



UNIVERSIDAD CATÓLICA ANDRÉS BELLO

FACULTAD DE INGENIERÍA

ESCUELA DE INGENIERÍA INFORMÁTICA

Servidor IoT - Manual técnico

Realizado por

Rozas Bolívar, Gabriela Carolina

Tutor industrial

Bolívar Sánchez, María Victoria

Tutor académico

Fonseca Droy, Francisco José

Fecha

Septiembre, 2023

Índice de Contenido

Requerimientos de desarrollo	10
Softwares y entornos de desarrollo utilizados para el trabajo de grado.....	10
Lenguajes de programación utilizados y conocimientos necesarios.....	10
Configuración necesaria antes de programar en Arduino IDE.	10
Instalación del Servidor	13
Instalación de VNC Viewer en el equipo que controlará el servidor	13
Configuración del Raspberry Pi 4 Modelo B para el desarrollo y uso del servidor	13
Configuración del broker MQTT Eclipse Mosquitto.....	16
Configuración del bróker Redis	17
Configuración necesaria antes de programar y ejecutar el servidor en Visual Studio Code	17
Ejecutar el servidor y todas sus funciones.	20
Ficheros de Arduino.....	21
Modulo_ESP8266_NodeMCU	21
Librerías utilizadas.....	21
Constantes	21
Variables globales	23
Funciones	23
Modulo_ArduinoMEGA2560.....	31
Librerías utilizadas.....	31
Constantes	31
Variables globales	32
Funciones	33

	2
Modulo_ArduinoUNO	38
Librerías utilizadas	38
Constantes	38
Funciones	39
Librerías utilizadas para la programación del servidor	42
Ficheros importantes del proyecto	44
Manage.py	44
Requirements.txt	44
Celerybeat-schedule	44
Db.sqlite3	45
Media	45
Aplicaciones del servidor	46
Server	46
__pycache__	46
static	46
__init__.py	48
asgi.py	48
celery.py	50
settings.py	51
urls.py	52
wsgi.py	53
Iot	54

	3
__pycache__	54
config/i2c.py	54
Management/commands	55
Migrations	59
Templates	60
__init__.py	60
admin.py.....	60
apps.py	61
consumers.py	63
custom_datetime.py	67
models.py	67
routing.py	69
tasks.py.....	70
tests.py	72
Urls.py.....	73
Views.py	73
Mqtt.....	74
__pycache__	74
Migrations	74
__init__.py	74
Admin.py	75
Apps.py	75

	4
Models.py.....	76
Tests.py	76
Views.py	76
Users	76
__pycache__	76
Migrations.....	77
Templates.....	77
__init__.py	77
Admin.py	78
Apps.py	78
Forms.py	78
Models.py.....	79
Signals.py.....	81
Tests.py	81
Views.py	81
Web_plataform	82
__pycache__	82
Migrations.....	82
Templates.....	82
__init__.py	83
Admin.py	83
Apps.py	83

Models.py.....	83
Tests.py	84
Urls.py	84
Views.py	84

Índice de figuras

Figura 1. Arduino IDE – Preferencias	11
Figura 2. Arduino IDE - Board Manager.....	11
Figura 3. Izquierda: VNC Connect en el Raspberry Pi - Derecha: VNC Viewer en el equipo que controla el Raspberry Pi	14
Figura 4. Dirección IP que utilizó el VNC Viewer para controlar el Raspberry Pi.....	14
Figura 5. Agregando una dirección estática wlan0 al Raspberry Pi	15
Figura 6. wlan0 del Raspberry Pi.....	16
Figura 7. Configuración del broker Mosquitto	16
Figura 8. Usuarios con acceso al broker	17
Figura 9. Explorador de Visual Studio Code	18
Figura 10. Interprete utilizado para el entorno virtual del proyecto (venv).....	19
Figura 11. MQTT_CONFIG en Server/Server/Settings.py	19
Figura 12. Modulo_ESP8266_NodeMCU - connectToWifi	24
Figura 13. Modulo_ESP8266_NodeMCU - onWifiConnect.....	24
Figura 14. Modulo_ESP8266_NodeMCU - onWifiDisconnect	24
Figura 15. Modulo_ESP8266_NodeMCU - connectToMqtt.....	25
Figura 16. Modulo_ESP8266_NodeMCU - onMqttConnect	25
Figura 17. Modulo_ESP8266_NodeMCU - onDisconnect	26
Figura 18. Modulo_ESP8266_NodeMCU - onMqttPublish.....	26
Figura 19. Modulo_ESP8266_NodeMCU – PublishTemp	27
Figura 20. Modulo_ESP8266_NodeMCU - PublishHum	28

Figura 21. Modulo_ESP8266_NodeMCU - PublishWater.....	29
Figura 22. Modulo_ESP8266_NodeMCU - setup.....	30
Figura 23. Modulo_ESP8266_NodeMCU - loop	31
Figura 24. Modulo_ArduinoMEGA2560 - setup.....	33
Figura 25. Modulo_ArduinoMEGA256 - request_Event	34
Figura 26. Modulo_ArduinoMEGA256 - temperature_readings	35
Figura 27. Modulo_ArduinoMEGA256 - humidity_readings	35
Figura 28. Modulo_ArduinoMEGA256 - pir_readings	36
Figura 29. Modulo_ArduinoMEGA256 - ldr_readings	37
Figura 30. Modulo_ArduinoMEGA256 - boolean_readings.....	37
Figura 31. Modulo_ArduinoUNO - receiveEvent	39
Figura 32. Modulo_ArduinoUNO - setup.....	41
Figura 33. Modulo_ArduinoUNO - loop	41
Figura 34. manage.py.....	44
Figura 35. Server - __init__.py	48
Figura 36. Server - asgi.py	49
Figura 37. Configuración de Celery.py.....	50
Figura 38. Configuración del cronograma de tareas, en Celery.py del módulo Server	51
Figura 39. Lista de urls creados en el módulo Server.....	53
Figura 40. Server - wsgi.py.....	53
Figura 41. iot - config/i2c.py	54
Figura 42. Clase Command de i2creader.py	56

Figura 43. Clase Command de i2Cwriter.py	57
Figura 44. iot - Funciones cls y write_module de i2cwriter.py	58
Figura 45. Clase Command de mqtt_connect.py	59
Figura 46. iot - admin.py.....	61
Figura 47. iot - apps.py	62
Figura 48. Acceso a la tabla Temperatura_exterior de la base de datos con Channels	64
Figura 49. Código de la función get_previousweek_data() para obtener la temperatura de lunes a viernes, de 7am a 7pm	66
Figura 50. Mensaje enviado por el URL ws/lab-two/.....	67
Figura 51. iot - routing.py	70
Figura 52. Función event_post_add() del archivo task.py del módulo iot.....	72
Figura 53. iot - urls.py.....	73
Figura 54. iot - views.py	74
Figura 55. users - admin.py.....	78
Figura 56. users - apps.py	78
Figura 57. users - forms.py	79
Figura 58. users - models.py	80
Figura 59. users - signals.py	81
Figura 60. web_plataform - urls.py.....	84

Índice de tablas

Tabla 1. Librerías utilizadas para programar el ESP8266 NodeMCU, autores y sus funciones	21
Tabla 2. Librerías utilizadas para programar el Arduino MEGA 2560 R3, autores y sus funciones	31
Tabla 3. Librerías utilizadas para programar el Arduino UNO R3, autores y sus funciones	38
Tabla 4. Eventos que lleva a cabo el Arduino UNO R3 según el valor de x	40
Tabla 5. Librerías utilizadas para programar el servidor en Python, autores y sus funciones	42
Tabla 6. Tipos de mensajes que pueden enviarse por el Channel_layer lab_events.....	65
Tabla 7. Tópicos y modelo de la base de datos asociada.....	76

Requerimientos de desarrollo

Softwares y entornos de desarrollo utilizados para el trabajo de grado

- Sistema operativo Raspbian.
- Arduino IDE, versión 1.8.19 o superior.
- Visual Studio Code.
- Eclipse Mosquitto.
- Redis.
- VNC Viewer y VNC Connect.

Lenguajes de programación utilizados y conocimientos necesarios

- Conocimiento nivel medio utilizando Python 3.9.2 o superior.
- Conocimiento nivel medio utilizando el framework de desarrollo Django 4.1.4.
- Conocimiento nivel medio utilizando Arduino, que está basado en el lenguaje C.
- Conocimiento medio en el desarrollo web, utilizando los lenguajes: HTML, CSS y Javascript.
- Conocimientos nivel medio utilizando el SO Raspbian, que está basado en Debian.

Configuración necesaria antes de programar en Arduino IDE.

- Instalar el Plugin del ESP8266 para Arduino.

Para que el Arduino IDE reconozca el ESP8266 NodeMCU como una tarjeta es necesario instalar el Plugin del ESP8266. Para ello, en el fichero Modulo_ESP8266_NodeMCU nos ubicamos en File > Preferences, tal y como se muestra en la figura 1, y en la casilla “Additional Boards Manager URLs:” agregamos el link: http://arduino.esp8266.com/stable/package_esp8266com_index.json

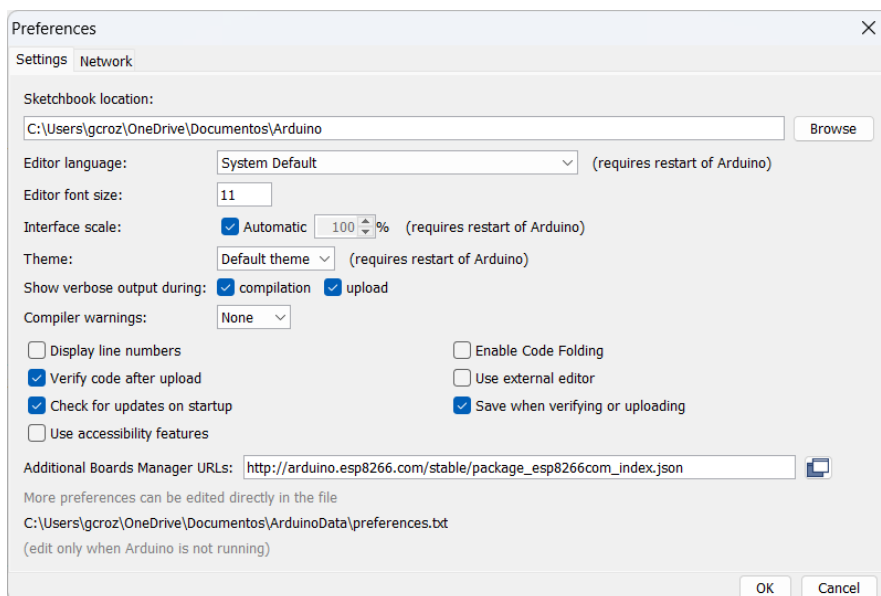


Figura 1. Arduino IDE – Preferencias

Adaptado de *Arduino IDE*

Seguidamente, nos ubicamos en Tools > Board > Boards Manager..., y buscamos en la lista “esp8266” by ESP8266 Community (ver en la figura 2).

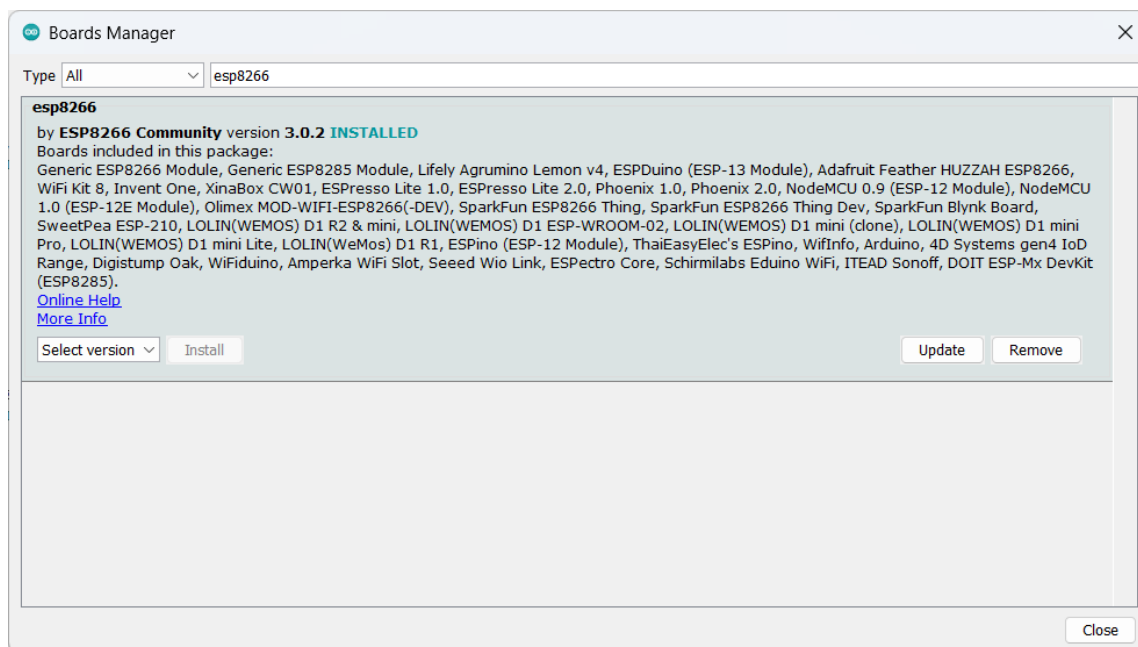


Figura 2. Arduino IDE - Board Manager

Una vez instalado, podemos utilizar esta placa para programar el ESP8266 NodeMCU en el Arduino IDE.

- Instalar el driver CH340 (opcional).

En caso de utilizar Arduinos genericos que no sean de la marca original y el Arduino IDE no lo reconozca al conectar el dispositivo, se recomienda instalar el driver CH340.

Instalación del Servidor

Instalación de VNC Viewer en el equipo que controlará el servidor

Con el fin de controlar el Raspberry Pi desde otra computadora/laptop se instala el software RealVNC en ambos equipos.

Primero, desde el sitio oficial de RealVNC se descarga e instala la aplicación VNC Viewer en el equipo que desea acceder al ordenador servidor.

Después, se descarga e instala VNC Connect en el ordenador servidor, que en este caso sería el Raspberry Pi 4 modelo B. VNC Connect consta de una aplicación VNC Server y otros programas de apoyo. Para más información, revise la sección de Configuración del Raspberry Pi 4 modelo B para el desarrollo y uso del servidor, que se muestra a continuación.

Configuración del Raspberry Pi 4 Modelo B para el desarrollo y uso del servidor

La instalación del sistema operativo Raspbian en el Raspberry Pi 4 modelo B se realizó con la guía del tutorial de instalación de Creatividad Ahora en youtube, donde se configuró la red WiFi a la que se conectaría el dispositivo y el Secure Shell, SSH, para permitir el acceso remoto al equipo, ya sea conectándose a él con un cable Ethernet o a través de Internet, y resolver la necesidad de poder contar con un puerto HDMI para ver la interfaz del Raspberry Pi en un monitor.

Con el fin de controlar el Raspberry Pi desde otra computadora/laptop se instaló el software RealVNC en ambos equipos (ver en la figura 3).

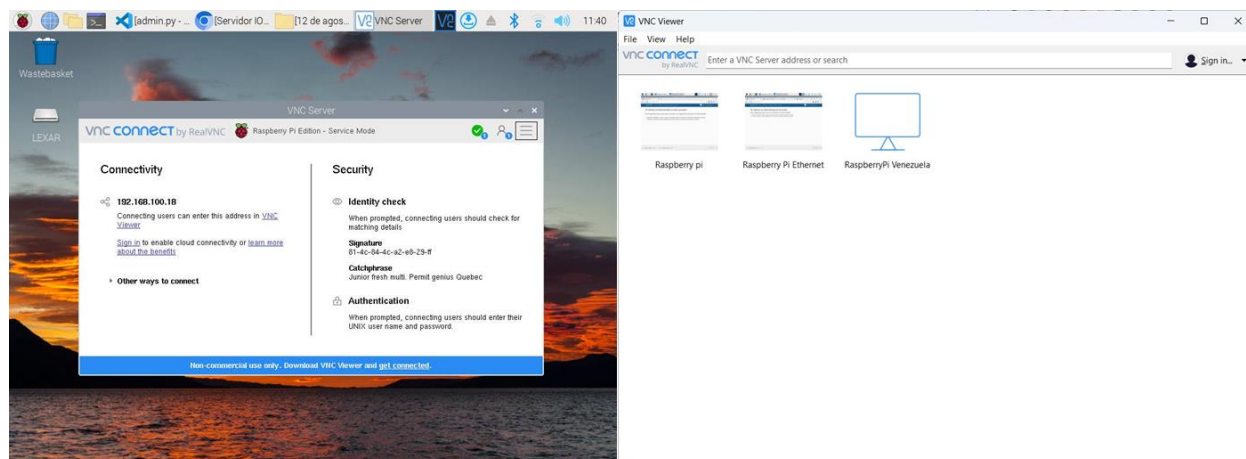


Figura 3. Izquierda: VNC Connect en el Raspberry Pi - Derecha: VNC Viewer en el equipo que controla el Raspberry Pi

Adaptado de RealVNC

Desde la página de configuración del router, que en este caso fue la dirección IP 192.168.100.1, se identificó que la dirección IP asignada por el router Rozas al Raspberry Pi era la 192.168.100.18, la cual fue agregada en las propiedades de la conexión que se creó en VNC Viewer (ver en la figura 4).

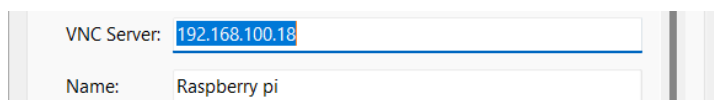


Figura 4. Dirección IP que utilizó el VNC Viewer para controlar el Raspberry Pi

Adaptado de VNC Viewer

Una vez establecida la conexión con el equipo, solo fue necesario iniciar sesión como el usuario “pi” (que viene predefinido en la instalación del SO), bajo la clave de “raspberrypi” para acceder al equipo.

Una vez que se tuvo acceso al equipo se cambió la clave de acceso del Raspberry Pi por una diferente, y se actualizaron los paquetes del sistema operativo. Seguidamente, se procedió a configurar una dirección IP estática para la interface WiFi (wlan0) en el Raspberry Pi.

Siguiendo el tutorial de Luis Llamas, para configurar una IP estática en Raspbian debemos editar el fichero /etc/dhcpd.conf desde la terminal, con el comando:

```
$ sudo nano /etc/dhcpd.conf
```

En el fichero encontraremos unas líneas comentadas (que empiezan con “#”) que tienen un ejemplo de configuración de IP estática. Para crear nuestra propia dirección estática para el wlan0 modificamos el archivo agregando el texto que se muestra en la figura, donde:

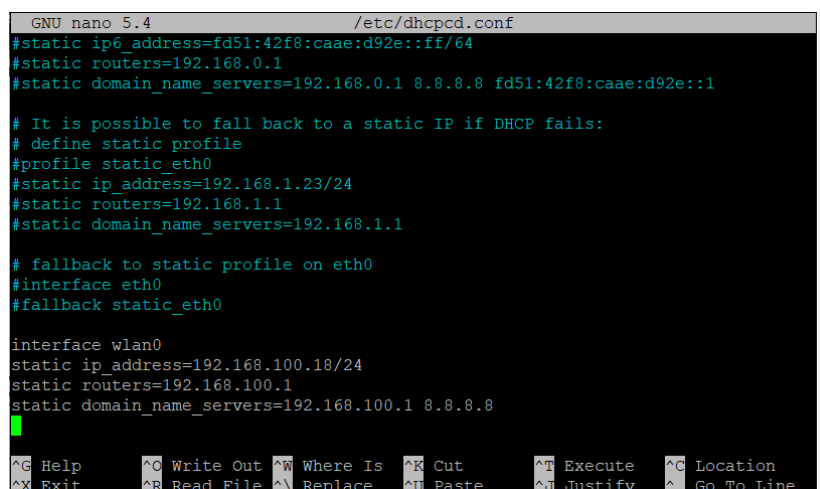
interface = Nombre de la interface que queremos configurar.

static ip_address = Dirección fija que queremos, dejando el /24 al final.

static routers = Dirección del gateway del router.

static domain_name_servers = Dirección del servidor DNS, que normalmente sería la del router o unas externas como las de Google 8.8.8.8.

Que mostró el resultado que se muestra en la figura 5.



```
GNU nano 5.4 /etc/dhcpd.conf
#static ip6_address=fd51:42f8:caae:d92e::ff/64
#static routers=192.168.0.1
#static domain_name_servers=192.168.0.1 8.8.8.8 fd51:42f8:caae:d92e::1

# It is possible to fall back to a static IP if DHCP fails:
# define static profile
#profile static_eth0
#static ip_address=192.168.1.23/24
#static routers=192.168.1.1
#static domain_name_servers=192.168.1.1

# fallback to static profile on eth0
#interface eth0
#fallback static_eth0

interface wlan0
static ip_address=192.168.100.18/24
static routers=192.168.100.1
static domain_name_servers=192.168.100.1 8.8.8.8
```

Figura 5. Agregando una dirección estática wlan0 al Raspberry Pi

Adaptado de /etc/dhcpd.conf

A continuación, se guardaron los cambios y se reinició el Raspberry Pi desde la terminal, con el comando:

```
$ sudo reboot
```

Finalmente, comprobamos que teníamos la dirección IP configurada ejecutando el comando:

```
$ ifconfig wlan0
```

Que mostró el resultado que se muestra en la figura 6.


```
wlan0: flags=4163<UP,BROADCAST,RUNNING,MULTICAST> mtu 1500
    inet 192.168.100.18 netmask 255.255.255.0 broadcast 192.168.100.255
    inet6 2800:300:6435:dd70::5 prefixlen 128 scopeid 0x0<global>
    inet6 fe80::8afa:713e:8e1e:7608 prefixlen 64 scopeid 0x20<link>
    ether dc:a6:32:75:3b:90 txqueuelen 1000 (Ethernet)
    RX packets 1837 bytes 1525977 (1.4 MiB)
    RX errors 0 dropped 0 overruns 0 frame 0
    TX packets 1349 bytes 162104 (158.3 KiB)
    TX errors 0 dropped 0 overruns 0 carrier 0 collisions 0
```

Figura 6. wlan0 del Raspberry Pi

Posterior a eso, se procedió a instalar el editor de código fuente Visual Studio Code para comenzar a programar la aplicación web del servidor.

Configuración del broker MQTT Eclipse Mosquitto

Luego de descargar e instalar Eclipse Mosquitto en el Raspberry Pi, se configuró el servicio para que se ejecutara cada vez que el sistema del equipo iniciara, ejecutando el siguiente comando en la terminal:

```
$ sudo systemctl enable mosquitto.service
```

Además, tal y como se ver en la figura 7, se editó el fichero /etc/mosquitto/mosquitto.conf para que los usuarios que quisieran utilizar el broker MQTT solo pudieran hacerlo si ingresaban los usuarios y claves incluidos en el fichero passwd.

```
mosquitto.conf x
1  # Place your local configuration in /etc/mosquitto/conf.d/
2  #
3  # A full description of the configuration file is at
4  # /usr/share/doc/mosquitto/examples/mosquitto.conf.example
5
6  per_listener_settings true
7  pid_file /run/mosquitto/mosquitto.pid
8
9  persistence true
10 persistence_location /var/lib/mosquitto/
11
12 log_dest file /var/log/mosquitto/mosquitto.log
13
14 include_dir /etc/mosquitto/conf.d
15 allow_anonymous false
16 listener 1883
17 password_file /etc/mosquitto/passwd
18
```

Figura 7. Configuración del broker Mosquitto

Adaptado de /etc/mosquitto/mosquitto.conf

Para generar el fichero passwd se ingresó el siguiente comando por consola:

```
$ sudo mosquito_passwd -c /etc/mosquito/passwd Rozas
```

Y se agregó un segundo usuario y clave al fichero con el siguiente comando:

```
$ sudo mosquito_passwd -b /etc/mosquito/passwd rasp-broker
```

Dando como resultado el fichero `/etc/mosquito/passwd` que se muestra en la figura 8, cuya clave de cada usuario está encriptada.



```
passwd x
1 Rozas:$7$101$s0rCLz/86uLuQtcf$iktAr5Hy+MZB1Bk05+f1rxA9+400Bhsmhtgd6nQI04ucU0rUx/oxoQGI
2 rasp-broker:$7$101$sZH9U2mJ3X22xP98$EmVgroPnl72lrtgma1Bm8yXZmE1xtQNkuIWT3U0x3v6xVj153i
3
```

Figura 8. Usuarios con acceso al broker

Adaptado de `/etc/mosquito/passwd`

Configuración del bróker Redis

Para instalar Redis en el Raspberry Pi, se siguieron los pasos del tutorial de Awais Khan, donde se usó el siguiente comando en la terminal:

```
$ sudo apt install redis-server -y
```

Después de terminar la instalación, para que Redis se ejecute de manera automática en cada reinicio del Raspberry Pi se usó el siguiente comando:

```
$ sudo systemctl enable redis
```

Configuración necesaria antes de programar y ejecutar el servidor en Visual Studio Code

- Abrir el proyecto

Para abrir el proyecto, nos dirigimos a Archivo > Abrir carpeta... Y seleccionamos la carpeta “Server”, donde se encuentra guardados todos los ficheros y módulos que conforman el proyecto. Luego de abrir el proyecto, se mostrará una lista de todas las carpetas y archivos que contiene en el Explorador, tal y como se muestra en la figura 9.

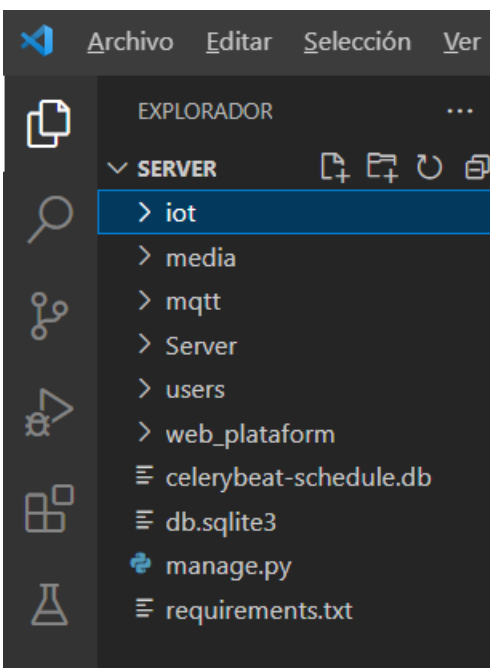


Figura 9. Explorador de Visual Studio Code

Adaptado de *Visual Studio Code*

- Crear un entorno virtual (venv)

Para programar el servidor que permitiera comunicar el Raspberry Pi con los dispositivos IoT, se creó un proyecto llamado “Server” en el editor Visual Studio Code. En dicho proyecto se creó un entorno virtual con la librería estándar virtualenv de Python, donde se instalarían todas las librerías necesarias para el funcionamiento del servidor. A diferencia del paquete venv, virtualenv es una librería que ofrece más funcionalidades.

Para acceder a la terminal que ofrece Visual Studio Code, nos dirigimos a Terminal > Nueva Terminal, y abriendo una terminal.

Después, para instalar la librería se utilizó el siguiente comando en la terminal:

```
$ pip install virtualenv
```

Una vez instalada, para crear el entorno virtual del proyecto se creó la ruta venv utilizando el siguiente comando:

```
$ python -m virtualenv venv
```

Por defecto, la extensión de Python de Visual Studio Code busca y usa el primer intérprete que se encuentra en la ruta del sistema. Aunque, es posible indicar un intérprete específico con las teclas Ctrl + Shift + P, y seleccionando el intérprete en la paleta de comandos, tal y como se muestra en la figura 10.

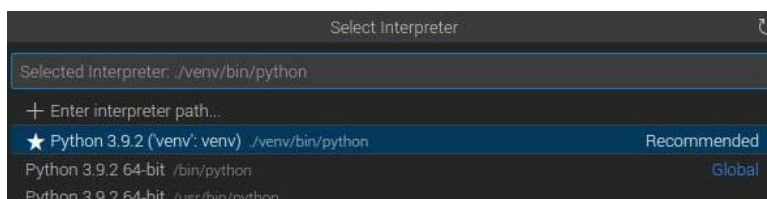


Figura 10. Intérprete utilizado para el entorno virtual del proyecto (venv)
Adaptado de *Visual Studio Code*

- Instalar el paquete de librerías del proyecto

Una vez que esté activo el entorno virtual del servidor, utilizamos el siguiente comando para instalar el paquete de librerías del proyecto, que se encuentra en el archivo de configuración requirements.txt

```
$ pip install -r requirements.txt
```

- Editar settings.py (opcional).

Específicamente, cambiar el valor '192.168.100.18' del elemento mqtt_broker en la variable MQTT_CONFIG que se muestra en la figura 11, por la dirección del Raspberry Pi asignada.

```
MQTT_CONFIG = {
    "mqtt_broker": '192.168.100.18', #ip del servidor central
    "mqtt_port": 1883, #puerto al que se conectara el host del servidor. Default: 1883 y 8883 cuando funciona sobre TLS
    "mqtt_qos": 0,
    "mqtt_username": 'rasp-broker',
    "mqtt_password": 'ucab*ucab'
}
```

Figura 11. MQTT_CONFIG en Server/Server/Settings.py

- Restablecer la base de datos y crear un superuser.

Cuando tenemos que restablecer toda la base de datos SQLite3 predeterminada de Django, debemos eliminar el archivo de base de datos db.sqlite3 que se encuentra en la carpeta

del proyecto, luego, eliminar todas las carpetas de migrations dentro de todas las aplicaciones, que en este caso serían las carpetas: `iot`, `mqtt`, `users` y `web_plataform`.

Después de eliminar las carpetas migrations, podemos rehacer las migraciones y migrarlas con dos comandos; a saber:

```
$ python manage.py makemigrations
$ python manage.py migrate
```

Seguidamente creamos un superusuario. Para crear un superusuario que sea administrador y programador del servidor, escribimos el siguiente comando:

```
$ python manage.py createsuperuser
```

Ejecutar el servidor y todas sus funciones.

De último, para poner en funcionamiento el servidor y todas las funciones, se abren tres terminales en Visual Studio Code.

En la primera terminal se inicializa el worker como un proceso de fondo con el siguiente comando:

```
$ celery -A Server worker --loglevel=INFO
```

En la segunda terminal se inicializa el servicio celery beat con el siguiente comando:

```
$ celery -A Server beat
```

En la tercera terminal inicializamos el servidor con el siguiente comando:

```
$ python manage.py runserver
```

Para acceder a la aplicación web del servidor desde el Raspberry Pi mientras se está ejecutando, escribimos en el navegador la dirección IP `127.0.0.1:8000`.

Ficheros de Arduino

Modulo_ESP8266_NodeMCU

Fichero que incluye el código que se desarrolló para que el ESP8266 MCU leyera información de los sensores conectados al dispositivo y envíe la información recibida al Raspberry Pi 4 Modelo B, utilizando el protocolo de comunicación MQTT.

Librerías utilizadas

Tabla 1. *Librerías utilizadas para programar el ESP8266 NodeMCU, autores y sus funciones*

Librería	Autor	Función
DHT.h	Adafruit	Librería desarrollada para sensores de temperatura/humedad como DHT11, DHT22, entre otros.
ESP8266WiFi.h	A-Vision Software	Librería WiFi para ESP8266 desarrollada basándose en el SDK de ESP8266, usando nombres convencionales y la filosofía de funcionalidades generales de la librería WiFi de Arduino.
Ticker.h	Stefan Staub	Librería desarrollada para llamar una función a un intervalo de tiempo determinado, con ayuda de las funciones micros() / millis()
AsyncMqttClient.h	Marvin Roger	Librería desarrollada para implementar un cliente MQTT asíncrono en ESP8266 y ESP31

Constantes

WIFI_SSID: Define el identificador de red SSID, que en este caso sería la red de area local inalámbrica “Rozas”.

WIFI_PASSWORD: Define la contraseña para acceder a la red definida en la variable WIFI_SSID que, en este caso, fue “Bolívar23”

MQTT_HOST IPAddress (192,168,100,18): Es la IP del equipo donde se encuentra instalado el broker MQTT que, en este caso, sería la misma IP que utiliza el Raspberry Pi para conectarse a la red definida en WIFI_SSID: 192.168.100.18

MQTT_HOST: Variable comentada, utilizada para pruebas, en lugar de una dirección IP. Contiene el link “https://test.mosquitto.org/” para que el dispositivo se conecte al server/broker MQTT que ofrece Eclipse Mosquitto de manera gratuita.

MQTT_PORT: Contiene el número de puerto al que se conectara el dispositivo, que en este caso sería 1883.

MQTT_QOS: Define la calidad de los mensajes que enviará el dispositivo, que en este caso, sería 1. Donde los mensajes se enviarán el publicador (ESP8266 NodeMCU) hasta que se garantiza la entrega. En caso de fallo, el suscriptor (Raspberry Pi) puede recibir algún mensaje duplicado.

MQTT_PUB_TEMP: Contiene el nombre del tópico donde el ESP8266 Node MCU publicará los mensajes de temperatura, que en este caso es: “esp8266/temperature”

MQTT_PUB_HUM: Contiene el nombre del tópico donde ESP8266 Node MCU publicará los mensajes de humedad relativa, que en este caso es: “esp8266/humidity”

MQTT_PUB_WATER: Contiene el nombre del tópico donde ESP8266 Node MCU publicará los mensajes de humedad del piso, que en este caso es: “esp8266/water”

Int WATERPIN: Pin donde se encuentra conectado el sensor de agua, que en este caso sería el pin 5.

DHTPIN: Pin donde se encuentra conectado el sensor DHT11 de temperatura y humedad relativa, que en este caso sería el pin 4.

DHTType: Define el tipo de sensor de temperatura y humedad, o DHT, que estará conectado al dispositivo, que en este caso es un DHT11.

Unsigned long interval_dht: Determina el intervalo de tiempo, en milisegundos, que se leerá el sensor DHT11 y se publicará el mensaje correspondiente, que en este caso sería 60000 (equivalente a 1 minuto).

Unsigned long interval_water: Determina el intervalo de tiempo, en milisegundos, que se leerá el sensor de agua y se publicará el mensaje correspondiente, que en este caso sería 300000 (equivalente a 5 minutos).

Variables globales

DHT dht(DHTPin, DHTType): En la variable dht definimos el pin al que está conectado el sensor DHT y tipo de sensor DHT utilizado.

AsyncMqttClient mqttClient: La variable mqttClient sirve para instanciar un cliente MQTT que se conectará de manera asincrónica.

Ticker mqttReconnectTimer: La variable mqttReconnectTimer sirve para instanciar un temporizador que permitirá al dispositivo reconectarse al MQTT bróker.

WiFiEventHandler wifiConnectHandler: Define a la variable como de clase genérica WiFiEventHandler para manejar los eventos relacionados a: Conectarse a la red WiFi.

WiFiEventHandler wifiDisconnectHandler: Define a la variable como de clase genérica WiFiEventHandler para manejar los eventos relacionados a: Desconectarse de la red WiFi.

Ticker wifiReconnectTimer: La variable wifiReconnectTimer sirve para instanciar un temporizador que permitirá al dispositivo reconectarse a la red WiFi.

Unsigned long event_dht: Definida inicialmente en 0, esta variable la utilizamos como condición de parada

Unsigned long event_water: Definida inicialmente en 0, esta variable la utilizamos como condición de parada

Funciones

Void connectToWifi()

Como se puede ver en la figura 12, esta función es utilizada para conectarse a la red WiFi definida en la variable WIFI_SSID con la contraseña definida en la variable WIFI_PASSWORD. Se imprimirá en el Serial de Arduino el mensaje: “Conectandose a la red WiFi...”.


```
void connectToWifi() {
    Serial.println("Conectadose a la red WiFi...");
    WiFi.begin(WIFI_SSID, WIFI_PASSWORD);
}
```

Figura 12. Modulo_ESP8266_NodeMCU - connectToWifi

Void onWifiConnect(const WiFiEventStationModeGotIP& event)

Como se puede ver en la figura 13, esta función es utilizada para imprimir en el Serial de Arduino mensajes que contiene la red WiFi a la que se está conectando el ESP8266 Node MCU, y la dirección IP que le asignó el router. Seguidamente, si la conexión fue exitosa, llamará a la función connectToMqtt() para conectarse al broker.

```
void onWifiConnect(const WiFiEventStationModeGotIP& event) {
    Serial.println("");
    Serial.print("Conectado a la red ");
    Serial.println(WIFI_SSID);
    Serial.print("IP address: ");
    Serial.println(WiFi.localIP());
    connectToMqtt();
}
```

Figura 13. Modulo_ESP8266_NodeMCU - onWifiConnect

Void onWifiDisconnect(const WiFiEventStationModeDisconnected& event)

Como se puede ver en la figura 14, esta función es utilizada para imprimir en el Serial de Arduino que se desconectó de la red WiFi, en caso de que se haya caído la conexión. Debido a esto, llamará al método detach() del objeto mqttReconnectTimer para evitar reintentar conectarse al bróker. Seguidamente, intentará reconectarse a la red WiFi con la función wifiReconnectTimer.

```
void onWifiDisconnect(const WiFiEventStationModeDisconnected& event) {
    Serial.println("Desconectado de la red WiFi.");
    mqttReconnectTimer.detach(); // Nos aseguramos que no intente conectarse a MQTT mientras intentamos reconectarnos al WiFi
    wifiReconnectTimer.once(2, connectToWifi);
}
```

Figura 14. Modulo_ESP8266_NodeMCU - onWifiDisconnect

Void connectToMqtt()

Como se puede ver en la figura 15, esta función es utilizada para conectarse al broker MQTT, imprimiendo en el Serial de Arduino el mensaje “Conectandose al broker MQTT...” y llamando al método connect() del objeto mqttClient.

```
void connectToMqtt() {
    Serial.println("Conectandose al broker MQTT...");
    mqttClient.connect();
}
```

Figura 15. Modulo_ESP8266_NodeMCU - connectToMqtt

Void onMqttConnect(bool sessionPresent)

Como se puede ver en la figura 16, esta función inicia una sesión una vez el ESP8266 NodeMCU ha podido conectarse al broker MQTT exitosamente. Imprimiendo en el Serial de Arduino el mensaje “Conectado al broker MQTT.”, y la sesión presente que fue pasado por parámetro.

```
void onMqttConnect(bool sessionPresent) {
    Serial.println("Conectado al broker MQTT.");
    Serial.print("Session: ");
    Serial.println(sessionPresent);
}
```

Figura 16. Modulo_ESP8266_NodeMCU - onMqttConnect

Void onMqttDisconnect(AsyncMqttClientDisconnectReason reason)

Como se puede ver en la figura 17, esta función que avisa que el ESP8266 NodeMCU se ha desconectado del bróker, escribiendo en el Serial de Arduino el mensaje “Desconectado del bróker MQTT.”. Seguidamente, si el dispositivo aún está conectado a la red WiFi, intentará reconectarse al broker.

```

void onMqttDisconnect(AsyncMqttClientDisconnectReason reason) {
    Serial.println("Desconectado del broker MQTT.");

    if (WiFi.isConnected()) {
        mqttReconnectTimer.once(2, connectToMqtt);
    }
}

```

Figura 17. Modulo_ESP8266_NodeMCU - onDisconnect

Void onMqttPublish(uint16_t packetId)

Como se puede ver en la figura 18, esta función escribirá un mensaje en el serial de Arduino cada vez que el ESP8266 NodeMCU publique un mensaje en el bróker MQTT.

```

void onMqttPublish(uint16_t packetId) {
    Serial.print("Se publico un mensaje.");
    Serial.print("  packetId: ");
    Serial.println(packetId);
}

```

Figura 18. Modulo_ESP8266_NodeMCU - onMqttPublish

Void PublishTemp()

Como ve en la figura 19, esta función lee la temperatura del sensor DHT11 en grados Celsius. El resultado es un número decimal, que es redondeado y convertido en una variable entera guardada en “temp”. Si la variable resultante no es un número, en caso de que haya ocurrido un error durante la lectura, el payload será igual a “error”. Por lo tanto, en el Serial de Arduino se imprimirá el mensaje: “Mensaje: error.”.

En el caso contrario, si la variable resultando es un número, el payload será igual al número, transformado en una variable String, y se imprimirá en el Serial de Arduino el mensaje que será publicado a continuación.

Seguidamente, se enviará el mensaje (payload) al tópico asignado en la variable MQTT_PUB_TEMP, y la calidad del mensaje será la misma que fue definida en la variable MQTT_QOS. En paralelo, se creará un paquete de 2 bytes llamado packetIdPub, el cual contiene los datos del envío, que será impreso en el Serial de Arduino.

```

void PublishTemp() {
    String payload;
    int temp = int(round(dht.readTemperature())); // Lee la temperatura en Celsius (default)
    // Lee la temperatura en Fahrenheit (isFahrenheit = true)
    //float temp = dht.readTemperature(true);
    if (isnan(temp)) {
        payload = "error";
        Serial.println("Mensaje: error.");
    } else{
        payload = String(temp);
        Serial.printf("Mensaje: %i \n", temp);
    }
    // Publica un mensaje MQTT al topico esp8266/temperature
    uint16_t packetIdPubl = mqttClient.publish(MQTT_PUB_TEMP, MQTT_QOS, true, payload.c_str());
    Serial.printf("Publicando al topico %s en QoS 1, packetId: %i \n", MQTT_PUB_TEMP, packetIdPubl);
}

```

Figura 19. Modulo_ESP8266_NodeMCU – PublishTemp

Void PublishHum()

Como se puede ver en la figura 20, en esta función se lee la humedad relativa del sensor DHT11. El resultado es un número decimal que es redondeado y convertido en una variable entera guardada en “hum”.

Si la variable resultante no es un número, en caso de que haya ocurrido un error durante la lectura, el payload será igual a “error”. Por lo tanto, en el Serial de Arduino se imprimirá el mensaje: “Mensaje: error.”.

En el caso contrario, si la variable resultando es un número, el payload será igual al número, transformado en una variable String, y se imprimirá en el Serial de Arduino el mensaje que será publicado a continuación.

Seguidamente, se enviará el mensaje (payload) al tópico asignado en la variable MQTT_PUB_HUM, y la calidad del mensaje será la misma que fue definida en la variable MQTT_QOS. En paralelo, se creará un paquete de 2 bytes llamado packetIdPub2, el cual contiene los datos del envío, que será impreso en el Serial de Arduino.

```
void PublishHum() {
    String payload;
    int hum = int(round(dht.readHumidity())); // Lee la humedad relativa
    if (isnan(hum)) {
        payload = "error";
        Serial.println("Mensaje: error.");
    } else{
        payload = String(hum);
        Serial.printf("Mensaje: %i \n", hum);
    }
    // Publica un mensaje MQTT al topico esp8266/humidity
    uint16_t packetIdPub2 = mqttClient.publish(MQTT_PUB_HUM, MQTT_QOS, true, payload.c_str());
    Serial.printf("Publicando al topico %s en QoS 1, packetId: %i \n", MQTT_PUB_HUM, packetIdPub2);
}
```

Figura 20. Modulo_ESP8266_NodeMCU - PublishHum

Void PublishWater()

Como se puede ver en la figura 21, en esta función se lee la humedad del piso del sensor. El resultado varía entre dos estados: “1” si el sensor está mojado, y “0” si el sensor está seco; el resultado será guardado en una variable payload.

Se imprimirá en el Serial de Arduino el mensaje que será publicado a continuación.

Seguidamente, se enviará el mensaje (payload) al tópico asignado en la variable MQTT_PUB_WATER, y la calidad del mensaje será la misma que fue definida en la variable MQTT_QOS. En paralelo, se creará un paquete de 2 bytes llamado packetIdPub3, el cual contiene los datos del envío, que será impreso en el Serial de Arduino.

```
void PublishWater(){
    String payload;
    if (! digitalRead(WATERPIN)){
        payload = "1"; //Hay humedad (el sensor esta mojado)
    } else {
        payload = "0"; //No hay humedad (el sensor esta seco)
    }
    Serial.printf("Mensaje: %s \n",payload);
    uint16_t packetIdPub3 = mqttClient.publish(MQTT_PUB_WATER, MQTT_QOS, true, payload.c_str());
    Serial.printf("Publicando al topico %s en QoS 1, packetId %i: \n", MQTT_PUB_WATER, packetIdPub3);
}
```

Figura 21. Modulo_ESP8266_NodeMCU - PublishWater

Void setup()

Como se puede ver en la figura 22, esta función es llamada cuando se ejecuta el sketch en el Arduino. Primero, abre el Puerto Serial y especifica la velocidad de transmisión. La velocidad típica para comunicación con el ordenador es de 9600.

Después, inicializa la variable dht para la lectura de temperatura y humedad relativa; luego, inicializa las variables que manejaran los eventos de conexión/desconexión al WiFi, seguidamente, inicializa el MQTT client para conectarse al bróker y enviar mensajes.

Para utilizar el bróker MQTT que se encuentra en el Raspberry Pi, se realiza una autenticación con el método setCredentials, pasando por parámetros el usuario “rasp-broker” y la contraseña “ucab*ucab”.

Una vez terminado las configuraciones pertinentes, se llama a la función connectToWifi() para inicializar todos los procesos.

La función setup() solo se ejecutará una sola vez, luego de que el Arduino recibe alimentación, o después de que es reseteado.

```

void setup() {
  Serial.begin(9600);
  pinMode(WATERPIN, INPUT);

  dht.begin();

  wifiConnectHandler = WiFi.onStationModeGotIP(onWifiConnect);
  wifiDisconnectHandler = WiFi.onStationModeDisconnected(onWifiDisconnect);

  mqttClient.onConnect(onMqttConnect);
  mqttClient.onDisconnect(onMqttDisconnect);

  mqttClient.onPublish(onMqttPublish);
  mqttClient.setServer(MQTT_HOST, MQTT_PORT);
  // Si el broker requiere autenticacion (usuario y contraseña), y posee una credencial, se configura con la siguiente funcion
  mqttClient.setCredentials("rasp-broker", "ucab*ucab");

  connectToWifi();
}

```

Figura 22. Modulo_ESP8266_NodeMCU - setup

Void loop()

Como se puede ver en la figura 23, esta función es el núcleo del programa en Arduino y se usa para el control activo de la placa. Se ejecutará continuamente, justo después de la función setup().

En esta función se hizo uso de la función millis, para calcular cuando segundos han pasado y ejecutar la función correspondiente:

- Cada 60000 milisegundos (1 minuto) se ejecutan las funciones PublishTemp() y PublishHum(), donde el ESP8266 NodeMCU procede a leer la temperatura y la humedad relativa del sensor DHT11, para luego enviar un mensaje a los tópicos “esp8266/temperature” y “esp8266/humidity” con los datos (en formato entero).
- Cada 300000 milisegundos (5 minutos) se ejecuta la función PublishWater(), donde el ESP8266 NodeMCU procede a leer el sensor de agua, para luego enviar un mensaje al tópico “esp8266/water” con el dato (“1” si el sensor estaba mojado, “0” si el sensor estaba seco).

```

void loop(){
  unsigned long currentMillis = millis();

  if (currentMillis - event_dht >= interval_dht) {
    PublishTemp(); //Publica la temperatura
    PublishHum(); //Publica la humedad
    event_dht = currentMillis; // Guarda el tiempo en que una nueva lectura fue publicada
  }

  if (currentMillis - event_water >= interval_water) {
    PublishWater(); //Publica si hay o no humedad en el sensor de agua
    event_water = currentMillis; // Guarda el tiempo en que una nueva lectura fue publicada
  }
}

```

Figura 23. Modulo_ESP8266_NodeMCU - loop

Modulo_ArduinoMEGA2560

Fichero que incluye el código que se desarrolló para que el Arduino MEGA 2560 R3 (Esclavo) leyera información de los sensores conectados al dispositivo y envíe la información recibida al Raspberry Pi 4 Modelo B (Maestro), utilizando el protocolo de comunicación I2C, cuando sea solicitada.

Librerías utilizadas

Tabla 2. Librerías utilizadas para programar el Arduino MEGA 2560 R3, autores y sus funciones

Librería	Autor	Función
Wire.h	Librería estándar de Arduino	Librería que le permite a Arduino implementar la comunicación I2C, ya sea como maestro a otros dispositivos o como esclavo recibiendo peticiones y respondiendo datos.
DHT.h	Adafruit	Librería desarrollada para sensores de temperatura/humedad como DHT11, DHT22, entre otros.

Constantes

SLAVE_ADDRESS: Define una dirección I2C para que el Arduino MEGA 2560 (Esclavo) pueda comunicarse con el Raspberry Pi 4 Modelo B (Maestro), en este caso su dirección sería 11.

DHTPin: Pin de entrada para el sensor de temperatura y humedad, que en este caso sería el pin digital 7.

Int LDRPin_A: Pin de entrada para el sensor de luz A, que en este caso sería el pin analógico A0.

Int LDRPin_B: Pin de entrada para el sensor de luz B, que en este caso sería el pin analógico A1.

Int PIR_Pin_A: Pin de entrada para el sensor PIR de movimiento A, que en este caso sería el pin digital 2.

Int PIR_Pin_B: Pin de entrada para el sensor PIR de movimiento B, que en este caso sería el pin digital 8.

Int DOORPin: Pin de entrada para la cerradura de la puerta, que en este caso sería el pin digital 3

Int ACPin_A: Pin de entrada para leer el estado del aire acondicionado A, que en este caso sería el pin digital 10.

Int ACPin_B: Pin de entrada para leer el estado del aire acondicionado B, que en este caso sería el pin digital 11.

DHTType: Define el tipo de sensor de temperatura y humedad, o DHT, que estará conectado al dispositivo, que en este caso es un DHT22.

Variables globales

DHT dht(DHTPin, DHTType): En la variable dht definimos el pin al que esta conectado el sensor DHT y tipo de sensor DHT utilizado.

Funciones

Void setup()

Como se puede ver en la figura 24, esta función es llamada cuando se ejecuta el sketch en el Arduino. Primero, configura al Arduino para que se una al bus I2C con la dirección definida en la variable SLAVE_ADDRESS, seguidamente, configura los pines para que sean entradas, inicializa la variable dht para la lectura de temperatura y humedad relativa, de último, pone en espera al Arduino (Esclavo) para ejecutar la función request_Event cuando reciba una request del Raspberry Pi (Maestro). La función setup() solo se ejecutará una sola vez, luego de que el Arduino recibe alimentación, o después de que es reseteado.

```
void setup() {
  Wire.begin(SLAVE_ADDRESS); //Inicializa la comunicacion I2C con el Maestro
  pinMode(PIRPin_A, INPUT); //Pin digital de entrada conectado al sensor PIR A
  pinMode(PIRPin_B, INPUT); //Pin digital de entrada conectado al sensor PIR B
  pinMode(ACPin_A, INPUT); //Pin digital de entrada conectado a la salida de relay asociada al aire acondicionado A
  pinMode(ACPin_B, INPUT); //Pin digital de entrada conectado la salida de relay asociada al aire acondicionado A
  pinMode(DOORPin, INPUT); //Pin digital de entrada conectado al sensor de puerta
  pinMode(LDRPin_A, INPUT); //Pin analogico de entrada conectado al sensor de luz A
  pinMode(LDRPin_B, INPUT); //Pin analogico de entrada conectado al sensor de luz B

  dht.begin(); //Inicializa el sensor de temperatura y humedad

  Wire.onRequest(request_Event); //Cuando el maestro solicite informacion al esclavo, se activara esta funcion
}
```

Figura 24. Modulo_ArduinoMEGA2560 - setup

Void request_Event()

Como se puede ver en la figura 25, esta función se ejecuta cada vez que el Arduino recibe una Request del Raspberry Pi (Maestro). Define una variable array dato[8] de tipo byte para guardar la información que reciba de los sensores: la temperatura, la humedad relativa, la presencia, la luminosidad de las secciones A y B, el estado de la puerta, y los estados de los aires acondicionados A y B; para después enviar el array al Raspberry Pi con la función Wire.write().

```

//Eventos solicitados
void request_Event(){
  int i = 0;
  byte dato[8]; // Esta dato en bytes que se envia al raspberry pi
  while(i<8){
    switch(i){
      case 0:
        dato[i] = temperature_readings(); //Leemos la temperatura
        break;
      case 1:
        dato[i] = humidity_readings(); //Leemos la humedad
        break;
      case 2:
        dato[i] = pir_readings(); //Leemos si hay o no movimiento
        break;
      case 3:
        dato[i] = ldr_readings(LDRPin_A); //Leemos la luminosidad del ambiente
        break;
      case 4:
        dato[i] = ldr_readings(LDRPin_B); //Leemos la luminosidad del ambiente
        break;
      case 5:
        dato[i] = boolean_readings(DOORPin); //Leemos si la puerta esta abierta o cerrada
        //HIGH: Puerta cerrada. El circuito esta cerrado
        //LOW: Puerta abierta. El circuito esta abierto
        break;
      case 6:
        dato[i] = boolean_readings(ACPin_A); //Leemos si el aire acondicionado A esta encendido o apagado
        break;
      case 7:
        dato[i] = boolean_readings(ACPin_B); //Leemos si el aire acondicionado B esta encendido o apagado
        break;
    }
    i++;
  }
  Wire.write(dato,8); //Envia los datos leidos al maestro
}

```

Figura 25. Modulo_ArduinoMEGA256 - request_Event

Byte temperature_readings()

Como se puede ver en la figura 26, esta función se encarga de leer la temperatura, en entero, desde el sensor DHT22 conectado al Arduino, para después retornar 0 si la lectura recibida no es un número, o retornar el valor de la variable transformado en bytes.

```

byte temperature_readings(){
  int temp = dht.readTemperature(); //leyendo la temperatura por el pin conectado al sensor dht22
  byte temp_bytes;
  if (isnan(temp)){
    return 0;
  } else {
    temp_bytes=(byte)temp;
    return temp_bytes;
  }
}

```

Figura 26. Modulo_ArduinoMEGA256 - temperature_readings

Byte humidity_readings()

Como se puede ver en la figura 27, esta función que se encarga de leer la humedad relativa, en entero, desde el sensor DHT22 conectado al Arduino, para después retornar 0 si la lectura recibida no es un número, o retornar el valor de la variable transformado en bytes.

```

byte humidity_readings(){
  int hum = dht.readHumidity(); //leyendo la humedad por el pin conectado al sensor dht22
  byte hum_bytes;
  if (isnan(hum)){
    return 0;
  } else {
    hum_bytes=(byte)hum;
    return hum_bytes;
  }
}

```

Figura 27. Modulo_ArduinoMEGA256 - humidity_readings

Byte Pir_readings()

Como se puede ver en la figura 28, esta función se encarga de leer los sensores PIR de movimiento conectados al Arduino, para después realizar una operación lógica OR entre la variable pir_valueA y la variable pir_valueB para devolver un valor entero 1 (TRUE) u 0 (FALSE).

Si alguno de los dos sensores detectó movimiento, el valor de la variable pir_total será 1.

Si ninguno de los dos sensores detectó movimiento, el valor de la variable pir_total será 0.

El resultado será convertido en un valor en byte para ser devuelto por la función.

```

byte pir_readings(){
  int pir_valueA = digitalRead(PIRPin_A);
  int pir_valueB = digitalRead(PIRPin_B);
  int pir_total = (pir_valueA or pir_valueB);
  byte pir_bytes = (byte)pir_total;
  return pir_bytes;
}

```

Figura 28. Modulo_ArduinoMEGA256 - pir_readings

Byte ldr_readings(const int pin_value)

Como se puede ver en la figura 29, esta función se encarga de leer una resistencia ldr desde el pin análogo pin_value que se pase por referencia.

El valor que recibiremos de la variable entera analogValue será un número entre 0 y 1023, equivalente de 0 a 5 analógicos.

En la maqueta física se conectaron las resistencias LDR en pull-up por lo que el valor de la variable analogValue sería el siguiente, dependiendo de la cantidad de luz que reciba la resistencia:

- analogValue < 10: Brillante
- analogValue < 200: Muy iluminado
- analogValue < 500: Iluminado
- analogValue < 800: Poco iluminado
- analogValue >= 800: Oscuro

Para poder transformar la variable en byte se utilizó la función map() para sacar el equivalente a 8 bits, que sería un número que se encuentra entre 0 y 255, guardado en la variable entera analogValue.

Finalmente, variable analogValue será transformada en una variable byte y retornada por la función.

```

byte ldr_readings(const int pin_value){
    int analogValue = analogRead(pin_value); // Leyendo el sensor de luz por el pin analogo A0

    //Mapea un valor analogo a 8 bits (0 a 255)
    analogValue = map(analogValue, 0, 1023, 0, 255);
    byte byte_value = (byte)analogValue;
    return byte_value;

    /*Luz que detecta el sensor
    analogValue < 10: Oscuro
    analogValue < 200: Poco iluminado
    analogValue < 500: Iluminado
    analogValue < 800: Brillante
    analogValue >= 800: Muy brillante
    */
}

```

Figura 29. Modulo_ArduinoMEGA256 - ldr_readings

Byte Boolean_readings(const int pin_value)

Como se puede ver en la figura 30, esta función se encarga de leer una entrada digital y retornar el valor en byte, pasando por referencia la variable pin_value que contiene el número del pin a leer. En este caso, se utiliza la función tres veces durante la ejecución del programa, para leer el estado del pin conectado a la cerradura de la puerta, y el estado de los dos pines conectados a los aires acondicionados.

```

byte boolean_readings(const int pin_value){
    int sensor_value = digitalRead(pin_value); //lectura digital de pin conectado al sensor
    byte sensor_byte=(byte)sensor_value;
    return sensor_byte;
}

```

Figura 30. Modulo_ArduinoMEGA256 - boolean_readings

Void loop()

Esta función es el núcleo de todos los programas de Arduino y se usa para el control activo de la placa. Se ejecutará continuamente, justo después de la función setup(). Esta función se dejó vacía, ya que cualquier proceso que se ejecutaba en esta función interfería con la comunicación bus I2C, por tanto, se colocaron en la función setup().

Modulo_ArduinoUNO

Fichero que incluye el código que se desarrolló para que el Arduino UNO R3 (Esclavo) enviara información de los actuadores conectados al dispositivo y estos realizaran una acción, de acuerdo a la información recibida al Raspberry Pi 4 Modelo B (Maestro), utilizando el protocolo de comunicación I2C.

Librerías utilizadas

Tabla 3. *Librerías utilizadas para programar el Arduino UNO R3, autores y sus funciones*

Librería	Autor	Función
Wire.h	Librería estándar de Arduino	Librería que le permite a Arduino implementar la comunicación I2C, ya sea como maestro a otros dispositivos o como esclavo recibiendo peticiones y respondiendo datos.

Constantes

SLAVE_ADDRESS: Define una dirección I2C para que el Arduino UNO (Esclavo) pueda comunicarse con el Raspberry Pi 4 Modelo B (Maestro), en este caso su dirección sería 12.

Int ledPin_A: Pin del Arduino que se comunica con el conector de entrada del Módulo de 4 relés (o relays) para encender/apagar uno de los LEDs azules. En este caso sería el pin 6.

Int ledPin_B: Pin del Arduino que se comunica con el conector de entrada del Módulo de 4 relés (o relays) para encender/apagar uno de los LEDs azules. En este caso sería el pin 5.

Int ledPin_C: Pin del Arduino que se comunica con el conector de entrada del Módulo de 4 relés (o relays) para encender/apagar uno de los LEDs amarillos. En este caso sería el pin 4.

Int ledPin_D: Pin del Arduino que se comunica con el conector de entrada del Módulo de 4 relés (o relays) para encender/apagar uno de los LEDs amarillos. En este caso sería el pin 3.

Funciones

Void receiveEvent(int howMany)

Como se puede ver en la figura 32, esta función se ejecuta cuando el Arduino UNO (Esclavo) recibe un mensaje del Raspberry Pi (Maestro) a través del bus I2C.

```
void receiveEvent(int howMany){
  while (Wire.available()){
    int x = Wire.read(); //Recibe el byte como un entero
    switch (x){
      case 10:
        digitalWrite(ledPin_D, HIGH);
        break;
      case 11:
        digitalWrite(ledPin_D, LOW);
        break;
      case 20:
        digitalWrite(ledPin_C, HIGH);
        break;
      case 21:
        digitalWrite(ledPin_C, LOW);
        break;
      case 30:
        digitalWrite(ledPin_B, HIGH);
        break;
      case 31:
        digitalWrite(ledPin_B, LOW);
        break;
      case 40:
        digitalWrite(ledPin_A, HIGH);
        break;
      case 41:
        digitalWrite(ledPin_A, LOW);
        break;
    }
  }
}
```

Figura 31. Modulo_ArduinoUNO - receiveEvent

Mientras el esclavo permanezca conectado al bus, realizará una acción dependiendo del mensaje recibido, el mensaje será guardado en una variable de tipo entero “x” (ver en la tabla 4).

Tabla 4. *Eventos que lleva a cabo el Arduino UNO R3 según el valor de x*

Valor de x	Evento
10	Apagar luces de la sección 1
11	Encender luces de la sección 1
20	Apagar luces de la sección 2
21	Encender luces de la sección 1
30	Apagar AC 1
31	Encender AC 1
40	Apagar AC 2
41	Encender AC 2

void setup()

Como se puede ver en la figura 33, esta función es llamada cuando se ejecuta el sketch en el Arduino. Primero, configura los pines definidos para que sean salidas. Seguidamente, envía una señal de HIGH a los pines definidos, con el fin de inicializar los relés y apagar los LEDs. De último, configura el Arduino para que se una al bus I2C con la dirección definida en la variable SLAVE_ADDRESS y pone en espera al Arduino (Esclavo) para ejecutar la función receive_Event cuando reciba un mensaje del Raspberry Pi (Maestro). La función setup() solo se ejecutará una sola vez, luego de que el Arduino recibe alimentación, o después de que es reseteado.

```

void setup() {
  //Configura los pines como salidas
  pinMode(ledPin_A, OUTPUT);
  pinMode(ledPin_B, OUTPUT);
  pinMode(ledPin_C, OUTPUT);
  pinMode(ledPin_D, OUTPUT);

  digitalWrite(ledPin_A, HIGH);
  digitalWrite(ledPin_B, HIGH);
  digitalWrite(ledPin_C, HIGH);
  digitalWrite(ledPin_D, HIGH);

  Wire.begin(SLAVE_ADDRESS); //Se une al bus I2C como un esclavo con la direccion correspondiente

  //Llama a la funcion receiveEvent cuando recibe la data
  Wire.onReceive(receiveEvent);
}

```

Figura 32. Modulo_ArduinoUNO - setup

loop()

Esta función es el núcleo de todos los programas de Arduino y se usa para el control activo de la placa. Se ejecutará continuamente, justo después de la función setup(). Como se puede ver en la figura 34, esta función se dejó vacía, ya que cualquier proceso que se ejecutaba en esta función interfería con la comunicación bus I2C, por tanto, se colocaron en la función setup().

```

void loop() {
  // put your main code here, to run repeatedly:

}

```

Figura 33. Modulo_ArduinoUNO - loop

Librerías utilizadas para la programación del servidor

Tabla 5. *Librerías utilizadas para programar el servidor en Python, autores y sus funciones*

Librería	Versión	Autor	Función
asgiref	3.6.0	Django Software Foundation	También llamado ASGI, provee funciones y clases que permite la comunicación asíncrona en aplicaciones web y servidores, siendo sucesor de WSGI
Django	4.1.4	Django Software Foundation	Utilizado para desarrollar aplicaciones web de forma rápida y eficiente.
Channels	4.0.0	Django Software Foundation	Le otorga a Django la capacidad de manejar protocolos que requieren una conexión persistente como, por ejemplo, websockets.
Channels-redis	4.0.0	Django Software Foundation	Provee a Django Channels de channel layers que utilicen Redis como almacén de respaldo
Daphne	4.0.0	Django Software Foundation	Dar soporte a Django Channels, implementando los protocolos HTTP, HTTP2 y WebSocket a ASGI y ASGI-HTTP en el servidor
Smbus2	0.4.2	Karl-Petter Lindegaard	Diseñado para ser el reemplazo de la librería smbus, conservando la misma sintaxis. Utiliza las estructuras y uniones de I2C en mayor medida que otras implementaciones puras de Python como lo hace pysmbus.
Paho-mqtt	1.6.1	Roger Light	Provee una clase cliente que permite a las aplicaciones conectarse al MQTT bróker, ya sea para publicar mensajes, o suscribirse a tópicos para recibir mensajes publicados.
Django-crispy-forms	1.14.0	Miguel Araujo	Estiliza los formularios de Django con la ayuda de paquetes de plantillas incorporadas.
Django-celery-beat	2.4.0	Asif Uddin, Saif Ask Solem	Extensión que permite almacenar cronogramas de tareas periódicas en una base de datos. Las tareas periódicas pueden ser manejadas desde la interfaz de Django Admin, donde se pueden crear, editar, borrar y ejecutar.
Django-redis	5.2.0	Andrei Antoukh	Provee a Django de funciones que le permitan implementar un sistema de caché con Redis
Celery	5.2.7	Ask Solem	En combinación con Django sirve para resolver la falta de asincronía de la aplicación web y mejorar su rendimiento, permitiendo la creación de tareas periódicas

pytz	2023.3	Stuart Bishop	Permite cálculos precisos y multiplataformas de zonas horarias, resolviendo el problema de los horarios ambiguos al final del horario de verano
xlwt	1.3.0	John Machin	Generar hojas de cálculo compatibles con Microsoft Excel, versiones 95 y 2003.
Pillow	9.3.0	Jeffrey A. Clark	Pillow es una bifurcación de PIL que provee soporte al formato de ficheros, permitiendo agregar y procesar imágenes desde el intérprete de Python
Flower	1.2.0	Mher Movsisyan	Provee información sobre el estatus de los workers y tareas de Celery
time	-	Estándar de Python	Proporcionar funciones relacionadas con el tiempo
datetime	-	Estándar de Python	Proporcionar clases para manipular fechas y horas
os	-	Estándar de Python	Provee una manera versátil de usar funcionalidades dependientes del sistema operativo
json	-	Estándar de Python	Codificar y decodificar formato JSON
threading	-	Estándar de Python	Construir interfaces de hilado de alto nivel sobre el módulo de más bajo nivel _thread.
logging	-	Estándar de Python	Provee funciones y clases para rastrear los eventos que ocurren cuando se ejecuta el sistema

Ficheros importantes del proyecto

Manage.py

Como se puede ver en la figura 35, el script manage.py ayuda con la administración del sitio. Con él se inicializa el servidor desde la terminal de comandos, sin necesidad de instalar otras herramientas o softwares.

```
#!/usr/bin/env python
"""Django's command-line utility for administrative tasks."""
import os
import sys

def main():
    """Run administrative tasks."""
    os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'Server.settings')
    try:
        from django.core.management import execute_from_command_line
    except ImportError as exc:
        raise ImportError(
            "Couldn't import Django. Are you sure it's installed and "
            "available on your PYTHONPATH environment variable? Did you "
            "forget to activate a virtual environment?"
        ) from exc
    execute_from_command_line(sys.argv)

if __name__ == '__main__':
    main()
```

Figura 34. manage.py

Requirements.txt

Contiene las librerías y paquetes requeridos por el proyecto, facilitando su instalación.

Celerybeat-schedule

Es la base de datos generada por la librería celery-beat, el cual guarda la lista de tareas que el worker de Celery ejecutará en ciertos intervalos de tiempo.

Db.sqlite3

Es la base de datos generada por Django, el cual guardará de manera local la información del servidor.

Media

Esta carpeta contiene el siguiente archivo y subcarpeta:

- default.png: Es la imagen predeterminada que se le coloca como avatar a los usuarios recién registrados en la aplicación web del servidor.
- Profile_pics: Es una subcarpeta que contiene todas las imágenes utilizadas como avatar por los usuarios registrados en la aplicación web del servidor.

Aplicaciones del servidor

Server

Es el núcleo del servidor, un módulo creado por defecto al momento de crear el proyecto Django Server. En el momento de su creación el módulo “Server” trajo consigo una serie de archivos que fueron modificados durante el desarrollo del proyecto.

__pycache__

Es una carpeta que contiene bytecode. Cuando se ejecuta un programa en Python, lo primero que el intérprete hace es compilarlo en bytecode para simplificar y guardar en esta carpeta. Como programador, esta carpeta se ignora, ya que su única función en el sistema es hacer que el software se ejecute más rápido.

static

Esta carpeta contiene los archivos estáticos, como: imágenes, archivos CSS y Javascript; los cuales serán cargados por diferentes aplicaciones del servidor para evitar cargar muchas cosas a la vez, estos archivos estáticos serán manejados por la función que ofrece Django: `django.contrib.staticfiles`.

Entre los archivos estáticos que contiene esta carpeta se encuentran los siguientes:

Css

- Backup.css: Define la apariencia visual del botón de “Exportar en Excel (.xls)” en `backup.html`, como el tamaño del botón, el tamaño y color de la letra, entre otros. También contiene las funciones necesarias para que cambie de color cuando el usuario pase el puntero del mouse por encima.
- Dashboard.css: Define la apariencia visual de la tabla de control en `lab_one_dashboard.html`, como el tamaño de la letra, la separación entre las secciones, y la ubicación de cada sección e imagen en la plantilla.

- Gauge.css: Define la apariencia de los indicadores de nivel de temperatura, de humedad relativa y luminosidad en lab_one_dashboard.html; también, define la alineación del texto que se muestra debajo del indicador.
- Historical.css: Define la apariencia visual de la página, como el tamaño y posición de las gráficas, de la caja de selección, la separación entre las secciones, y la ubicación de las mismas en la plantilla.
- List.css: Define la apariencia visual de las listas y viñetas. Este archivo es utilizado por home.html y about.html
- Main.css: Define la apariencia de algunos elementos que pertenecen a base.html.
- Readings.css: Define la apariencia de las secciones en la tabla de control en lab_one_dashboard.html, así como el tamaño de la letra y posición del texto en cada sección.
- Switch.css: Define la apariencia de los botones que controlan las luces y aires acondicionados en lab_one_dashboard.html, es decir, cómo se verán cuando se enciende o apagan.
- Tables.css: Define la apariencia de las tablas en lab_two_dashboard.html, donde se muestra el registro histórico semanal.

Images

- Favicon.ico: Es un ícono que la aplicación web mostrará en cada página del servidor. La imagen que se utilizó pertenece a la Open Automation Software.
- Hum.png: Es una imagen del ícono de humedad relativa, utilizado por lab_one_dashboard.html.
- Lab_map.png: Es una imagen que muestra el mapa del laboratorio de prototipos, utilizado por lab_one_dashboard.html.
- Temp.png: Es una imagen del ícono de la temperatura, utilizado por lab_one_dashboard.html.

Js

- **Historical.js:** Añade las características interactivas de `lab_two_dashboard.html` para que funcione con websockets, y muestre el histórico de la semana de la temperatura, humedad relativa, luminosidad y presencia, junto a sus respectivas las gráficas.
- **Realtime.js:** Añade las características interactivas de `lab_one_dashboard.html` para que funcione como un panel de control del laboratorio, usando el canal websockets para recibir la información desde `consumers.py` a tiempo real.

Pdf

- **SevidorIOT_ManualDeUsuario.pdf:** Es un archivo PDF que contiene el manual de usuario, el cual es usado por `about.html` para mostrárselo a los usuarios registrados y administradores del servidor.
- **ServidorIOT_ManualTecnico.pdf:** Es un archivo PDF que contiene el manual de usuario, el cual es usado por `about.html` para mostrárselo a los administradores del servidor.

__init__.py

Es un método especial, reservado entre las clases de Python, que se llama cada vez que se instancia una clase. Se modificó para cargar la app Celery.py cada vez que Django se ejecute, tal y como se muestra en la figura 36.

```
# This will make sure the app is always imported when
# Django starts so that shared_task will use this app.
from Server.celery import app as celery_app

__all__ = ['celery_app']
```

Figura 35. **Server - __init__.py**

asgi.py

Como se puede ver en la figura 37, este archivo fue modificado específicamente para integrar las librerías Channels y Daphne a la aplicación, de modo que el servidor pueda trabajar con los protocolos de comunicación HTTP y Websockets.

```
import os

from channels.routing import ProtocolTypeRouter, URLRouter
from channels.auth import AuthMiddlewareStack
from django.core.asgi import get_asgi_application

from iot.routing import ws_urlpatterns

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'core.settings')

#application = get_asgi_application()

application = ProtocolTypeRouter({
    'http': get_asgi_application(),
    'websocket': AuthMiddlewareStack(
        URLRouter(ws_urlpatterns),
    )
    # Agregar otros protocolos de comunicacion
})
```

Figura 36. Server - asgi.py

celery.py

Este archivo fue creado para ejecutar funciones de las librerías Celery y django-celery-beats. Contiene un cronograma de las tareas que se ejecutarán en el servidor y el tiempo en que serán ejecutadas.

Siguiendo el tutorial del Red Eyed Coder Club en YouTube, se adaptó la configuración para definir la instancia Celery en el proyecto, tal y como se muestra en la figura 38.

```
# Librerías
import os
from celery import Celery
from celery.schedules import crontab

# Set the default Django settings module for the 'celery' program.
os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'Server.settings')

app = Celery('Server') #Nombre del proyecto

# Using a string here means the worker doesn't have to serialize
# the configuration object to child processes.
# - namespace = 'CELERY' means all celery-related configuration keys
# should have a 'CELERY_' prefix.
app.config_from_object('django.conf:settings', namespace='CELERY')

# Load task modules from all registered Django apps
app.autodiscover_tasks()
```

Figura 37. Configuración de Celery.py

Siguiendo la guía oficial de Celery, posteriormente se instaló la librería Django-celery-beat para ejecutar una lista de tareas, o tasks, definidas en el archivo tasks.py a ciertos intervalos de tiempos, agregando la configuración que se muestra en la figura 39, en el archivo Celery.py.

```

app.conf.beat_schedule = {
    'i2creader_60s': {
        'task': 'iot.tasks.i2creader', #Leemos los dispositivos iot por i2c
        # La tarea se ejecuta cada 1 minuto
        'schedule': crontab(minute='*/1')
    },
    'average_data_60m': {
        'task': 'iot.tasks.average_data',
        # La tarea se ejecuta cada 1 hora, en el minuto 1
        'schedule': crontab(minute=1, hour='*/1')
    },
    'turn_on_AC': {
        'task': 'iot.tasks.turn_on_AC',
        # La tarea se ejecuta a las 6:50 am, de Lunes a Viernes
        'schedule': crontab(minute=50, hour=6, day_of_week='mon,tue,wed,thu,fri')
    }
}

```

Figura 38. Configuración del cronograma de tareas, en Celery.py del módulo Server

Donde los tasks que ejecutara el worker, definidos en tasks.py del módulo iot son los siguientes:

- I2creader: Que se ejecutará cada 1 minuto.
- Average_data: Que se ejecutará cada 1 hora, en el minuto 1.
- Turn_on_AC: Que se ejecutará a las 6:50 am, de Lunes a Viernes.

settings.py

Este archivo representa el núcleo del proyecto, contiene los ajustes del servidor y el registro de todas las aplicaciones/módulos que se crearon, así como la localización de los ficheros estáticos, los detalles de configuración de la base de datos, entre otros. A lo largo del desarrollo del servidor se fue modificando el código que viene por default en el fichero, agregando lo siguiente:

- Se modificó LANGUAGE_CODE a “es-CL” para que el formato de las vistas del servidor se muestre en español.
- En TIME_ZONE se modificó a “América/Santiago” para que tomara de internet la hora de Santiago de Chile cada vez que se ejecutara el servidor.

- En `INSTALLED_APPS` se agregaron los módulos que se definieron en el diseño y que conforman el proyecto, también se agregó el nombre de las librerías instaladas.
- Se agregó las variables globales `ASGI_APPLICATION` y `CHANNEL_LAYERS` como parte de la implementación de las funciones de la librería Channels.
- Se agregó las opciones de configuración de Celery: `CELERY_TIMEZONE`, `CELERY_TASK_TRACK_STARTED`, `CELERY_TASK_TIME_LIMIT` y `CELERY_BROKER_URL`.
- Se agregó las rutas `"/static"` y `"/media"` para que el servidor pueda ubicar los ficheros correspondientes que se encuentran en estas rutas en cada ejecución.
- Se agregaron variables globales que servirían para la configuración del servidor como cliente MQTT.

urls.py

Contiene el esquema de URLs de la aplicación web que se muestra en la figura 40, diseñado para dar acceso al usuario a algunas páginas del servidor cuando se esté ejecutando, incluyendo la configuración necesaria para que Django pueda localizar la carpeta static, donde se encuentra los archivos estáticos y los avatares usados por los usuarios.

```
# Server/urls.py
from django.contrib import admin
from django.urls import path,include
from django.contrib.auth import views as auth_views
from django.conf import settings
from django.conf.urls.static import static
from users import views as user_views
from iot import views as iot_views
from iot.admin import server_site

urlpatterns = [
    path('admin/', server_site.urls),
    path('register/',user_views.register,name='server-register'),
    path('profile/',user_views.profile,name='server-profile'),
    path('lab-one/',iot_views.lab_one, name='server-lab-one'),
    path('lab-two/',iot_views.lab_two, name='server-lab-two'),
    path('login/',auth_views.LoginView.as_view(template_name='login.html'),name='server-login'),
    path('logout/',auth_views.LogoutView.as_view(template_name='logout.html'),name='server-logout'),
    path('', include('web_plataform.urls')), #Pagina principal
] + static(settings.STATIC_URL,document_root=settings.STATIC_ROOT)

if settings.DEBUG:
    urlpatterns += static(settings.MEDIA_URL, document_root=settings.MEDIA_ROOT)
```

Figura 39. Lista de urls creados en el módulo Server

Al buscar acceder a alguna de las URLs de la lista, se llama a la función correspondiente que se encuentra en views.py, que realiza el proceso correspondiente.

wsgi.py

Django trabaja con un Web Server Gateway Interface, WSGI, el cual es un estándar que permite escribir programas que puedan comunicarse a través del protocolo HTTP, por lo tanto, existe para todos los servidores y aplicaciones web.

El archivo estaba configurado como se muestra en la figura 41.

```
import os

from django.core.wsgi import get_wsgi_application

os.environ.setdefault('DJANGO_SETTINGS_MODULE', 'Server.settings')

application = get_wsgi_application()
```

Figura 40. Server - wsgi.py

Iot

Este módulo contiene todos los archivos y funciones que fueron desarrollados para la implementación del Internet de la Cosas en el prototipo, a saber: La página /lab-one para mostrar las lecturas los sensores y el estado de los actuadores en tiempo real, la página /lab-two para mostrar el registro histórico de la temperatura, humedad relativa, luminosidad/horas de encendido y presencia semanal en el laboratorio de prototipo. Así como también, las funciones necesarias para la implementación del protocolo de comunicación I2C, y la implementación de la librería Channels para la comunicación websockets entre Cliente-Servidor.

__pycache__

Es una carpeta que contiene bytecode. Cuando se ejecuta un programa en Python, lo primero que el intérprete hace es compilarlo en bytecode para simplificar y guardar en esta carpeta. Como programador, esta carpeta se ignora, ya que su única función en el sistema es hacer que el software se ejecute más rápido.

config/i2c.py

Config es una carpeta que contiene variables globales que sirven para la configuración de funciones que realizara el módulo iot, los cuales no fueron incluidas en settings.py

En esta carpeta se encuentra el archivo i2c.py que, como se muestra en la figura 42, contiene las direcciones de los dispositivos (esclavos) que contiene el bus I2C y con los que el Raspberry Pi se comunicará, como Maestro.

```
# I2C
# Direccion de los esclavos en el bus
I2C_SLAVE1_ADDRESS = 11 #0X0b ou 11
I2C_SLAVE2_ADDRESS = 12 #0X0c ou 12
I2C_SLAVE3_ADDRESS = 13 #0X0d ou 13
```

Figura 41. iot - config/i2c.py

Management/commands

Esta subcarpeta se creó para los comandos de Django `i2cwriter`, `i2creader`, `mqtt_connect` que sirvieron como base para desarrollar el servidor utilizando los protocolos de comunicación I2C y MQTT para enviar/recibir información de los dispositivos IoT del laboratorio de prototipos.

__pycache__

Es una carpeta que contiene bytecode. Cuando se ejecuta el programa, lo primero que el intérprete hace es compilarlo en bytecode para simplificar y guardar en esta carpeta. Esta carpeta se ignora, ya que su función en el sistema es hacer que el software se ejecute más rápido.

I2creader.py

Permite la comunicación entre el Raspberry Pi y el Arduino MEGA 2560 R3 que se encuentra conectado a los sensores, usando el protocolo de comunicación I2C; donde el Raspberry Pi es el Maestro y el Arduino es el Esclavo. Como se muestra en la figura 43, dentro del archivo se creó la clase `Command`, a la que se le añadió el método `handle()` que ejecutará un bucle donde: Limpiará la terminal con la función `cls()`, y ejecutará la función `read_module()` cada 10 segundos. A la función `read_module()` se pasa como referencia el valor de la variable global `I2C_SLAVE1_ADDRESS`, la cual contiene la dirección del Arduino: 11.


```
class Command(BaseCommand):  
  
    help = ('Muestra la lecturas de los sensores'  
            + 'del modulo 1')  
  
    def handle(self, *args, **kwargs):  
        try:  
            print("Conectando...")  
            while True:  
                cls() #limpia la pantalla  
                datos = read_module(I2C_SLAVE1_ADDRESS)  
                #save(datos)  
                time.sleep(60) # Tiempo de espera de 1 minuto  
        except KeyboardInterrupt:  
            print("La comunicacion se detuvo")
```

Figura 42. Clase Command de i2creader.py

La función `read_module()` permite al Raspberry Pi conectarse al esclavo 11 y recibir la lectura de los sensores conectados a ese módulo en un bloque de datos en byte, para luego imprimir en pantalla la información de manera ordenada. Para ejecutar el comando `i2creader`, se escribe en la terminal de Visual Studio Code lo siguiente:

```
python manage.py i2creader
```

I2cwriter.py

Permite la comunicación entre el Raspberry Pi y el Arduino UNO que se encuentra conectado a los actuadores usando el protocolo I2C, siendo el Raspberry Pi el maestro y el Arduino el esclavo. Como se muestra en la figura 44, dentro de `i2cwriter.py` se creó la clase `Command`, a la que se le añadió el método `handle()` que ejecutará un bucle donde: Limpiará la pantalla con la función `cls()` y permitirá al usuario ingresar el número asociado al evento que se desea llevar a cabo, para que de último se ejecutó la función `write_module()` y haya que esperar 10 segundos. A la función `write_module()` se pasa como referencia el valor de la variable global o `I2C_SLAVE2_ADDRESS` que contienen la dirección: 12.

```
class Command(BaseCommand):
    def handle(self, *args, **kwargs):
        try:
            while True:
                cls()
                #print("Conectando...")
                #print("Activando actuadores...")
                flag = True
                while (flag):
                    print("Luces (1-2)")
                    print("AC (3-4)")
                    relaySelect = int(input("Opcion seleccionada: "))
                    if ((relaySelect > 0) and (relaySelect < 5)):
                        status = int(input("Encender/Apagar (1-0): "))
                        if ((status == 1) or (status==0)):
                            message = (relaySelect * 10) + status
                            #print("orden: ",str(message))
                            write_module(I2C_SLAVE2_ADDRESS, message)
                            flag = False
                        time.sleep(10) #Tiempo de espera de 10 segundos
                except KeyboardInterrupt:
                    print("La comunicacion se detuvo")
```

Figura 43. Clase `Command` de `i2cwriter.py`

Como se muestra en la figura 45, la función `write_module()` permite al Raspberry Pi conectarse al esclavo 12 con la librería `smbus2`, y enviar un valor entero en bytes al Arduino correspondiente.

```
# Librerias
import os
import smbus2 as smbus
import time

from django.core.management.base import BaseCommand

# Direccion bus del servidor
DEVICE_BUS = 1
# Direccion de los esclavos en el bus
I2C_SLAVE2_ADDRESS = 12 #0X0c ou 12

#Instancia del bus I2C
I2Cbus = smbus.SMBus(1)

#Borra el texto de la terminal
def cls():
    os.system('cls' if os.name=='nt' else 'clear')

#Envia un comando en concreto al modulo
def write_module(slave_address,message):
    with smbus.SMBus(1) as I2Cbus:
        try:
            I2Cbus.write_byte_data(slave_address, 0, message)
        except Exception as e:
            print("Error remoto i/o")
            print(e)
```

Figura 44. *iot* - Funciones `cls` y `write_module` de `i2cwriter.py`

Para ejecutar `i2cwriter.py`, se escribe en la terminal de Visual Studio Code lo siguiente:

```
python manage.py i2cwriter
```

mqtt_connect.py

Permite la comunicación entre el Raspberry Pi y el ESP8266 NodeMCU conectado a los sensores, a través del protocolo MQTT. Como se muestra en la figura 46, dentro del archivo se creó la clase `Command`, donde se añadió el método `handle()` que ejecutará un bucle donde: Se crea una instancia `client`, se intenta conectar al bróker MQTT con la función `connect_mqtt()`,

luego se llama a la función `loop_start()` para crear un hilo, que llama un método `loop`, o bucle, cada cierto tiempo sin ser bloqueado por los otros procesos que se ejecuten en paralelo en Django. En el loop se limpiará la pantalla, se imprimirá el número de lectura realizado para comprobar que no se repiten mensajes, y se ejecutará la función `subscribe()` que permite suscribirse a unos tópicos definidos e imprime los mensajes recibidos por el bróker MQTT. De último, el loop se detendrá por 10 segundos antes de volver a ejecutarse.

```
class Command(BaseCommand):

    help = ('Muestra la lecturas de los sensores'
            + 'del modulo 1')

    def handle(self, *args, **kwargs):
        cont = 1
        try:
            client = connect_mqtt()
            client.loop_start()
            print("Conectando...")
            #subscribe(client)
            while True:
                cls()
                print("Lectura " + str(cont) + ":")
                subscribe(client)
                cont= cont + 1
                sleep(10)
        except KeyboardInterrupt:
            print("La comunicacion se detuvo")
```

Figura 45. Clase `Command` de `mqtt_connect.py`

Para ejecutar `mqtt_connect.py`, se escribe en la terminal de Visual Studio Code lo siguiente:

```
python manage.py mqtt_connect.py
```

Migrations

Esta carpeta sirve para que Django procese los cambios que se han realizado a los Modelos que se encuentran en el esquema de la base de datos del proyecto, como agregar campos, borrar modelos, entre otros. Esta carpeta se crea de manera automática luego de que se

ejecuta el comando `makemigrations`, y crea nuevos archivos con el mismo comando. Las migraciones de esta carpeta se aplican en el proyecto con el comando `migrate`.

Templates

Esta carpeta se crea para que Django pueda generar Vistas de manera dinámica. Por lo tanto, esta carpeta contiene las plantillas HTML del módulo `iot`, a saber:

- `Lab_one_dashboard.html`: Archivo que sirve para generar la vista que muestra un panel de control del laboratorio de prototipos, que permitirá visualizar la temperatura interior y exterior, la humedad relativa interior y exterior, los niveles de temperatura y humedad relativa, los niveles de luminosidad, el estado del seguro de la puerta, las luces y aire acondicionados; detectar presencia y fugas; y encender/apagar las luces y aire acondicionados.
- `Lab_two_dashboard.html`: Archivo que sirve para generar la vista que mostrará los registros históricos de la temperatura, humedad relativa, luminosidad/horas de encendido y presencia semanal en el laboratorio de prototipos.

`__init__.py`

Es un archivo usado para indicar que el directorio presente es un paquete de Python, volviendo posible la importación de módulos a sub-paquetes del directorio en el que se encuentra. También puede usarse para definir código de inicialización o variables que se ejecuten cuando el paquete sea importado.

`admin.py`

Este archivo usa los modelos que se codificaron en `models.py` del módulo `iot` para construir automáticamente dentro del Área Administrativa un sitio que se pueda usar para crear, consultar, actualizar y borrar registros. Todo esto con la finalidad de ahorrar tiempo de desarrollo y poder probar los modelos para verificar que los datos son correctos.

En este archivo también se personalizó el Área Administrativa para que en la cabecera mostrara el nombre “Servidor IoT – Área Administrativa” y mostrará una lista de modelos específicos en el sitio (ver en la figura 47).

```
from django.contrib import admin
from django.contrib.auth import get_user_model
from django.contrib.auth.models import User, Group

from .models import Temperatura_exterior, Humedad_exterior, Humedad_piso
from .models import Interior, Promedio_temperatura, Promedio_luminosidad, Promedio_presencia, Promedio_humedad

class ServerAdminArea(admin.AdminSite):
    site_header = 'Servidor IoT - Área Administrativa'
    site_title = 'Servidor IoT'
    index_title = 'Laboratorio'

server_site = ServerAdminArea(name='ServerAdmin')

server_site.register(Temperatura_exterior)
server_site.register(Humedad_exterior)
server_site.register(Humedad_piso)
server_site.register(Interior)
server_site.register(User)
server_site.register(Group)
server_site.register(Promedio_temperatura)
server_site.register(Promedio_humedad)
server_site.register(Promedio_luminosidad)
server_site.register(Promedio_presencia)
```

Figura 46. `iot - admin.py`

apps.py

Como se puede ver en la figura 48, este archivo sirve para realizar configuraciones en la aplicación “iot”, a saber: Incluir los modelos creados en `models.py` dentro de la base de datos, que pueden ser accedidos desde el Área Administrativa en la sección “iot” e implementar las configuraciones realizadas en `admin.py` en el sitio.

```
from django.apps import AppConfig
from django.contrib.admin.apps import AdminConfig
class ServerAdminConfig(AdminConfig):
    default_site = 'iot.admin.ServerAdminArea'
class IotConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'iot'
```

Figura 47. iot - apps.py

consumers.py

Haciendo uso de la librería Channels, en este archivo se encuentran los Consumers que manejarán los canales websocket del servidor. Un consumer es una abstracción de un canal o channel en forma de clase, esta clase implementa métodos que se encargarán de manejar los eventos de los usuarios.

Para el desarrollo del proyecto se crearon dos consumidores:

- **Lab2Consumer:** Es un consumidor, que hereda la clase `AsyncWebsocketConsumer`, la cual funciona igual que una clase `WebsocketConsumer` pero de manera asíncrona. Está enlazado con `lab_one_dashboard.html`. A diferencia de un `WebsocketConsumer`, sus métodos: `connect()`, `disconnect()` y `receive()` están definidos de manera asíncrona, con la sintaxis `def async`; para llamar funciones síncronas desde esta clase se usa la función `await`. Entre métodos asíncronos que conforman el consumidor podemos encontrar:

Connect(): Evento que se ejecutará cuando un usuario se conecte al canal websockets. Con la función `self.accept()` acepta la conexión, solo si se da la condición de que el usuario haya iniciado sesión previamente en la aplicación web. Luego de conectarse exitosamente, el usuario es agregado a un `channel_layer` definido como “`lab_event`”. Seguidamente, se llevan a cabo los siguientes procesos para que al momento de conectarse al URL “`lab-one/`”, la vista no se encuentre vacía:

- Intentar llamar al método `get_lab_data()` definido en la clase, con el fin de obtener el último elemento registrado en la tabla “Interior” en la base de datos, con el cual crea un objeto en formato JSON para poder enviarlo como mensaje por el canal websockets con el método `send()`.
- Intentar llamar al método `get_temp_ext_data()` definido en la clase, con el fin de obtener el último elemento registrado en la tabla `Temperatura_exterior` en la base de datos, con el cual crea un objeto en formato JSON para poder enviarlo como mensaje por el canal websockets con el método `send()`.

- Intentar llamar al método `get_hum_ext_data()` definido en la clase, con el fin de obtener el último elemento registrado en la tabla `Humedad_exterior` en la base de datos, con el cual crea un objeto en formato JSON para poder enviarlo como mensaje por el canal websockets con el método `send()`.
- Y de último, intentar llamar al método `get_hum_floor_data()` definido en la clase, con el fin de obtener el último elemento registrado en la tabla `Humedad_piso` en la base de datos, con el cual crea un objeto en formato JSON para poder enviarlo como mensaje por el canal websockets con el método `send()`.

`Disconnect()`: Desconecta al usuario del channel_layer “lab_event”.

`Receive()`: Se ejecuta cuando un websocket envía información. Los mensajes recibidos estarán dentro de un objeto JSON, por lo que se utilizara el método `json.loads()` de la librería `Json` para convertir la cadena en un diccionario/registro de Python, guardando la información en una variable llamada `text_data_json`. El registro `text_data_json` estará formado por los siguientes elementos: `message` y `type`; los cuales serán guardados en unas variables individuales llamadas `message` y `type_message` respectivamente. Si la variable `type_message` es igual a “action_event” se llamará a la función `write_module` del archivo `i2cwriter.py` que definimos anteriormente, pasando por referencia la dirección 12 (que pertenece al Arduino UNO R3) y `message` (que será la acción o evento que llevará a cabo el Arduino UNO R3)

El mapeo objeto-relacional, ORM, de Django es una pieza de código síncrona, por lo que para acceder a la base de datos desde el consumidor `Lab2Consumer` fue necesario crear un grupo de métodos que usaran la función `database_sync_to_async` decorator, tal y como se muestra de ejemplo en la figura 49, donde la función síncrona `get_temp_ext_data()` devuelve el contenido más reciente registrado en la tabla `Temperatura_exterior`.

```
@database_sync_to_async
def get_temp_ext_data(self):
    myData = Temperatura_exterior.objects.order_by('dia_hora_leida').last()
    return myData #El ultimo contenido registrado en la tabla
```

Figura 48. Acceso a la tabla `Temperatura_exterior` de la base de datos con Channels

Lo mismo se aplicó a los métodos síncronos: `get_lab_data()`, `get_hum_ext_data()` y `get_hum_floor_data()`; los cuales manejan diferentes tablas de la base de datos y devuelven el último valor registrado.

Entre los diferentes métodos que puede ejecutar cada consumer que se conecte al `Channel_layer`, donde enviará un tipo de mensaje a través del canal websockets, tenemos los que se definen en la tabla 6.

Tabla 6. *Tipos de mensajes que pueden enviarse por el Channel_layer lab_events*

Tipos de mensaje	Mensaje
<code>Update_lab_data()</code>	Contiene la última información agregada a la tabla Interior de la base de datos, enviada desde el consumidor en formato JSON
<code>Update_temp_ext()</code>	Contiene la última información agregada a la Temperatura_exterior de la base de datos, enviada desde el consumidor en formato JSON
<code>Update_hum_ext()</code>	Contiene la última información agregada a la Humedad_exterior de la base de datos enviada desde consumidor en formato JSON
<code>Update_hum_floor()</code>	Contiene la última información agregada a la Humedad_piso de la base de datos enviada desde consumidor en formato JSON
<code>Action_event()</code>	Contiene el mensaje enviado desde <code>historical.js</code> , por un usuario que se haya conectado al grupo <code>lab_event</code>

- **RecordConsumer:** Es un consumidor que hereda la clase `WebsocketConsumer`. Esta enlazado con `lab-two-dashboard.html`. Funciona de manera síncrona y sus métodos principales son:

Connect(): Evento que se ejecutará cuando un usuario se conecte al canal websockets. Con la función `self.accept()` acepta la conexión, solo si se da la condición de que el usuario haya iniciado sesión previamente en la aplicación web.

Disconnect(): Desconecta al usuario del websocket.

Receive(): Recibe los mensajes del websocket, donde `message` es objeto JSON que al ser decifrado contiene un número entre 0 y 3; la cual pasa por parámetro como la variable `option` junto con el número 0 como la variable `week`, que representa la semana actual, para ejecutar una función llamada `get_previousweek_data()`.

Get_previous_week_data(): Dependiendo del valor de la variable “option”, extrae información de un campo en específico de la tabla Interior, que fue registrada en la semana entre las 7 am y 7 pm en la base de datos. Si la opción es 0, devuelve la temperatura; si la opción es 1, devuelve la humedad relativa; si la opción es 2, devuelve la luminosidad; y si la opción es 3, devuelve la presencia.

Tal y como se muestra de ejemplo en la figura 50, se escribió el siguiente código para obtener la temperatura del laboratorio registrada en la semana, calculando el rango de fechas entre lunes y viernes con la ayuda de funciones de la librería datetime; y usar la función filter() de Django para extraer de la tabla las lecturas que fueron tomadas entre las 7 am y 7 pm. Para guardar el resultado en un arreglo vacío definido como myData[].

```
def get_previousweek_data(self, option, week):
    current_date = datetime.today() #Obtenemos la fecha y tiempo de ahora
    current_weekday = current_date.weekday() # Obtenemos el día de la semana de hoy
    past_days = current_weekday + week # Numero de días a retroceder, hasta llegar al lunes de la semana correspondiente

    previous_week = current_date - timedelta(days=past_days) # Nos ubicamos el día lunes de esa semana
    end_of_week = previous_week + timedelta(days=5) # Día viernes de esa semana

    date_begin = previous_week.date() # Fecha del lunes de esa semana
    date_end = end_of_week.date() # Fecha del sábado de esa semana

    myData = []

    if (option==0):
        # Extrae la información entre el día lunes y viernes de esa semana
        readings = Promedio_temperatura.objects.filter(dia_inicio__range=[date_begin,date_end])

        if (readings):
            for reading_info in readings:
                # Revisa si la lectura es de entre las 7 am y las 7 pm
                if ((reading_info.dia_inicio.hour >= 7) and (reading_info.dia_inicio.hour < 19)):
                    dataInfo = {
                        "value": str(reading_info.temp),
                        "fecha": reading_info.dia_inicio.strftime('%d/%m/%Y'),
                        "dia_semana": str(reading_info.dia_inicio.weekday()),
                        "hora_inicio": reading_info.dia_inicio.strftime('%H:%M'),
                        "hora_fin": reading_info.dia_fin.strftime('%H:%M')
                    }
                    myData.append(dataInfo)
```

Figura 49. Código de la función get_previousweek_data() para obtener la temperatura de lunes a viernes, de 7am a 7pm

Seguidamente, tal y como se ve en la figura 64, el arreglo myData sería guardado en un objeto JSON que sería enviado por el canal de websockets.

```
#print(myData)
# Envía los datos de esa semana
self.send(text_data=json.dumps({
    'type': 'user_request',
    'request': option,
    'weeks': week // 7,
    'date_begin': date_begin.strftime('%d/%m/%Y'),
    'date_end': date_end.strftime('%d/%m/%Y'),
    'message': myData}))
```

Figura 50. Mensaje enviado por el URL ws/lab-two/

custom_datetime.py

Este archivo se creó para comprobar el funcionamiento de la librería datetime y realizar pruebas, específicamente sus funciones date, timedelta y datetime. Codificando funciones con para imprimir en la terminal la fecha de hoy, de ayer, de hace una semana, el tiempo de hace una hora, el día de la semana en el calendario, entre otros.

La sentencia `if __name__ == "__main__":` nos permite manejar el código, ejecutándolo individualmente desde la terminal, sin la necesidad de ejecutar el servidor.

models.py

En este archivo están definidos los modelos de la base de datos, utilizados por la aplicación “iot”, a saber:

- **Temperatura_exterior:** Se encargará de guardar valor de la temperatura exterior que recibe del ESP8266 NodeMCU en caracteres, con la fecha-hora en la que se guardó la lectura. Cuando se ingrese al sitio Área administrativa, se mostrará los registros ordenados desde el más reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Hora: minutos leída – Temperatura - C
- **Humedad_exterior:** Se encargará de guardar el valor de la humedad exterior que recibe del ESP8266 NodeMCU en caracteres, con la fecha-hora en la que se guardó la lectura. Cuando se ingrese al sitio Área administrativa, se mostrará los registros ordenados desde el más reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Hora: minutos leída – Humedad relativa - %

- **Humedad_piso:** Se encargará de guardar cada 5 minutos el valor de la humedad de piso que recibe del ESP8266 NodeMCU en caracteres, donde los valores variaran entre “0” y “1”, donde “0” representa el estado “seco” y 1 representa el estado “mojado”, con la fecha-hora en la que se guardó la lectura. Cuando se ingrese al sitio Área administrativa, se mostrará los registros ordenados desde el más reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Hora: minutos leída – Humedad de piso (“Seco” si es “0” o “Mojado” si es “1”, “Error” si no es ninguno de los dos anteriores)
- **Interior:** Se encargara de guardar las variables que recibe del Arduino MEGA 2560, tales como: temperatura, humedad relativa, presencia (que varía entre 1 y 0, donde 1 significa que hubo presencia y 0 que no hubo presencia), luminosidad de la sección 1 y de la sección 2 donde valores entre 0 y 255, donde entre mayor sea el valor menos luminosidad hay), estado del seguro de la puerta (que varía entre 1 y 0, donde 1 significa que está bloqueado y 0 desbloqueado), estado del aire acondicionado 1 y el aire acondicionado 2 (que varía entre 1 y 0, donde 1 significa encendido y 0 significa apagado) y la fecha-hora que el Raspberry Pi recibió la información. Cuando se ingrese al sitio Área administrativa, se mostrará los registros ordenados desde el más reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Hora: minutos leída – Temperatura – C - Humedad relativa - % - Presencia
- **Promedio_temperatura:** Se encarga de guardar el resultado del promedio de temperatura de un grupo de muestras; donde dia_inicio representa la fecha-hora que se inició a tomar las lecturas y el dia_fin representa fecha-hora que se terminó de tomar las lecturas. Por ejemplo: 18 °C, de 8:00 am a 9:00 am. Cuando se ingrese al sitio Área administrativa, se mostrará los registros ordenados desde el más reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Hora: minutos inicio – Hora: minutos fin - Temperatura - C
- **Promedio_humedad:** Se encarga de guardar el resultado del promedio de temperatura de un grupo de muestras; donde dia_inicio representa la fecha-hora que se inició a tomar las lecturas y el dia_fin representa fecha-hora que se terminó de tomar las lecturas. Por ejemplo: 68%, de 8:00 am a 9:00 am. Cuando se ingrese al sitio Área administrativa, se mostrará los

registros ordenados desde el más reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Hora: minutos inicio – Hora: minutos fin – Humedad relativa - %

- **Promedio_presencia:** Se encarga de guardar el resultado del promedio de presencia de un grupo de muestras; donde `dia_inicio` representa la fecha-hora que se inició a tomar las lecturas y el `dia_fin` representa fecha-hora que se terminó de tomar las lecturas. Por ejemplo: 1 (Hay presencia), de 8:00 am a 9:00 am. Cuando se ingrese al sitio Área administrativa, se mostrará los registros ordenados desde el más reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Hora: minutos inicio – Hora: minutos fin – Presencia (“Si” si el valor es “1”, “No” si el valor es “0”, “Error” si no es ninguno de los dos anteriores).
- **Promedio_luminosidad:** Se encarga de guardar el resultado del promedio de luminosidad de un grupo de muestras y el objeto/lugar de donde se tomaron; donde `dia_inicio` representa la fecha-hora que se inició a tomar las lecturas y el `dia_fin` representa fecha-hora que se terminó de tomar las lecturas. Por ejemplo: Sección 1, 143, de 8:00 am a 9:00 am. Cuando se ingrese al sitio Área administrativa, se mostrará los registros ordenados desde el mas reciente hasta el más viejo, mostrando el día de la semana – Fecha leída – Sección - Hora: minutos inicio – Hora: minutos fin – valor de la luminosidad

También se hace uso de la función que ofrece Django: `timezone`; para registrar y mostrar en la zona administrativa la fecha-hora registrada según la zona horaria.

routing.py

En este archivo se enlaza los consumers creados en `consumers.py` con los websockets, tal y como se muestra en la figura 65. Donde todas las conexiones a la url `ws://127.0.0.1:8000/ws/lab-one/` crearán una instancia de `Lab2Consumer`; y donde todas las conexiones a la url `ws:// 127.0.0.1:8000/ws/lab-two/` crearán una instancia de `RecordConsumer`.

Tal y como podemos ver en la figura 52, es importante que mientras se enruta los consumidores, se llama al método `as_asgi()`, ya que esto retornará un ASGI wrapper application que instanciara un nuevo consumidor por cada conexión; que seria similar al método de Django

as_view(), el cual juega un rol similar para las instancias pre-request o las vistas basadas en clases.

```
from django.urls import re_path
from .consumers import Lab2Consumer, RecordConsumer

ws_urlpatterns = [
    re_path(r'ws/lab-one/', Lab2Consumer.as_asgi()),
    re_path(r'ws/lab-two/', RecordConsumer.as_asgi()),
]
```

Figura 51. **iot - routing.py**

tasks.py

Este archivo contiene las tareas, o tasks, que realizará el worker de Celery. Los tasks que ejecutara el worker, definidos en tasks.py del módulo iot son los siguientes:

- **I2creader:** Intenta comunicarse con esclavo 11, el Arduino MEGA2560 R3, para recibir la lectura de los sensores y guardarla en la base de datos; con una función definida dentro de tasks.py llamada save(), a la que se pasa la información como un objeto JSON. La función i2creader() se ejecuta cada 1 minuto.
- **Average_data:** Calcula el promedio de la temperatura, humedad relativa, luminosidad y presencia para guardarla en las tablas Promedio_temperatura, Promedio_humedad, Promedio_luminosidad y Promedio_presencia. Para ello, se extrae la información que fue guardada en la tabla Interior entre la hora anterior actual, en el minuto 0, y la hora actual en el minuto 0, 59 segundos y 59 milisegundos. Como por ejemplo: 14:00:00:00 – 15:00:59:59. Si se encontraron resultados, se separa los resultados en 4 arreglos: temperature, humidity, lum1, lum2 y mov; para entonces calcula los promedios de manera separada, guardarlas en su respectiva tabla y de último borrar la información de la tabla Interior, cuyo campo dia_hora_leida sea menor a la hora actual. La función average_data() se ejecuta cada 1 hora, en el minuto 1.
- **Turn_on_AC:** Utiliza la función write_module() declarada en i2cwriter.py para enviar dos mensajes a la dirección del esclavo 12, el Arduino UNO, para indicarle que debe encender

los aires acondicionados. La función `turn_on_AC()` se ejecuta a las 6:50 am, de Lunes a Viernes.

Dentro del archivo `task.py` se declararon unos decorator: `receiver(post_save, sender)`, que permite a algunas funciones ejecutarse en el momento que una tabla dentro de la base de datos recibe información nueva. También se declaró una variable global llamada `channel_layer`, que recibe la función `get_channel_layer()` de la librería `Channels`, permitiendo enviar mensajes a una `channel_layer` fuera del consumer. Entre las funciones que fueron declaradas con el decorator mencionado se encuentran:

- `Event_post_add`: Cuando ingresa nueva información a la tabla Interior (Lab) de la base de datos, extrae el último dato registrado para enviarla en un objeto JSON al `channel layer lab_events`, como se muestra en la figura 53.


```

@receiver(post_save, sender=Lab)
def event_post_add(sender, instance, created, **kwargs):
    if created:
        print ("Datos del laboratorio registrados")

        # Obtenemos informacion de la base de datos
        db_data = Lab.objects.order_by('dia_hora_leida').last()

        # Obtenemos la fecha-hora en la que la informacion fue leida
        some_datetime = db_data.dia_hora_leida
        iso_datetime = some_datetime.isoformat() #Transformandola a formato ISO para guardarla en JSON

        dict_data = {
            'type': 'update_lab_data',
            'temperature': db_data.temp,
            'humidity': db_data.hum,
            'presence': db_data.presencia,
            'lum1': db_data.lum_1,
            'lum2': db_data.lum_2,
            'door': db_data.seg_puerta,
            'ac1': db_data.AC_1,
            'ac2': db_data.AC_2,
            'iso_datetime': iso_datetime
        }
        # Enviamos la data al grupo 'lab_events'
        async_to_sync(channel_layer.group_send)('lab_events',dict_data)

```

Figura 52. Función event_post_add() del archivo task.py del módulo iot

- Event_post_temp_ext: Cuando ingresa nueva información a la tabla Temperatura_exterior de la base de datos, extrae el último dato registrado para enviarla en un objeto JSON al channel layer lab_events.
- Event_post_hum_ext: Cuando ingresa nueva información a la tabla Humedad_exterior de la base de datos, extrae el último dato registrado para enviarla en un objeto JSON al channel layer lab_events.
- Event_post_hum_floor: Cuando ingresa nueva información a la tabla Humedad_piso de la base de datos, extrae el último dato registrado para enviarla en un objeto JSON al channel layer lab_events.

tests.py

Es un archivo creado automáticamente por Django después de crear la aplicación, que sirve para realizar pruebas adicionales automatizadas.

Urls.py

Contiene el esquema de URLs de la aplicación “iot”, que se muestra en la figura 54, diseñado para dar acceso al usuario a las páginas que se encuentran en la carpeta templates: lab_one_dashboard.html y lab_two_dashboard.html, cuando el servidor se esté ejecutando.

```
# web_plataform/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('lab-one/', views.lab_one, name='server-lab-one'),
    path('lab-two/', views.lab_two, name='server-lab-two'),
]
```

Figura 53. iot - urls.py

Views.py

Este archivo contiene funciones que se encargan de recibir la web request del usuario y retornar una web response en forma de contenido HTML. Tal y como se ve en la figura 55, entre las funciones que podemos encontrar en este archivo se encuentra:

lab_one: Posee un decorador @login_required que pasa por argumento una señal (signal) que contiene la url al que el usuario accederá si no ha iniciado sesión con su cuenta: “login/”. Si el usuario ha iniciado sesión, tendrá acceso a la página lab_one_dashboard.html.

lab_two: Posee un decorador @login_required que pasa por argumento una señal (signal) que contiene la url al que el usuario accederá si no ha iniciado sesión con su cuenta: “login/”. Si el usuario ha iniciado sesión, tendrá acceso a la página lab_two_dashboard.html.

```

from django.shortcuts import render
from django.contrib.auth.decorators import login_required

@login_required(login_url='/login/')
def lab_one(request):
    return render(request, 'lab_one_dashboard.html', {'title': 'Laboratorio_uno'})

@login_required(login_url='/login/')
def lab_two(request):
    return render(request, 'lab_two_dashboard.html', {'title': 'Laboratorio_dos'})

```

Figura 54. `iot - views.py`

Mqtt

Este módulo contiene todos los archivos y funciones que fueron desarrollados para la implementación del protocolo de comunicación MQTT en el prototipo.

`__pycache__`

Es una carpeta que contiene bytecode. Cuando se ejecuta un programa en Python, lo primero que el intérprete hace es compilarlo en bytecode para simplificar y guardar en esta carpeta. Como programador, esta carpeta se ignora, ya que su única función en el sistema es hacer que el software se ejecute más rápido.

Migrations

Esta carpeta sirve para que Django procese los cambios que se han realizado a los Modelos que se encuentran en el esquema de la base de datos del proyecto, como agregar campos, borrar modelos, entre otros. Esta carpeta se crea de manera automática luego de que se ejecuta el comando `makemigrations`, y crea nuevos archivos con el mismo comando. Las migraciones de esta carpeta se aplican en el proyecto con el comando `migrate`.

`__init__.py`

Es un archivo usado para indicar que el directorio presente es un paquete de Python, volviendo posible la importación de módulos a sub-paquetes del directorio en el que se encuentra. También puede usarse para definir código de inicialización o variables que se ejecuten cuando el paquete sea importado.

Admin.py

Este archivo usa los modelos que sean codificados en `models.py` del módulo “`mqtt`” para construir automáticamente dentro del Área Administrativa un sitio que se pueda usar para crear, consultar, actualizar y borrar registros. Todo esto con la finalidad de ahorrar tiempo de desarrollo y poder probar los modelos para verificar que los datos son correctos.

Apps.py

Este archivo contiene las funciones necesarias para implementar el protocolo MQTT como un background thread, cada vez que se ejecuta el servidor.

Primero, existe una clase llamada `MqttClient()`, que inicia un background thread cuando el servidor está en ejecución. A esta clase se le pasa por parámetros la IP del broker 192.168.100.18, el puerto 1883 y un arreglo que contiene los tópicos a los que se suscribirá el servidor: “`esp8266/temperature`”, “`esp8266/humidity`”, “`esp8266/water`”. Para que la clase conservara sus atributos más importantes a lo largo de la ejecución del servidor se utilizó la función `super` de Python.

Al iniciar el hilo de la clase `MqttClient()` se ejecuta el método `connect_to_broker()`, que configura la instancia cliente para conectarse al bróker de manera asíncrona, teniendo acceso a este con el usuario “`rasp-broker`” y la clave “`ucab*ucab`”, seguidamente, cuando ya esté conectado, se suscriba a los tópicos definidos.

De esta forma, mientras el servidor estuviese en ejecución, la aplicación web también estaría conectándose al bróker de manera asíncrona. Este proceso se realiza como un background thread en Django, con el fin de que el servidor pueda ejecutar sus otras funciones sin esperar a que las tareas asociadas a MQTT terminen.

Cada vez que el servidor recibiera un mensaje de los tópicos, la información sería guardada en la base de datos en el modelo correspondiente, junto con la fecha y hora en la que se recibió el mensaje, tal y como se muestra en la tabla 7.

Tabla 7. *Tópicos y modelo de la base de datos asociada*

Tópico	Modelo de la base de datos asociada
esp8266/temperature	Temperatura_exterior
esp8266/humidity	Humedad_exterior
esp8266/water	Humedad_piso

Tras guardar el mensaje recibido desde el bróker en la base de datos, el mensaje anterior a este es borrado de la base de datos; pues solo se necesita conocer el estado más reciente de la temperatura y la humedad relativa exterior cada vez, y si en el momento el piso este húmedo o no.

Models.py

En este archivo se definen los modelos de la base de datos, utilizados por la aplicación “mqtt”.

Tests.py

Es un archivo creado automáticamente por Django después de crear la aplicación, que sirve para realizar pruebas adicionales automatizadas.

Views.py

Es un archivo que contiene funciones que se encargan de recibir la web request del usuario y retornar una web response en forma de contenido HTML.

Users

En este donde se encuentra en código Python las funciones necesarias para generar el formulario que permite el registro de usuarios en el servidor.

__pycache__

Es una carpeta que contiene bytecode. Cuando se ejecuta un programa en Python, lo primero que el intérprete hace es compilarlo en bytecode para simplificar y guardar en esta

carpeta. Como programador, esta carpeta se ignora, ya que su única función en el sistema es hacer que el software se ejecute más rápido.

Migrations

Esta carpeta sirve para que Django procese los cambios que se han realizado a los Modelos que se encuentran en el esquema de la base de datos del proyecto, como agregar campos, borrar modelos, entre otros. Esta carpeta se crea de manera automática luego de que se ejecuta el comando `makemigrations`, y crea nuevos archivos con el mismo comando. Las migraciones de esta carpeta se aplican en el proyecto con el comando `migrate`.

Templates

Esta carpeta se crea para que Django pueda generar Vistas de manera dinámica. Por lo tanto, esta carpeta contiene las plantillas HTML del módulo `users`, a saber:

- `Login.html`: Archivo que genera la vista con el formulario que permitirá al usuario iniciar sesión en la aplicación web.
- `Logout.html`: Archivo que genera una vista cuando el usuario registrado ha cerrado sesión en la aplicación web.
- `Profile.html`: Archivo que genera la vista con el formulario que contiene los datos del usuario que ha iniciado sesión.
- `Register.html`: Archivo que genera la vista con el formulario que permitirá el registro de usuario en la aplicación web.

`__init__.py`

Es un archivo usado para indicar que el directorio presente es un paquete de Python, volviendo posible la importación de módulos a sub-paquetes del directorio en el que se encuentra. También puede usarse para definir código de inicialización o variables que se ejecuten cuando el paquete sea importado.

Admin.py

Como se muestra en la figura 56, este archivo usa el modelo que se codificó en models.py del módulo users para construir automáticamente dentro del Área Administrativa un sitio que se pueda usar para crear, consultar, actualizar y borrar registros. Todo esto con la finalidad de ahorrar tiempo de desarrollo y poder probar los modelos para verificar que los datos son correctos.

```
from django.contrib import admin
from .models import Perfil
from iot.admin import server_site

server_site.register(Perfil)
```

Figura 55. users - admin.py

Apps.py

Como se puede ver en la figura 57, este archivo sirve para realizar configuraciones en la aplicación “users”, a saber: Incluir los modelos creados en models.py dentro de la base de datos, que pueden ser accedidos desde el Área Administrativa en la sección “users” e implementar las configuraciones realizadas en admin.py en el sitio. Además, cuando se cargue la aplicación cargará las funciones creadas en signals.py.

```
from django.apps import AppConfig

class UsersConfig(AppConfig):
    default_auto_field = 'django.db.models.BigAutoField'
    name = 'users'

    def ready(self):
        import users.signals
```

Figura 56. users - apps.py

Forms.py

Como se muestra en la figura 58, ste archivo contiene las clases UserRegisterForm y UserUpdateForm, que representan la lógica de los formularios de “Registrar” y “Editar perfil”

que llena el usuario para dichos procesos, que incluye datos como: username, email, password1, password2, y avatar. Para facilitar el proceso de creación de usuario y validación de los campos, se importó el formulario de creación de usuario que viene en Django: UserCreationForm; también se importó la clase Form de Django para la creación del formulario, que contiene atributos con predefinidos, tales como: email.

```
from django import forms
from django.contrib.auth.models import User
from django.contrib.auth.forms import UserCreationForm
from .models import Perfil

class UserRegisterForm(UserCreationForm):
    email = forms.EmailField()

    class Meta:
        model = User
        fields = ['username', 'email', 'password1', 'password2']

class UserUpdateForm(forms.ModelForm):
    email = forms.EmailField()

    class Meta:
        model = User
        fields = ['username', 'email']

class ProfileUpdateForm(forms.ModelForm):
    class Meta:
        model = Perfil
        fields = ['avatar']
```

Figura 57. users - forms.py

Models.py

En este archivo define el modelo Perfil en la base de datos. Ubicado en la aplicación users, para guardar los avatares de usuario haciendo uso de la librería Pillow, o PIL, (ver la figura 59).


```

from django.db import models

from django.contrib.auth.models import User
from PIL import Image

class Perfil(models.Model):
    user = models.OneToOneField(User, on_delete=models.CASCADE)
    avatar = models.ImageField(default='default.jpg', upload_to='profile_pics')

    class Meta:
        verbose_name = ("Perfil reistrado")
        verbose_name_plural = ("Perfiles registrados")
        ordering = ['user']

    def __str__(self):
        return self.user.username

    def save(self, *args, **kwargs):
        super().save(*args, **kwargs)

        img = Image.open(self.avatar.path)

        if img.height > 300 or img.width > 300:
            output_size = (300,300)
            img.thumbnail(output_size)
            img.save(self.avatar.path)

```

Figura 58. users - models.py

Signals.py

Como se muestra en la figura 60, este archivo contiene funciones que se activan por medio de signals, o señales, cuando se presentan ciertos eventos, tales como:

- `Create_profile()`: Crea un Perfil en la base de datos cuando se crea un nuevo usuario.
- `Save_profile()`: Se actualiza el Perfil luego de que se actualiza los datos asociados a User.

```
from django.db.models.signals import post_save
from django.contrib.auth.models import User
from django.dispatch import receiver
from .models import Perfil

@receiver(post_save, sender=User)
def create_profile(sender, instance, created, **kwargs):
    if created:
        Perfil.objects.create(user=instance)

@receiver(post_save, sender=User)
def save_profile(sender, instance, created, **kwargs):
    instance.perfil.save()
```

Figura 59. users - signals.py

Tests.py

Es un archivo creado automáticamente por Django después de crear la aplicación, que sirve para realizar pruebas adicionales automatizadas.

Views.py

Este archivo se encarga de recibir la web request del usuario y retornar una web response en forma de contenido HTML. Entre las funciones que podemos encontrar en este archivo se encuentra:

- `Register()`: Permite el registro de usuario en la aplicación web, dirigiendo al usuario a la vista `register.html`. Para facilitar este proceso, se importó a esta función la clase `UserRegisterForm` que construimos en `forms.py` para las casillas del formulario. Luego de llenar formulario en `register.html`, que contiene el token CSRF como una etiqueta escondida, el usuario realiza una petición POST. Si se ha llenado el formulario correctamente, se envía la información a la base de datos para crear un usuario nuevo, si no, debe corregir la casilla correspondiente.

- **Profile():** Permite al usuario editar su perfil, dirigiendo al usuario a la vista `profile.html`. Si el usuario ha iniciado sesión en la aplicación web, puede realizar una petición POST para actualizar su username, email o avatar. En `profile.html` también agregamos la etiqueta oculta del token CSRF. Una vez que se valida la información, se actualiza la información del usuario en la base de datos.

Web_plataform

__pycache__

Es una carpeta que contiene bytecode. Cuando se ejecuta un programa en Python, lo primero que el intérprete hace es compilarlo en bytecode para simplificar y guardar en esta carpeta. Como programador, esta carpeta se ignora, ya que su única función en el sistema es hacer que el software se ejecute más rápido.

Migrations

Esta carpeta sirve para que Django procese los cambios que se han realizado a los Modelos que se encuentran en el esquema de la base de datos del proyecto, como agregar campos, borrar modelos, entre otros. Esta carpeta se crea de manera automática luego de que se ejecuta el comando `makemigrations`, y crea nuevos archivos con el mismo comando. Las migraciones de esta carpeta se aplican en el proyecto con el comando `migrate`.

Templates

Esta carpeta se crea para que Django pueda generar Vistas de manera dinámica. Por lo tanto, esta carpeta contiene las plantillas HTML del módulo `users`, a saber:

- **About.html:** En esta Vista se muestra los PDFs guardados en el directorio `Server/static/PDF/`, que contiene el manual de usuario y técnico de la aplicación web. Si el usuario no ha iniciado sesión no se mostrarán los manuales, mientras que el manual de usuario se mostrará a los usuarios registrados, y el manual de usuario y técnico se mostrará a los superusuarios.

- Backup.html: En esta vista se muestra un botón que da acceso al Área Administrativa, y un botón que permite descargar un backup en formato excel (.xls) del histórico del laboratorio de prototipos, que se encuentra almacenado en la base de datos.
- Base.html: Es la plantilla base para todas las Vistas que son utilizadas por el servidor, la cual genera la cabecera con las opciones correspondientes dependiendo que tipo usuario que se encuentra conectado.
- Home.html: Genera la vista que ve el usuario al entrar en la aplicación web desde el navegador.

`__init__.py`

Es un archivo usado para indicar que el directorio presente es un paquete de Python, volviendo posible la importación de módulos a sub-paquetes del directorio en el que se encuentra. También puede usarse para definir código de inicialización o variables que se ejecuten cuando el paquete sea importado.

`Admin.py`

Este archivo usa los modelos que se codifican en `models.py` del módulo `web_plataform` para construir automáticamente dentro del Área Administrativa un sitio que se pueda usar para crear, consultar, actualizar y borrar registros. Todo esto con la finalidad de ahorrar tiempo de desarrollo y poder probar los modelos para verificar que los datos son correctos.

`Apps.py`

Como se puede ver en la figura, este archivo sirve para realizar configuraciones en la aplicación “web_plataform”, a saber: Incluir los modelos creados en `models.py` dentro de la base de datos, que pueden ser accedidos desde el Área Administrativa en la sección “web_plataform” e implementar las configuraciones realizadas en `admin.py` en el sitio.

`Models.py`

En este archivo se definen los modelos de la base de datos, utilizados por la aplicación “web_plataform”.

Tests.py

Es un archivo creado automáticamente por Django después de crear la aplicación, que sirve para realizar pruebas adicionales automatizadas.

Urls.py

Este archivo especifica la relación que existe entre una Vista concreta del proyecto y la URL en la que aparece esa vista. Como se muestra en la figura 61, el fichero contiene una configuración de paths en una lista de urlpatterns.

```
# web_plataform/urls.py
from django.urls import path
from . import views

urlpatterns = [
    path('', views.home, name='server-home'),
    path('about/', views.about, name='server-about'),
    path('backup/', views.backup, name='server-backup'),
    path('export_excel', views.export_excel,
         name='export-excel'),
]
```

Figura 60. web_plataform - urls.py

Al buscar acceder a alguna de las URLs de la lista, se llama a la función correspondiente que se encuentra en views.py, que realiza el proceso correspondiente.

Views.py

Se encarga de recibir la web request del usuario y retornar una web response en forma de contenido HTML. Entre las funciones que podemos encontrar en este archivo se encuentra:

- Home(): Es una función que recibe por parámetros una web request de cualquier usuario que quiere acceder al url ' ', dando acceso a la vista Inicio o home.html que se encuentra en la carpeta templates del módulo.
- Backup(): Posee un decorador @login_required que pasa por argumento una señal (signal) que contiene la url al que el usuario accederá si no ha iniciado sesión con su cuenta: "login/". En la función, si el usuario que quiere acceder al URL "backup/" no es un administrador

(superusuario), se le redirigirá a la página `home.html`, pero si cumple con las condiciones se dirigirá a la página `backup.html` que se encuentra en la carpeta `templates` del módulo, donde se le da la opción al superusuario de entrar al Área Administrativa, creada por Django al momento de crear el proyecto, y de descargar un backup en formato Excel (.xls) del histórico del laboratorio de prototipos, que se encuentra almacenado en la base de datos. También se declaró la URL que dirigirá al usuario administrador a la página del Área Administrativa que provee Django. En el Área Administrativa añadir, modificar y eliminar información de la base de datos; como, por ejemplo, cambiar el rol de los usuarios para que sean administradores, o borrar usuarios registrados.

- `Export_excel()`: Función que se ejecuta cuando se presiona el botón “Exportar en Excel (.xls)”. Posee un decorador `@login_required` que pasa por argumento una señal (signal) que contiene la URL al que el usuario accederá si no ha iniciado sesión con su cuenta: “`login/`”. En la función, si el usuario desea exportar la información de la base de datos y no es un administrador (superusuario), se le redirigirá a la página `home.html`, pero si cumple con las condiciones se creará un archivo Excel de nombre “registro_lab_prototipos_<<fecha actual en formato día/mes/año>>”, donde se muestra un histórico de la temperatura, humedad relativa, luminosidad en la sección 1 y sección 2, y presencia que se ha guardado en la base de datos durante el tiempo que el servidor ha estado en ejecución.