项目后端技术文档

1. 介绍

欢迎来到树上鸟儿成双队项目后端的介绍。

本项目旨在构建一个现代化、可扩展的Web温度控制和管理应用后端，提供稳定、高效的服务。

后端负责实现用户的登录和注册、处理用户请求、管理传感器数据存储与检索，并通过API向前端提供必要的信息。

1. 技术栈

编程语言： Python 3.8

框架： Django

数据库： PostgreSQL

ORM： SQLAlchemy

1. 安装和部署

3.1 安装依赖

pip install -r requirements.txt

* 1. 配置nginx

#user nobody;

worker\_processes 1;

#error\_log logs/error.log;

#error\_log logs/error.log notice;

#error\_log logs/error.log info;

#pid logs/nginx.pid;

events {

worker\_connections 1024;

}

http {

include mime.types;

default\_type application/octet-stream;

#log\_format main '$remote\_addr - $remote\_user [$time\_local] "$request" '

# '$status $body\_bytes\_sent "$http\_referer" '

# '"$http\_user\_agent" "$http\_x\_forwarded\_for"';

#access\_log logs/access.log main;

sendfile on;

#tcp\_nopush on;

#keepalive\_timeout 0;

keepalive\_timeout 65;

#gzip on;

server\_names\_hash\_bucket\_size 128;

server {

listen 80;

server\_name localhost;

location / {

root html;

index index.html index.htm;

}

error\_page 500 502 503 504 /50x.html;

location = /50x.html {

root html;

}

}

server {

listen 5555;

server\_name 127.0.0.1;

# 这里是你要代理的测试环境域名加上-local

# 比如你的项目测试环境为a.test.com,你本地此处可以设置为a-local.test.com,当然你可以随便设置

#这里说明了如果是http://test-local.test.com:8888/则说明跨域，用以下http://127.0.0.1:666/请求

location /{

add\_header 'Access-Control-Allow-Origin' '\*';

add\_header 'Access-Control-Allow-Credentials' 'true';

add\_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';

add\_header 'Access-Control-Allow-Headers' 'DNT,web-token,app-token,Authorization,Accept,Origin,Keep-Alive,User-Agent,X-Mx-ReqToken,X-Data-Type,X-Auth-Token,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range';

add\_header 'Access-Control-Expose-Headers' 'Content-Length,Content-Range';

#proxy\_pass 你要跨域的的接口地址

proxy\_pass http://127.0.0.1:5501/;

}

#这里说明了如果是http://test-local.test.com:8888/api/则说明跨域，用以下http://127.0.0.1:666/请求

location ^~/aaa/{

add\_header 'Access-Control-Allow-Origin' '\*';

add\_header 'Access-Control-Allow-Credentials' 'true';

add\_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';

add\_header 'Access-Control-Allow-Headers' 'DNT,web-token,app-token,Authorization,Accept,Origin,Keep-Alive,User-Agent,X-Mx-ReqToken,X-Data-Type,X-Auth-Token,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range';

add\_header 'Access-Control-Expose-Headers' 'Content-Length,Content-Range';

rewrite ^/aaa(.\*)$ /$1 break;

#proxy\_pass 你要跨域的的接口地址

proxy\_pass http://127.0.0.1:8000/;

}

location ^~/bbb/{

add\_header 'Access-Control-Allow-Origin' '\*';

add\_header 'Access-Control-Allow-Credentials' 'true';

add\_header 'Access-Control-Allow-Methods' 'GET, POST, OPTIONS';

add\_header 'Access-Control-Allow-Headers' 'DNT,web-token,app-token,Authorization,Accept,Origin,Keep-Alive,User-Agent,X-Mx-ReqToken,X-Data-Type,X-Auth-Token,X-Requested-With,If-Modified-Since,Cache-Control,Content-Type,Range';

add\_header 'Access-Control-Expose-Headers' 'Content-Length,Content-Range';

rewrite ^/bbb(.\*)$ /$1 break;

#proxy\_pass 你要跨域的的接口地址

proxy\_pass http://127.0.0.1:5000/;

}

}

}

* 1. 启动应用

Python manage.py

1. 项目结构

/main

├─.idea

│ └─inspectionProfiles

├─backend

│ └─\_\_pycache\_\_

├─DataProcess

│ ├─migrations

│ │ └─\_\_pycache\_\_

│ └─\_\_pycache\_\_

├─langchain\_model

│ ├─migrations

│ │ └─\_\_pycache\_\_

│ └─\_\_pycache\_\_

├─Login

│ ├─migrations

│ │ └─\_\_pycache\_\_

│ └─\_\_pycache\_\_

├─Product

│ ├─migrations

│ │ └─\_\_pycache\_\_

│ └─\_\_pycache\_\_

└─utils

└─\_\_pycache\_\_

1. 功能介绍

5.1 用户注册与登录

5.2 数据接受与推送

5.3 API服务

5.4 数据持久化

1. API文档

6.1 用户管理

userlist/

查看已有的用户列表，主要用于网站管理者使用

login/

实现登录功能，实现密码检查，用户名检查等内容

register/

实现注册功能，实现新用户的注册

* 1. 模型使用

answer/

获取请求中的问题, 使用chatBloom模型预测问题, 最后返回预测结果

* 1. 项目管理

product/

查看项目后台设备运行情况，主要用于网站管理者使用

category/

查看项目后台已注册设备目录情况，主要用于网站管理者使用

* 1. 数据处理

sendData/

将数据库中的光照、温度、湿度等数据发送给前端进行调用

1. 数据库设计
   1. 数据接收

首先，利用apschedule库每一个小时向网关发送请求，将申请到的数据存放到设置好的数据库中。

from apscheduler.schedulers.blocking import BlockingScheduler

import requests

# 定义向网关发送请求的函数

def send\_request\_to\_gateway():

#这里省略

# 创建调度器实例

scheduler = BlockingScheduler()

# 添加定时任务，每小时执行一次 send\_request\_to\_gateway 函数

scheduler.add\_job(send\_request\_to\_gateway, 'interval', hours=1)

# 启动调度器

try:

print("Scheduler started. Press Ctrl+C to exit.")

scheduler.start()

except KeyboardInterrupt:

print("Scheduler stopped by user.")

* 1. 数据库介绍

数据库中一共有三个表，分别是用户表、传感器表和状态数据表。

（1）对于用户数据表：

from django.db import models

# 定义User模型，用于存储用户信息

class User(models.Model):

# 定义用户ID字段，最大长度为100，唯一，verbose\_name为'用户ID'

user\_id = models.CharField(max\_length=100, unique=True, verbose\_name="用户ID")

# 定义年龄字段，verbose\_name为'年龄'

age = models.IntegerField(verbose\_name="年龄")

# 定义性别字段，最大长度为10，verbose\_name为'性别'

gender = models.CharField(max\_length=10, verbose\_name="性别")

# 定义登录账号字段，最大长度为100，唯一，verbose\_name为'登录账号'

username = models.CharField(max\_length=100, unique=True, verbose\_name="登录账号")

# 定义密码字段，最大长度为100，verbose\_name为'密码'

password = models.CharField(max\_length=100, verbose\_name="密码")

# 定义拥有的传感器模型外键关联，on\_delete为models.CASCADE，verbose\_name为'拥有的传感器模型'

sensors = models.ManyToManyField("Sensor", verbose\_name="拥有的传感器模型")

# 定义\_\_str\_\_方法，返回username

def \_\_str\_\_(self):

return self.username

这是一个用于存储用户信息的Django模型。简要解释一下每个字段的含义：

• user\_id: 用户的唯一标识符。

• age: 用户的年龄。

• gender: 用户的性别。

• username: 用户的登录账号，唯一标识用户。

• password: 用户的密码，需要注意在实际应用中应使用安全的密码哈希存储方式。

• sensors: 一个多对多关系字段，表示用户拥有的传感器模型。这里使用了ManyToManyField，表示一个用户可以拥有多个传感器。

在这个模型中，User 和 Sensor 之间存在外键关系，通过 sensors 字段建立了它们之间的关联。这样，一个用户可以与多个传感器相关联，而一个传感器也可以被多个用户拥有。

（2）对于传感器数据表：

# 定义Sensor模型，用于存储传感器信息

class Sensor(models.Model):

# 定义传感器ID字段，最大长度为100，唯一，verbose\_name为'传感器ID'

sensor\_id = models.CharField(max\_length=100, unique=True, verbose\_name="传感器ID")

# 定义传感器类型字段，最大长度为100，verbose\_name为'传感器名称'

type = models.CharField(max\_length=100, verbose\_name="传感器名称")

# 定义用户字段，最大长度为100，verbose\_name为'用户'，默认值为'none'

user = models.CharField(max\_length=100, verbose\_name="用户", default="none")

# 定义\_\_str\_\_方法，返回sensor\_id

def \_\_str\_\_(self):

return self.sensor\_id

这个传感器数据库模型包含以下字段：

• sensor\_id: 传感器的唯一标识符。

• type: 传感器的类型或名称。

• user: 与传感器相关联的用户，这是一个字符字段，表示用户的登录账号。在数据库设计中，我们使用了一个字符字段而不是外键来关联用户，这样只是适用于这种情况，不是很规范。

（3）对于状态数据表：

# 定义StatusInfo模型，用于存储传感器状态信息

class StatusInfo(models.Model):

# 定义id字段，自增，主键

id = models.AutoField(primary\_key=True)

# 定义时间字段，自动添加，verbose\_name为'时间'

time = models.DateTimeField(auto\_now\_add=True, verbose\_name="时间")

# 定义传感器外键，on\_delete为models.CASCADE，verbose\_name为'传感器'

sensor = models.ForeignKey(Sensor, on\_delete=models.CASCADE, verbose\_name="传感器")

# 定义属性字段，最大长度为100，verbose\_name为'属性'

attribute = models.CharField(max\_length=100, verbose\_name="属性")

# 定义值字段，最大长度为100，verbose\_name为'值'

value = models.CharField(max\_length=100, verbose\_name="值")

# 定义\_\_str\_\_方法，返回sensor.sensor\_id-attribute-value

def \_\_str\_\_(self):

return f"{self.sensor.sensor\_id}-{self.attribute}-{self.value}"

这个状态数据库模型包含以下字段：

• id: 传感器状态信息的唯一标识符，采用自增主键。

• time: 记录状态信息的时间戳，采用 DateTimeField 字段，并在每次添加数据时自动设置为当前时间。

• sensor: 外键字段，关联到传感器数据库中的 Sensor 模型。这表示每个状态信息都与一个特定的传感器关联。

• attribute: 属性字段，表示状态信息的属性。

• value: 值字段，表示状态信息的具体数值。

我们还在 StatusInfo 模型中定义了 str 方法，用于返回这个模型的字符串表示。

* 1. 数据库操作

1. 插入数据
2. 查询数据
3. 更新数据
4. 删除数据
   1. 与前端的数据交互

后端收到前端指定的传感器的序列号，查询对应数据库信息，然后将状态信息发送给前端。

from django.shortcuts import render

from django.http import JsonResponse

from .models import StatusInfo, Sensor

def get\_sensor\_status(request, sensor\_id):

try:

# 查询传感器状态信息

sensor = Sensor.objects.get(sensor\_id=sensor\_id)

status\_info = StatusInfo.objects.filter(sensor=sensor).latest('time')

# 构建响应数据

response\_data = {

'sensor\_id': sensor.sensor\_id,

'attribute': status\_info.attribute,

'value': status\_info.value,

'time': status\_info.time.strftime('%Y-%m-%d %H:%M:%S')

}

# 返回 JSON 格式的响应

return JsonResponse(response\_data)

except Sensor.DoesNotExist:

return JsonResponse({'error': f'Sensor with ID {sensor\_id} does not exist'}, status=404)

except StatusInfo.DoesNotExist:

return JsonResponse({'error': f'No status information found for sensor with ID {sensor\_id}'}, status=404)

1. 代码示例

注册函数

@api\_view(['POST'])

def Register(request, format=None):

print(request.data)

username = request.data['username']

password = str(request.data["password"])

flag = Userinfo.objects.filter(username=username)

if not flag.exists():

instance = Userinfo.objects.create(username=username, password=password)

result = default\_response()

result['data']['reseaion'] = "创建成功"

result['data']['pk'] = instance.id

# 需要传什么参数这块可以给

return Response(result)

else:

result = default\_response()

result['data'] = "用户名已存在"

return Response(result, status=status.HTTP\_400\_BAD\_REQUEST)

登录函数

@api\_view(['POST'])

def login\_view(request, format=None):

print(request.data)

username = request.data['username']

password = str(request.data["password"])

if not Userinfo.objects.filter(username=username).exists():

result = default\_response()

result['data'] = "用户名不存在"

return Response(result, status=status.HTTP\_400\_BAD\_REQUEST)

user\_info = Userinfo.objects.filter(username=username)[0]

if user\_info.password != password:

result = default\_response()

result['data'] = "密码错误"

return Response(result, status=status.HTTP\_400\_BAD\_REQUEST)

result = default\_response()

result['data']['reseaion'] = "登录成功"

return Response(result)

返回数据

def Return\_Data(request):

# 创建一个空列表，用于存储数据

Info = []

# 遍历IDList列表

for item in IDList:

# 创建一个空列表，用于存储每一项数据

item\_List = []

# 调用Get\_RoomID函数，获取Roomid\_dict字典

Roomid\_dict = Get\_RoomID(item, TemperatureInformation)

# 调用Get\_serial函数，获取serial\_dict字典

serial\_dict = Get\_serial(item, TemperatureInformation)

# 调用merge\_Properties函数，获取properties\_dict字典

properties\_dict = merge\_Properties(item, TemperatureInformation)

# 将Roomid\_dict、serial\_dict、properties\_dict合并

merged\_dict = {\*\*Roomid\_dict, \*\*serial\_dict, \*\*properties\_dict}

# 将合并后的字典添加到item\_List列表中

item\_List.append(merged\_dict)

# 将item\_List列表添加到Info列表中

Info.append(item\_List)

# 后期删除这一行

Info = makedata(Info)

# 返回一个JsonResponse对象，其中包含Info列表

return JsonResponse(Info,safe=False)

读取温度信息

def Get\_Temperature(item, TemperatureInformation):

# 获取温度信息

Temperature = TemperatureInformation[item]["properties"]["TEMPERATURE"]

ID = Temperature["id"]

# 获取温度名称

CHS\_NAME = Temperature["chs\_name"]

# 获取温度是否可控制

CONTROLLABLE = Temperature["controllable"]

# 获取温度值

VALUE = Temperature["value"]

# 获取温度更新时间

UPDATE\_TIME = timestamp\_to\_timestr(Temperature["update\_time"])

# 将温度信息存入字典

dict = {

"id": ID,

"chs\_name": CHS\_NAME,

"controllable": CONTROLLABLE,

"value": VALUE,

"update\_time": UPDATE\_TIME,

}

# 返回温度信息字典

return dict

1. 性能优化

我们对性能的优化主要集中在对数据库的优化上

9.1明晰的表结构

我们在数据库的设计中注明了每个字段的注释，包括字段名、数据类型、约束条件等。确保表的结构合理、符合范式，并避免不必要的冗余。

* 1. 索引的使用与原则

我们在传感器的序列号上创建了单列索引。在数据写入和数据查询之间保持了较好的平衡性，数据查询速度提升20%到35%不等。