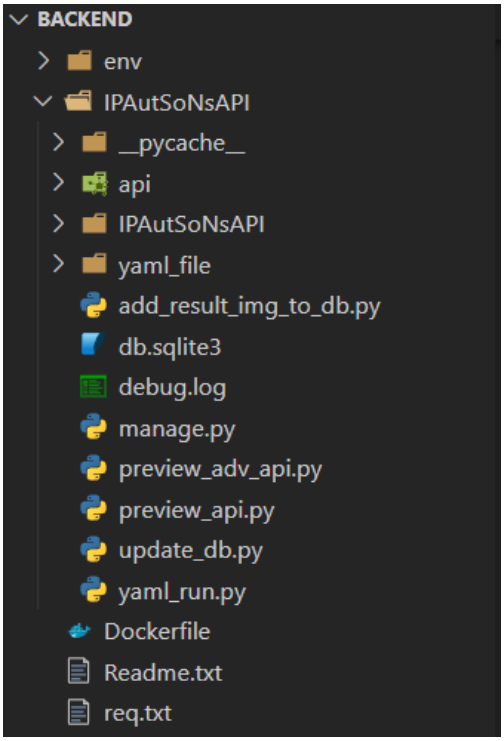
คู่มือสำหรับนักพัฒนา

1. ส่วนของ Back-End

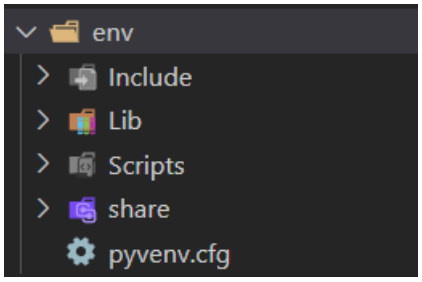
ภายในโฟลเดอร์ backend ซึ่งถูกพัฒนาด้วย Python – Django ซึ่งสามารถ clone ได้จาก https://github.com/SuteeSaraphan/IPAuTSoNS ประกอบไปด้วยไฟล์และโฟลเดอร์ดังรูป 1



รูป 1 ไฟล์และโฟลเดอร์ภายในโฟลเดอร์ backend

1.1 โฟลเดอร์ env

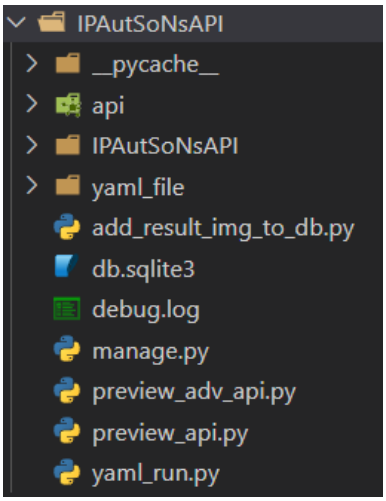
โฟลเดอร์ env เป็นโฟลเดอร์ที่ถูกสร้างขึ้นมาเพื่อสร้าง environment ในการพัฒนาให้กับโฟลเดอร์ IPAutSoNsAPI เพื่อลดปัญหาที่อาจจะเกิดขึ้นจาก version ของ library เสริมที่ติดตั้งเข้ามาเพื่อใช้งาน และ ลดปัญหาการหา path file ไม่พบอีกด้วย โดยมีไฟล์และโฟลเดอร์ดังรูป 2



รูป 2 ไฟล์และโฟลเดอร์ภายในโฟลเดอร์ env

1.2 โฟลเดอร์ IPAutSoNsAPI

โฟลเดอร์ IPAutSoNsAPI เป็นโฟลเดอร์หลักของการเก็บไฟล์ที่ใช้งานในการสร้างส่วน Backend โดยจะมี Django project ที่ถูกสร้างเอาไว้ภายใน โดยมีไฟล์และโฟลเดอร์ดังรูป 3

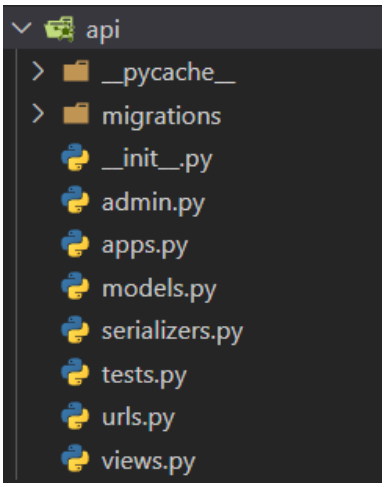


รูป 3 ไฟล์และโฟลเดอร์ภายในโฟลเดอร์ IPAutSoNsAPI

โดยตัว project Django จะแบ่งเป็น 2 ส่วนหลักคือ โฟลเดอร์ IPAutSoNsAPI และ โฟลเดอร์ api และไฟล์คำสั่งย่อยอื่นๆ ประกอบไปด้วย

1.2.1 โฟลเดอร์ api

โดยโฟลเดอร์จะเป็นส่วนหลักของการทำงานด้าน API ที่เชื่อมต่อเข้ากับ Frontend , การประมวลผลและการจัดเก็บข้อมูลต่างๆ ของ Web application โดยมีไฟล์และโฟลเดอร์ดังรูป 4



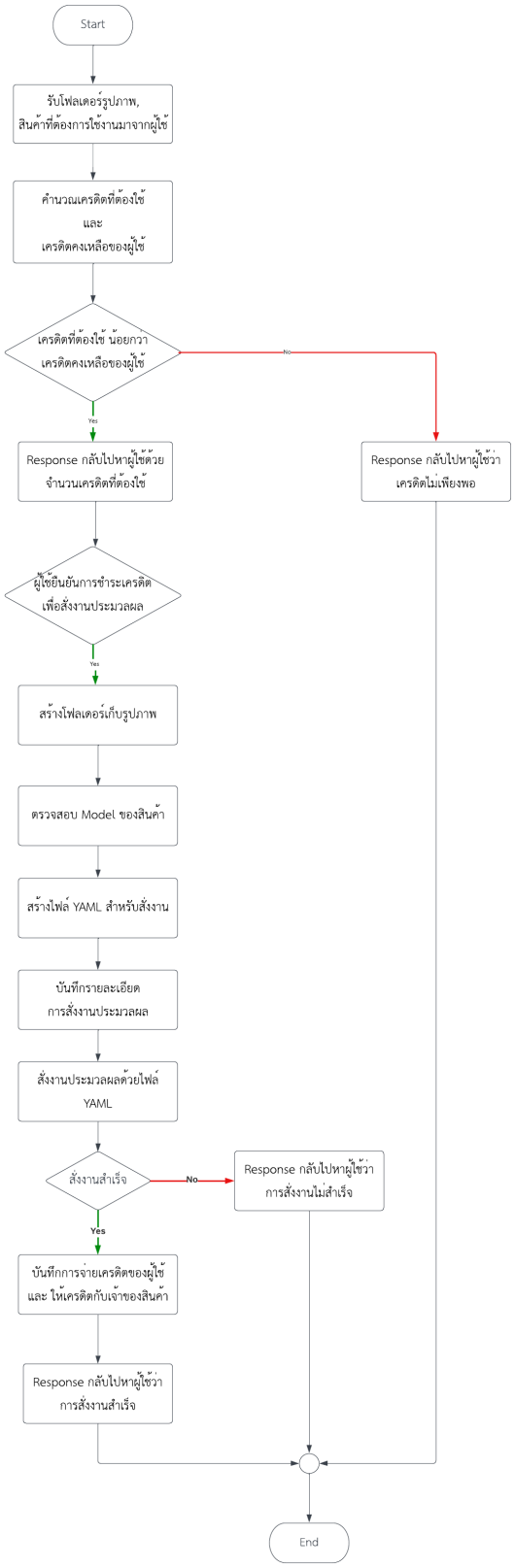
รูป 4 ไฟล์และโฟลเดอร์ภายในโฟลเดอร์ api

1) model.py เป็นไฟล์ที่ใช้สำหรับกำหนดโครงสร้างของ database โดยสามารถกำหนดโครงสร้าง Entity และ Attribute ของ Entity ได้โดยจะอยู่ในรูปแบบของ Class
2) serializers.py เป็นไฟล์ที่ใช้สำหรับปรับรูปแบบการเข้าถึง Model ต่างๆเพื่อให้ง่ายต่อการเรียกใช้ Model
3) url.py เป็นไฟล์สำหรับตั้ง URL เพื่อเข้าถึง API เรียกใช้ Functions ต่างๆ โดยสามารถใช้ method ได้หลากหลายในการเข้าถึง เช่น GET,POST,PUT,DELETE เป็นต้น ซึ่งสามารถกำหนดให้รับค่า parameter เพื่อนส่งต่อไปให้ Functions เรียกใช้งานได้อีกด้วย โดยส่วนปนะสอบสามารถดูได้ที่รูป 5

```
from django.urls import path
from . import views

urlpatterns = [
    path('make_docker_file', views.MakeDockerFile.as_view()),
    path('yolo_export', views.YoloExport.as_view()),
    path('gan_export', views.GanExport.as_view()),
    path('version', views.VersionCheck.as_view()),

    path('register', views.RegisterView.as_view()),
    path('login', views.LoginView.as_view()),
    path('user', views.UserView.as_view()),
    path('password', views.PasswordView.as_view()),

    path('image', views.ImageView.as_view()),
    path('image/<str:folder_id>', views.ImageView.as_view()),
    path('image/<str:type>/<str:folder_id>', views.ImageView.as_view()),

    path('all_images', views.AllImageView.as_view()),
    path('folder_img', views.FolderView.as_view()),
    path('folder_img/<str:folder_id>', views.FolderView.as_view()),

    path('product', views.ProductView.as_view()),
    path('product/<str:type>/<str:key>', views.ProductView.as_view()),

    path('market/<str:key>', views.MarketView.as_view()),

    path('payment', views.PaymentView.as_view()),
    path('price_check/<str:product_id>/<str:folder_name>', views.PriceCheckView.as_view()),
    path('feed', views.FeedView.as_view()),

    path('user_history/<str:type>', views.UserHistoryView.as_view()),
    path('product_history/<str:product_id>/<str:type>',
         views.ProductHistoryView.as_view()),
    path('job_history/<str:type>', views.JobHistoryView.as_view()),

    path('open_product', views.OpenProductView.as_view()),


    path('preview', views.PreviewNormalView.as_view()),
    path('preview_adv', views.PreviewAdvanceView.as_view()),

    path('download_img/<str:img_id>',views.download_file, name='download-file'),
    path('dowload_folder/<str:folder_id>',views.download_folder, name='download-folder'),

]
```

รูป 5 URL ทั้งหมดที่ใช้งาน API ได้

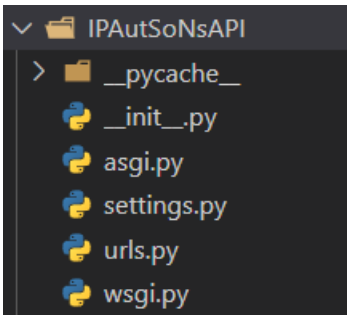
4) view.py เป็นไฟล์ที่รบรวมการทำงานประมวลผลและจัดเก็บข้อมูลทั้งหมดเอาไว้โดยจะถูกแบ่งออกเป็น class ตามการทำงานที่เกี่ยวข้อง โดยการทำงานสำหรับสั่งงานประมวลผลภาพก็จะอยู่ในไฟล์นี้เช่นกัน โดยจะอยู่ใน 4 class คือ class PriceCheckView , MakeDockerFile, YoloExport และ GanExport โดยมีขั้นตอนการทำงานดังรูป 6



รูป 6 Flowchart สำหรับ ขั้นตอนการทำงานสั่งงานประมวลผลภาพ

1.2.2 โฟลเดอร์ IPAutSoNsAPI

เป็นโฟลเดอร์หลักของ Django Project ที่ได้มีการเก็บไฟล์ config ค่าต่างๆ ของ backend เอาไว้ โดยมีไฟล์และโฟลเดอร์ดังรูป 7

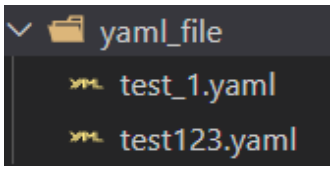


รูป 7 ไฟล์และโฟลเดอร์ที่อยู่ในโฟลเดอร์ IPAutSoNsAPI

1) setting.py ไฟล์รวมการตั้งค่าทั้งหมดของ Project โดยสามารถปรับค่าต่างๆ เฉพาะด้านได้ภายในไฟล์นี้ เช่น การเชื่อมต่อ Database, การกำหนด IP address สำหรับใช้งาน Server ไปจนถึงการกำหนด path เก็บไฟล์
2) url.py ไฟล์ไว้สำหรับการตั้งค่า URL ที่จะให้เข้าถึงการใช้งาน API

1.2.3 โฟลเดอร์ yaml_file

เป็นโฟลเดอร์สำหรับเก็บไฟล์ประเภท yaml เพื่อเรียกสั่งงาน order image processing job ซึ่งจะสามารถปรับเปลี่ยนย้ายที่ได้หากผู้พัฒนาต้องการ โดยรูป 7 คือรูปภายในโฟล์เดอร์ yaml_file และ รูป 8 คือตัวอย่างไฟล์ yaml ที่ใช้สำหรับสั่งงาน



รูป 8 ไฟล์ที่อยู่ในโฟลเดอร์ yaml_file

```yaml
apiVersion: batch/v1

kind: Job

metadata:

  name: test123

spec:

  template:

    spec:

      containers:

      - name: test123

        image: suteesaraphan27/ascii
        volumeMounts:
            - name: nfs-share
              mountPath: /ipautsons

        command: ["python","ASCII.py","test123","/ipautsons/img"]

      restartPolicy: Never
      volumes:
      - name: nfs-share
        persistentVolumeClaim:
          claimName: example
```

รูป 9 ตัวอย่างไฟล์ yaml สำหรับสั่งงาน

1.2.4 ไฟล์ manage.py

ไฟล์สำหรับสั่งเปิดใช้งาน backend API server ให้สามารถเข้าใช้งานได้ผ่าน URL ที่ผู้พัฒนานั้นตั้งเอาไว้

1.2.5 ไฟล์ preview_adv_api.py

ไฟล์สำหรับการสั่งงานแบบ Preview เพื่อให้ได้รูปที่ผ่านการประมวลผลแบบเร็วที่สุด โดยที่ไม่ห่วงคุณภาพ เพื่อให้สามารถนำไปแสดงผลให้ผู้ใช้ได้ไวที่สุด โดยจะเป็นการ Preview ในรูปแบบใช้งาน weight จากสินค้าที่มีผู้ใช้เพิ่มขึ้นมา โดยรูป 10 คือภายในของไฟล์

```
5   class PreviewADVAPI():
6       def __init__(self,img,img_type,model,weight_path):
7           self.url = 'http://192.168.1.46:'
8           self.img = img
9           self.img_type = img_type
10          self.model = model
11          self.weight_path = weight_path
12
13      def __str__(self) -> str:
14          return str(self.weight_path)
15
16      def do_preview(self):
17          logger.error('runed preview functions')
18          headers = {'accept': 'application/json'}
19          files = {'file': self.img}
20          docker_url = None
21          logger.error('preview 01')
22          if(self.model == 'YOLOv5'):
23              print("2")
24              docker_url = self.url+'4050/detect-to-img?modelse='+str(self.weight_path)
25          elif(self.model == 'GANs'):
26              logger.error('preview 02')
27              docker_url = self.url+'4070/gan?modelse='+str(self.weight_path)
28          try:
29              logger.error('preview 03')
30              headers = {'accept': 'application/json'}
31              files = {'file': self.img}
32              response = requests.post(docker_url, headers=headers, files=files)
33              logger.error('preview 04')
34              logger.error(response.content)
35              return response.content
36          except Exception as error:
37              logger.error('runed preview functions Error : '+str(error))
38              return error
```

รูป 10 ภายในของไฟล์ preview_adv_api.py

1.2.6 ไฟล์ preview_api.py

ไฟล์สำหรับการสั่งงานแบบ Preview เพื่อให้ได้รูปที่ผ่านการประมวลผลแบบเร็วที่สุด โดยที่ไม่ห่วงคุณภาพ เพื่อให้สามารถนำไปแสดงผลให้ผู้ใช้ได้ไวที่สุด โดยจะเป็นการ Preview ในรูปแบบใช้งาน weight จากสินค้าที่มีเป็นของทาง Web application โดยรูป 11 คือ ภายในของไฟล์

```python
class PreviewAPI():
    def __init__(self,img,img_type,filter_id,filter_value):
        self.url = 'http://192.168.1.46:'
        self.img = img
        self.img_type = img_type
        self.filter_id = filter_id
        self.filter_value = filter_value

    def __str__(self) -> str:
        return str(self.img)

    def do_preview(self):
        logger.error('runed preview functions')
        headers = {'accept': 'application/json'}
        files = {'file': self.img}


        if(self.filter_id == 'Black and White'):
            docker_url = self.url+'4020/blackwhite'
        elif(self.filter_id == 'ASCII'):
            docker_url = self.url+'4020/ascii'
        elif(self.filter_id == 'PixelArt'):
            docker_url = self.url+'4020/pixelart'
            docker_url += "?pixel="+str(self.filter_value).split(" ")[0]
        elif(self.filter_id == 'Mosaic'):
            docker_url = self.url+'4020/mosaig?folders=/'+(str(self.filter_value))
            print(docker_url)
        try:
            headers = {'accept': 'application/json'}
            files = {'file': self.img}
            response = requests.post(docker_url, headers=headers, files=files)
            print('preview complete')
            return response.content
        except Exception as error:
            logger.error('runed preview functions Error : '+str(error))
            return error
```

รูป 11 ภายในของไฟล์ preview_api.py

1.2.7 ไฟล์ yaml_run.py

ไฟล์สำหรับสั่งงานการทำงาน Image processing โดยผ่านการสั่งให้ไฟล์ yaml ให้ทำงาน ซึ่งจะสามารถส่งค่าสถานการณ์สั่งงานกลับมาได้ว่าสั่งงานสำเร็จหรือไม่ โดยรูปภาพ 12 คือตัวอย่างภายในไฟล์

```python
from kubernetes import client, config, utils
import os
import logging
logger = logging.getLogger(__name__)


class YamlRunner:
    def __init__(self,job_id):
        self.yaml_file = 'yaml_file/'+job_id+'.yaml' # stable
    def __self__(self):
        print("runner of "+str(self.yaml_file))

    def run_yaml(self):
        logger.error('run '+self.yaml_file)
        try:
            config.load_incluster_config()
            k8s_client = client.ApiClient()
            utils.create_from_yaml(k8s_client, self.yaml_file)
            logger.error('runed run_yaml functions')
            return 1
        except (config.ConfigException,BaseException,utils.FailToCreateError,Exception) as error:
            logger.error('Fail to run yaml because : ' + str(error))
            return str(error)
```

รูป 12 ภายในของไฟล์ yaml_run.py

1.3 ไฟล์ Dockerfile

ใช้สำหรับการสร้างหรือ build Docker image เพื่อใช้สำหรับการ Deploy ส่วน Backend ในเครื่อง Server จริงโดยรูปภาพ 13 คือตัวอย่างภายในไฟล์

```
FROM --platform=$BUILDPLATFORM python:3.10-slim-buster AS backend


RUN mkdir /backend

WORKDIR /backend

COPY req.txt /backend

RUN apt update

RUN apt install gcc -y

RUN pip3 install -U setuptools

RUN pip3 install --upgrade pip

RUN pip3 install -r req.txt

RUN pip3 install pymongo[srv]

COPY . /backend


WORKDIR /backend/IPAutSoNsAPI

ENTRYPOINT ["python3","manage.py", "runserver", "192.168.1.134:8000"]
```

รูป 13 ภายในของไฟล์ Dockerfile

1.4 ไฟล์ Readme.txt

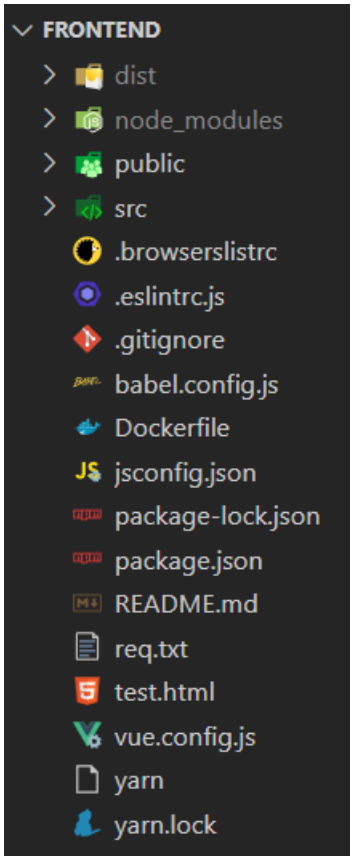
เป็นไฟล์สำหรับแสดงรายละเอียดของโฟลเดอร์ backend

1.5 ไฟล์ req.txt

เป็นไฟล์สำหรับจัดการ package ต่าง ๆ ที่ใช้ในโปรเจค

2. ส่วนของ Front-End

ภายในโฟลเดอร์ Frontend ซึ่งถูกพัฒนาด้วย JavaScript – Vue JS ซึ่งสามารถ clone ได้จาก https://github.com/SuteeSaraphan/IPAuTSoNS ประกอบไปด้วยไฟล์และโฟลเดอร์ดังรูป 14



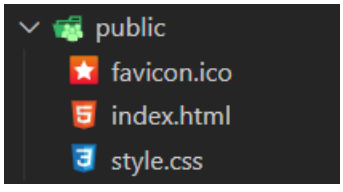
รูป 14 ภายในของโฟลเดอร์ Frontend

2.1 โฟลเดอร์ node_modules

โฟลเดอร์สำหรับเก็บรวบรวมไฟล์ package ต่างๆของ project node JS ที่ได้สร้างขึ้นมาในรูปแบบ Vue JS ซึ่งจะรวบรวมไฟล์ package ทั้งหลัก และส่วนเสริมอื่นๆที่ถูกติดตั้งเพิ่มเข้ามา

2.2 โฟลเดอร์ public

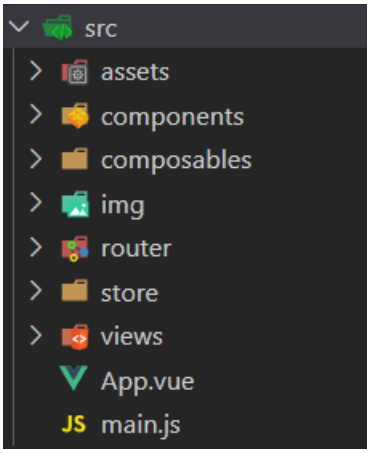
โฟลเดอร์สำหรับเก็บไฟล์ในส่วนที่แสดงผลหลักให้ผู้ใช้เห็นในหน้า Web application โดยมีส่วนประกอบภายในโฟลเดอร์ดังรูป 15



รูป 15 ภายในของโฟลเดอร์ public

2.3 โฟลเดอร์ src

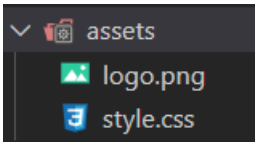
โฟลเดอร์หลักสำหรับเก็บ source code ของ project นี้ โดยจะมีส่วนประกอบดังรูป 16



รูป 16 ภายในของโฟลเดอร์ public

2.3.1 โฟลเดอร์ assets

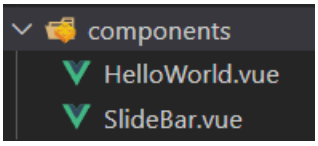
เป็นโฟลเดอร์สำหรับใช้ในการเก็บไฟล์ CSS หรือ รูปภาพ ที่เป็น static file ที่จำนำมาใช้งานแสดงผลบนหน้า Web application โดยจะมีส่วนประกอบดังรูป 17



รูป 17 ภายในของโฟลเดอร์ assets

2.3.2 โฟลเดอร์ components

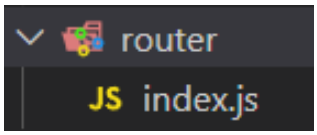
เป็นโฟลเดอร์ที่มีไว้สำหรับการสร้างและจัดเก็บ component ต่างๆ ที่จะนำไปใช้แสดงผลในหน้า Web application โดยจะมีส่วนประกอบดังรูป 18



รูป 18 ภายในของโฟลเดอร์ components

2.3.3 โฟลเดอร์ router

เป็นโฟลเดอร์สำหรับเก็บการเรียกใช้ไฟล์ต่างๆ โดยที่จะเน้นไปที่การเรียกใช้ view อื่นๆขึ้นมาเพื่อแสดงผลในหน้า Web application โดยจะมีส่วนประกอบดังรูป 19 และ 20 ไฟล์ index.js



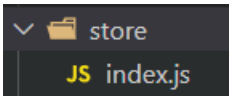
รูป 19 ภายในของโฟลเดอร์ router

```
import { createRouter, createWebHistory } from "vue-router";
import HomeView from "@/views/HomeView.vue";
import DriveView from "@/views/DriveView.vue";
import ImgAppView from "@/views/ImgAppView.vue";
import LoginView from "@/views/LoginView.vue";
import RegisView from "@/views/RegisView.vue";
import SettingView from "@/views/SettingView";
import ChangePassView from "@/views/ChangePassView";
import ImgFolderView from "@/views/ImgFolderView";
import MaketView from "@/views/MarketView";
import ProductView from "@/views/ProductView";
import AddProductView from "@/views/AddProductView";
import ProductHistoryView from "@/views/ProductHistoryView";
import HistoryView from "@/views/HistoryView";
import EditProductView from "@/views/EditProductView";
import JobHistoryView from "@/views/JobHistoryView";

const routes = [ ···
];

const router = createRouter({
  history: createWebHistory(process.env.BASE_URL),
  routes,
});

export default router;
```

รูป 20 ภายในไฟล์ index.js ของโฟลเดอร์ router

2.3.4 โฟลเดอร์ store

เป็นโฟลเดอร์สำหรับการใช้งานพื้นที่เก็บข้อมูลที่เป็น Local storage ใน Web browser เพื่อใช้ในการประมวลผลและแสดงผลในหน้า Web application โดยจะมีส่วนประกอบดังรูป 21 และ 22 ไฟล์ index.js



รูป 21 ภายในของโฟลเดอร์ store

```
import { createStore } from "vuex";
import createPersistedState from "vuex-persistedstate";

export default createStore({
  plugins: [createPersistedState()],
  state: {
    isLoading: false,
    first_name: "",
    last_name: "",
    jwt: "",
    isAuthenticated: false,
    storageSize : 0,
    creditTotal : 0,
    productUse : [],
  },
  getters: {},
  mutations: {
    setToken(state, token) {
      state.jwt = token;
      state.isAuthenticated = true;
      console.log('set token done')
    },

    setUserData(state, data) {
      state.first_name = data[0];
      state.last_name = data[1];
      state.storageSize = data[2];
      state.creditTotal = data[3];
    },

    setCredit(state, data) {
      state.creditTotal = data;
    },

    setDriveSize(state,data){
      state.storageSize = data
    },


    addProduct(state, product){
      state.productUse.push(product)
    },

    clearProduct(state){
      state.productUse = []
    },

    logout(state) {
      state.jwt = ''
      state.productUse = []
      state.isAuthenticated = false
  },
  },
  actions: {},
  modules: {},
  computed: {},
});
```

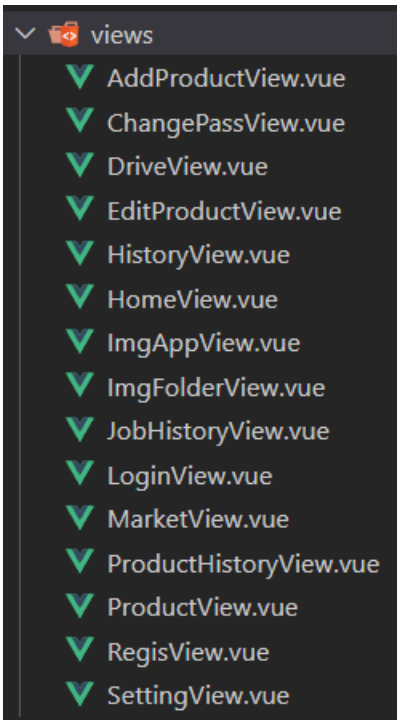
รูป 22 ภายในไฟล์ index.js ของโฟลเดอร์ store

2.3.5 โฟลเดอร์ views

เป็นโฟลเดอร์สำหรับเก็บหน้า GUI ต่างๆ ในรูปแบบของ View โดยจะถูกแยกออกเป็นแต่ละหน้าของ Web application โดยภายในแต่ละไฟล์ของ View จะประกอบไปด้วย

1) ส่วน HTML ที่เป็นส่วนแสดงผลหลักๆของแต่ละหน้า Web page
2) ส่วน JavaScript ที่เป็นตัวช่วยให้การแสดงผลนั้นสามารถมีความยืดหยุ่นและแสดงข้อมูลได้ออกมาตรงตามที่ได้ออกแบบเอาไว้ รวมไปถึงการส่ง-รับข้อมูลจาก API backend อีกด้วย

โดยจะมีส่วนประกอบดังรูป 23



รูป 23 ภายในของโฟลเดอร์ views

ซึ่งในแต่ละไฟล์ก็จะมีหน้าของ Web application ที่แสดงผลต่างกันออกไปดังนี้

1) AddProductView - หน้าสำหรับการเพิ่มสินค้าเข้าสู่ Marketplace
2) ChangePassView - หน้าสำหรับการเปลี่ยน Password ของผู้ใช้
3) DriveView – หน้าสำหรับการจัดการโฟลเดอร์เก็บรูปภาพของผู้ใช้
4) EditProductView - หน้าสำหรับแก้ไขข้อมูลสสินค้า
5) HistoryView – หน้าสำหรับดูประวัติการได้รับและจ่ายเครดิตของผู้ใช้
6) HomeView - หน้าสำหรับฟีดข่าวหลัก

7) ImgAppView – หน้าสำหรับการใช้งาน Image processing application
8) ImgFolderView - หน้าสำหรับการใช้งานภายในโฟลเดอร์เก็บรูปภาพ
9) JobHistoryView - หน้าสำหรับดูประวัติการสั่งงานประมวลผลของผู้ใช้
10) LoginView - หน้าสำหรับการลงทะเบียนเข้าใช้งาน
11) MarketView - หน้าสำหรับการใช้งานตลาดซื้อขาย
12) ProductHistoryView - หน้าสำหรับการดูประวัติการใช้งานสินค้า
13) ProductView – หน้าสำหรับการดูข้อมูลสินค้าและเปิดใช้งานสินค้า
14) RegisView - หน้าสำหรับการลงทะเบียนผู้ใช้ใหม่
15) SettingView – หน้าสำหรับการจัดการข้อมูลผู้ใช้

2.3.6 ไฟล์ app.vue

ไฟล์สำหรับการตั้งค่าเบื้องต้นได้และสามารถใส่ Function บางอย่างที่มีการกระทำเรียกใช้บ่อยๆได้ โดยจะมีส่วนประกอบดังรูป 24

```
<template>
  <div>
    <router-view></router-view>
  </div>
</template>

<script>

export default {
  name: "app",

  data() {
    return {

    }
  },
  methods: {

  },
  created() {
    //console.log("created is work");

  },

}

</script>
```

รูป 24 ภายในของไฟล์ app.vue

2.3.7 ไฟล์ main.js

ไฟล์สำหรับการเรียกใช้งาน Project สำหรับการเปิดใช้งานหน้า Web application และสามารถตั้งค่าบางอย่างได้เช่น การตั้งค่า URL ของ API ที่ต้องการเชื่อมต่อเป็นต้น โดยจะมีส่วนประกอบดังรูป 25

```
1   import { createApp } from 'vue'
2   import App from './App.vue'
3   import router from './router'
4   import store from './store'
5   import "./assets/style.css"
6   import Axios from 'axios';
7   //Axios.defaults.baseURL = 'http://127.0.0.1:8000/api/';
8   Axios.defaults.baseURL = 'http://192.168.1.46:8000/api/';
9
10
11  createApp(App).use(store).use(router).mount('#app')
```

รูป 25 ภายในของไฟล์ main.js

2.4 ไฟล์ Dockerfile

ใช้สำหรับการสร้างหรือ build Docker image เพื่อใช้สำหรับการ Deploy ส่วน Frontend ในเครื่อง Server จริงโดยรูปภาพ 12 คือตัวอย่างภายในไฟล์ โดยจะมีส่วนประกอบดังรูป 26

```
1   FROM --platform=$BUILDPLATFORM node:16.16.0-alpine AS frontend
2
3   RUN mkdir /frontend
4   WORKDIR /frontend
5
6   COPY . .
7
8   RUN yarn global add @vue/cli
9   RUN yarn install
10  ENV HOST=0.0.0.0
11
12  ENTRYPOINT ["yarn", "run", "serve"]
```

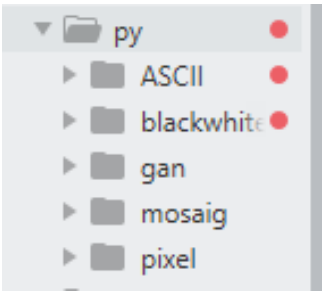
รูป 26 ภายในของไฟล์ main.js

2.5 ไฟล์ req.txt

เป็นไฟล์สำหรับจัดการ package ต่าง ๆ ที่ใช้ในโปรเจค

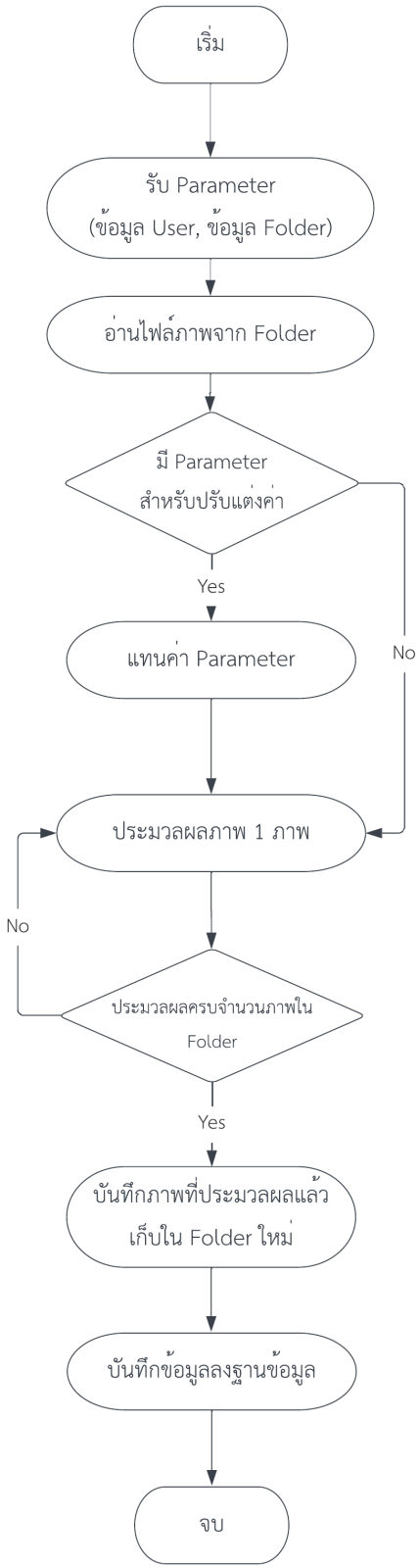
3. ส่วนของ Application ประมวลผลภาพ

ภายในโฟลเดอร์แอพพลิเคชั่นพัฒนาด้วย Python ประกอบด้วยโฟลเดอร์ และ ไฟล์ทั้งหมดตามโครงสร้างดังรูป 27 สามารถ clone ได้จาก https://github.com/SuteeSaraphan/IPAuTSoNS



รูป 27 ภายในของโฟลเดอร์ Application

โดยมี Flowchart อธิบายการทำงานเบื้องต้นของ Application ประมวลผลภาพทั้งหมดที่ได้จัดทำขึ้น ASCII, BlackWhite, Mosaic, Pixelart, Yolo และ GAN โดยมีพื้นฐาน และ ขั้นตอนการทำงานตาม Flowchart ที่ได้กล่าวมาข้างต้นในทุก ๆ Application ดังรูป 28



รูป 28 Flowchart การทำงานของแอพพลิเคชั่นประมวลผลภาพเบื้องต้นทั้งหมดในระบบ

3.1 โฟลเดอร์ ASCII

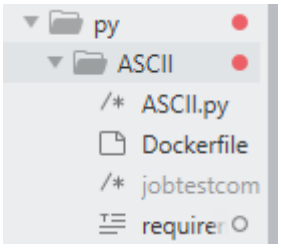
a. ascii.py สำหรับเป็น Script ในการประมวลผลภาพ

b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

c. jobtestcom1.yaml เป็นคำสั่งสำหรับสั่งงานประมวลผลในรูปแบบ Job ของ Kubernetes เพื่อทดสอบ Python Script ตัวนี้บนรูปแบบ Docker Image

d. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile

โดยการทำงานหลักของโปรแกรมจะต้องใส่ Parameter เป็น JobID, UserID, Folder(ภาพที่ต้องการประมวผล), NewFolder(ตำแหน่งใหม่ที่ต้องการเก็บภาพ) โดยตัว Folder ข้างในของ ASCII จะแสดงดังรูป 29



รูป 29 ภายในของโฟลเดอร์ ASCII

3.2 โฟลเดอร์ blackwhite

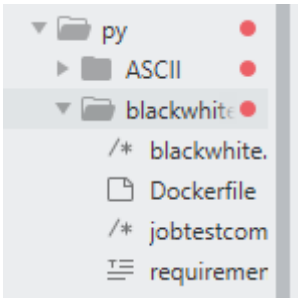
a. blackwhite.py สำหรับเป็น Script ในการประมวลผลภาพ

b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

c. jobtestcom1.yaml เป็นคำสั่งสำหรับสั่งงานประมวลผลในรูปแบบ Job ของ Kubernetes เพื่อทดสอบ Python Script ตัวนี้บนรูปแบบ Docker Image

d. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile

โดยการทำงานหลักของโปรแกรมจะต้องใส่ Parameter เป็น JobID, UserID, Folder(ภาพที่ต้องการประมวผล), NewFolder(ตำแหน่งใหม่ที่ต้องการเก็บภาพ) โดยตัว Folder ข้างในของ blackwhite จะแสดงดังรูป 30



รูป 30 ภายในของโฟลเดอร์ BlackWhite

3.3 โฟลเดอร์ mosaig

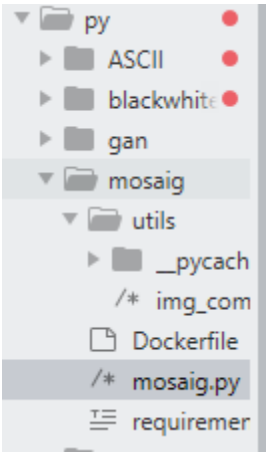
a. mosaig.py สำหรับเป็น Script ในการประมวลผลภาพ

b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

c. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile

d. ในโฟลเดอร์ utils ประกอบด้วยไฟล์ img_common_util.py สำหรับเป็น Function ในการ convert_image_to_tensor, convert_tensor_to_image เพื่อทำงานประมวลใน mosaig.py

โดยการทำงานหลักของโปรแกรมจะต้องใส่ Parameter เป็น JobID, UserID, Folder(ภาพที่ต้องการประมวผล), SelectImage(ภาพที่ต้องการเลือกให้เป็นภาพหลักในการทำ Mosaig), NewFolder(ตำแหน่งใหม่ที่ต้องการเก็บภาพ) โดยตัว Folder ข้างในของ mosaig จะแสดงดังรูป 31



รูป 31 ภายในของโฟลเดอร์ Mosaic

3.4 โฟลเดอร์ pixel

a. pixel.py สำหรับเป็น Script ในการประมวลผลภาพ

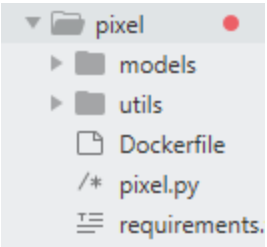
b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

c. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile

d. ในโฟลเดอร์ utils ประกอบด้วยไฟล์ img_common_util.py สำหรับเป็น Function ในการ convert_image_to_tensor, convert_tensor_to_image เพื่อทำงานประมวลใน mosaig.py

e. ในโฟลเดอร์ models จะประกอบไปด้วย Function ในการตรวจจับ Edge ของภาพ, การแปลงภาพเป็น pixel และ แต่งเอฟเฟ็คของ pixel ภาพ โดยมี Function ดังนี้ edge_detector, photo2pixel และ pixel_effect

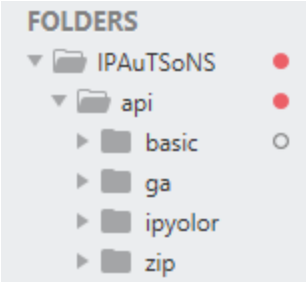
โดยการทำงานหลักของโปรแกรมจะต้องใส่ Parameter เป็น JobID, UserID, Folder(ภาพที่ต้องการประมวผล), ค่า Pixel ที่ต้องการใช้ในการประมวลผลภาพ,NewFolder(ตำแหน่งใหม่ที่ต้องการเก็บภาพ) โดยตัว Folder ข้างในของ Pixelart จะแสดงดังรูป 32



รูป 32 ภายในของโฟลเดอร์ Pixel

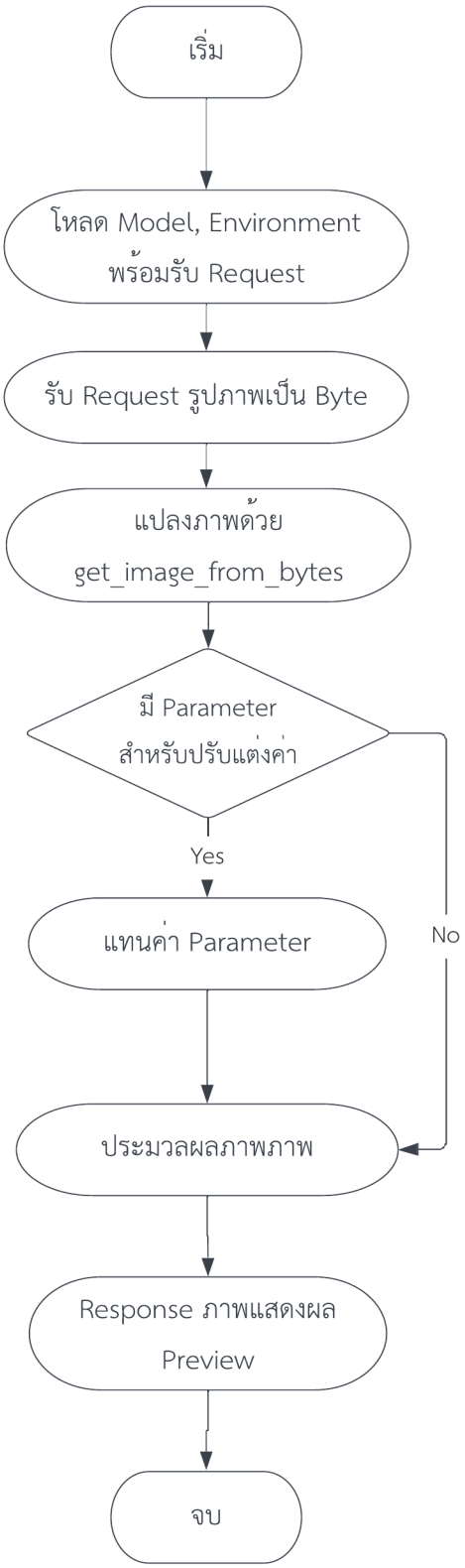
4. ส่วนของ API

ภายในโฟลเดอร์ API พัฒนาด้วย Python ประกอบด้วยโฟลเดอร์ และ ไฟล์ทั้งหมดตามโครงสร้างดังรูป 33 สามารถ clone ได้จาก https://github.com/SuteeSaraphan/IPAuTSoNS



รูป 33 ภายในของโฟลเดอร์ API

โดยมี Flowchart อธิบายการทำงานเบื้องต้นของ API Application ประมวลผลภาพแสดงผลตัวอย่างทั้งหมดที่ได้จัดทำขึ้น ASCII, BlackWhite, Mosaic, Pixelart, Yolo และ GAN โดยมีพื้นฐาน และ ขั้นตอนการทำงานตาม Flowchart ที่ได้กล่าวมาข้างต้นในทุก ๆ API Application ดังรูป 34



รูป 34 Flowchart การทำงานของ API แอพพลิเคชั่นประมวลผลภาพทั้งหมดในระบบ

4.1 โฟลเดอร์ basic

a. main.py สำหรับเป็น Script API ในการกำหนด Routing ต่าง ๆ เพื่อเข้าไปใช้งานประมวลผลในแต่ละงาน ASCII, BlackWhite, Pixelart และ Mosaic โดยจะมี Swagger UI สำหรับให้ Developer สามารถเข้ามาอ่าน Documents เพื่อช่วยในการพัฒนาต่อ และ สามารถดูการกำหนด Parameter ต่าง ๆ ของตัว API ได้

b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

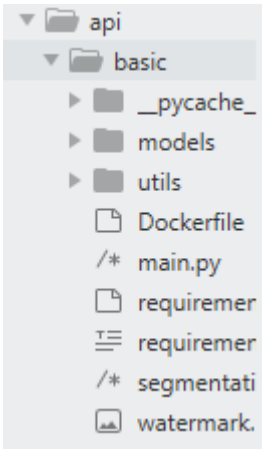
c. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile

d. models สำหรับจัดเก็บ models ฟังก์ชั่นต่าง ๆ ของ Pixelart และ Mosaic ในระบบ API

e. utils สำหรับเก็บฟังก์ชั่นของ Pixelart

f. watermark.png สำหรับการทำลายน้ำในส่วนของการส่งไฟล์ภาพเป็นรูปแบบ Preview

g. segmentation.py สำหรับการประมวลผลภาพจาก Byte ให้ภาพขนาดเล็กลง และ นำไปประมวลผลต่อได้ในส่วนของงานประมวลผลอื่น ๆ ที่มีในระบบ



รูป 35 ภายในของโฟลเดอร์ API Basic

4.2 โฟลเดอร์ gan

a. main.py สำหรับเป็น Script API ในการกำหนด Routing ต่าง ๆ เพื่อเข้าไปใช้งานประมวลผลใน GAN Model โดยจะมี Swagger UI สำหรับให้ Developer สามารถเข้ามาอ่าน Documents เพื่อช่วยในการพัฒนาต่อ และ สามารถดูการกำหนด Parameter ต่าง ๆ ของตัว API ได้

b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

c. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile

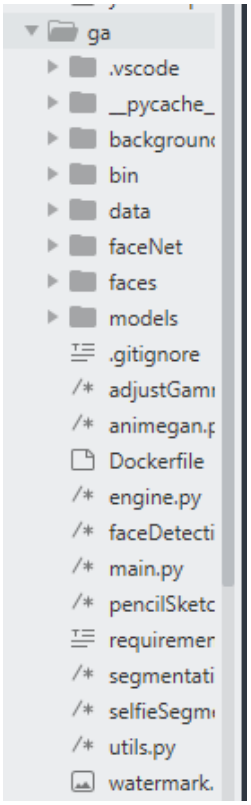
d. models สำหรับจัดเก็บ models ฟังก์ชั่นต่าง ๆ ของ Pixelart และ Mosaic ในระบบ API

e. utils สำหรับเก็บฟังก์ชั่นของ Pixelart

f. watermark.png สำหรับการทำลายน้ำในส่วนของการส่งไฟล์ภาพเป็นรูปแบบ Preview

g. segmentation.py สำหรับการประมวลผลภาพจาก Byte ให้ภาพขนาดเล็กลง และ นำไปประมวลผลต่อได้ในส่วนของงานประมวลผลอื่น ๆ ที่มีในระบบ

h. ไฟล์อื่น ๆ ที่ใช้ในการจับโมเดล หน้าคน หรือ ลบพื้นหลังของวัตถุต่าง ๆ ในการใช้งานในการประมวลผลด้วย GAN Model



รูป 36 ภายในของโฟลเดอร์ API Gan

4.3 โฟลเดอร์ ipyolor

a. main.py สำหรับเป็น Script API ในการกำหนด Routing ต่าง ๆ เพื่อเข้าไปใช้งานประมวลผลใน Yolov5 Model โดยจะมี Swagger UI สำหรับให้ Developer สามารถเข้ามาอ่าน Documents เพื่อช่วยในการพัฒนาต่อ และ สามารถดูการกำหนด Parameter ต่าง ๆ ของตัว API ได้

b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

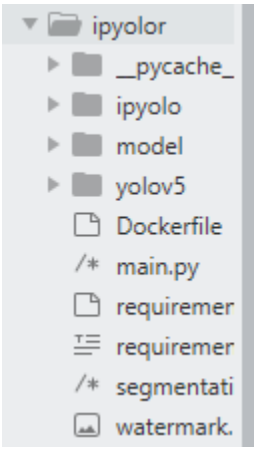
c. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile

d. models สำหรับจัดเก็บ models ต่าง ๆ ของ Yolov5 เพื่อใช้เป็นตัว Default ของ Model

f. watermark.png สำหรับการทำลายน้ำในส่วนของการส่งไฟล์ภาพเป็นรูปแบบ Preview

g. segmentation.py สำหรับการประมวลผลภาพจาก Byte ให้ภาพขนาดเล็กลง และ นำไปประมวลผลต่อได้ในส่วนของงานประมวลผลอื่น ๆ ที่มีในระบบ

h. ไฟล์อื่น ๆ ที่ใช้ในการจับโมเดล หน้าคน หรือ ลบพื้นหลังของวัตถุต่าง ๆ ในการใช้งานในการประมวลผลด้วย Yolov5 Model



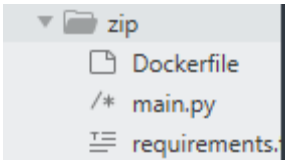
รูป 37 ภายในของโฟลเดอร์ Yolov5 Model

4.4 โฟลเดอร์ zip

a. main.py สำหรับเป็น Script API ในการกำหนด Routing ต่าง ๆ เพื่อเข้าไปใช้งานการบีบอัดไฟล์ทั้งหมดเป็นรูปแบบ Zip โดยจะมี Swagger UI สำหรับให้ Developer สามารถเข้ามาอ่าน Documents เพื่อช่วยในการพัฒนาต่อ และ สามารถดูการกำหนด Parameter ต่าง ๆ ของตัว API ได้

b. Dockerfile ในการสร้าง Python Script ให้อยู่ในรูปแบบของ Docker Image

c. requirement.txt เป็นไฟล์ในการเก็บ Requirement ที่จำเป็นของ Python Script นี้เพื่อนำไปใช้งานกับ Dockerfile



รูป 38 ภายในของโฟลเดอร์ Zip

5. ส่วนของ Yaml ในการทำงาน Kubernetes

5.1 backend

a. backend.yaml สำหรับเป็น Yaml ในการสั่งงาน Deployment ของ backend Web Application บน Kubernetes

b. backend-service.yaml สำหรับเป็น Yaml ในการจัดการ Expose Port ของตัว backend Web Application

5.2 fronend

a. fronend.yaml สำหรับเป็น Yaml ในการสั่งงาน Deployment ของ fronend Web Application บน Kubernetes

b. fronend -service.yaml สำหรับเป็น Yaml ในการจัดการ Expose Port ของตัว fronend Web Application

5.3 baisc API

a. basic.yaml สำหรับเป็น Yaml ในการสั่งงาน Deployment ของ basic API บน Kubernetes

b. basic -service.yaml สำหรับเป็น Yaml ในการจัดการ Expose Port ของตัว basic API

5.3 Yolo API

a. yoloapi.yaml สำหรับเป็น Yaml ในการสั่งงาน Deployment ของ Yolo Model API บน Kubernetes

b. yoloapi-service.yaml สำหรับเป็น Yaml ในการจัดการ Expose Port ของตัว basic API

5.4 GAN API

a. ganapi.yaml สำหรับเป็น Yaml ในการสั่งงาน Deployment ของ GAN Model API บน Kubernetes

b. ganapi-service.yaml สำหรับเป็น Yaml ในการจัดการ Expose Port ของตัว basic API

5.5 Zip API

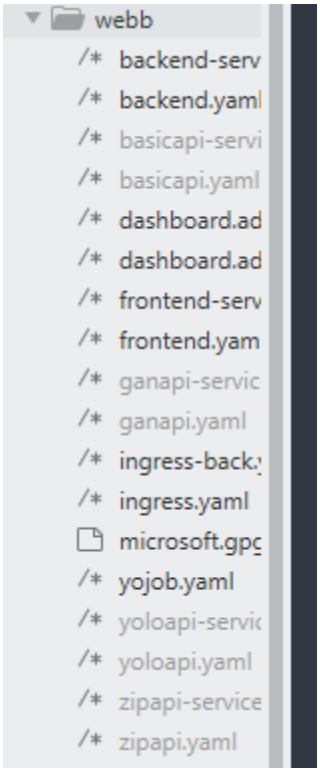
a. zipapi.yaml สำหรับเป็น Yaml ในการสั่งงาน Deployment ของ Zip API บน Kubernetes

b. zipapi-service.yaml สำหรับเป็น Yaml ในการจัดการ Expose Port ของตัว Zip API

5.6 dashboard

a. dashboard.admin-user-role.yml สำหรับจัดการ Role ในการสั่งงาน Type Job ใน Kubernetes เพื่อความปลอดภัยของระบบ

b. dashboard.admin-user.yml สำหรับสร้าง User ขึ้นมาเพื่อจัดการกับ Role



รูป 39 ภายในของโฟลเดอร์ Zip

6. ส่วนของการโหลด Model และ Weight

การโหลด Model และ Weight ของ Yolov5 นั้นจะต้องทำการ Clone repo ของตัว YoloV5 Model เข้ามาก่อนจาก https://github.com/ultralytics/yolov5 เมื่อทำการโหลดเสร็จเรียบร้อยแล้วทำการตั้ง path ที่จะต้องการใช้งาน และ Import lib ที่ชื่อ torch สำหรับในการโหลด Model เข้ามาในรูปแบบ pre-load โดยใช้คำสั่ง torch.hub.load() เพื่อเป็นการสั่งให้โหลดตัว Model และ Weight เข้ามาพร้อม ๆ กัน ดังรูป 40

```python
def get_yolov5(models):
    # local best.pt
    model = torch.hub.load('./yolov5', 'custom', path='./model/'+models, source='local')  # local repo
    model.conf = 0.3
    return model
```

รูป 40 ฟังก์ชั่นในการโหลด Model และ Weight ด้วย Tourch

เมื่อทำการสร้างฟังก์ชั่นในการโหลด Model และ Weight แล้วสามารถเรียกใช้งานโดยการนำเข้าตัว path ของ Weight เข้ามาในคำสั่ง get_yolov5() เพื่อให้ทำการ pre-load ตัว model และ weight ในฟังก์ชั่นที่ได้สร้างขึ้นดังรูป 41

```python
@app.post("/detect-to-img")
async def detect_garbage_return_base64_img(folders = "",modelse = "yolov5n.pt",job_id = ""):
    folder = folders+"/*"

    All_files = glob(folder+'.png') + glob(folder+'.jpg') + glob(folder+'.jpeg') + glob(folder+'.tiff')

    model = get_yolov5(modelse)
```

รูป 41 การเรียกใช้ฟังก์ชั่นในการโหลด Model และ Weight

7. ส่วนของการตั้งค่า Load Balancer และ ตรวจสอบทรัพยากรที่ใช้

7.1 ส่วนของการตั้งค่า Load Balancer

จำเป็นต้องสร้าง Service ของ Kubernetes ในส่วนของแต่ละ Application ที่ทำงานอยู่บนระบบ Cluster ขึ้นมา และ ใส่คำสั่งในการตั้งค่าดังรูป 42 และ คำอธิบายคำสั่งเบื้องต้น

a.) type: ClusterIP กำหนดให้ใช้ IP ภายใน Cluster เพื่อใช้งาน Load ภายใน Cluster

b.) sessionAffinity: Client IP กำหนดให้ใช้ IP ของ Client ในการเก็บบันทึกการเข้าใช้งานเพื่อให้เข้าใช้งานเครื่องเดิมจากที่เคยเข้ามาอยู่แล้ว

c.) externalTrafficPolicy: Local กำหนดให้ใช้งานภายใน Local Network เท่านั้นสำหรับการเข้ามาใช้งานจากภายนอกให้ Load Balance ระหว่างระบบ Local

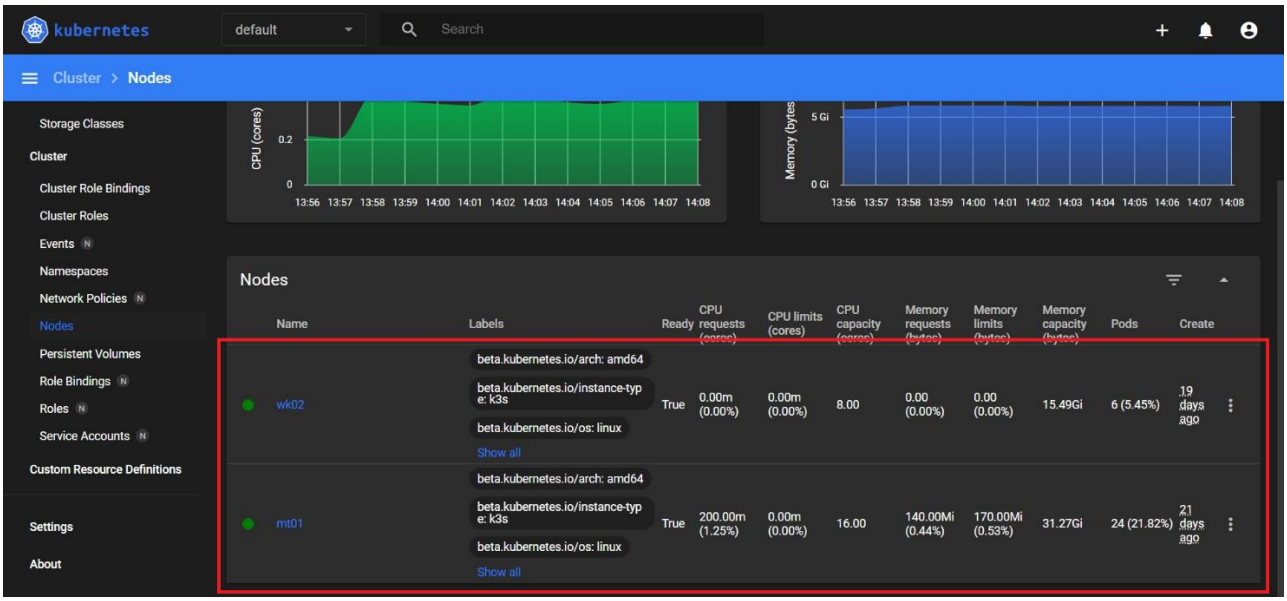
d.) LoadBalancerAlgorithm: least_conn เพื่อกำหนดว่าในการ Load Balance จะใช้ Algorithm ใดในการ Load สามารถเปลี่ยนเป็น round-robin หรือ ทำการลบบรรทัดนี้ออกเนื่องจาก round-robin เป็น Default Algorithm ในการ Load Balancer ของ Kubernetes Service

```
spec:
  selector:
    app: my-app
  type: ClusterIP
  sessionAffinity: ClientIP
  externalTrafficPolicy: Local
  loadBalancerAlgorithm: least_conn
  ports:
    - name: http
      port: 80
      targetPort: 8080
```

รูป 42 คำสั่งในการตั้งค่า Load Balancer

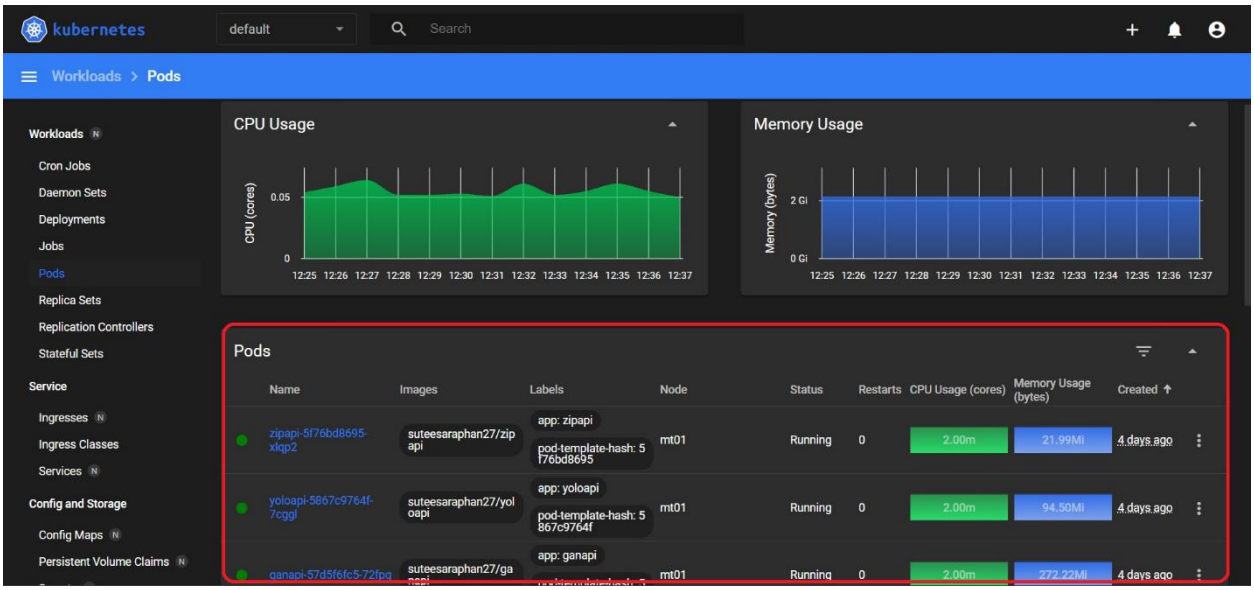
7.2 ส่วนของการตรวจสอบทรัพยากรที่ใช้

สามารถเข้ามาตรวจสอบทรัพยากรที่ใช้งานใน Kubernetes ในแต่ละ Nodes ได้ด้วยการเข้าใช้งานบน Dashboard ของ Kubernetes และ ตรวจสอบการทำงานของ Node ใช้งานทรัพยากรอยู่เท่าไหร่ และมีการทำงานของ Pods อยู่กี่ Pods ในแต่ละ Nodes สามารถใช้คำสั่ง kubectl top nodes เพื่อตรวจเช็คใน Command-Line ได้เช่นกัน ดังรูป 43



รูป 43 Dashboard ตรวจสอบทรัพยากรที่ใช้ภายใน Nodes

สามารถเข้ามาตรวจสอบทรัพยากรที่ใช้งานใน Kubernetes ในแต่ละ Pods ที่ทำงานอยู่ได้ด้วยการเข้าใช้งานบน Dashboard ของ Kubernetes และ ตรวจสอบการทำงานของ Pods ใช้งานทรัพยากรอยู่เท่าไหร่ มีการทำงานของ Pods อยู่บน Nodes เครื่องไหน สามารถใช้คำสั่ง kubectl top pods เพื่อตรวจเช็คใน Command-Line ได้เช่นกัน ดังรูป 44



รูป 44 Dashboard ตรวจสอบทรัพยากรที่ใช้ภายใน Pods