를 사용해 버전 지정할 수 있음

3.3 Model Artifact

- @bentoml.artifacts : 사용자가 만든 모델을 저장해 pretrain model을 읽어 Serialization, Deserialization
- 여러 모델을 같이 저장할 수 있음
- A 모델의 예측 결과를 B 모델의 Input으로 사용할 경우
- 보통 하나의 서비스 당 하나의 모델을 권장

3.4 Model Artifact Metadata

- 해당 모델의 Metadata(Metric - Accuracy, 사용한 데이터셋, 생성한 사람, Static 정보 등)
- Pack에서 metadata 인자에 넘겨주면 메타데이터 저장
- 메타데이터는 Immutable

Metadata에 접근하고 싶은 경우

- 1) CLI : bentoml get model:version
- 2) REST API : bentoml serve 한 후, /metadata로 접근
- 3) Python

```
from bentoml import load
svc = load( 'path_to_bento_service' )
print(svc.artifacts[ 'model' ].metadata)
```

3.5 Model Management & Yatai

bentoml list

- BentoService의 save 함수는 BentoML Bundle을 ~/bentoml/repository/{서비스 이름}/{서비스 버전}에 저장
- 모델 리스트 확인

bentoml get IrisClassifier

- BentoService의 save 함수는 BentoML Bundle을 ~/bentoml/repository/{서비스 이름}/{서비스 버전}에 저장
- 특정 모델 정보 가져오기

bentoml yatai-service-start

- YataiService: 모델 저장 및 배포를 처리하는 컴포넌트
- node.js 설치 필요

▼ 3.6 API Function and Adapters

- BentoService API는 클라이언트가 예측 서비스에 접근하기 위한 End Point 생성
- Adapter는 Input / Output을 추상화해서 중간 부분을 연결하는 Layer
 - 예) csv 파일 형식으로 예측 요청할 경우 => DataFrameInput을 사용하고 있으면 내부적으로 pandas.DataFrame 객체로 변환하고 API 함수에 전달함
- @bentoml.api를 사용해 입력 받은 데이터를 InputAdapter 인스턴스에 넘김
- 데이터 처리하는 함수도 작성할 수 있음
- Input 데이터가 이상할 경우 오류 코드를 반환할 수 있음
- BentoService가 여러 API를 포함할 수 있음

3.7 Model Serving

BentoService가 벤트로 저장되면, 여러 방법으로 배포할 수 있음

1. Online Serving
2. Offline Batch Serving
3. Edge Serving

3.8 Labels

3.9 Retrieving BentoServices

bentoml retrieve ModelServe --target_dir=~/.bentoml_bundle/

- 학습한 모델을 저장한 후, Artifact bundle을 찾을 수 있음
- -target_dir flag를 사용

3.10 WEB UI

@bentoml.web_static_content를 사용하면 웹 프론트엔드에 추가할 수 있음

- 참고 링크 : <https://github.com/bentoml/gallery/tree/master/scikit-learn/iris-classifier>

▶ 4. BentoML으로 Serving 코드 리팩토링하기

4.1 BentoService 정의하기

↳ 숨겨진 셀 7개

▼ 4.2 Airflow

▼ 1. Apache Airflow 소개

1.1 Batch Process란?

- 예약된 시간에 실행되는 프로세스
- 일회성(1회)도 가능하고, 주기적인 실행도 가능

예시

- 이번 주 일요일 07:00에 1번 실행되는 프로세스
- 매주 일요일 07:00에 실행되는 프로세스

Batch Process를 AI 엔지니어가 알아야 하는 이유

- 모델을 주기적으로 학습시키는 경우 사용(Continuous Training)
- 주기적인 Batch Serving을 하는 경우 사용
- 그 외 개발에서 필요한 배치성 작업

▼ 1.2 Batch Process - Airflow 등장 전

대표적인 Batch Process 구축 방법 : Linux Crontab

- (서버에서) crontab -e 입력
- 실행된 에디터에서 0 **** predict.py 입력
- (0 **** 은 크론탭 표현으로 매 시 0분에 실행하는 것을 의미)
- OS에 의해 매 시 0분에 predict.py가 실행
- Linux는 일반적인 서버 환경이고, Crontab은 기본적으로 설치되어 있기 때문에 매우 간편
- 간단하게 Batch Process를 시작하기에 Crontab은 좋은 선택

크론 표현식

- Batch Process의 스케줄링을 정의한 표현식
- 이 표현식은 다른 Batch Process 도구에서도 자주 사용됨

```
# |----- minute (0 - 59)
# |----- hour (0 - 23)
# |----- day of the month (1 - 31)
# |----- month (1 - 12)
# |----- day of the week (0 - 6) (Sunday to Saturday;
# |                          7 is also Sunday on some systems)
# |
# |
# * * * * * <command to execute>
```

이미지 출처 : <https://crontab.cronhub.io/>

크론 표현식 예시

Cron expression	Schedule
* * * * *	Every minute
0 * * * *	Every hour
0 0 * * *	Every day at 12:00 AM
0 0 * * FRI	At 12:00 AM, only on Friday
0 0 1 * *	At 12:00 AM, on day 1 of the month

크론 표현식 제너레이터 사이트

- <http://www.cronmaker.com>

cron 표현식 해석

- <https://crontab.guru/>

Linux Crontab의 문제

- 재실행 및 알람
 - 파일을 실행하다 오류가 발생한 경우, 크론탭이 별도의 처리를 하지 않음
 - 예) 매주 일요일 07:00에 predict.py를 실행하다가 에러가 발생한 경우, 알람을 별도로 받지 못함
- 실패할 경우, 자동으로 몇 번 더 재실행(Retry)하고, 그래도 실패하면 실패했다는 알람을 받으면 좋음
- 과거 실행 이력 및 실행 로그를 보기 어려움
- 여러 파일을 실행하거나, 복잡한 파이프라인을 만들기 힘들

Crontab은 간단히 사용할 수는 있지만, 실패 시 재실행, 실행 로그 확인, 알람 등의 기능은 제공하지 않음

- 정교한 스케줄링 및 워크플로우 도구 => Airflow

▼ 1.3 Airflow 소개

스케줄링, 워크플로우 도구의 표준

- 에어비앤비(Airbnb)에서 개발
- 현재 릴리즈된 버전은 2.2.0으로, 업데이트 주기가 빠름
- 스케줄링 도구로 무거울 수 있지만, 거의 모든 기능을 제공하고, 확장성이 넓어 일반적으로 스케줄링과 파이프라인 작성 도구로 많이 사용
- 특히 데이터 엔지니어링 팀에서 많이 사용

기능

- 파이썬을 사용해 스케줄링 및 파이프라인 작성
- 스케줄링 및 파이프라인 목록을 볼 수 있는 웹 UI 제공
- 실패 시 알람
- 실패 시 재실행 시도
- 동시 실행 워커 수
- 설정 및 변수 값 분리

▼ 2. Apache Airflow 실습하며 배워보기

2.1 설치하고 실행하기

```
pip install pip --upgrade

pip install "apache-airflow==2.2.0"

# Airflow 기본 디렉토리 경로 지정
export AIRFLOW_HOME=.

# 윈도우
set AIRFLOW_HOME=.
# 출처 : https://kamang-it.tistory.com/228

# Airflow에서 사용할 DB 초기화
airflow db init

# Airflow에서 사용할 어드민 계정 생성
airflow users create W
-- username admin W
-- password ???? W
-- firstname h W
-- lastname j W
-- role Admin W
-- email hj@g.com

# Airflow 웹서버 실행
airflow webserver --port 8080

# Airflow 스케줄러 실행
airflow scheduler
```

▼ 2.2 DAG 작성하기

Batch Scheduling을 위한 DAG 생성

- Airflow에서는 스케줄링할 작업을 DAG이라고 부름
- DAG은 Directed Acyclic Graph의 약자로, Airflow에 한정된 개념이 아닌 소프트웨어 자료구조에서 일반적으로 다루는 개념
- DAG은 이름 그대로, 순환하지 않는 방향이 존재하는 그래프를 의미

Airflow는 Crontab처럼 단순히 하나의 파일을 실행하는 것이 아닌, 여러 작업의 조합도 가능함

- DAG 1개 : 1개의 파이프라인
- Task : DAG 내에서 실행할 작업
- 하나의 DAG에 여러 Task의 조합으로 구성

정리

- Airflow는 DAG이라는 단위로 스케줄링 관리
- 각 DAG은 Task로 구성
- DAG 내 Task는 순차적으로 실행되거나, 동시에(병렬로) 실행할 수 있음

DAG 작성하기

- 먼저 DAG을 담을 디렉토리 생성(이름은 무조건 dags)

DAG 작성하기 정리

- AIRFLOW_HOME 으로 지정된 디렉토리에 dags 디렉토리를 생성하고 이 안에 DAG 파일을 작성
- DAG은 파이썬 파일로 작성. 보통 하나의 .py 파일에 하나의 DAG을 저장
- DAG 파일은 크게 다음으로 구성
 - DAG 정의 부분
 - Task 정의 부분
 - Task 간 순서 정의 부분
- DAG 파일을 저장하면, Airflow 웹 UI에서 확인할 수 있음
- Airflow 웹 UI에서 해당 DAG을 ON으로 변경하면 DAG이 스케줄링되어 실행
- DAG 세부 페이지에서 실행된 DAG Run의 결과를 볼 수 있음

▼ 2.3 유용한 Operator 간단 소개

출처 : https://airflow.apache.org/docs/apache-airflow/stable/_api/airflow/operators/index.html

PythonOperator

- 파이썬 함수를 실행
- 함수 뿐 아니라, Callable한 객체를 파라미터로 넘겨 실행할 수 있음
- 실행할 파이썬 로직을 함수로 생성한 후, PythonOperator로 실행

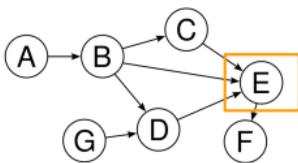
BashOperator

- Bash 커맨드를 실행
- 실행해야 할 프로세스가 파이썬이 아닌 경우에도 BashOperator로 실행 가능
- ex. gcp 명령어, shell 스크립트, scala 파일 등

DummyOperator

- 아무것도 실행하지 않음
- DAG 내에서 Task를 구성할 때, 여러 개의 Task의 SUCCESS를 기다려야 하는 복잡한 Task 구성에서 사용

E가 SUCCESS 된 후, F가 실행, E는 사실상 아무 작업도 하지 않고, 작업을 모아두는 역할



SimpleHttpOperator

- 특정 호스트로 HTTP 요청을 보내고 Response를 반환
- 파이썬 함수에서 requests 모듈을 사용한 뒤 PythonOperator로 실행시켜도 무방
- 다만 이런 기능이 Airflow Operator에 이미 존재하는 것을 알면 좋음

이 외에도

- BranchOperator
- DockerOperator
- KubernetesOperator
- CustomOperator (직접 Operator 구현)

클라우드의 기능을 추상화한 Operator도 존재(AWS, GCP 등) - Provider Packages

- <https://airflow.apache.org/docs#providers-packages-docs-apache-airflow-providers-index-htm>

Tip

- 외부 Third Party와 연동해 사용하는 Operator의 경우 (docker, aws, gcp 등) Airflow 설치 시에 다음처럼 extra package를 설치해야 함

```
pip install "apache-airflow[aws]"
```

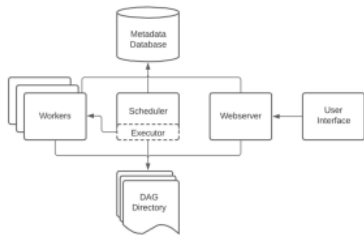
다루지 않은 내용

Airflow DAG을 더 풍부하게 작성할 수 있는 방법으로 다음 내용

- Variable : Airflow Console에서 변수(Variable)를 저장해 Airflow DAG에서 활용
- Connection & Hooks : 연결하기 위한 설정(MySQL, GCP 등)
- Sensor : 외부 이벤트를 기다리며 특정 조건이 만족하면 실행
- Marker
- XComs : Task 끼리 결과를 주고받은 싶은 경우 사용

▼ 3. Apache Airflow 아키텍처와 활용방안

3.1 기본 아키텍처



DAG Directory

- DAG 파일들을 저장
- 기본 경로는 \$AIRFLOW_HOME/dags
- DAG_FOLDER 라고도 부르며, 이 폴더 내부에서 폴더 구조를 어떻게 두어도 상관없음
- Scheduler에 의해 .py 파일은 모두 탐색되고 DAG이 파싱

Scheduler

- Scheduler는 각종 메타 정보의 기록을 담당
- DAG Directory 내 .py 파일에서 DAG을 파싱하여 DB에 저장
- DAG들의 스케줄링 관리 및 담당
- 실행 진행 상황과 결과를 DB에 저장
- Executor를 통해 실제로 스케줄링된 DAG을 실행
- Airflow에서 가장 중요한 컴포넌트

Scheduler - Executor

- Executor는 스케줄링된 DAG을 실행하는 객체로, 크게 2종류로 나뉨
- Local Executor
- Remote Executor

Scheduler - Local Executor

- Local Executor는 DAG Run을 프로세스 단위로 실행하며, 다음처럼 나뉨
- Local Executor
 - 하나의 DAG Run을 하나의 프로세스로 띄워서 실행
 - 최대 생성할 프로세스 수를 정해야 함
 - Airflow를 간단하게 운영할 때 적합
- Sequential Executor
 - 하나의 프로세스에서 모든 DAG Run들을 처리
 - Airflow 기본 Executor로, 별도 설정이 없으면 이 Executor를 사용
 - Airflow를 테스트로 잠시 운영할 때 적합

Scheduler - Remote Executor

- DAG Run을 외부 프로세스로 실행
- Celery Executor
 - DAG Run을 Celery Worker Process로 실행
 - 보통 Redis를 중간에 두고 같이 사용
 - Local Executor를 사용하다, Airflow 운영 규모가 좀 더 커지면 Celery Executor로 전환
- Kubernetes Executor

- 쿠버네티스 상에서 Airflow를 운영할 때 사용
- DAG Run 하나가 하나의 Pod(쿠버네티스의 컨테이너 같은 개념)
- Airflow 운영 규모가 큰 팀에서 사용

Scheduler - Remote Executor

- DAG Run을 외부 프로세스로 실행
- 이 외에도 CeleryKubernetes Executor, Dask Executor 등이 있지만, 여기서는 생략
- Executor의 동작 과정, 라인 블로그 글
 - <https://engineering.linecorp.com/ko/blog/data-engineering-with-airflow-k8s-1/>

Workers

- DAG을 실제로 실행
- Scheduler에 의해 생기고 실행
- Executor에 따라 워커의 형태가 다름
 - Celery 혹은 Local Executor인 경우, Worker는 프로세스
 - Kubernetes Executor인 경우, Worker는 pod.
- DAG Run을 실행하는 과정에서 생긴 로그를 저장

Metadata Database

- 메타 정보를 저장
- Scheduler에 의해 Metadata가 쌓임
- 보통 MySQL이나 Postgres를 사용
- 파싱한 DAG 정보, DAG Run 상태와 실행 내용, Task 정보 등을 저장
- User와 Role (RBAC)에 대한 정보 저장
- Scheduler와 더불어 핵심 컴포넌트
 - 트러블 슈팅 시, 디버깅을 위해 직접 DB에 연결해 데이터를 확인하기도 함
- 실제 운영 환경에서는 GCP Cloud SQL이나, AWS Aurora DB 등 외부 DB 인스턴스를 사용

Webserver WEB UI를 담당

- Metadata DB와 통신하며 유저에게 필요한 메타 데이터를 웹 브라우저에 보여주고 시각화
- 보통 Airflow 사용자들은 이 웹서버를 이용하여 DAG을 ON/OFF 하며, 현 상황을 파악
- REST API도 제공하므로, 꼭 WEB UI를 통해서 통신하지 않아도 괜찮음
- 웹서버가 당장 작동하지 않아도, Airflow에 큰 장애가 발생하지 않음(반면 Scheduler의 작동 여부는 매우 중요)

3.2 Airflow 실제 활용 사례

Airflow를 구축하는 방법으로 보통 3가지 방법을 사용

1. Managed Airflow (GCP Composer, AWS MWAA)

- 보통 별도의 데이터 엔지니어가 없고, 분석가로 이루어진 데이터 팀의 초기에
- 장점
 - 설치와 구축을 클릭 몇번으로 클라우드 서비스가 다 진행
 - 유저는 DAG 파일을 스토리지(파일 업로드) 형태로 관리
- 단점
 - 비용
 - 자유도가 적음. 클라우드에서 기능을 제공하지 않으면 불가능한 제약이 많음

2. VM + Docker compose

- VM + Docker compose는 직접 VM 위에서 Docker compose로 Airflow를 배포하는 방법
- Airflow 구축에 필요한 컴포넌트(Scheduler, Webserver, Database 등)를 Docker container 형태로 배포
- 보통 데이터 팀에 데이터 엔지니어가 적게 존재하는 데이터 팀 성장 초반에 적합
- 장점
 - Managed Service 보다는 살짝 복잡하지만, 어려운 난이도는 아님
 - (Docker와 Docker compose에 익숙한 사람이라면 금방 익힐 수 있음)
 - 하나의 VM만을 사용하기 때문에 단순
- 단점
 - 각 도커 컨테이너 별로 환경이 다르므로, 관리 포인트가 늘어남
 - 예를 들어, 특정 컨테이너가 갑자기 죽을 수도 있고, 특정 컨테이너에 라이브러리를 설치했다면, 나머지 컨테이너에도 하나씩 설치해야 함

3. Kubernetes + Helm

- Kubernetes + Helm은 Kubernetes 환경에서 Helm 차트로 Airflow를 배포하는 방법
- Kubernetes는 여러 개의 VM을 동적으로 운영하는 일종의 분산환경으로, 리소스 사용이 매우 유연한게 대표적인 특징(필요에 따라 VM 수를 알아서 늘려주고 줄여줌)
- 이런 특징 덕분에, 특정 시간에 배치 프로세스를 실행시키는 Airflow와 궁합이 매우 잘 맞음
- Airflow DAG 수가 몇 백개로 늘어나도 노드 오토 스케일링으로 모든 프로세스를 잘 처리할 수 있음
- 하지만 쿠버네티스 자체가 난이도가 있는만큼 구축과 운영이 어려움
- 보통 데이터 팀에 엔지니어링 팀이 존재하고, 쿠버네티스 환경인 경우에 적극 사용

3.3 MLOps 관점의 Airflow

Airflow는 데이터 엔지니어링에서 많이 사용하지만, MLOps에서도 활용할 수 있음

“주기적인 실행”이 필요한 경우

- Batch Training : 1주일 단위로 모델 학습
- Batch Serving(Batch Inference) : 30분 단위로 인퍼런스
- 인퍼런스 결과를 기반으로 일자별, 주차별 모델 퍼포먼스 Report 생성
- MySQL에 저장된 메타데이터를 데이터 웨어하우스로 1시간 단위로 옮기기
- S3, GCS 등 Object Storage
- Feature Store를 만들기 위해 Batch ETL 실행

1. 버킷플레이스 - Airflow 도입기

- <https://www.bucketplace.co.kr/post/2021-04-13-버킷플레이스-airflow-도입기/>

2. 라인 엔지니어링 - Airflow on Kubernetes

- <https://engineering.linecorp.com/ko/blog/data-engineering-with-airflow-k8s-1/>

3. 쏘카 데이터 그룹 - Airflow와 함께한 데이터 환경 구축기

- <https://tech.socarcorp.kr/data/2021/06/01/data-engineering-with-airflow.html>

4. Airflow Executors Explained

- <https://www.astronomer.io/guides/airflow-executors-explained>

▼ 4.3 머신러닝 디자인 패턴

▼ 1. 디자인 패턴

1.1 디자인 패턴이란?

디자인 패턴

- 문제를 해결하는 방법을 패턴화해서 표현
- 반복적으로 발생하는 문제를 어떻게 해결할지에 대한 솔루션
- 추상화된 패턴
- 개발할 때 구조화된 패턴을 설명하는 용어

안티 패턴 : 좋지 않은 패턴

▼ 1.2 머신러닝 디자인 패턴

- 일반적인 개발 디자인 패턴에도 포함할 수 있지만, 머신러닝의 특수성으로 별도의 디자인 패턴이 생김
- 머신러닝 개발의 특수성 : Data, Model, Code
- 소프트웨어 개발은 Code
- 학습, 예측, 운영하면서 생기는 노하우 => 패턴화
- 꼭 이 방법이 Best가 아닐 수 있음(상황에 따라 다른 것) => 참고할 수 있는 지침서로 활용할 수 있음
- 이 강의에선 “이런 패턴도 있구나, 염두해놔야겠다”라는 느낌으로 파악하면 좋음
- 소개하는 여러 패턴을 합쳐서 하나의 패턴으로 될 수도 있음

크게 4가지 패턴

- Serving 패턴 : 모델을 Production 환경에 서빙하는 패턴
- Training 패턴 : 모델을 학습하는 패턴
- QA 패턴 : 모델의 성능을 Production 환경에서 평가하기 위한 패턴

- 비즈니스 메트릭 파악

- Operation 패턴 : 모델을 운영하기 위한 패턴

MLOps와 디자인 패턴은 서로 동반자

▼ 2. Serving 패턴

Online Serving, Batch Serving도 패턴 중 하나

2.1 Web Single 패턴

Usecase

- 가장 간단한 아키텍처
- 예측 서버를 빠르게 출시하고 싶은 경우

Architecture

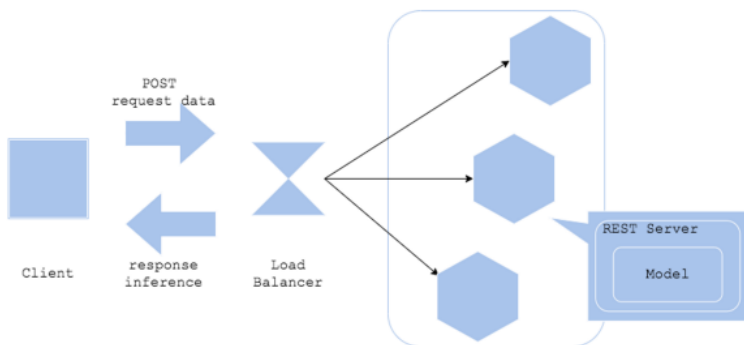
- FastAPI, Flask 등으로 단일 REST 인터페이스 생성
- 요청시 전처리도 같이 포함
- 간단하게 생성할 수 있는 구조

장점

- 하나의 프로그래밍 언어로 진행
- 아키텍처의 단순함
- 처음 사용할 때 좋은 방식

단점

- 구성 요소 하나가 바뀌면 전체 업데이트를 해야함



▼ 2.2 Synchronous 패턴 (우리 모델 해당)

Usecase

- 예측의 결과에 따라 로직이 달라지는 경우
- 예) 예측 결과가 강아지라고 하면 => 강아지 관련 화면, 고양이이면 고양이 화면

Architecture

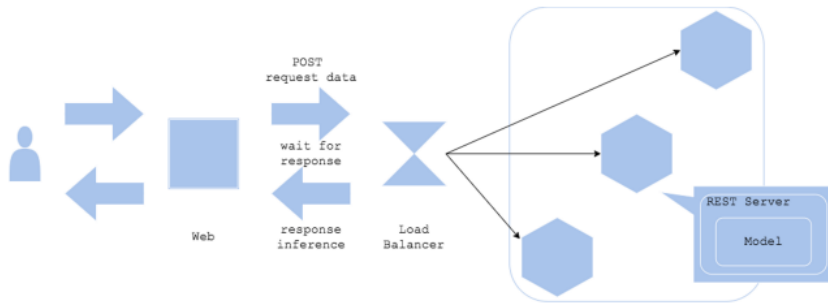
- 예측이 끝날 때까지 프로세스를 Block
- REST API는 대부분 Synchronous 패턴

장점

- 아키텍처의 단순함
- 예측이 완료될 때까지 프로세스가 다른 작업을 할 필요가 없어서 Workflow가 단순해짐

단점

- 예측 속도가 병목이 됨(동시에 1000개의 요청이 올 경우 대기 시간)
- 예측 지연으로 사용자 경험이 악화될 수 있음



▼ 2.3 Asynchronous 패턴

Usecase

- 예측과 진행 프로세스의 의존성이 없는 경우
- 비동기로 실행됨. 예측 요청을 하고 응답을 바로 받을 필요가 없는 경우
- 예측을 요청하는 클라이언트와 응답을 반환하는 목적지가 분리된 경우

Architecture

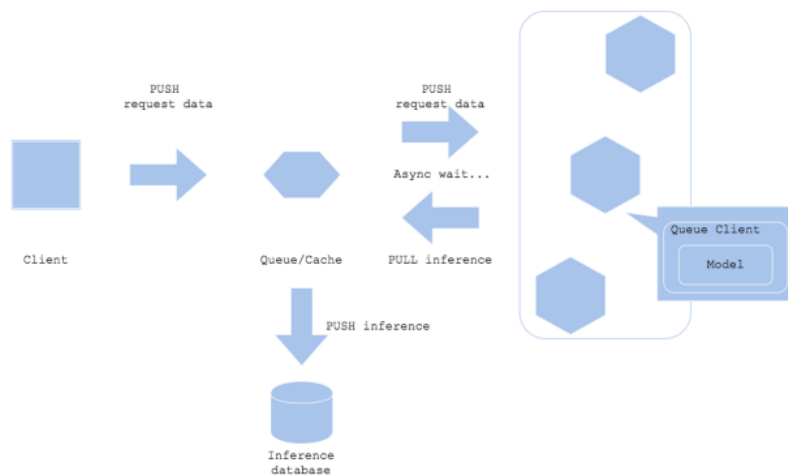
- 클라이언트와 예측 서버 사이에 메세지 시스템(Queue)을 추가
- 특정 메세지에 Request 데이터를 메세지 Queue에 저장(Push)
- 특정 서버는 메세지 Queue의 데이터를 가져와서 예측(Pull)

장점

- 클라이언트와 예측을 분리
- 클라이언트가 예측을 기다릴 필요가 없음

단점

- 메세지 Queue 시스템을 만들어야 함
- 실시간 예측엔 적절하지 않음



▼ 2.4 Batch 패턴

Usecase

- 예측 결과를 실시간으로 얻을 필요가 없는 경우
- 대량의 데이터에 대한 예측을 하는 경우
- 예측 실행이 시간대별, 월별, 일별로 스케줄링해도 괜찮은 경우

Architecture

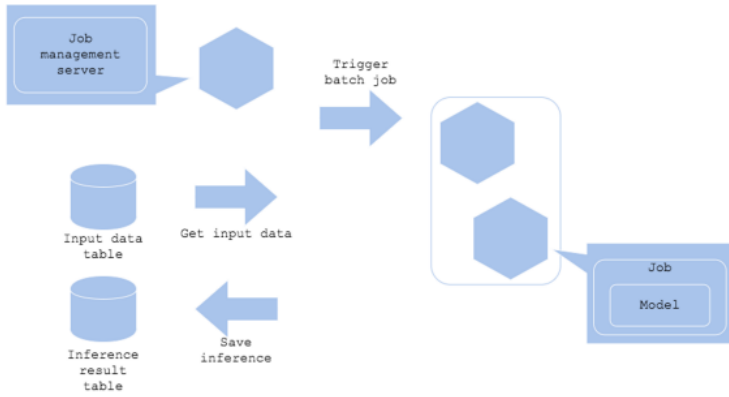
- 실시간으로 진행할 필요가 없는 경우 사용
- Airflow 등으로 Batch 작업을 스케줄링에 맞게 트리거링
- Input, Output 데이터는 데이터 웨어하우스 등에 저장

장점

- API 서버 등을 개발하지 않아도 되는 단순함
- 서버 리소스를 유연하게 관리할 수 있음

단점

- 스케줄링을 위한 서버 필요



▼ 2.5 Preprocess - Prediction 패턴

Usecase

- 전처리와 예측을 분리하고 싶은 경우
- 전처리와 예측에서 사용하는 언어가 다른 경우
- 리소스를 분리해서 효율성을 향상하고 싶은 경우

Architecture

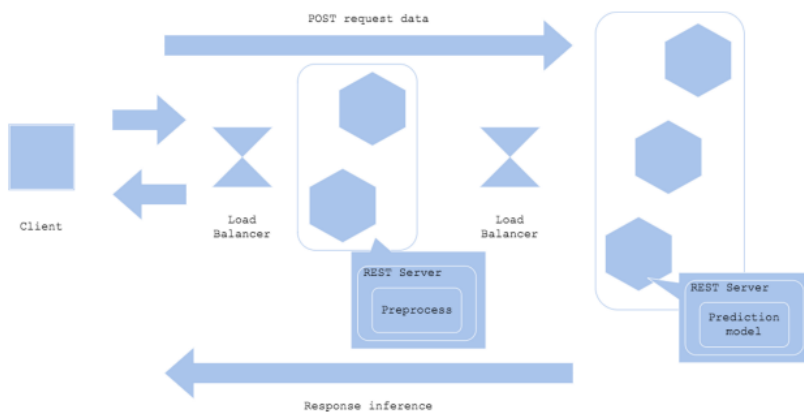
- 전처리 서버와 예측 서버를 분리
- Request를 할 경우 맨 처음엔 전처리 서버로 가서 전처리하고, 그 데이터를 예측 서버로 Request

장점

- 전처리 서버와 예측 서버를 분리해서 효율적으로 리소스를 사용할 수 있음
- Fault Isolation : 장애를 격리할 수 있음
- 딥러닝에선 전처리가 많이 필요해서 이렇게 활용하는 경우도 존재

단점

- 서버 2개를 운영해야 해서 운영 비용이 증가함
- 전처리 서버와 예측 서버 네트워크 연결이 병목 포인트가 될 수 있음



▼ 2.6 Microservice Vertical 패턴

Usecase

- 여러 모델이 순차적으로 연결되는 경우
- A 모델의 결과를 B 모델의 Input으로 사용하는 경우
- 예측끼리 의존 관계가 있는 경우

Architecture

- 각각의 모델을 별도의 서버로 배포
- 동기적으로 순서대로 예측하고, 예측 결과를 다른 모델에 또 Request

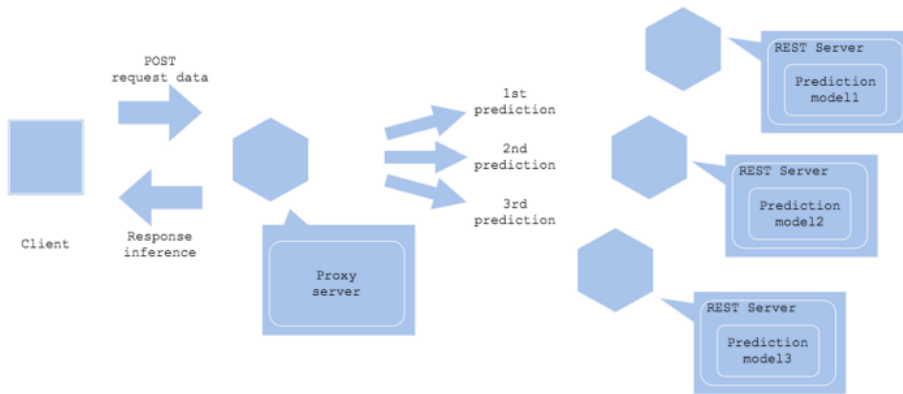
장점

- 여러 예측을 순서대로 실행할 수 있음

- 이전 예측 결과에 따라 다음 예측 모델을 여러개로 분기할 수 있음

단점

- 동기식으로 실행되기 때문에 대기 시간이 더 필요 함
- 하나의 포인트에서 병목이 생길 수 있음
- 복잡한 시스템 구조



▼ 2.7 Microservice Horizontal 패턴

Usecase

- 하나의 Request에 여러 모델을 병렬로 실행하고 싶은 경우
- 보통 이런 경우 마지막에 예측 결과를 통합함
- 마스크 분류 모델에서 한번에 예측할 수도 있지만, 연령대 예측 모델 + 마스크 분류 모델 + 성별 예측 모델 등으로 구성할 수도 있음

Architecture

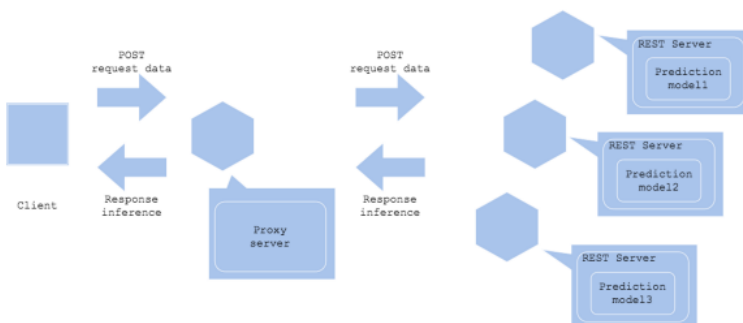
- Microservice Vertical 패턴과 유사하게 모델을 각각의 서버로 배포
- Request가 올 경우 여러 모델 서버로 예측

장점

- 리소스 사용량을 독립적으로 사용할 수 있고, 장애를 격리할 수 있음
- 다른 모델에 의존성 없이 개발할 수 있음

단점

- 시스템이 복잡해질 수 있음



▼ 2.8 Microservice Horizontal 패턴 Serving 패턴

Usecase

- Request할 때 데이터를 저장하고, 예측 결과도 별도로 저장해야 하는 경우
- 예측 결과가 자주 변경되지 않는 경우
- 입력 데이터를 캐시로 활용할 수 있는 경우
- 예측의 속도를 개선하고 싶은 경우

Architecture

- Request가 올 경우 해당 데이터로 예측한 결과가 있는지 캐시에서 검색함(주로 Redis 사용)
 - Redis 메모리 기반으로 속도가 빠름
- 만약 예측 결과가 있다면 해당 데이터를 바로 Return, 예측 결과가 없다면 모델에 예측

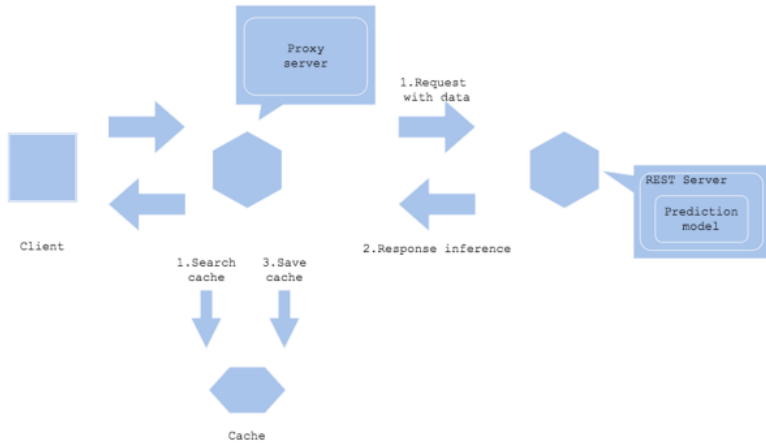
- 오래된 예측이 있다면 주기적으로 삭제하는 로직이 필요함

장점

- 반복되는 요청이 있는 경우 성능을 개선할 수 있음

단점

- 캐시 서버를 운영해야 함



처음 설계 시 Anti 패턴은 피하자

2.9 Serving Anti 패턴 - Online Bigsize 패턴

- 실시간 대응이 필요한 온라인 서비스에 예측에 오래 걸리는 모델을 사용하는 경우
- 속도와 비용 Tradeoff를 조절해 모델 경량화하는 작업이 필요
- 실시간이 아닌 배치로 변경하는 것도 가능한지 검토
- 중간에 캐시 서버를 추가하고, 전처리를 분리하는 것도 Bigsize를 탈피하는 방법

2.10 Serving Anti 패턴 - All-in-one 패턴

- 하나의 서버에 여러 예측 모델을 띄우는 경우
- predict1, predict2, predict3으로 나눠서 하나의 서버에서 모두 실행하는 경우

단점

- 라이브러리 선택 제한이 존재함
- 장애가 발생할 경우(서버가 갑자기 다운) 로그를 확인하기 어려움

▼ 3. Training 패턴

3.1 Batch Training 패턴

Usecase

- 주기적으로 학습해야 하는 경우

Architecture

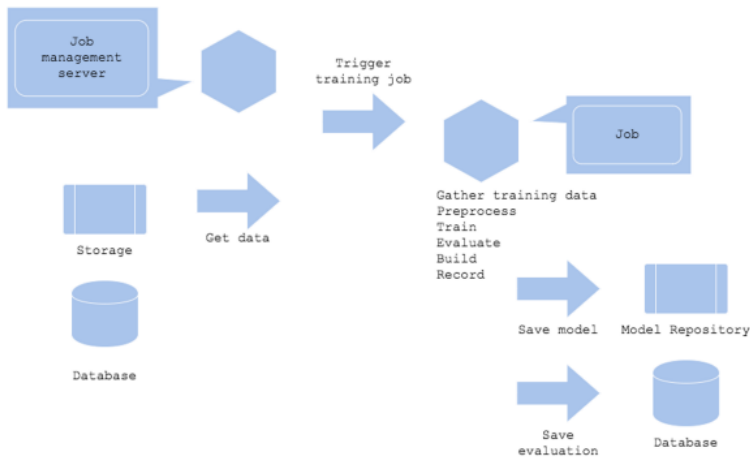
- Batch Serving 패턴처럼 스케줄링 서버 필요
- 학습 과정에서 데이터 전처리, 평가 과정 모두 필요
- 저장한 모델 파일을 사용할 수 있도록 저장하는 작업이 필요

장점

- 정기적인 재학습과 모델 업데이트

단점

- 데이터 수집, 전처리, 학습, 평가 과정에서 오류가 발생할 상황을 고려해야 함



▼ 3.2 Pipeline Training 패턴

Usecase

- 학습 파이프라인 단계를 분리해 각각을 선택하고 재사용할 수 있도록 만드는 경우
- 각 작업을 별도로 컨트롤하고 싶은 경우

Architecture

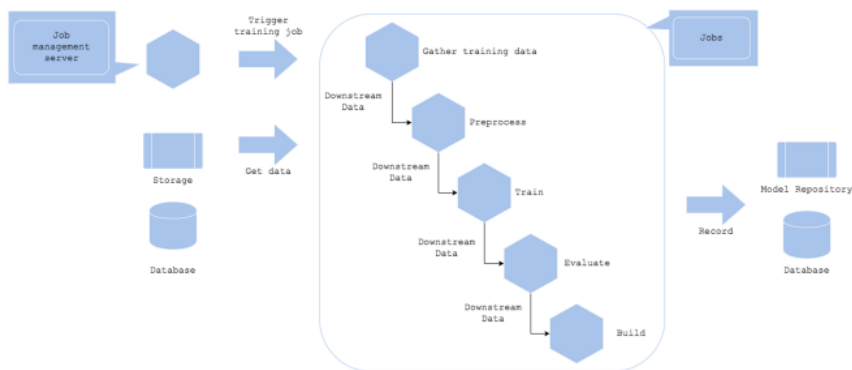
- Batch Training 패턴의 응용 패턴
- 각 작업을 개별 리소스로 분할(서버, 컨테이너 등) => 전처리 서버는 메모리가 크게, 서빙 서버는 GPU 등
- 시간이 많이 걸리는 작업은 자주 실행하고, 다른 작업은 적게 실행
- 이전 작업의 실행 결과가 후속 작업의 Input
- 처리 완료된 데이터를 데이터 웨어하우스에 중간 저장

장점

- 작업 리소스, 라이브러리를 유연하게 선택할 수 있음
- 장애 분리
- Workflow 기반 작업
- 컨테이너를 재사용할 수 있음

단점

- 다중 구조로 여러 작업을 관리해야 함



▼ 3.3 Training Anti 패턴 - Training code in Serving

- 학습, 실험, 평가에 사용해야 하는 코드가 서빙 코드에 들어간 경우
- 학습, 실험, 평가를 위한 환경과 서빙을 같이 처리하는 경우
- Research 단계와 Production 단계에서 필요한 코드와 로직은 다름 => 리소스도 마찬가지로 분리

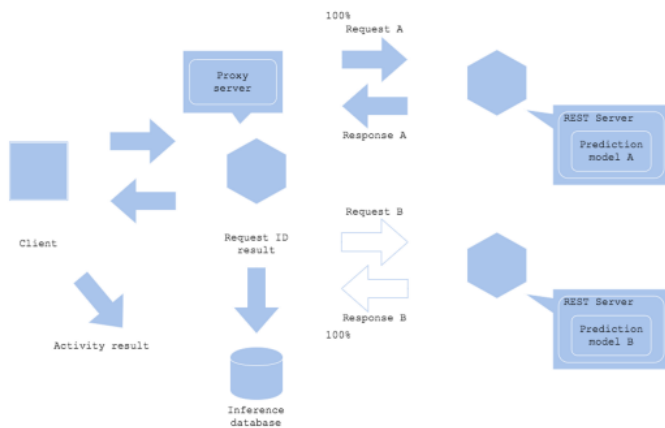
- 예측 모델, 서버를 Production 환경에 배포하기 전에 사용
- Request가 들어온 경우 기존 모델과 새로운 모델에게 모두 전달되고, Response는 기존 모델 서버에만 전달됨(새로운 모델의 결과는 별도로 저장만)
- 모델이 잘 예측하는지 동시에 2개의 모델을 실행해서 파악할 수 있음
- 새로운 모델이 문제가 생기면 AB Test에서 제거하고 다시 개선
- Shadow AB Test 패턴은 Risk가 적음(현재 모델은 그대로 운영)

장점

- Production 환경에 영향을 주지 않고 새로운 모델 성능을 확인할 수 있음
- 여러 모델의 예측 결과를 수집해 분석할 수 있음

단점

- 새로운 예측 서버에 대한 비용이 발생



▼ 4.2 Online AB Test 패턴

Usecase

- 새로운 모델이 Production 환경에서 잘 동작하는지 확인하고 싶은 경우
- 새로운 서버가 Production 환경의 부하를 견딜 수 있는지 확인하고 싶은 경우
- 온라인으로 여러 예측 모델을 측정하고 싶은 경우

Architecture

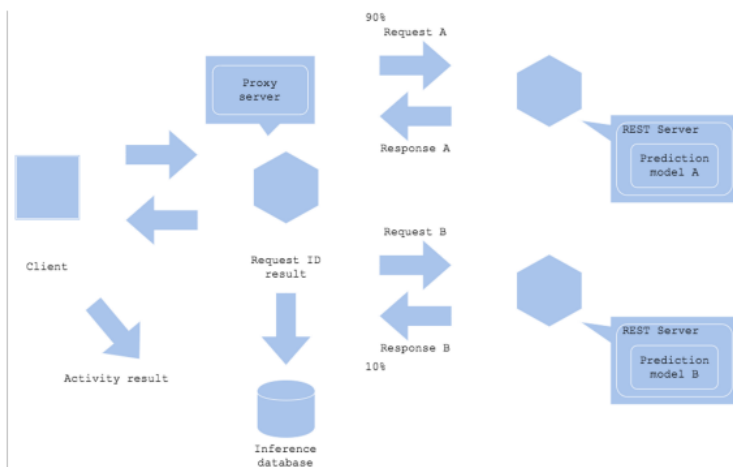
- Shadow AB Test 패턴과 큰 방식은 동일
- Request가 들어오면 지정된 비율(예:1:1)로 트래픽을 나눠서 절반은 기존 모델, 절반은 신규 모델에 예측
- 처음부터 1:1로 하진 않고, 새로운 모델을 10% 정도로 조절하곤 함

장점

- Production 환경에서 새로운 모델의 예측 결과, 속도를 확인할 수 있음
- 여러 모델의 예측 결과를 수집해 분석할 수 있음

단점

- 새로운 모델이 바로 비즈니스에 노출되므로 부정적인 비즈니스 영향이 발생할 수 있음
- 새로운 예측 서버에 대한 비용이 발생

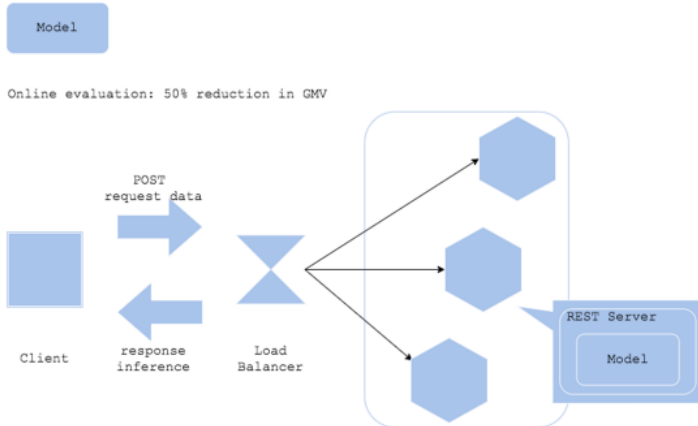


▼ 4.3 QA Anti 패턴 - Offline Only 패턴

- 머신러닝 모델이 Online Test를 하지 않고, Offline Test Data로만 진행되는 경우

- 머신러닝 모델의 비즈니스 가치를 입증하기 어려움
- Production 환경에도 꼭 사용하는 시기가 필요함

Offline evaluation: 99.9999% accuracy, 99.99 precision, 99.99 recall



5. Operation 패턴

머신러닝 시스템의 설정, 로깅, 모니터링 등 운영을 위한 패턴

- 모델의 이미지를 함께 Docker Image로 만들 것인지, 별도로 만들 것인지
- 로그를 어떻게 저장할지, 저장된 로그로 모니터링할지

5.1 Model in Image 패턴

Usecase

- 서비스 환경과 모델을 통합해서 관리하고 싶은 경우
- Docker Image 안에 모델이 저장되어 있는 경우

Architecture

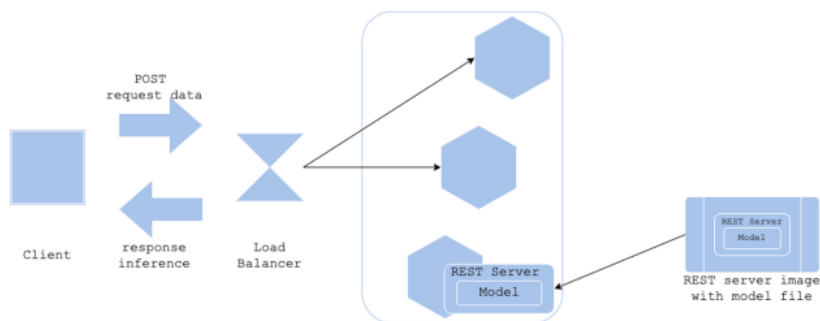
- Docker Image로 모델 코드와 모델 파일(pk1 등)을 저장해서 사용
- Production 환경에선 이 이미지를 Pull해서 사용

장점

- Production 환경과 Dev 환경을 동일하게 운영할 수 있음

단점

- 모델을 수정하는 일이 빈번할 경우, Docker Image Build를 계속 수행해야 함



5.2 Model Load 패턴

Usecase

- Docker 이미지와 모델 파일을 분리하고 싶은 경우
- 모델 업데이트가 빈번한 경우
- 서버 이미지는 공통으로 사용하되, 모델은 여러개를 사용하는 경우

Architecture

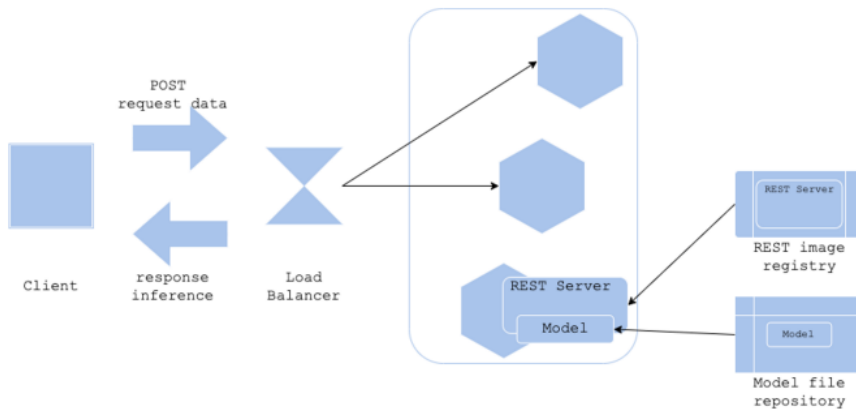
- 개발 코드는 Docker Image로 Build
- 모델 파일은 Object Storage 등에 업로드하고 프로세스 시작할 때 모델 파일을 다 (프리트레인 모델처럼)
- 분리를 통해 서버 이미지를 경량화할 수 있는 패턴

장점

- 모델과 서버 이미지를 구분
- 서버 이미지를 재사용할 수 있으며, 서버 이미지가 경량화됨

단점

- 모델 파일을 가지고 와야하기 때문에 서비스 시작하는데 더 오래 걸릴 수 있음
- 서버 이미지, 모델 관리를 해야 함



▼ 5.3 Prediction Log 패턴

Usecase

- 서비스 개선을 위해 예측, 지연 시간(latency) 로그를 사용하려고 할 경우
- Data Validation, 예측 결과 등을 확인하고 싶은 경우

Architecture

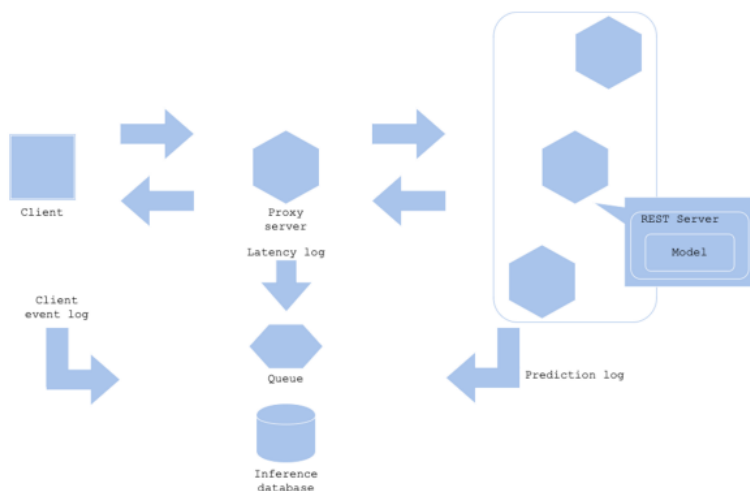
- 프로세스에서 로그를 저장하지 않고, 메세지 시스템으로 넘겨서 프로세스가 저장에 신경쓰는 시간을 줄임
- 장애 등을 파악할 수 있도록 로그도 기록하고 모니터링도 할 수 있도록 대비해야 함

장점

- 예측 결과, 모델의 Latency 등을 분석할 수 있음

단점

- 로그가 많아지면 저장 비용이 발생함



▼ 5.4 Condition Based Serving 패턴

Usecase

- 상황에 따라(특정 조건) 예측해야 하는 대상이 다양한 경우
- 룰 베이스 방식으로 상황에 따라 모델을 선택하는 경우

Architecture

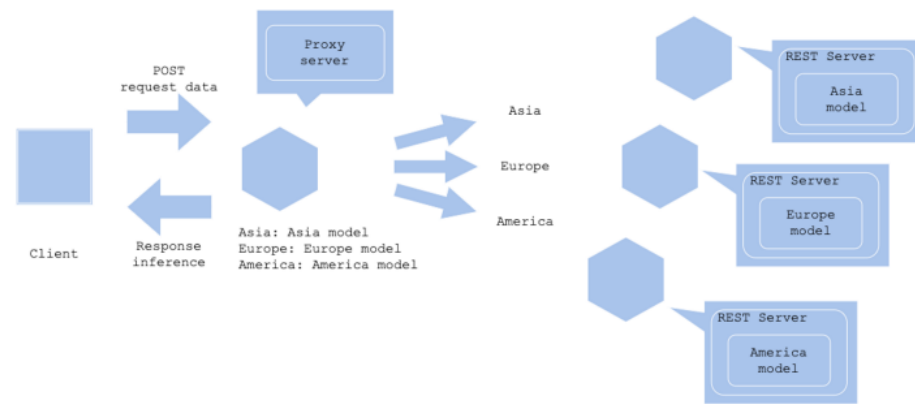
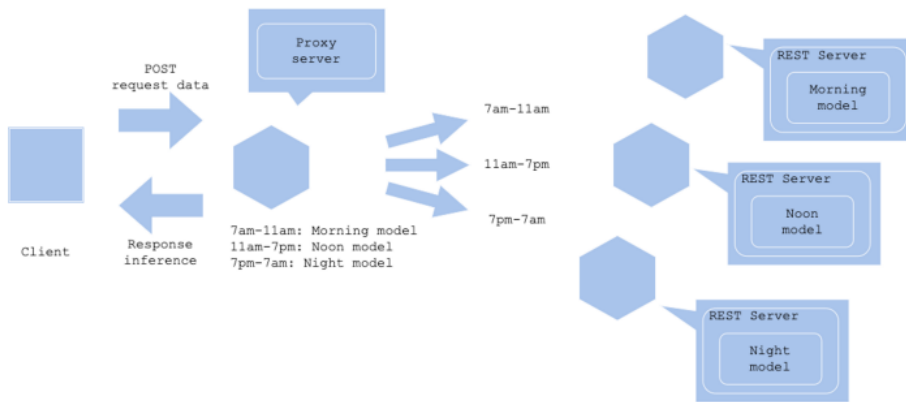
- 사용자의 상태, 시간, 장소에 따라 예측 대상이 바뀔 수 있음

장점

- 상황에 따라 알맞은 모델 제공

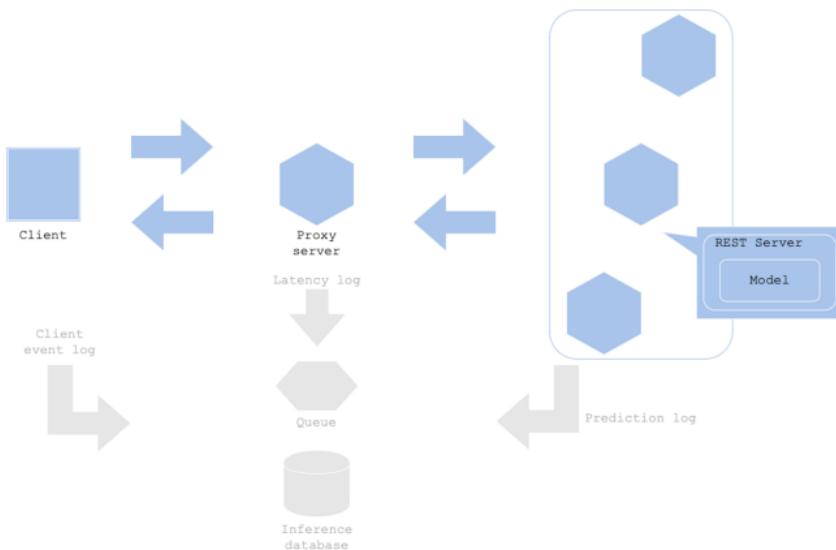
단점

- 모델 수에 따라 운영 비용 증가



▼ 5.5 Operation Anti 패턴 - No Logging 패턴

- 별도의 로그를 남기지 않는 경우



정리

- 머신러닝 시스템 개발도 점점 패턴화되고 있음
- 어떻게 설계할지 고민이 된다면 머신러닝 디자인 패턴을 찾아본 후, 설계하는 것도 방법
- 위 방법이 항상 절대 진리는 아니고, 프로젝트 상황에 따라 다름
- 여러 패턴이 어떤 장단점이 있는지 하나씩 파악하기

추가학습 자료

인터넷

- ml-system-design-pattern : https://mercari.github.io/ml-system-design-pattern/README_ko.html
- ml-system-design-pattern code : <https://github.com/shibuiwilliam/ml-system-in-actions>

교재

- 머신러닝 시스템 디자인 패턴, 위키북스
- 머신러닝 디자인 패턴, 오라일리

▼ 4.4 앞으로 더 공부하면 좋을 내용

▼ 1. 앞으로 더 공부하면 좋을 내용

1.1 개발

파이썬

- 파이썬 코딩의 기술
- [인프런 파이썬 중급 프로그래밍 강좌](#)

리눅스

- 데이터 전처리 속도 빠름
- 김태용의 리눅스 셸 스크립트 프로그래밍 입문
- 만화로 배우는 리눅스 시스템 관리
- 유닉스 리눅스 셸 스크립트 예제 사전

클린 코드/아키텍처/DDD(Domain Driven Development)

- 파이썬 클린 코드 (쉬움, 정답은 아님)
- 파이썬으로 살펴보는 아키텍처 패턴
- 클린코드, 클린코드 아키텍처 (어려움)

▼ 1.2 데이터 엔지니어링

AI Engineer는 데이터 모델링 + 데이터 개발 영역을 커버하길 원하는 추세

- 실시간 데이터를 어떻게 전처리하고, 가져올지에 대한 이해도가 있을수록 좋음
- 데이터 엔지니어링 관련 생태계도 엄청 넓은 상태. 하나씩 격파하기

추천 도서

- 빅데이터를 지탱하는 기술 (초급)
- 데이터 중심 애플리케이션 설계 (중급)

데이터 엔지니어링 영역

- 실시간 데이터 처리 : Kafka(대세), Apache Spark Streaming ...
 - 배치하다가 실시간
- 메시지 시스템 : Kafka, Redis, AWS SQS, GCP PubSub, Celery
- 분산 처리 : Ray, Apache Spark
 - 데이터 엔지니어링 및 연산을 위해 사용
- 데이터 웨어하우스 : GCP BigQuery, AWS Redshift
- 캐싱 : Redis
- BI(Business Intelligence) : Superset, Redash, Metabase
- ETL 파이프라인 : 어떻게 구성할 것인가?

1.3 Computer Science

Computer Science에 대한 이해

- 네트워크
- OS
- 자료 구조
- 알고리즘

▼ 1.4 Cloud, Infra

클라우드 환경 작업이 점점 익숙해지는 상황(큰 회사의 경우 아니지만) 어느정도 인프라에 대한 이해가 있을수록 인프라 엔지니어와 이야기할 때 수월함

- Docker를 넘어서서 Kubernetes 학습!
- CICD를 더 잘하기 위한 방법도 고민
 - Cloud build, Jenkins

추천 도서

- 핵심만 꼭 쿠버네티스
- laaC
- 우리가 직접 클릭하면서 클라우드 서비스를 사용했으나, 인프라를 코드로 관리할 수 있는 Terraform
- 인프라 환경 설정할 경우 매우 유용
- [44bits의 Terraform 글](#)

Monitoring

- 모니터링을 더 잘하기 위한 고민도 필요
- Prometheus, <https://github.com/prometheus/prometheus>
- Grafana, <https://github.com/grafana/grafana>

1.5 Database

- 데이터를 저장하는 곳은 Object Storage, NoSQL도 있지만 여전히 RDB에 저장을 많이 함
- 효율적으로 저장하려면 어떻게 해야할까?
- Index 전략 등
- 저장한 데이터를 추출할 수 있도록 SQL
- 데이터 웨어하우스로 데이터를 옮기기 위한 고민

1.6 Modeling

- 머신러닝 모델링에 대한 이해도 있고, Researcher나 Scientist 분들에게 “이 부분 불편하니 이렇게 개선해보면 어떨까요?” 제안할 수 있는 사람
- 최근 논문의 방향성, SOTA 등을 어느정도 파악해두는 정도라도!

▼ 1.7 추천 콘텐츠 - 기술 블로그 & 발표 영상 & 논문

공부할 때 “책”, “강의”를 보는 방법을 이젠 넘어서 시기

- 각종 회사의 기술 블로그 찾아보기
 - <https://github.com/seonggwonyoon/techblog>
 - 어떤 기술을 “왜” 사용했는가(어떤 상황이었고, 어떻게 문제 정의했는가)
 - 기술 블로그 내용을 잘 정리하고 => 추상화해서 기록해보기(이 산업에선 이 회사에선 이렇게 풀었다, 동종 업계인 다른 회사는 저런 방식으로 풀었다 등) => 취업이 아니라 긴 커리어를 가지기 위해서도 좋음
- 해외는 Uber, Door Dash 추천. Uber는 데이터 엔지니어링 논문도 종종 Publish
- 중국: 디디추싱

발표 영상

- 회사별로 컨퍼런스하는 추세, 커뮤니티 행사도 여전히 존재
- 네이버의 DEVVIEW, 카카오의 IFKAKO, 토스의 SLASH, 우아한형제들의 우아콘, 데이터야놀자, PyCon

기술 블로그와 마찬가지로 “왜?”를 고민하기

- 컨퍼런스를 날 잡고 보는 것도 추천 + 친구들과 같이 토론해보기
- DEVVIEW는 2009년부터 존재 => 과거엔 어떤 발표가 있었는지도 살펴보기(이 당시엔 이런 생각을 가지고 A 프레임워크가 제일 좋았구나 => 현재는 이 부분을 개선하기 위해 이런 것들을 만들었구나 등)

논문

- 머신러닝, 딥러닝 논문
- 데이터 엔지니어링 논문
- CS(Computer Science) 논문
- 라이브러리, 프레임워크의 논문(Hadoop, Ray 등)

▼ 2. 앞으로의 삶의 방향성

2.1 취업, 데이터 업무, 방향성

취업 자세

- 조금해하지 않기 나에 대한 파악하고 취업 고민하기
- 회사와 나 모두 성장할 수 있는 회사

데이터 업무

- 현재 맡은 업무와 내가 하고 싶은 것과 Align 되어있는지 고민
- Align 되어있다면 누구보다 빠르게 달려가기. 몰입하기
- 방향성이 맞지 않는 것 같아도, 결국 경험이 연결되어 시너지가 나기에 그 부분에 대해 이해하기
- 데이터 분석, 머신러닝 모델링, 데이터 엔지니어링, 개발 등에 대한 고민 계속 하기

방향성

- 5년 10년 후의 삶의 목표 정하기

2.2 동기부여

회사에 들어간 이후에도 지속적인 동기부여, Refresh에 대해 생각하기

- 정답은 없고, 동기부여가 꼭 필요한 것이 아닐 수도 있음
- 외부에서 주는 동기부여와 셀프 동기부여에 대해 고민해보기
- 학습 동기부여

2.3 태도, 목적성

- 지속적인 개선을 하겠다는 태도
- 내가 아는 것이 잘못될 수도 있고 인정하는 태도
- 목적을 가지고 작업에 임하는 태도
- 조직과 함께 성장하려는 태도