UNIVERSITAT POLITÈCNICA DE VALÈNCIA

Escuela Técnica Superior de Ingeniería Informática

Diseño e Implementación del Módulo Software de Predicciones para un Sistema de Gestión de Energía basado en PLC Industrial

Trabajo Fin de Grado

Grado en Ingeniería Informática

AUTOR/A: Moreno Aranda, Iván

Tutor/a: García-Nieto Rodríguez, Sergio

CURSO ACADÉMICO: 2023/2024

Resumen

En el presente Trabajo Final de Grado se ha diseñado e implementado un módulo software de predicción del coste de la energía eléctrica y de la climatológica para un sistema de gestión de energía basado en un PLC industrial de la marca Phoenix Contact, en particular el modelo AXC F 3152.

El tema de este proyecto es de gran interés porque aborda la gestión eficiente de la energía en un contexto de creciente preocupación por la sostenibilidad y el ahorro energético. La capacidad de predecir los costos de energía y las condiciones climáticas permite optimizar el uso de recursos en instalaciones de autoconsumo.

Para lograr este objetivo, se ha analizado y desarrollado un módulo software que integra varios componentes críticos. Se ha implementado un servidor Flask que se comunica con diferentes APIs de meteorología y electricidad para obtener datos en tiempo real. Este servidor también tiene la capacidad de servir páginas web implementadas en HTML y CSS para la visualización de los datos. Los datos obtenidos de las APIs se procesan en formato JSON y se envían a un proceso simulado en Matlab Simulink mediante un cliente UDP.

El desarrollo del módulo software se ha realizado utilizando una serie de herramientas y tecnologías. El entorno de desarrollo utilizado fue Visual Studio Code, y el lenguaje de programación principal fue Python. Además, se emplearon microframeworks como Flask y diversas bibliotecas para la gestión de datos y la comunicación en red. Por último, el módulo software es ejecutado en un controlador lógico programable (PLC).

Como resultado del proyecto, el módulo software es el encargado de aportar datos reales a un problema de optimización de tipo programación lineal entera mixta (MILP) para gestionar los flujos de energía eléctrica en una vivienda unifamiliar equipada con una instalación de autoconsumo (inversor, paneles solares, batería, etc).

En conclusión, en este proyecto se demuestra que la incorporación de hardware más potente y flexible en los PLCs industriales, junto con la implementación de módulos software avanzados, puede mejorar significativamente las capacidades de los sistemas de gestión de energía. La capacidad de predecir los costos de energía y las condiciones climáticas permite una optimización más eficiente de los recursos.

Palabras clave: módulo software, PLC, gestión de energía, Flask, APIs, JSON, UDP, Python, problema de optimización, MILP.

Abstract

In this Final Degree Project, a software module for predicting the cost of electrical energy and weather conditions has been designed and implemented for an energy management system based on an industrial PLC from Phoenix Contact, specifically the AXC F 3152 model.

The subject of this project is of great interest because it addresses efficient energy management in a context of growing concern for sustainability and energy savings. The ability to predict energy costs and weather conditions allows to optimize the use of resources in self-consumption installations.

To achieve this goal, a software module integrating several critical components has been analyzed and developed. A Flask server has been implemented to communicate with different weather and electricity APIs to obtain real-time data. This server also has the capability to serve web pages implemented in HTML and CSS for data visualization. The data obtained from the APIs are processed in JSON format and sent to a simulated process in Matlab Simulink via a UDP client.

The development of the software module has been carried out using a range of tools and technologies. The development environment used was Visual Studio Code, and the main programming language was Python. In addition, microframeworks such as Flask and various libraries for data management and network communication were employed. Finally, the software module is executed on a programmable logic controller (PLC).

As a result of the project, the software module is responsible for providing real data to a mixed-integer linear programming (MILP) optimization problem to manage the electrical energy flows in a single-family house equipped with a self-consumption installation (inverter, solar panels, battery, etc.).

In conclusion, this project demonstrates that the incorporation of more powerful and flexible hardware in industrial PLCs, along with the implementation of advanced software modules, can significantly improve the capabilities of energy management systems. The ability to predict energy costs and weather conditions allows for more efficient optimization of resources.

Keywords: software module, PLC, power management, Flask, APIs, JSON, UDP, Python, optimization problem, MILP.

Tabla de contenidos

Tabla de figuras

1 Introducción

1.1 Motivación

En la actualidad industrial, la automatización de procesos es ampliamente adoptada en la mayoría de los sectores. Esta transformación se realiza principalmente mediante el uso de controladores lógicos programables, conocidos como PLCs (Programmable Logic Controllers). Desde finales de la década de 1960, estos dispositivos han sido clave reemplazando los sistemas de control basados en circuitos eléctricos con relés, interruptores y otros componentes de la época, ofreciendo una solución más eficiente para la lógica combinacional.

La integración de PLCs en la automatización diaria ha resultado en significativas mejoras industriales. Por un lado, facilita la rápida adaptación y modificación de proyectos sin costos adicionales significativos. Por otro lado, los PLCs son compactos y requieren poco mantenimiento económico, reduciendo costos laborales y permitiendo el control simultáneo de múltiples máquinas con un solo equipo.

No obstante, durante el comienzo y hasta hace unos años, los PLCs solo podían ser programados para automatizar procesos. Aquellos que eran un poco más modernos permitían hacerlo mediante diagramas de contactos y lógica de estado, un lenguaje de programación de alto nivel. Actualmente algunos PLCs están en otro nivel. Esto amplía las posibilidades, lo que permite llevar a cabo tareas más complejas.

A lo largo de todo el trabajo va a ponerse a prueba el potencial de los PLCs actuales y se va a incorporar un módulo Python con un sistema de predicción basado en APIs de terceros, el cual enviará datos a un proceso simulado en Matlab para que este ejecute una rutina de optimización MILP con los datos aportados.

1.2 Objetivos

El objetivo principal de este trabajo es el desarrollo de un módulo software de predicción del coste de energía eléctrica y climatológica para un sistema de gestión de energía basado en PLC industrial.

Una vez alcanzado este objetivo, aplicar el trabajo realizado para mejorar un algoritmo de control basado en optimización matemática multivariable, aportando valores reales en un caso práctico. El algoritmo realizará la gestión de los flujos de energía para una vivienda unifamiliar con una instalación de autoabastecimiento.

Además, con respecto a los propios objetivos personales del creador del módulo software relacionado a este TFG:

- Aprendizaje de uno de los lenguajes de programación más relevantes en el panorama actual.
- Profundizar en el uso de componentes y herramientas dedicadas a la automatización.
- Conocer mecanismos de comunicación avanzados entre diferentes sistemas y dispositivos.
- Desarrollar los conocimientos adquiridos en un entorno de trabajo real.

1.3 Estructura de la memoria

Esta memoria recoge la documentación del desarrollo íntegro del módulo software de predicción. A continuación, se explica el contenido de cada uno de los capítulos que contiene:

- En el capítulo 2, se presenta el estado del arte, donde se plantea cual es la situación actual y se realiza un análisis del problema. Además, se realiza una investigación sobre las APIs a utilizar.
- En el capítulo 3, se detalla en profundidad cómo ha sido el desarrollo de la solución, abordando aspectos como el diseño, los elementos que intervienen en el proceso, cómo se ha implementado el módulo software y el resultado final.
- En el capítulo 4, se presentan las conclusiones a las que se han alcanzado durante el desarrollo de este proyecto y se exponen futuros trabajos a realizar.

Posteriormente, se presentan las referencias bibliográficas consultadas para la realización del presente TFG. Por último, se adjunta un anexo con el código desarrollado en el módulo software de predicción y los Objetivos de Desarrollo Sostenible (ODS).

2 Estado del arte

2.1 Introducción al problema

En las últimas décadas, los Controladores Lógicos Programables (PLCs) han evolucionado significativamente desde sus orígenes como dispositivos dedicados al control industrial. Anteriormente, los PLCs se distinguían por su capacidad de ejecutar lógica de control en tiempo real mediante relés, circuitos integrados o microcontroladores dedicados, lo que limitaba su adaptabilidad a nuevas funcionalidades sin modificaciones físicas significativas.

En contraste, la moderna generación de PLCs ha incorporado arquitecturas de hardware más potentes y flexibles, permitiendo la ejecución directa de software avanzado. Esto se logra mediante la integración de procesadores más rápidos y eficientes, junto con sistemas operativos embebidos que soportan la instalación de aplicaciones y módulos de software personalizados.

La capacidad de ejecutar módulos de software directamente en los PLCs no solo ha mejorado la eficiencia operativa y la flexibilidad en la producción industrial, sino que también ha allanado el camino para la adopción de tecnologías emergentes como el Internet de las Cosas Industrial (IIoT) y el Big Data[1]. Ahora, los PLCs pueden actuar como nodos inteligentes dentro de sistemas más amplios de automatización y control, permitiendo la monitorización remota, la optimización predictiva y la respuesta adaptativa a condiciones cambiantes del entorno industrial.

Este trabajo es consecuencia de la evolución de las capacidades de los PLCs, permitiendo así el desarrollo de un módulo Python para la obtención de datos en tiempo real. Se explicará durante este documento cómo se ha realizado esta implementación.

2.2 Análisis del problema

Actualmente, no se dispone de una vivienda unifamiliar para la aplicación de los algoritmos de gestión de flujos de energía. En cambio, se dispone de un proceso simulado mediante ecuaciones diferenciales en Matlab Simulink que actúa cómo dicha vivienda.

Para ello, el proceso en Matlab dispone de un modelo proporcionado y unas predicciones meteorológicas y eléctricas para la simulación del consumo de energía eléctrica. Esta información

[1] El Big Data es el conjunto de datos masivos y complejos que no pueden ser gestionados, procesados o analizados con herramientas tradicionales debido a su volumen, velocidad y variedad.

es la que se le pasa al algoritmo de gestión de flujos para que se resuelva el problema de optimización.

En consecuencia, la aplicación de este trabajo consiste en aportar datos reales al proceso simulado, para que este se los aporte al algoritmo de control. De esta forma, el algoritmo puede realizar pruebas con datos verdaderos y así, mejorar la precisión de los resultados.

2.2.1 Investigación de APIs

La forma de poder abordar el problema descrito anteriormente es aportando información meteorológica y eléctrica al proceso simulado. Para obtener esta información, se va a hacer uso de diferentes APIs que recojan información sobre estos dos grandes bloques.

En los siguientes apartados se explorarán diversas APIs y se elegirán aquellas que se adapten más a los requisitos de este proyecto. En el apartado 3.2.1 se comentará la configuración de cada API.

2.2.1.1 APIs Meteorológicas

Para una mayor precisión de los datos y un mayor desarrollo del proyecto, se decidió que se elegirían mínimo tres APIs diferentes. De esta forma, al abarcar más cantidad de datos, se podrían procesar cómo una única predicción y obtener un resultado más preciso.

Finalmente, se optó por implantar un total de cuatro APIs, eligiendo aquellas que tenían la mejor relación en base a las siguientes características:

- Precisión de los datos.
- Acceso gratuito.
- Buena documentación.

Después de analizar un gran número de posibilidades, se seleccionaron las siguientes APIs:

1. **OpenWeather**: Dispone de un plan gratuito denominado *One Call API 3.0* que ofrece una gran cantidad de datos en diferentes formatos (minuto a minuto por 1 hora, hora a hora para las futuras 48 horas y diario para los próximos 8 días). Además, la forma de realizar la solicitud es muy sencilla.
2. **Weatherstack**: El plan gratuito sólo ofrece información en tiempo real, pero es de las pocas que obtiene la radiación solar actual. Además, es una de las APIs más precisas en información en tiempo real y tiene una gran documentación con ejemplos de código para Python.

3. **Tomorrow.io**: Una de las APIs con mayor influencia internacional, se caracteriza por su información detallada a nivel global, su plan gratuito ofrece una enorme cantidad de datos y dispone de una plataforma amigable para la gestión de la API.
4. **AEMET**: Esta es la API de la Agencia Estatal de Meteorología española, por ello sólo ofrece datos para España. En cambio, es completamente gratuita, la más precisa y dispone de una buena documentación.

2.2.1.2 APIs Eléctricas

Para la elección de la API eléctrica se utilizó el mismo criterio comentado para las APIs meteorológicas. Pero en este caso, las opciones son mucho más escasas.

Esto es debido a que los precios sobre la electricidad son indicados a las 20:00h del día anterior para el siguiente día, con lo que después de establecerse ya no varían. Además, la empresa **red eléctrica**, perteneciente al grupo **redeia**, es la encargada de gestionar la electricidad en España y ofrece una API para la obtención de los datos eléctricos. El inconveniente que tiene esta API es que es muy compleja y no ofrece una buena documentación.

Finalmente, después de buscar otras opciones, se encontró **preciodelaluz.org**, qué como indican en su página web: “Esta api está desarrollada para simplificar la obtención de los precios diarios relativos a la tarifa regulada PVPC (Precio voluntario para el pequeño consumidor) del mercado eléctrico español”. Es completamente gratuita y muy fácil de usar, por ello se eligió cómo API para los datos eléctricos.

3 Desarrollo de la solución

3.1 Diseño de la solución

La solución propuesta busca mejorar la eficiencia y precisión en la gestión de flujos de energía aportando información clave al algoritmo de control. A continuación, se describen las fases de diseño, las herramientas tecnológicas, la metodología empleada para la implementación del módulo, las pruebas y resultados obtenidos durante el desarrollo del proyecto.

3.1.1 Metodología

La metodología empleada en este trabajo se diseñó para asegurar un desarrollo eficiente y estructurado del módulo Python destinado a la obtención y gestión de datos meteorológicos y eléctricos. El proceso se dividió en varias fases, cada una con objetivos y tareas específicas, que abarcaron desde la investigación inicial de las APIs más adecuadas hasta la implementación y despliegue final en el sistema. A continuación, se detallan las fases clave del desarrollo, describiendo los enfoques y técnicas utilizados en cada etapa para alcanzar los objetivos del proyecto.

1. **Fase de investigación**: Inicialmente, se llevó a cabo una investigación para identificar las APIs más adecuadas que proporcionaran datos meteorológicos y eléctricos. Se evaluaron varios proveedores y se seleccionaron aquellos que ofrecían la mejor combinación de datos relevantes, posibilidades económicas, facilidad de uso y documentación detallada.
2. **Implementación inicial**: El desarrollo comenzó con la creación de pequeños scripts individuales para cada API. Estos scripts se centraron en realizar solicitudes a las APIs y obtener los datos necesarios. Posteriormente, se implementaron procesos para manipular y limpiar estos datos según los requisitos específicos del proyecto.
3. **Unificación del código**: Después de la implementación inicial de los scripts individuales, se procedió a unificar el código en dos clases separadas: una para la gestión de datos meteorológicos y otra para los datos eléctricos. Esta unificación se realizó para encapsular la lógica de interacción con cada API y mejorar la mantenibilidad del código. Cada clase se encargó de manejar las peticiones a sus respectivas APIs y de procesar los datos obtenidos.
4. **Implantación de la solución**: La fase final del proyecto consistió en la integración y despliegue de la solución completa. Esto incluyó la configuración del servidor Flask para manejar las solicitudes web y presentar los datos a los usuarios a través de una interfaz

amigable, el envío de datos al proceso simulado a través del protocolo UDP y el despliegue del módulo en el PLC.

3.1.2 Entorno de desarrollo y herramientas

Respecto al entorno de desarrollo integrado (IDE), no se hizo una selección deliberada, sino que se optó por el que resultaba más familiar y cómodo, que en este caso es *Visual Studio Code*. Esta herramienta proporciona todas las facilidades y asistencias necesarias, como un sistema de depuración muy eficiente y fácil de usar. Además, se integra perfectamente con Python, hecho que mejora significativamente el desarrollo.

Asimismo, se emplearon herramientas adicionales imprescindibles para el flujo de trabajo. Entre ellas, los navegadores Safari y Mozilla Firefox, que son esenciales para poder probar la solución y depurar fallos. También se encuentra la consola o terminal, jugando un papel crucial, ya que es por donde se realizan los accesos al PLC y en donde se ejecuta el módulo software de predicción en un entorno virtual Python.

Por último, git se ha utilizado para el control de versiones y GitHub como repositorio de código, debido a que es un de las plataformas más utilizada y aceptada en la comunidad de desarrolladores.

3.1.3 Esquemas de elementos e interacciones

En la figura uno se puede ver un esquema general con los elementos involucrados en el módulo, el flujo de la información y la comunicación entre ellos.

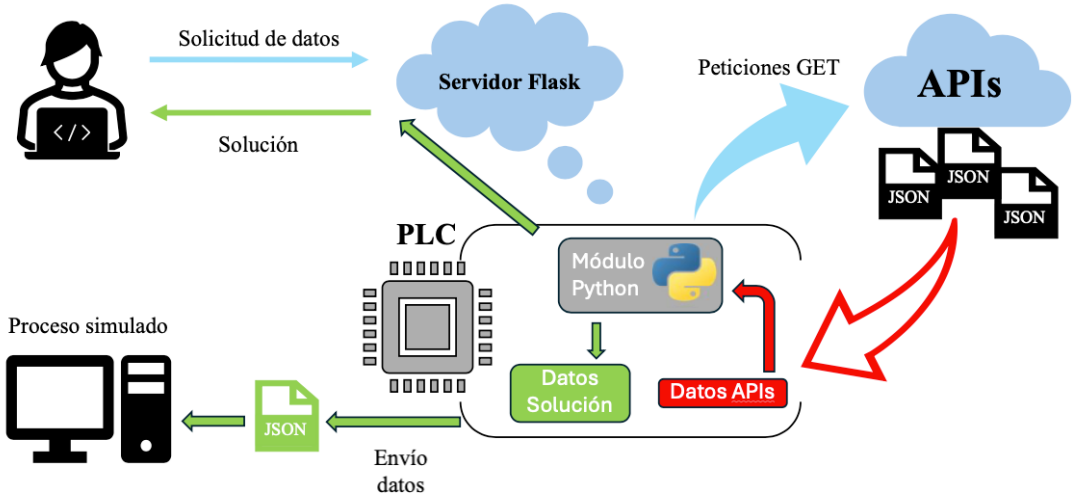

***Figura 1**: esquema general de los elementos*
***Fuente**: elaboración propia*

El elemento primordial es el PLC industrial AXC F 3125 de la compañía Phoenix Contact, en este elemento es dónde se ejecutará el módulo de predicción Python. Este módulo juega un papel crucial para la obtención de la solución final, ya que se encarga de realizar las peticiones a las APIs, obtener y manipular los datos y servirlos tanto al usuario solicitador cómo al proceso simulado.

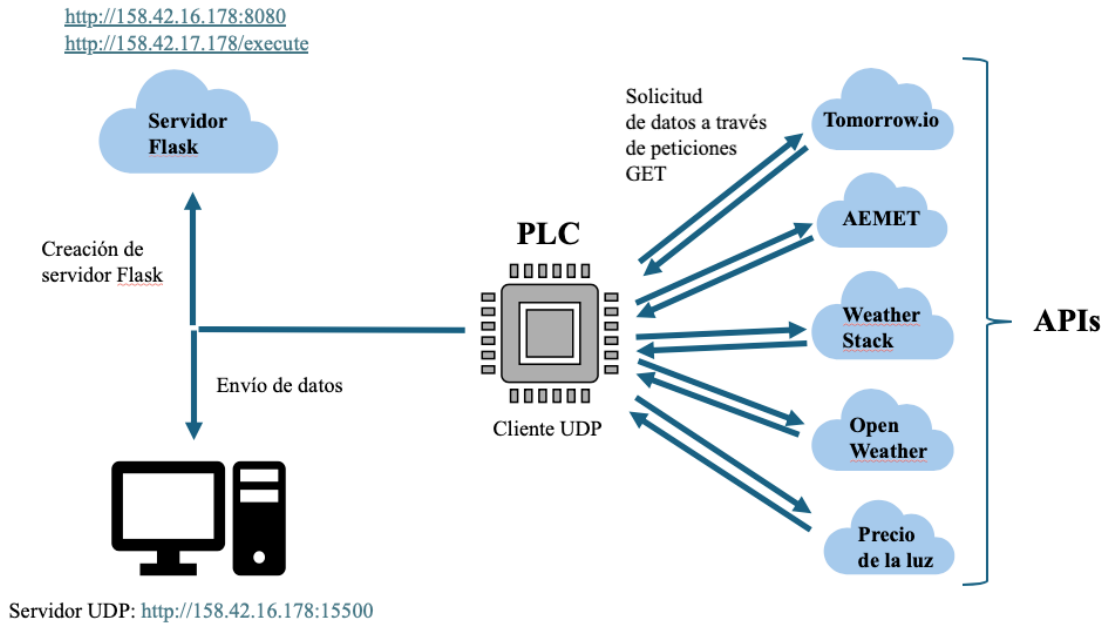

***Figura 2**: diagrama de red*
***Fuente**: elaboración propia*

En la figura dos se puede ver de manera gráfica cómo el PLC genera un servidor Flask que sirve una página web, accesible a través de la siguiente URL: http://158.42.16.178:8080, para que un usuario pueda realizar solicitudes de tipo GET [2]a diferentes APIs externas. De estas solicitudes se obtiene cómo respuesta un objeto JSON independiente por cada API.

Una vez obtenidos todos los datos en el módulo software, este los procesa y los unifica en un único JSON como datos finales. Estos datos son servidos al usuario de vuelta a través de una nueva página web en el servidor Flask en la URL: http://158.42.16.178/execute, y enviados al proceso simulado actuando cómo cliente UDP al puerto 15500.

[2] Las peticiones GET son solicitudes HTTP que se utilizan para obtener datos de un servidor, especificando los recursos a recuperar mediante una URL.

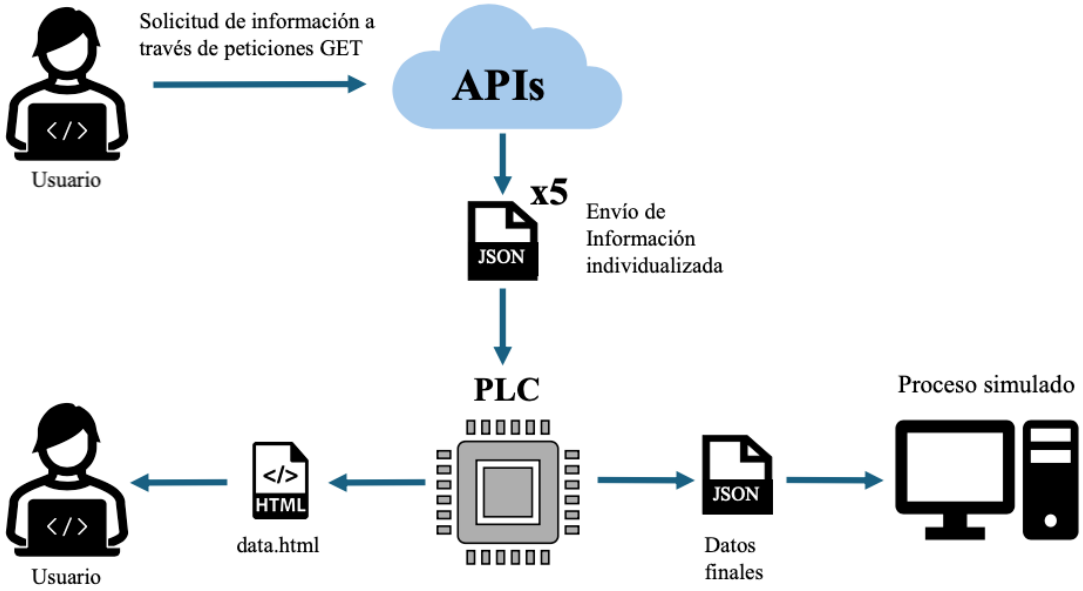

Figura 3: esquema del proceso completo
***Fuente**: elaboración propia*
***Icono html**: https://flaticon.com/*

Por último, la figura tres representa el flujo de datos desde la solicitud inicial hasta la visualización y envío de los datos finales.

3.1.4 Explicación de los elementos

En esta sección se detallarán todos los elementos clave que influyen significativamente en el desarrollo de la técnica del trabajo. Estos elementos se encuentran representados en la figura uno, junto con su interacción mutua.

3.1.4.1 PLC Industrial

Para la implementación del módulo software de control, se empleará un PLC industrial de la compañía Phoenix Contact. Entre los puntos a destacar de esta compañía, se encuentra la *PLCnext Technology* (Tecnología PLCnext), esto es un ecosistema para la automatización industrial que consiste en un hardware abierto, un software de ingeniería modular, una comunidad global y el mercado de software digital. El diseño de este ecosistema permite la programación paralela y la combinación de lenguajes de programación como C/C++, Python, etc. además del estándar de programación del PLC según la norma IEC 61131[3].

[3] La norma IEC 61131 es un estándar internacional que define lenguajes y métodos de programación para controladores lógicos programables (PLCs), facilitando la interoperabilidad y eficiencia en sistemas de automatización industrial.

En este trabajo, se utilizará el sistema de control PLC (AXC F 3152), que cuenta con tres interfaces Ethernet independientes, conector de conexión, un módulo de zócalo de bus y soporta ciertos lenguajes de programación. Este sistema de control se caracteriza porque son rápidos, robustos y sencillos.

***Figura 4**: PLC AXC F 3152.*
***Fuente**: Phoenix Contact*

En la figura cuatro se observa cómo es el PLC con el que se va a trabajar en este proyecto. Para obtener más información sobre estos módulos, se puede consultar el producto en la página web oficial de Phoenix Contact: https://www.phoenixcontact.com/es-es/productos/mando-axc-f-3152-1069208.

3.1.4.2 Python

Python es un lenguaje de programación de alto nivel, interpretado y de propósito general, creado por Guido van Rossum y lanzado por primera vez en 1991. Es conocido por su sintaxis clara y legible, lo que facilita su aprendizaje y uso, por ello es ampliamente utilizado en aplicaciones web, el desarrollo de software, la ciencia de datos y el *machine learning* (ML). Además, soporta múltiples paradigmas de programación, incluyendo la programación estructurada, orientada a objetos y funcional.

***Figura 5**: logo de Python*
***Fuente**: Python*

Al ser un lenguaje interpretado, el código es ejecutado línea por línea por un intérprete. Esto contrasta con los lenguajes compilados, que requieren que el código sea convertido a un lenguaje máquina antes de su ejecución. Python se ejecuta sobre una máquina virtual, generalmente en CPython, que es la implementación de referencia del lenguaje. Los pasos para la generación y ejecución estándar de un código en Python serían:

1. **Creación del código**: Se desarrolla el código en un archivo de texto con la extensión '.py'.
2. **Interpretación**: El intérprete de Python lee el código fuente.
3. **Compilación**: El código fuente es compilado a un bytecode intermedio, creando el archivo '.pyc'.
4. **Ejecución**: La máquina virtual de Python ejecuta el bytecode.

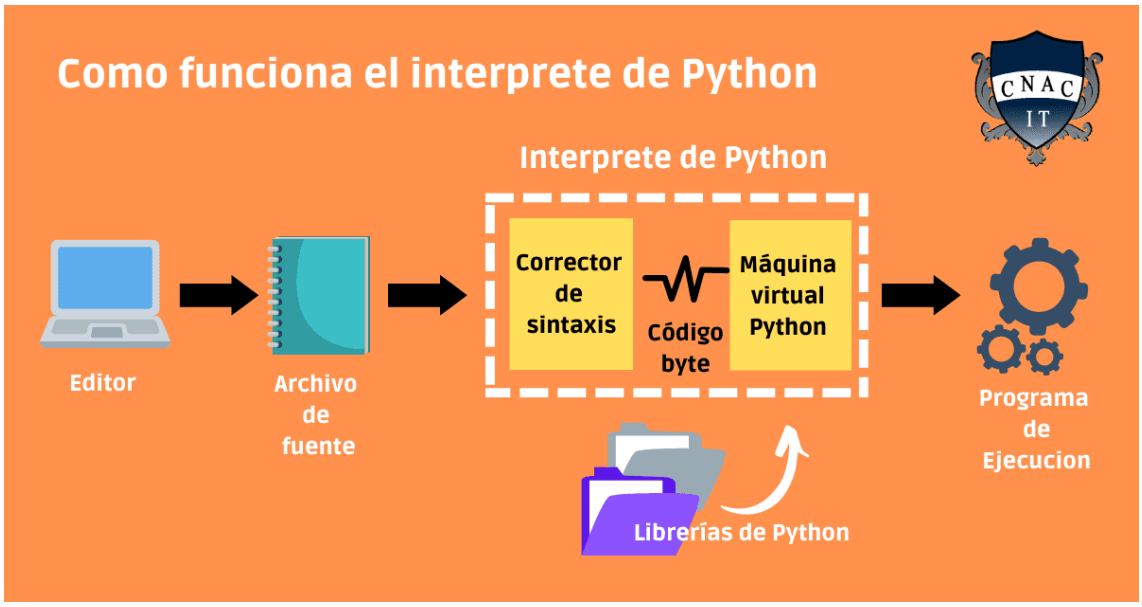

Figura 6: funcionamiento de generación y ejecución de código Python
Fuente: https://www.cnac.es/noticias/que-es-python/

Python ha ganado una popularidad considerable en la última década debido a su versatilidad, facilidad de uso y una extensa colección de bibliotecas y frameworks. Sin embargo, como cualquier herramienta, Python tiene sus desventajas y ventajas que deben considerarse antes de adoptarlo para un proyecto específico.

Entre sus desventajas se encuentran:

- **Rendimiento**: Como es un lenguaje interpretado, es generalmente más lento que los lenguajes compilados.
- **Limitaciones en el desarrollo móvil**: Pese a disponer de herramientas, no es tan robusto como Java o Swift en este ámbito.
- **Consumo de memoria**: Python puede consumir memoria de más debido a la gestión dinámica de memoria y su naturaleza de alto nivel.

En contrapartida, las principales ventajas de Python son:

- **Multiplataforma**: Python es compatible con múltiples sistemas operativos.
- **Sintaxis sencilla y legible**: Python tiene una sintaxis clara y concisa que facilita tanto la escritura cómo la lectura del código.

- **Versatilidad**: Python es adecuado para una gran variedad de aplicaciones, desde aprendizaje automático hasta desarrollo web.
- **Bibliotecas**: Python cuenta con una extensa comunidad que contribuye a una amplia variedad de bibliotecas y frameworks.

Para el desarrollo del módulo software de predicción se ha utilizado Python, la versión 3.12.3, cómo lenguaje de programación, junto con varios paquetes y herramientas. Entre ellos se encuentra **pip**, que es el sistema de gestión de paquetes para Python, utilizado para instalar y gestionar paquetes software. Otro paquete utilizado es **Numpy**, un módulo fundamental para el cálculo numérico en Python, que proporciona soporte para arrays y matrices, además de funciones matemáticas de alto nivel para operar con estos datos de manera eficiente. También se ha utilizado **Requests**, un módulo que simplifica el envío de solicitudes HTTP, permitiendo interactuar con servicios web y APIs de manera sencilla y eficaz.

Además, se ha empleado **Flask** para la creación del servidor. Flask es un microframework [4]web para Python que permite crear aplicaciones web de manera sencilla y rápida. Proporciona las herramientas necesarias para construir rutas, manejar solicitudes HTTP, renderizar plantillas y más, sin requerir una estructura compleja de proyectos. Su simplicidad y potencia lo hacen ideal para desarrollar y desplegar aplicaciones web que requieren una interacción dinámica y en tiempo real con el usuario, como en el caso del módulo de predicción que se describe en este trabajo.

3.1.4.3 HTML y CSS

HTML (HyperText Markup Language) es el componente más básico de la web. Es el lenguaje estándar utilizado para crear y estructurar páginas web. Se trata de un lenguaje de marcado que define la estructura básica de una página web mediante etiquetas o elementos que indican cómo debería presentarse y qué tipo de contenido debería contener. Cada elemento HTML representa diferentes tipos de contenido, como texto, imágenes, vídeos, formularios, entre otros.

CSS (Cascading Style Sheets) por otro lado, complementa HTML al definir cómo se visualiza el contenido HTML en la página web. CSS permite establecer el diseño, los colores, las fuentes y otros aspectos visuales de un sitio web. Utilizando reglas de estilo, que especifican cómo los elementos HTML deben ser presentados en el navegador, se puede controlar la apariencia y el diseño de una página web de manera eficiente y coherente. Las reglas CSS se aplican a elementos específicos utilizando selectores, como nombres de etiquetas HTML, clases o identificadores, lo que permite un alto grado de personalización y adaptabilidad en el diseño.

[4] Un microframework es un framework de software minimalista que proporciona solo las funcionalidades básicas necesarias para construir aplicaciones web.

La combinación entre HTML y CSS es fundamental para la creación de páginas web modernas y atractivas. Ya que esta separación entre estructura (HTML) y presentación (CSS) facilita la mantenibilidad del código, mejora la accesibilidad y permite una mayor flexibilidad en el diseño y la adaptación de las páginas web a diferentes dispositivos y tamaños de pantalla.

En este trabajo, se ha desarrollado la página de configuración y la página de visualización para los datos del módulo de predicción con estos componentes.

3.1.4.4 SSH

SSH (Secure Shell) es un protocolo de red criptográfico que permite a los usuarios acceder y gestionar dispositivos de manera segura a través de una red no segura. SSH proporciona una forma segura de autenticar, encriptar y transmitir datos entre dos dispositivos, lo cual es esencial para mantener la confidencialidad e integridad de la información.

Las características principales son:

- **Autenticación Segura**: Utiliza autenticación basada en contraseñas o en llaves criptográficas para asegurar que solo usuarios autorizados puedan acceder al sistema.
- **Encriptación**: Los datos transmitidos entre el cliente y el servidor SSH son encriptados, protegiéndolos de interceptaciones y ataques de intermediario (man-in-the-middle).
- **Integridad de datos**: Utiliza algoritmos de hash para asegurar que los datos no sean alterados durante la transmisión.

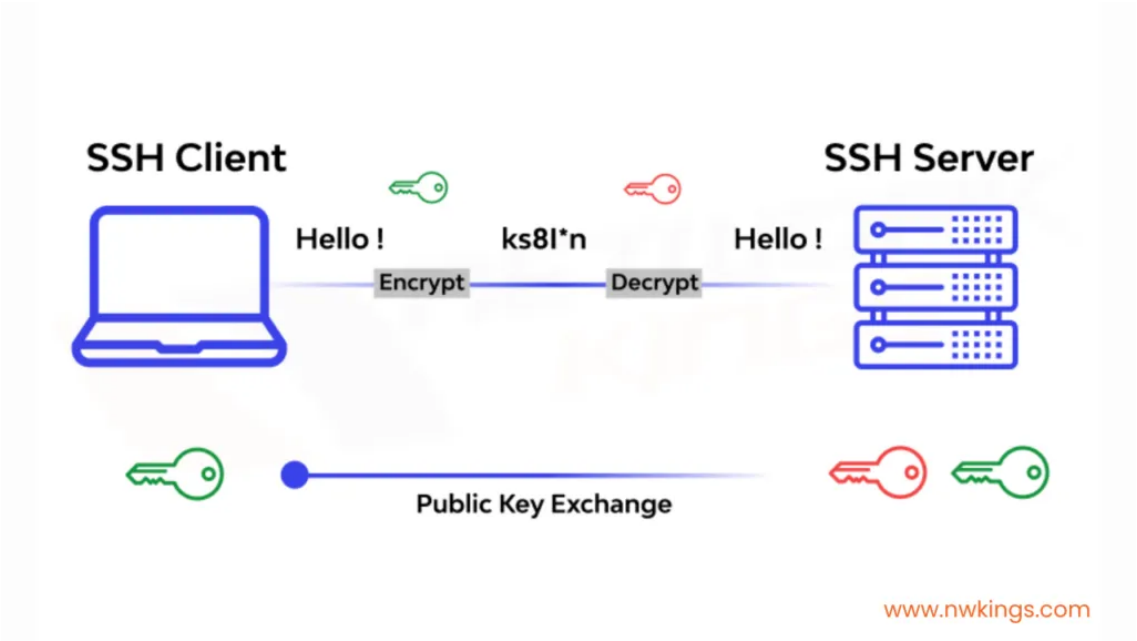

***Figura** 7: funcionamiento de la autenticación para SSH*
***Fuente**: nwkings.com*

En el contexto del trabajo con el PLC que ejecuta el módulo de predicción, SSH ha sido clave por varias razones:

1. **Acceso Remoto Seguro**: SSH permite conectarse al PLC de manera remota desde cualquier ubicación, garantizando que la conexión es segura. Esto es crucial cuando el PLC está en un entorno industrial o, en este caso, en un laboratorio de la UPV (Universidad Politécnica de Valencia).
2. **Gestión y configuración**: A través de SSH, se puede acceder a la consola del PLC para ejecutar comandos, configurar el sistema, y supervisar su funcionamiento.
3. **Transferencia de Archivos**: SSH facilita la transferencia segura de archivos entre el dispositivo de desarrollo y el PLC. Herramientas como SCP (Secure Copy Protocol) y SFTP (SSH File Transfer Protocol) se basan en SSH para mover archivos de manera segura, como scripts, módulos de Python y archivos de configuración necesarios para el módulo de predicción.
4. **Ejecución de Scripts Remotos**: Permite ejecutar scripts de manera remota en el PLC, lo que es esencial para la gestión y el despliegue del módulo de predicción. Esto ayuda a automatizar tareas y realizar actualizaciones sin interrumpir el funcionamiento del sistema.

3.1.4.5 APIs

Una API (Application Programming Interface), o Interfaz de Programación de Aplicaciones, es un conjunto de funciones y procedimientos que facilita la integración de sistemas, permitiendo que sus funcionalidades sean reutilizadas por otras aplicaciones o programas. Las APIs permiten el intercambio de datos entre diferentes tipos de software, lo que ayuda a automatizar procesos y desarrollar nuevas funcionalidades.

Sirven de puente que conecta diferentes tipos de software o aplicaciones y pueden crearse en varios lenguajes de programación. La forma de establecer una conexión segura es mediante el uso de API KEYs, que son identificadores únicos para autentificar las solicitudes.

Estas API KEYs consisten en una cadena alfanumérica que permite identificar y controlar el acceso, asegurando que sólo los usuarios autorizados puedan interactuar con la API. Para facilitar su implementación, una API no solo necesita estar bien desarrollada, sino también contar con una documentación clara y precisa. Los pasos seguidos en el funcionamiento de una API son:

1. El cliente inicia las solicitudes a través del URI (Identificador uniforme de recursos) de la API.
2. La API realiza una llamada al servidor después de recibir la solicitud.

3. El servidor envía la respuesta a la API con la información.
4. Finalmente, la API transfiere los datos al cliente.

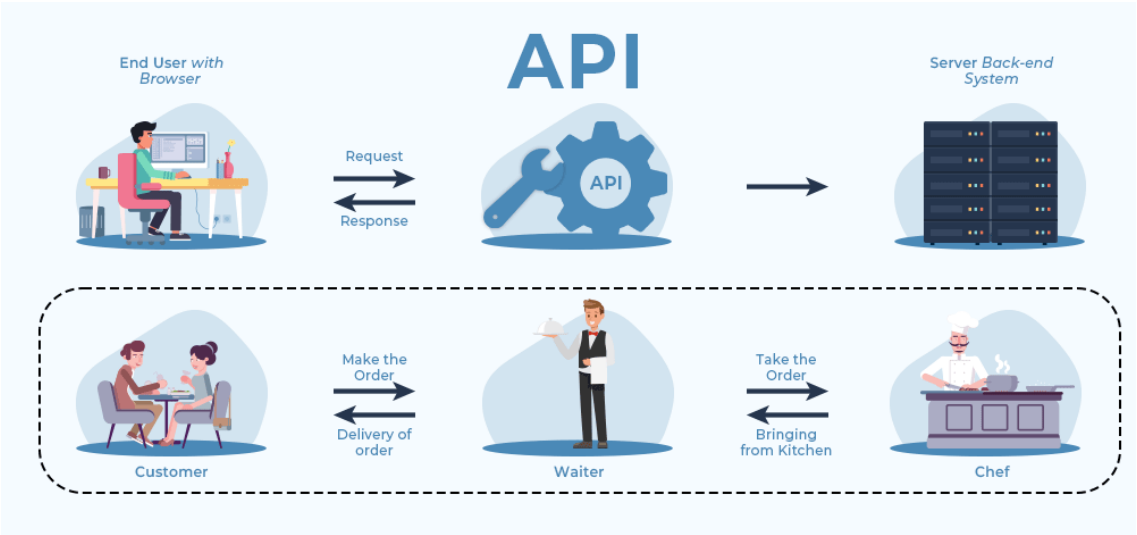

***Figura 8**: ejemplos para el procedimiento de las APIs*
***Fuente**: https://www.geeksforgeeks.org/what-is-an-api/*

Las APIs suelen utilizar formatos de datos predefinidos para compartir información entre sistemas, favoreciendo así su integración. Entre los formatos más comunes se encuentran XML (Extensible Markup Language), YAML (Yet Another Markup Language) y JSON (JavaScript Object Notation), en este trabajo se utilizará JSON cómo formato.

Dependiendo del caso de uso, existen diferentes tipos de API, como las APIs de datos, APIs de sistemas operativos, y APIs web, entre otras. En este proyecto se trabajará específicamente con varias APIs web, las cuales son las más comunes debido a su capacidad para proporcionar datos que los dispositivos pueden leer y transferir entre sistemas basados en la web o arquitecturas cliente-servidor.

Adicionalmente, las APIs webs utilizan protocolos para estandarizar el intercambio de datos entre distintos servicios web. Este trabajo accede a diferentes APIs de meteorología y de datos relacionados con la electricidad y todas las APIs seleccionadas utilizan el protocolo REST (Representational State Transfer).

REST es un estilo de arquitectura de software que se basa en seis restricciones, diseñadas para crear aplicaciones que funcionan sobre HTTP, principalmente servicios web. Estas restricciones aseguran que los servicios web sean escalables, eficientes y fáciles de mantener, lo que resulta en una integración más robusta y flexible entre los diferentes sistemas involucrados.

3.1.4.6 JSON

En este trabajo, durante el intercambio de datos entre las APIs y el módulo software de predicción, se dedicará una parte del desarrollo al análisis y el tratamiento de los datos en formato JSON (JavaScript Object Notation). JSON es un formato ligero de intercambio de datos, fácil de leer y escribir para los humanos, y sencillo de interpretar y generar para las máquinas.

Aunque JSON es independiente del lenguaje de programación, sigue convenciones definidas por ECMA-404, conocidas por los programadores de lenguajes de la familia C, Java, JavaScript, Python, entre otros. Estas características hacen de JSON un formato ideal para el intercambio de datos.

Un JSON está compuesto por dos estructuras principales: una colección de pares nombre/valor, conocida como objeto, y una lista ordenada de valores, implementada como arreglos o vectores en la mayoría de los lenguajes de programación. Los valores pueden incluir números enteros, números reales, booleanos, cadenas de caracteres, etc.

Para más información sobre el formato JSON, se puede consultar su página web: https://json.org/json-es.html o el documento ECMA-404, donde se describen en detalle las convenciones del formato: https://www.ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf.

3.1.4.7 Protocolo UDP

UDP es un protocolo ligero de transporte de datos que funciona sobre protocolo de internet (IP). Es decir, permite el envío rápido de datagramas en redes IP sin necesidad de establecer una conexión previa.

Presenta varias características importantes. Para empezar, trabaja sin conexión, lo cual significa que no usa sincronización alguna entre el origen y el destino. Además, utiliza paquetes o datagramas enteros, con lo que intercambia información en forma de bloques de bytes, en vez de hacerlo con bytes individuales como lo hace TCP[5].

También posibilita una comunicación sencilla, rápida y sin retardos, al no requerir configurar una conexión. Por otro lado, no emplea control del flujo ni ordena los paquetes, por ello se le califica como no fiable.

En este proyecto se hace uso del protocolo UDP para el envío de los datos, previamente extraídos de las APIs y procesados, al proceso simulado. Indicar que la configuración de la comunicación

[5] TCP (Transmission Control Protocol) es un protocolo de comunicación que garantiza la entrega ordenada y confiable de datos entre dispositivos en una red.

entre los diferentes elementos a través del protocolo no es materia de este proyecto, por ello simplemente se desarrollará un cliente UDP con el objetivo de enviar un objeto JSON a un servidor UDP ya creado y configurado.

3.2 Implementación de la solución

En este apartado se detallará el proceso de implementación del módulo de predicción meteorológica y eléctrica, desde su concepción hasta su instalación y operación en un PLC industrial. La implementación de esta solución implica pasos técnicos que aseguran que el módulo funcione eficiente y confiablemente en un entorno industrial. Se describirán las etapas clave del desarrollo, incluyendo la conexión con las APIs para la obtención de datos, el análisis y manipulación de estos datos, la creación del módulo software, su instalación en el PLC, así como los problemas encontrados y las versiones del software. Finalmente, se presentarán los resultados obtenidos tras la implementación.

3.2.1 Conexión y manipulación de datos de las APIs

En esta sección se explicará cómo se establecieron las conexiones con las APIs de predicción meteorológica y eléctrica, incluyendo los métodos de autenticación, la estructura de las solicitudes y respuestas, y cómo se procesan y manipulan los datos obtenidos para su uso en el módulo.

3.2.1.1 Aemet

Para poder acceder a la API de acceso público que tiene AEMET (Agencia Estatal de Meteorología), se tuvo que solicitar una API KEY, sin ella, la creación de peticiones a esta API resultaría en errores por falta de autenticación. La forma de obtenerla es a través de la siguiente página https://opendata.aemet.es/centrodedescargas/inicio y siguiendo estos pasos:

1. Acceder al apartado Obtención de API KEY.
2. Seleccionar Solicitar.
3. En la nueva pantalla, rellenar el formulario.
4. Enviar la información.
5. Después de este proceso, se recibirá en el buzón de correos asociado al correo electrónico indicado un email con una API KEY.

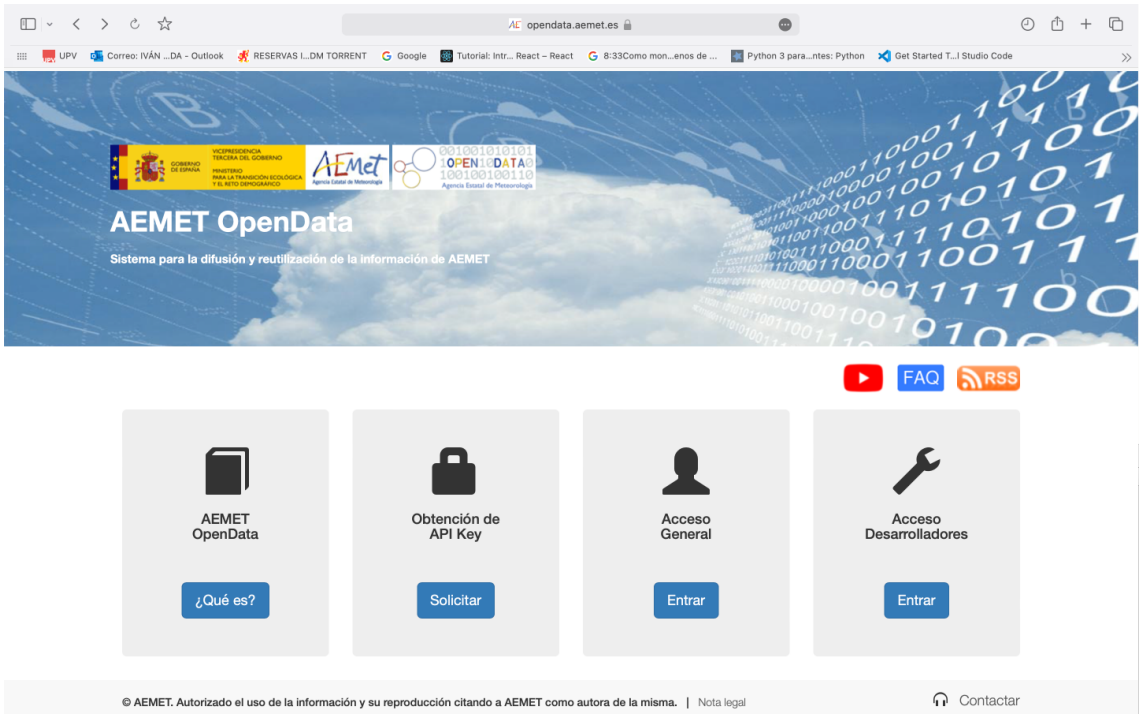

***Figura 9**: página web de AEMET relacionada con la API que ofrece*
***Fuente**: AEMET*

Obtenida esta API KEY, AEMET ofrece servicios que permiten familiarizarse con la API, como acceder a la documentación, hacer pruebas y generar un código de petición de datos muy sencillo en diferentes lenguajes de programación. Todo esto a través del apartado **Acceso Desarrolladores**.

Dentro del apartado de desarrolladores, se encuentra una página de documentación con todas las peticiones disponibles que ofrece la API, además de una breve descripción y la capacidad de hacer pruebas.

En este caso, se ha seleccionado la predicción horaria para el municipio que se pasa como parámetro, que se encuentra representada a través de la siguiente url: https://opendata.aemet.es/opendata/api/prediccion/especifica/municipio/horaria/. Faltaría entonces seleccionar el municipio, para ello la propia documentación ofrece un archivo (diccionario24.xlsx) en formato Excel con los códigos de los municipios. Para este proyecto se ha seleccionado València como municipio, al cual le corresponde el siguiente código: 46250.

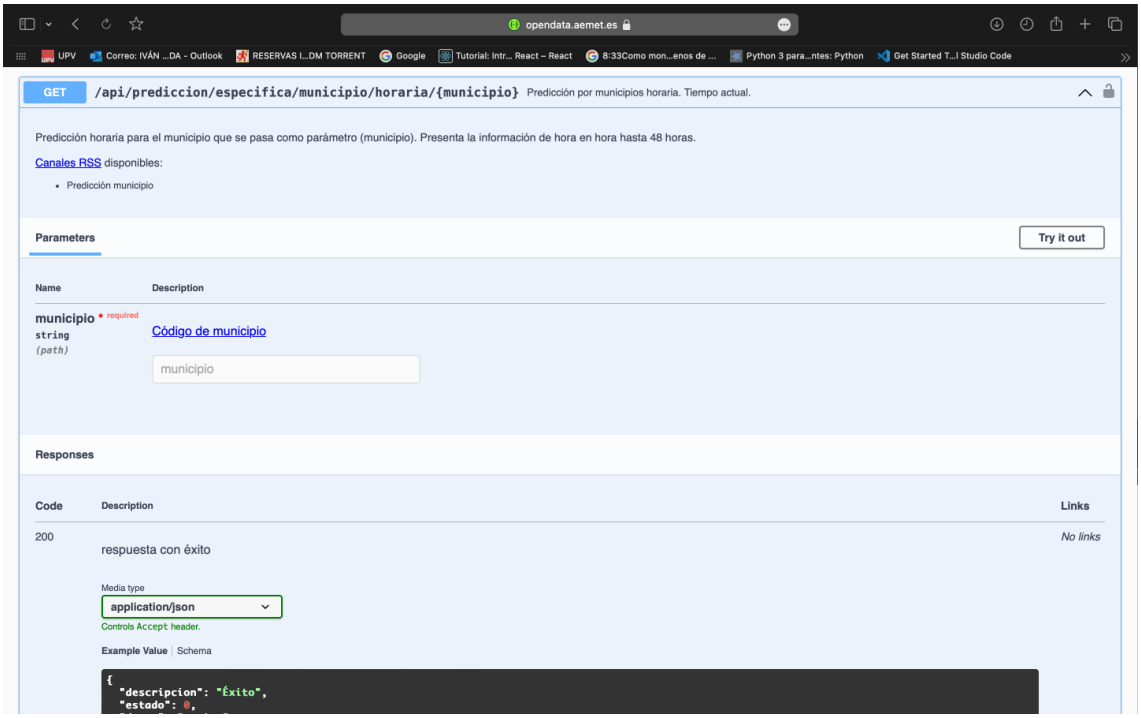

***Figura 10**: endpoint y entorno de pruebas de la API*
***Fuente**: AEMET*

Una vez se conoce el código del municipio, dado que la página ofrece la posibilidad de hacer peticiones y ver las respuestas, se procedió a probar esta funcionalidad. Los pasos necesarios fueron:

1. Indicar el municipio en el campo correspondiente.
2. Introducir la API KEY recibida.
3. Seleccionar el botón *Try it out* que aparece en la Figura 8.

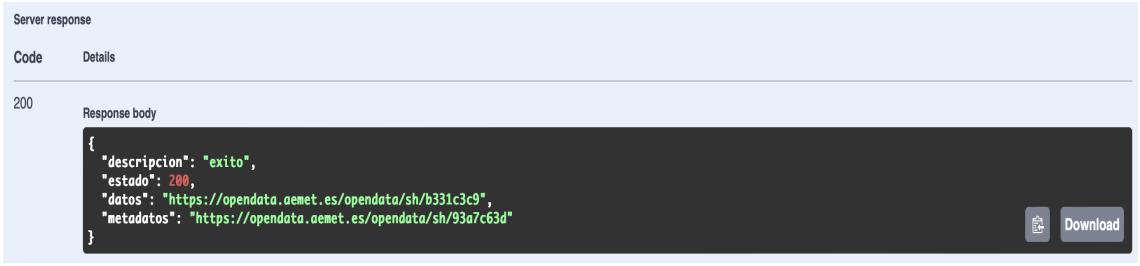

***Figura 11**: respuesta obtenida de la ejecución de prueba en la web*
***Fuente**: AEMET*

Como se puede observar en la figura once, lo que se obtiene al realizar la solicitud es una URL. En este momento, se entendió que sería necesario realizar una nueva petición a la URL indicada en la variable “datos”.

Para ello, se utilizó la extensión **RESTED** en el navegador Mozilla Firefox, la cual permite realizar peticiones https. Al indicar en esta extensión la URL obtenida y ejecutar la petición, se

obtuvo cómo respuesta un objeto JSON con datos sobre el estado del cielo, la probabilidad de precipitación, la probabilidad de tormenta, la sensación térmica, la humedad relativa, etc., separados por horas hasta un máximo de 48 horas.

Después de analizar toda la información obtenida y seleccionar aquella que resultaba clave para este proyecto, se desarrolló en Python un código capaz de realizar las peticiones, obtener los datos, manipularlo y devolver un objeto JSON denominado *AemetJS*.

```
AemetJs = {
    "Current": {
        "wind": None,
        "temp": None
    },
    "Wind": [],
    "Temperature": []
}
```

Como se puede observar en el código anterior, el resultado que se obtendrá de esta API se definirá en un objeto JSON que contendrá los siguientes datos:

- **Current (wind y temp)**: Valor de la velocidad del viento y la temperatura en el momento actual de la petición.
- **Wind**: Un listado de 24 valores con la predicción de la velocidad del viento para las futuras 24 horas contando desde el momento de la ejecución de la petición.
- **Temperature**: Un listado de 24 valores con el valor de la temperatura en Celsius para las futuras 24 horas contando desde el momento de la ejecución de la petición.

Para poder obtener los datos mencionados anteriormente, se ha desarrollado el código indicado a continuación. Se ha creado la función **AemetRequest(URL)**, esta función realiza una solicitud GET a la URL pasada cómo parámetro y devuelve un objeto JSON con la respuesta.

```
# Petición HTTPS
def AemetRequest(URL):
    querystring = {"api_key": AEMET_KEY}
    headers = {'cache-control': "no-cache"}
    response = requests.request("GET", URL, headers=headers,
                params=querystring)
    return response.json()
```

El código desarrollado a continuación realiza la manipulación de los datos. Define una función llamada **get_aemet()** que se encarga de obtener y procesar datos meteorológicos de la Agencia Estatal de Meteorología de España (AEMET). La función realiza las siguientes operaciones:

Primero, llama a la función **AemetRequest(URL)** con la URL **AEMET_URL_OBTAIN_URL** para obtener una URL de datos y luego llama de nuevo a **AemetRequest(URL)** con la URL **AEMET_URL_DATA** para obtener los datos meteorológicos en formato JSON.

```
def get_aemet(self):
        AemetRequest(AEMET_URL_OBTAIN_URL)
        data = AemetRequest(AEMET_URL_DATA)
```

A continuación, se inicializan algunas variables: *hora_actual* almacena la hora actual del sistema e *indice* se inicializa con el valor *None*. Se extrae la lista de temperaturas para el primer día del JSON obtenido y se almacena en *TempList*.

```
        # Variables
        hora_actual = datetime.now().time()
        indice_t = None
        indice_w = None
        day_t = 0
        day_w = 0

        #Datos Temperatura
        TempList = data[0]["prediccion"]["dia"][day_t]["temperatura"]

        for i, elemento in enumerate(TempList):
            if elemento["periodo"] == str(hora_actual.hour).zfill(2):
                indice_t = i
                break

        if indice_t is None:
            day_t += 1
            TempList =
            → data[0]["prediccion"]["dia"][day_t]["temperatura"]
            for i, elemento in enumerate(TempList):
                if elemento["periodo"] ==
                → str(hora_actual.hour).zfill(2):
                    indice_t = i
                    break
```

La función busca en *TempList* el índice del elemento cuya propiedad **periodo** coincide con la hora actual. Una vez encontrado el índice, se calcula el número de elementos restantes en *TempList* desde esa posición. La lista se recorta para incluir solo los elementos desde el índice encontrado hasta el final y luego se amplía con las temperaturas del segundo día.

Se elimina y guarda el primer elemento de *TempList* en *currentT*. Se actualiza el diccionario *AemetJs* con la temperatura actual y la lista de temperaturas restantes.

```
        indice_t = len(TempList) - indice_t
        TempList = TempList[-indice_t:]
```

```
TempList.extend(data[0]["prediccion"]["dia"][day_t +
→ 1]["temperatura"])

currentT = TempList.pop(0)

AemetJs["Current"]["temp"] = int(currentT["value"])
AemetJs["Temperature"] = [int(elemento["value"]) for elemento
→ in TempList]
```

Para los datos del viento, se extrae una lista similar (*WindList*) para el primer día. Se realiza la misma búsqueda y recorte de la lista de acuerdo con la hora actual. La lista se amplía con los datos del segundo día y se filtran los elementos para obtener solo aquellos en posiciones pares.

Se elimina y guarda el primer elemento de *WindList* en *currentW*. Se actualiza el diccionario *AemetJs* con la velocidad del viento actual y la lista de velocidades del viento restantes.

Finalmente, la función devuelve el diccionario *AemetJs* que contiene la temperatura actual, la lista de temperaturas futuras, la velocidad actual del viento y la lista de velocidades futuras del viento.

```
ivmoar@MBP-de-Ivan Prediction_module % /usr/local/bin/python3 /Users/ivmoar/Desktop/Prediction_module/script
s/weather.py
{'Current': {'wind': 8, 'temp': 25}, 'Wind': [11, 15, 18, 20, 19, 21, 20, 16, 13, 11, 8, 7, 8, 8, 7, 8, 7, 5, 4, 5
, 5, 6, 8, 9, 8, 8, 11, 14, 16, 19, 21, 19, 17, 15, 14, 9, 7], 'Temperature': [26, 27, 27, 27, 26, 26, 25, 25, 25,
 24, 24, 23, 23, 23, 23, 23, 23, 22, 22, 22, 21, 22, 23, 24, 25, 26, 26, 25, 25, 26, 26, 26, 25, 25, 25, 24, 24],
```

***Figura 12**: datos finales en objeto JSON de la llamada a la API de AEMET*

En la figura doce se ha realizado un **print(AemetJs)** para poder mostrar los datos finales, después de haber sido procesados, que contiene el objeto *AemetJs*.

3.2.1.2 Tomorrow.io

La forma de acceder a los servicios gratuitos que ofrece Tomorrow.io es mediante la creación de una cuenta. Una vez creada la cuenta y después de iniciar sesión, se accede a la plataforma personal de Tomorrow.io, donde se puede solicitar una API KEY.

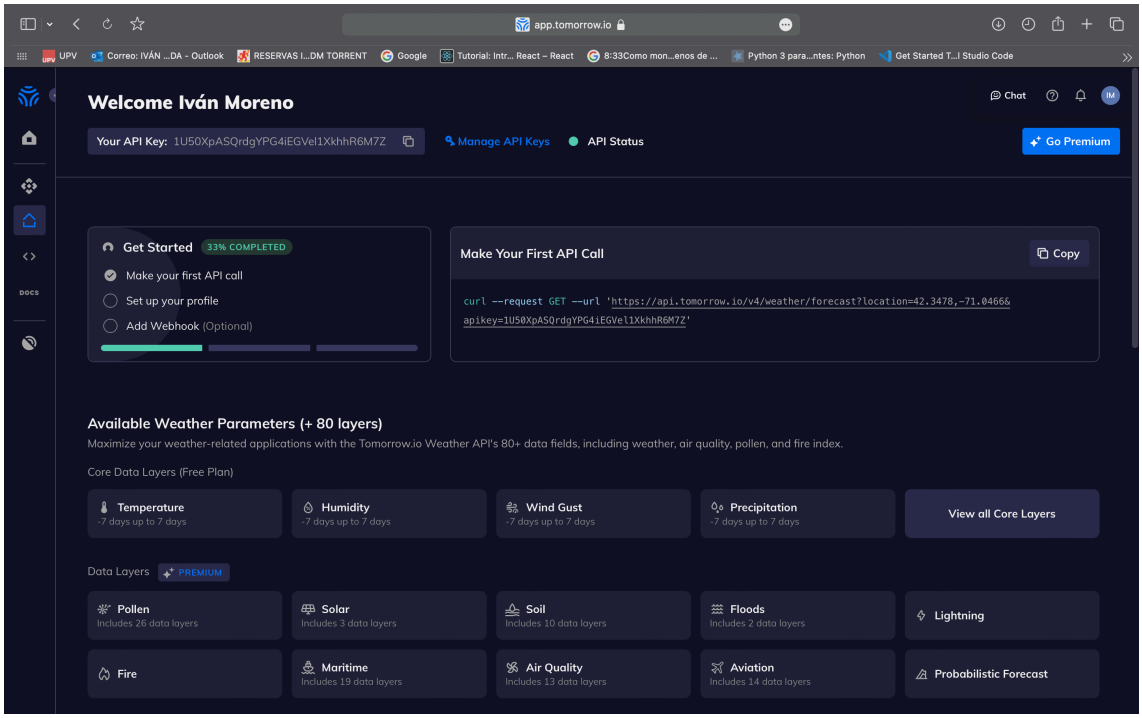

***Figura 13**: plataforma personal de Tomorrow.io*
***Fuente**: Tomorrow.io*

Dentro de la plataforma existe un apartado específico para la API, con una pantalla principal mostrada en la figura trece, una sección para la actividad realizada sobre la API, el manejo de las API KEYs y una sección de documentación.

La documentación de la API consiste en una página web que ofrece información general, ayuda con la autenticación, información detallada sobre cómo empezar a utilizar la API, generación de código sencillo en diferentes lenguajes de programación para crear peticiones, etc. Dentro del apartado *Weather Forecast*, que es la petición seleccionada para este proyecto, se puede indicar un lenguaje de programación, indicar los parámetros para la petición (localización e intervalos de tiempo) y realizar la llamada a la API.

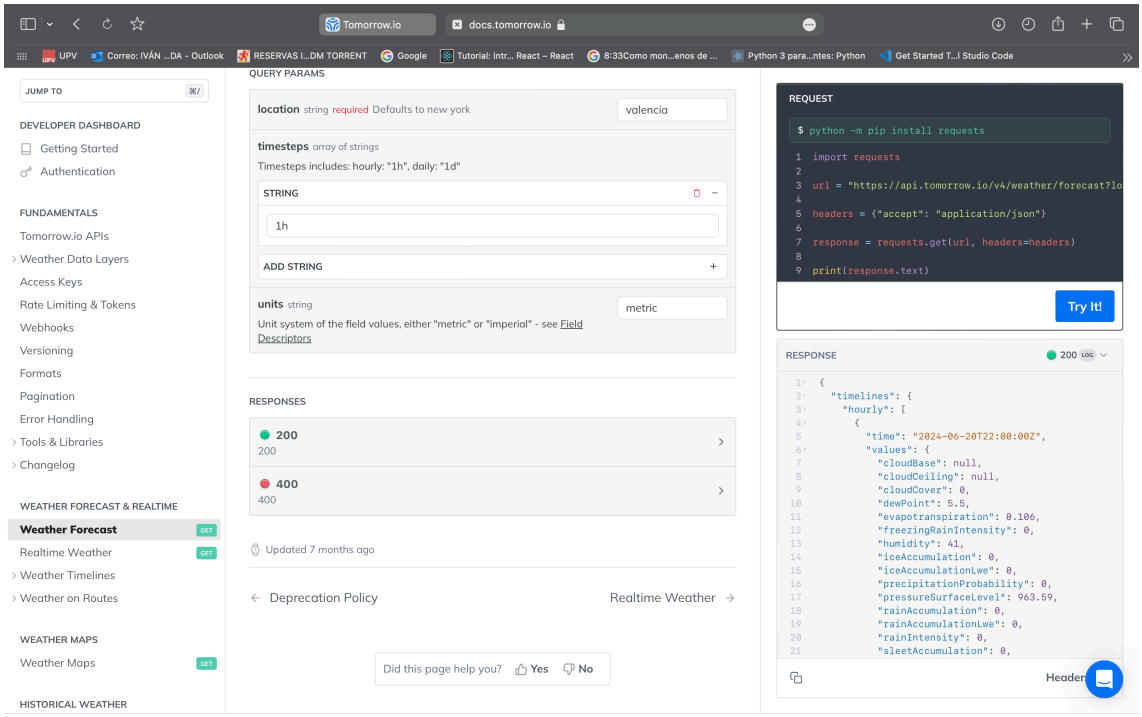

***Figura 14**: página de documentación de la API*
***Fuente**: AEMET*

Cómo se puede apreciar en la figura catorce, en la parte inferior derecha, se ha obtenido un JSON cómo respuesta después de ejecutar una petición a la URL: https://api.tomorrow.io/v4/weather/forecast?, indicando además Valencia como parámetro de localización y "1h" como intervalo de tiempo. En este JSON se recoge información meteorológica actualizada para la ubicación, incluyendo pronósticos por hora para las próximas 120 horas, y pronósticos diarios para los próximos 5 días.

Tras analizar toda la información recopilada y seleccionar los datos más relevantes para este proyecto, se ha desarrollado un código en Python que puede realizar solicitudes, obtener y manipular datos y devolver un JSON.

```
TomorrowJson = {
    "Temperature": [],
    "UV": [],
    "Wind": {
        "Speed": [],
        "Dir": [],
    }
}
```

Como se muestra en el código anterior, el resultado del procesamiento de esta API será un objeto JSON que incluirá los siguientes datos:

- **Temperature**: Un listado de 49 valores con la predicción de la temperatura en grados Celsius del momento actual y para las futuras 48 horas.
- **Wind (Speed y Dir)**: Un listado de 49 valores con la velocidad en km/h y la dirección del momento actual y para las futuras 48 horas.
- **UV**: Un listado de 49 valores con la predicción de la radiación ultravioleta del momento actual y para las futuras 48 horas.

Una vez se decidieron los valores que se iban a obtener de esta API, se procedió con el desarrollo del código. En el siguiente se define una función llamada **TomorrowRequest(URL)** que realiza una solicitud GET a una API usando la biblioteca **requests**. La función toma un parámetro URL, que es la dirección a la que se hará la solicitud. Esta dirección tiene como base: https://api.tomorrow.io/v4/weather/forecast?.

```
# Petición https - Tomorrow.io
def TomorrowRequest(URL):
    params = {
        'location': TOMORROWOI_LOCATION,
        'timesteps': TOMORROWOI_TIMESTEPS,
        'apikey': TOMORROWOI_KEY,
    }
    headers = {"accept": "application/json"}
    response = requests.request("GET", URL, headers=headers,
     → params=params)
    return response.json()
```

Por último, el código mostrado a continuación realiza una solicitud a la API de Tomorrow.io para obtener los datos del pronóstico. Luego, procesa estos datos para extraer la información específica relacionada con la temperatura, el índice UV, la velocidad y dirección del viento para el momento actual y las próximas 48 horas. Esta información se almacena en el objeto *TomorrowJson*, que luego es devuelto como resultado de la función **get_tomorrowoi()**.

```
def get_tomorrowoi(self):
        TomorrowData = TomorrowRequest('https://api.tomorrow.io
        → /v4/weather/forecast?')
        TomorrowData = TomorrowData["timelines"]["hourly"][:49]

        for data in TomorrowData:
            TomorrowJson["Temperature"].append(data
            → ['values']['temperature'])
            TomorrowJson["UV"].append(data['values']['uvIndex'])
            TomorrowJson["Wind"]["Speed"].append(data
            → ['values']['windSpeed'])
            TomorrowJson["Wind"]["Dir"].append(data[
            → 'values']['windDirection'])

        return TomorrowJson
```

```
ivmoar@MBP-de-Ivan Prediction_module % /usr/local/bin/python3 /Users/ivmoar/Desktop/Prediction_module/scripts/weat
her.py
{'Temperature': [23.5, 25.21, 26.16, 26.96, 29.17, 28.09, 24.97, 24.86, 24, 22.51, 21.02, 19.51, 19.18, 18.36, 17.
73, 17.32, 17.5, 17.35, 17.3, 16.8, 16.15, 18.18, 19.92, 21.39, 23.07, 23.54, 23.82, 26.2, 27.6, 26.98, 25.03, 24.
8, 23.26, 22.2, 21.06, 19.92, 19.4, 18.07, 17.7, 17.08, 16.69, 16.41, 16.11, 15.89, 16.26, 18.66, 21.3, 23.12, 24.
57], 'UV': [7, 7, 9, 7, 3, 3, 2, 2, 1, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, 5, 6, 5, 4, 4, 4, 4, 2, 1, 0,
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 1, 3, 5], 'Wind': {'Speed': [1.63, 2.17, 2.98, 2.39, 1.89, 3.95, 5.03, 4.3, 3.
78, 3.14, 2.27, 1.07, 0.65, 0.55, 1.17, 1.38, 0.73, 0.22, 0.07, 0.79, 0.84, 1.13, 0.13, 1.26, 1.64, 1.86, 2.7, 2.5
4, 3.12, 4.35, 5.34, 4.74, 4.83, 3.87, 2.86, 1.85, 1.04, 1.06, 0.7, 0.86, 0.95, 1.2, 1.26, 1.26, 1.36, 1.47, 0.45,
 1.03, 1.75], 'Dir': [166.38, 153.5, 164.06, 193.27, 209.55, 134.44, 122.78, 126.26, 128.98, 130.82, 122.07, 97.86
, 67.85, 319.46, 312, 306.09, 305.85, 164.73, 318.41, 334.25, 314.44, 309.88, 187.03, 145.96, 159.49, 146.99, 149.
42, 156.92, 146.4, 129.51, 119.84, 126.36, 114.66, 107.08, 95.37, 95.46, 72.77, 3.62, 315.51, 301.79, 299.23, 305.
34, 315.4, 316.57, 300.76, 292.78, 243.93, 149.58, 134.63]}}
```


***Figura 15**: datos finales en objeto JSON de la llamada a la API de Tomowwor.io*

En la figura quince se ha realizado un **print(TomorrowJson)** para poder mostrar los datos finales, después de haber sido procesados, que contiene el objeto *TomorrowJson*.

3.2.1.3 Open Weather

Para beneficiarse de la API de Open Weather, es necesario registrarse en su plataforma y seleccionar una tarifa. Para el desarrollo de este proyecto se ha subscrito al plan por defecto *One call API 3.0.* Dentro de las posibilidades que ofrece, se ha seleccionado el tipo de datos que denominan *Current and forecasts weather data*, el cual permite acceder a:

- Meteorología Actual
- El pronóstico minuto a minuto para la próxima hora.
- El pronóstico por hora para 48 horas.
- El pronóstico diario para 8 días.
- Alertas meteorológicas del gobierno.

Una vez se ha iniciado sesión en la plataforma, se obtiene una API KEY accediendo al apartado de *My API keys* en la siguiente página web: https://openweathermap.org.

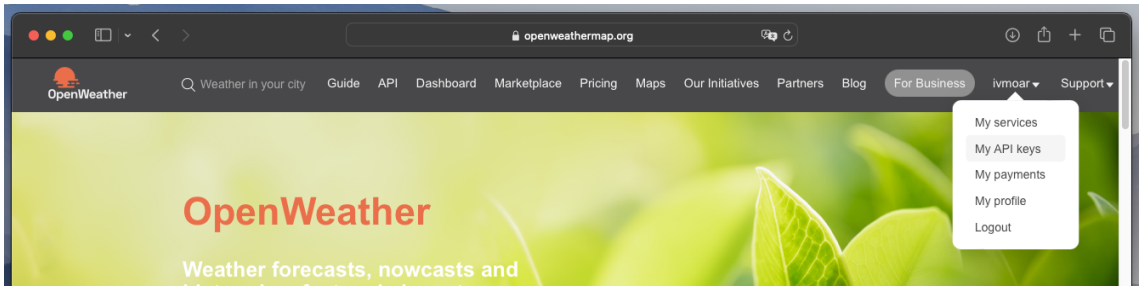

***Figura 16**: página web de OpenWeather*
***Fuente**: OpenWeather*

Entre toda la información, productos e informes que ofrece la plataforma, la sección de API, indicada en el menú superior que aparece en la figura dieciséis, es la más relevante para el desarrollo del proyecto, ya que contiene información clave para el desarrollo de la petición GET. En esta sección API se comenta cómo empezar a realizar peticiones a la API.

Ofrecen documentación sobre la URL base de petición, los parámetros y una respuesta a modo de ejemplo. En la figura diecisiete se puede observar la URL y en la figura dieciocho los parámetros.

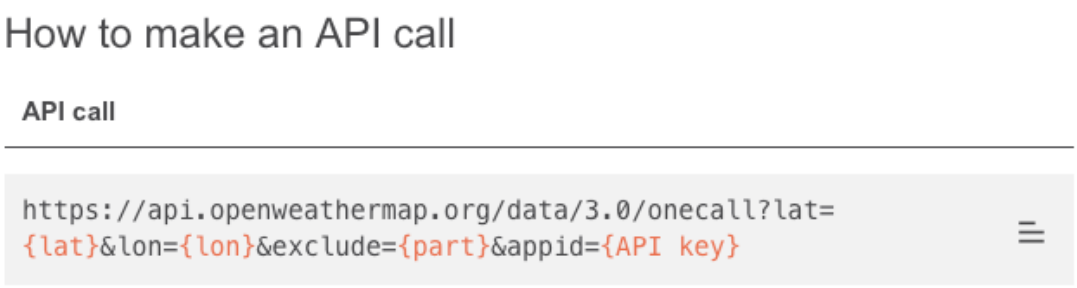

***Figura 17**: base URL para peticiones a la API de OpenWeather*
***Fuente**: OpenWeather*

Parameters

`lat`	required	Latitude, decimal (-90; 90). If you need the geocoder to automatic convert city names and zip-codes to geo coordinates and the other way around, please use our Geocoding API
`lon`	required	Longitude, decimal (-180; 180). If you need the geocoder to automatic convert city names and zip-codes to geo coordinates and the other way around, please use our Geocoding API
`appid`	required	Your unique API key (you can always find it on your account page under the "API key" tab)
`exclude`	optional	By using this parameter you can exclude some parts of the weather data from the API response. It should be a comma-delimited list (without spaces). Available values: • `current` • `minutely` • `hourly` • `daily` • `alerts`
`units`	optional	Units of measurement. `standard`, `metric` and `imperial` units are available. If you do not use the `units` parameter, `standard` units will be applied by default. Learn more
`lang`	optional	You can use the `lang` parameter to get the output in your language. Learn more

***Figura 18**: parámetros para generar la petición de la API*
***Fuente**: OpenWeather*

Después de haber analizado las características de esta API, se desarrolló el código Python para realizar las solicitudes y manejar la respuesta. El objeto JSON que recogerá los datos finales contiene:

- **Current (temp, wind y uvi)**: Valor de la temperatura, de la velocidad del viento y de la radiación en el momento actual de la petición.
- **Wind**: Un listado de 24 valores con la predicción de la velocidad del viento para las futuras 24 horas contando desde el momento de la ejecución de la petición.
- **Temperature**: Un listado de 24 valores con el valor de la temperatura en Celsius para las futuras 24 horas contando desde el momento de la ejecución de la petición.
- **UV**: Un listado de 24 valores con el valor de cantidad de radiación solar para las futuras 24 horas contando desde el momento de la ejecución de la petición.

```
OpenWeatherJs = {
    "Current": {
        "wind": None,
        "temp": None,
        "uvi": None
    },
    "Wind": [],
    "Temperature": [],
    "UV": []
}
```

A continuación, se definió el código que realizará las peticiones tipo GET a la API.

```
# Petición https - OpenWeather
def OpenWeatherRequest(URL):
    params = {
        "lat": OPENWEATHER_VALENCIA_LAT,
        "lon": OPENWEATHER_VALENCIA_LON,
        "exclude": OPENWEATHER_EXCLUDE,
        "appid": OPENWEATHER_KEY
    }
    response = requests.request("GET", URL, params=params)
    return response.json()
```

Para finalizar, se encuentra el código para la manipulación de los datos.

```
def get_openweather(self):
        OpenWeatherData = OpenWeatherRequest(OPENWEATHER_URL_3_0)

        OpenWeatherJs["Current"]["temp"] = round(kelvin_celsius(
        → OpenWeatherData["current"]["temp"]),2)
        OpenWeatherJs["Current"]["wind"] = OpenWeatherData["current"]
        → ["wind_speed"]
        OpenWeatherJs["Current"]["uvi"] = OpenWeatherData["current
```

```
→ "]["uvi"]

OpenWeatherData = OpenWeatherData["hourly"][:24]

for data in OpenWeatherData:
    OpenWeatherJs["Temperature"].append(round(kelvin_cels
    → ius(data['temp']),2))
    OpenWeatherJs["Wind"].append(data['wind_speed'])
    OpenWeatherJs["UV"].append(data['uvi'])

return OpenWeatherJs
```

Debido a que esta API devuelve los datos en un formato muy similar al deseado para este proyecto, el código simplemente recoge en la variable *OpenWeatherData* la respuesta JSON que obtiene de la API y la secciona en los diferentes elementos definidos en el objeto JSON de respuesta.

Al ejecutar **print(OpenWeatherJs)** se obtiene cómo resultado la información de la figura diecinueve.


```
{'Current': {'wind': 2.24, 'temp': 22.03, 'uvi': 0}, 'Wind': [1.11, 1.69, 2.09, 1.9, 1.61, 2.32, 3.88, 4.82, 5.53, 6.62, 7.48,
 7.23, 7.31, 7.2, 6.73, 6.04, 4.89, 3.4, 2.83, 2.38, 1.74, 1.28, 1.67, 1.36], 'Temperature': [21.78, 22.03, 21.68, 21.71, 22.0
6, 22.54, 23.46, 23.79, 23.6, 24.06, 24.64, 24.94, 24.42, 24.04, 23.99, 23.56, 22.95, 22.32, 22.12, 22.01, 21.69, 21.42, 21.24
, 21.01], 'UV': [0, 0, 0, 0.42, 1.32, 2.89, 4.94, 6.96, 8.47, 8.66, 8.5, 7.01, 4.95, 2.91, 1.34, 0.44, 0, 0, 0, 0, 0, 0, 0, 0]
}
```


***Figura 19**: datos finales en objeto JSON de la respuesta de la API de OpenWeather*

3.2.1.4 WeatherStack

El primer paso para poder utilizar esta API consiste en crearse una cuenta. Al hacerlo, se accede a la plataforma donde se puede encontrar información sobre los productos que ofrece, documentación y un apartado denominado *Dashboard*, el cual actúa cómo pantalla de gestión para el usuario.

Se ha investigado la plataforma para conocer el proceso de solicitud de información a través de peticiones de tipo GET. En el apartado de documentación, es donde se puede encontrar la forma de proceder, ya que indica la forma de conseguir la API KEY para autenticarse, posibles errores, tipos de ENDPOINTs a los que acceder e incluso pequeños ejemplos de código para diferentes lenguajes.

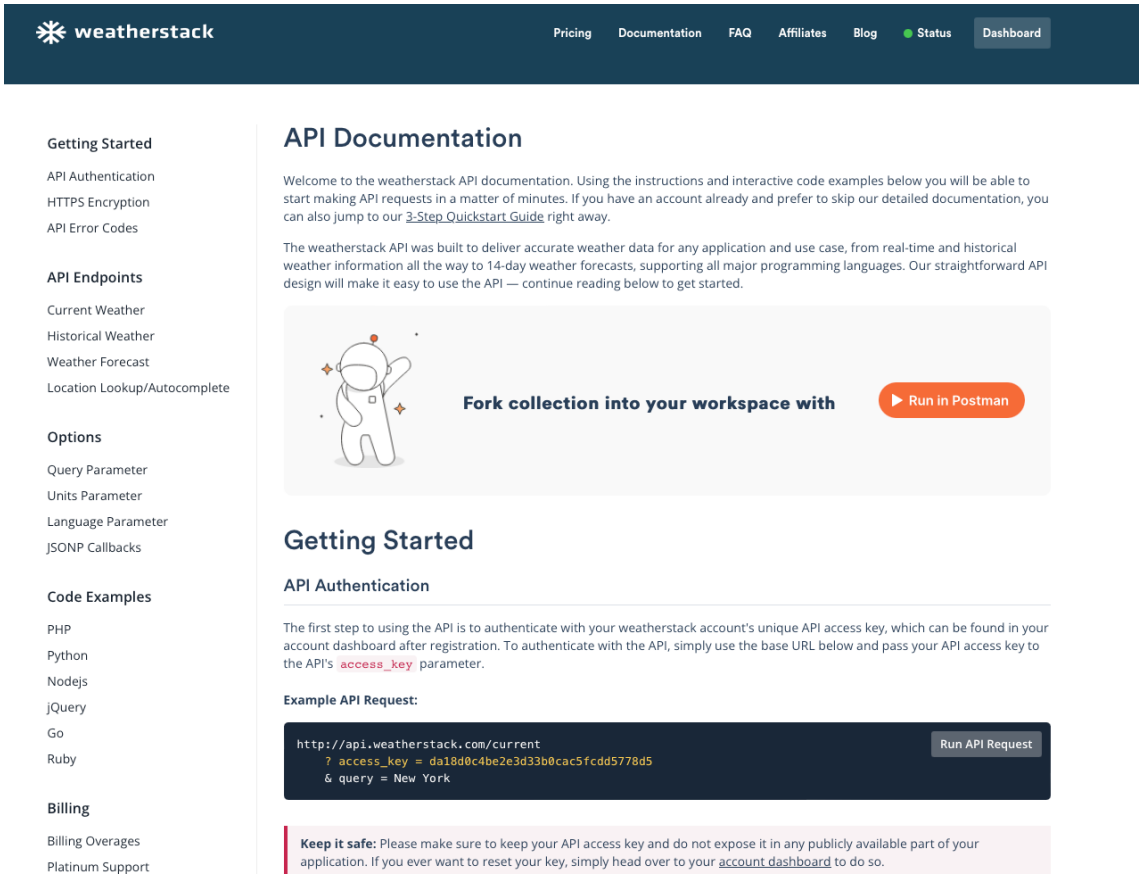

***Figura 20**: página web de documentación para la API de WeatherStack*
***Fuente**: WeatherStack*

El único ENDPOINT disponible en la versión gratuita es el *Current Weather*, el cual ofrece datos meteorológicos en tiempo real para una ubicación seleccionada. En este caso, la URL base para la petición sería http://api.weatherstack.com/current, y los parámetros que se pueden añadir son:

- **Acces_key** (API KEY): Parámetro obligatorio para la autenticación.
- **Query** (localización): Parámetro obligatorio que indica la localización sobre la cual obtener los datos.
- **Units** (Unidades): Parámetro opcional para obtener los datos en una medida seleccionada.
- **Language** (Idioma): Parámetro opcional para seleccionar el idioma con el que se creará la respuesta de la API.
- **Callback**: Parámetro opcional para definir una función de callback.

Cómo se puede apreciar en la figura veinte, aparece un botón **Run API Request**, al seleccionarlo se activa una funcionalidad que permite ejecutar una llamada con ciertos valores predeterminados y mostrar el resultado obtenido en una nueva pantalla.

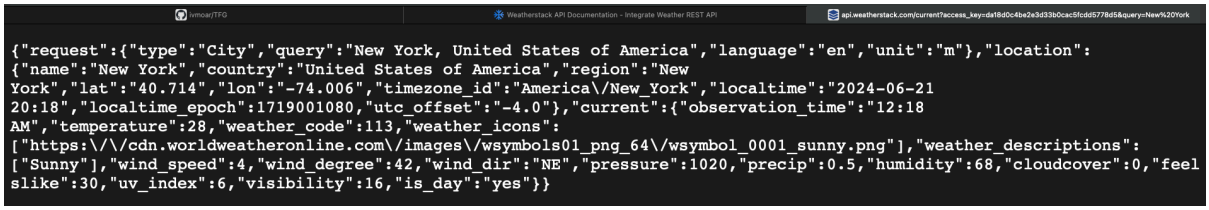

Figura 21: respuesta de la API a una llamada de prueba

Gracias a la información obtenida después de realizar la prueba y al código de ejemplo indicado en la propia documentación, se ha desarrollado el siguiente código Python para realizar la llamada, manipular los datos y obtener un resultado útil para el objetivo de este proyecto.

```
WeatherstackJson = {
    "Wind": {"wind_speed": None, "wind_degree": None, "wind_dir":
    → None},
    "UVindex": None,
    "Temperature": None
}
```

El código anterior refleja el objeto JSON desarrollado, el cual contendrá el resultado final de los datos, incluyendo los valores actuales de:

- **Wind (wind_speed, wind_degree, wind_dir)**: Valor en tiempo real de velocidad, grados y dirección del viento.
- **UVindex**: Valor en tiempo real de la cantidad de radiación ultravioleta.
- **Temperature**: Valor en tiempo real de la temperatura.

Teniendo clara la estructura final del JSON que se entregará como resultado de la ejecución de la llamada esta API, se procedió al desarrollo del código para realizar la petición GET. Los valores almacenados en las constantes de WEATHERSTACK_KEY, WEATHERSTACK_VALENCIA_QUERY y WEATHERSTACK_UNITS son:

- La API KEY obtenida: da18d0c4be2e3d33b0cac5fcdd5778d5
- El valor de la localización: Valencia, Spain.
- Las unidades con las que obtener los datos: m (metric).

`m` for Metric:

Parameter	Units
`units = m`	temperature: Celsius
`units = m`	Wind Speed/Visibility: Kilometers/Hour
`units = m`	Pressure: MB - Millibar
`units = m`	Precip: MM - Millimeters
`units = m`	Total Snow: CM - Centimeters

***Figura 22**: unidades de los datos de respuesta de la API*
***Fuente**: WeatherStack*

```
# Petición https - WeatherStack
def WeatherstackRequest(URL):
    params = {
        'access_key': WEATHERSTACK_KEY,
        'query': WEATHERSTACK_VALENCIA_QUERY,
        'units': WEATHERSTACK_UNITS
    }
    headers = {'cache-control': "no-cache"}
    response = requests.request("GET", URL, headers=headers,
     → params=params)
    return response.json()
```

Posteriormente, se desarrolló el código para la obtención de la información final. Este realiza una solicitud a la API para obtener los datos meteorológicos actuales. Extrae valores específicos de la respuesta JSON (indicados anteriormente), y los organiza en un diccionario llamado *WeatherstackJson*, el cual es retornado al finalizar el método.

```
def get_weatherstack(self):
        WeatherstackData = WeatherstackRequest(WEATHERSTACK_URL)
        WeatherstackJson["Wind"]["wind_speed"] = WeatherstackData[
        → "current"]["wind_speed"]
        WeatherstackJson["Wind"]["wind_degree"] = WeatherstackData["
        → current"]["wind_degree"]
        WeatherstackJson["Wind"]["wind_dir"] = WeatherstackData["
        → current"]["wind_dir"]
        WeatherstackJson["UVindex"] = WeatherstackData["cur
        → rent"]["uv_index"]
        WeatherstackJson["Temperature"] = WeatherstackData["
        → current"]["temperature"]

        return WeatherstackJson
```

```
ivmoar@MBP-de-Ivan Prediction_module % /usr/local/bin/python3 /Users/ivmoar/Desktop/Prediction_module/scripts/weat
her.py
{'Wind': {'wind_speed': 4, 'wind_degree': 149, 'wind_dir': 'SSE'}, 'UVindex': 7, 'Temperature': 25}
```

***Figura 23**: datos finales en objeto JSON de la llamada a la API de WeatherStack*

En la figura veintitrés se ha realizado un **print(WeatherstackJson)** para poder mostrar los datos finales, después de haber sido procesados, que contiene el objeto *WeatherstackJson*.

3.2.1.5 Preciodelaluz

El funcionamiento de esta API es el más sencillo de todas, ya que no requiere ni de la creación de una cuenta ni de la obtención de una APIKEY. La página web con la información para la creación de solicitudes es https://api.preciodelaluz.org.

***Figura 24**: página web de información sobre la API de Preciodelaluz*
***Fuente**: preciodelaluz.org*

Cómo se puede apreciar en la figura veinticuatro, el único procedimiento para obtener los datos es realizar una solicitud al ENDPOINT deseado, seleccionando como único parámetro la zona de la cual obtener la información. Las únicas dos posibilidades para el parámetro de zona son:

- **PCB**: Comprende la Península, Canarias y Baleares.
- **CYM**: Comprende a Ceuta y Melilla.

Antes de comenzar el desarrollo, se realizaron algunas llamadas gracias a la funcionalidad de *Test* que ofrece la página. Se pueden realizar peticiones de prueba para cada ENDPOINT seleccionando los botones **Test** indicados en la figura veinticuatro.

Para el desarrollo de este trabajo, se creó el siguiente JSON con el fin de recoger los datos más relevantes de todos los ENDPOINTS. Posteriormente, se creó la función para manipular estos datos.

```
ElectricityJSON = {
    "Current": {
        "Time": None,
        "Price": None
    },
    "AVG": None,
    "Max": {
        "Time": None,
        "Price": None
    },
    "Min": {
        "Time": None,
        "Price": None
    },
    "Next": None
}
```

- **Current (Time, Price)**: Recoge el valor del precio de la electricidad en el periodo que comprende la hora actual. Es decir, si son las 17:40, es el valor estipulado entre las 17:00 y las 18:00, ya que el precio se estipula para cada hora.
- **AVG**: Recoge el valor del precio promedio de la electricidad en el día.
- **Max (Time, Price)**: Recoge el precio máximo de la electricidad en el día y el periodo de tiempo en el que ha sido o será.
- **Min (Time, Price)**: Recoge el precio mínimo de la electricidad en el día y el periodo de tiempo en el que ha sido o será.
- **Next**: Listado de valores Periodo – Precio para las horas restantes del día, comenzando el periodo de tiempo inmediatamente posterior al momento actual.

```
# Petición https - preciodelaluz
def PreciodelaluzRequest(URL):
    params = {
        'zone': ZONE_PENINSULA_CANARIAS_BALEARS
    }
    headers = {'cache-control': "no-cache"}
    response = requests.request("GET", URL, headers=headers, params=
    → params)
    return response.json()
```

En el código anterior se visualiza el desarrollo para realizar las peticiones a los ENDPOINTs. En este caso, se ha seleccionado la constante ZONE_PENINSULA_CANARIAS_BALEARS, la cual almacena la zona PCB mencionada anteriormente, cómo parámetro de zona.

```
def get_preciodelaluz(self):
        AVGData = PreciodelaluzRequest('https://api.preciode
        → laluz.org/v1/prices/avg?')
        ElectricityJSON["AVG"] = AVGData["price"]

        MaxData = PreciodelaluzRequest('https://api.preciodel
        → aluz.org/v1/prices/max?')
        ElectricityJSON["Max"]["Time"] = MaxData["hour"]
        ElectricityJSON["Max"]["Price"] = MaxData["price"]

        MinData = PreciodelaluzRequest('https://api.preciod
        → elaluz.org/v1/prices/min?')
        ElectricityJSON["Min"]["Time"] = MinData["hour"]
        ElectricityJSON["Min"]["Price"] = MinData["price"]

        CurrentData = PreciodelaluzRequest('https://api.preciode
        → laluz.org/v1/prices/now?')
        ElectricityJSON["Current"]["Time"] = CurrentData["hour"]
        ElectricityJSON["Current"]["Price"] = CurrentData["price"]

        indice = CurrentData["hour"][-2:]
        aux_indice = int(indice) + 1
        cadena = indice + '-' + str(aux_indice)

        AllDayData = PreciodelaluzRequest('https://api.preciode
        → laluz.org/v1/prices/all?')
        AllDayData = {clave: valor for clave, valor in AllDayD
        → ata.items() if clave >= cadena}

        lista_json = [{"Time": valor["hour"], "Price": valor["price"]}
        → for valor in AllDayData.values()]
        ElectricityJSON["Next"] = lista_json

        return ElectricityJSON
```

Se define un método llamado **get_preciodelaluz()** que obtiene y organiza los datos obtenidos de la API. A continuación, se explica cada parte del proceso:

Primero, obtiene el precio promedio de la electricidad haciendo una solicitud a la API con la URL https://api.preciodelaluz.org/v1/prices/avg? usando la función **PreciodelaluzRequest(URL)**. Luego, almacena el precio promedio obtenido en el diccionario *ElectricityJSON* bajo la clave *AVG*.

Seguidamente, obtiene el precio máximo de la electricidad y la hora correspondiente haciendo otra solicitud a la API con la URL https://api.preciodelaluz.org/v1/prices/max?. Almacena la hora

y el precio máximo en el diccionario *ElectricityJSON* bajo las claves *["Max"]["Time"]* y *["Max"]["Price"]* respectivamente.

Después, obtiene el precio mínimo de la electricidad y la hora correspondiente haciendo una solicitud a la API con la URL https://api.preciodelaluz.org/v1/prices/min?. Almacena la hora y el precio mínimo en el diccionario *ElectricityJSON* bajo las claves *["Min"]["Time"]* y *["Min"]["Price"]* respectivamente.

A continuación, obtiene el precio actual de la electricidad y la hora correspondiente haciendo una solicitud a la API con la URL https://api.preciodelaluz.org/v1/prices/now?. Almacena la hora y el precio actual en el diccionario *ElectricityJSON* bajo las claves *["Current"]["Time"]* y *["Current"]["Price"]* respectivamente.

Luego, determina el índice de la hora siguiente extrayendo los últimos dos caracteres de la hora actual (*indice*), convirtiéndolos a entero, sumándoles uno (*aux_indice*), y creando una cadena (*cadena*) que representa el rango de la hora actual a la siguiente.

Después, obtiene los precios de electricidad para el resto del día haciendo una solicitud a la API con la URL https://api.preciodelaluz.org/v1/prices/all?. Filtra los datos obtenidos para incluir solo las horas a partir de *cadena* y almacena los precios y horas resultantes en una lista de diccionarios bajo la clave *Next* de *ElectricityJSON*.

Finalmente, el método devuelve el diccionario *ElectricityJSON* que ahora contiene los precios de electricidad promedio, máximo, mínimo, actual y los precios futuros para el resto del día.

```
ivmoar@MBP-de-Ivan Prediction_module % /usr/local/bin/python3 /Users/ivmoar/Desktop/Prediction_module/script
s/electricity.py
{'Current': {'Time': '10-11', 'Price': 42.2}, 'AVG': 77.2, 'Max': {'Time': '22-23', 'Price': 142.7}, 'Min':
{'Time': '11-12', 'Price': 41.38}, 'Next': [{'Time': '11-12', 'Price': 41.38}, {'Time': '12-13', 'Price': 48
.64}, {'Time': '13-14', 'Price': 42.99}, {'Time': '14-15', 'Price': 42.32}, {'Time': '15-16', 'Price': 43.69
}, {'Time': '16-17', 'Price': 50.96}, {'Time': '17-18', 'Price': 42.68}, {'Time': '18-19', 'Price': 48.9}, {
'Time': '19-20', 'Price': 55.6}, {'Time': '20-21', 'Price': 80.98}, {'Time': '21-22', 'Price': 122.27}, {'Ti
me': '22-23', 'Price': 142.7}, {'Time': '23-24', 'Price': 126.82}]}
```

***Figura 25**: datos finales en objeto JSON de la llamada a la API de Preciodelaluz*

En la figura veinticinco aparece el resultado de ejecutar **print(ElectricityJSON)**, este muestra los datos finales que contiene el objeto *ElectricityJSON*.

3.2.2 Creación del módulo software

En este apartado se abordará como fue el desarrollo del módulo software de predicción, más específicamente se verá la integración del código de las APIs mencionado en el apartado anterior en clases Python, el diseño de las páginas web que acompañan al servidor para poder interactuar con el módulo, la creación del servidor Flask y del cliente UDP.

En la siguiente figura veintiséis se encuentra una aproximación visual de la estructura interna del módulo software de predicción. Esta estructura es similar a la que se encontraría en aplicaciones web desarrolladas en el lenguaje de programación Python.

***Figura 26**: estructura de archivos del módulo software*
***Fuente**: Elaboración propia*

- **Start.py**: es el punto de entrada del servidor Flask. Configura y arranca el servidor, define las rutas para manejar las solicitudes HTTP y realiza el envío de los datos al proceso simulado.
- **Templates**: Contiene las plantillas HTML utilizadas por Flask para renderizar las páginas web.
    - Index.html: Página principal.
    - Data.html: Página para mostrar los datos obtenidos.
- **Static**: Contiene recursos estáticos.
    - Styles.css: Archivo CSS para estilizar las páginas web.
    - Images: carpeta para almacenar imágenes usadas en las páginas web.
- **Scritps**: contiene los scripts para la lógica
    - Constants.py: Define las constantes utilizadas para las creaciones de las peticiones a las APIs.
    - Functions.py: Contiene funciones de llamada para cada API y funciones auxiliares.
    - Init.py: Inicializa el paquete de scritps y convierte la carpeta en un módulo.

- Weather.py: Define la clase para manejar las peticiones a las APIs meteorológicas.
- Electricity.py: Define la clase para manejar las peticiones a la API eléctrica.

Se ha decidido seguir esta estructura debido a los beneficios que tiene en términos de organización, mantenibilidad y rendimiento. A continuación, se describirán con más detalle estos beneficios:

- **Organización y estructura**: Separar los archivos proporciona una estructura clara y comprensible, dado que cada tipo de recursos tiene su propio directorio. Además, con una estructura bien definida es más fácil comprender y mantener el código.
- **Reutilización de código**: El uso de plantillas (*templates*) permite la reutilización de componentes en una interfaz gráfica.
- **Rendimiento**: Los archivos estáticos (CSS, imágenes, etc.) se pueden cachear de manear eficiente por los navegadores, así como los scripts se pueden cargar de manera asíncrona permitiendo la carga de varios elementos simultáneamente.
- **Buenas prácticas de desarrollo**: Las estructuras comúnmente aceptadas mejoran la calidad del código y facilitan la integración con herramientas y bibliotecas de terceros. En efecto, muchos frameworks y bibliotecas (como Flask o Django) están diseñados para trabajar con una estructura de proyecto específica que incluye directorios *static* y *templates*.

3.2.2.1 Modelo de clases

El modelo de datos es una representación abstracta que define la estructura y las interacciones de las clases dentro de un sistema orientado a objetos. Estos modelos se utilizan principalmente en el contexto del diseño de software para planificar y visualizar la estructura de un sistema.

A continuación, se muestra en la figura veintisiete las clases más importantes que intervienen en el módulo software, así como las relaciones entre ellas. Es necesario indicar que no se mencionan explícitamente los archivos HTML y CSS debido a que no son relevantes para las relaciones entre las clases de Python.

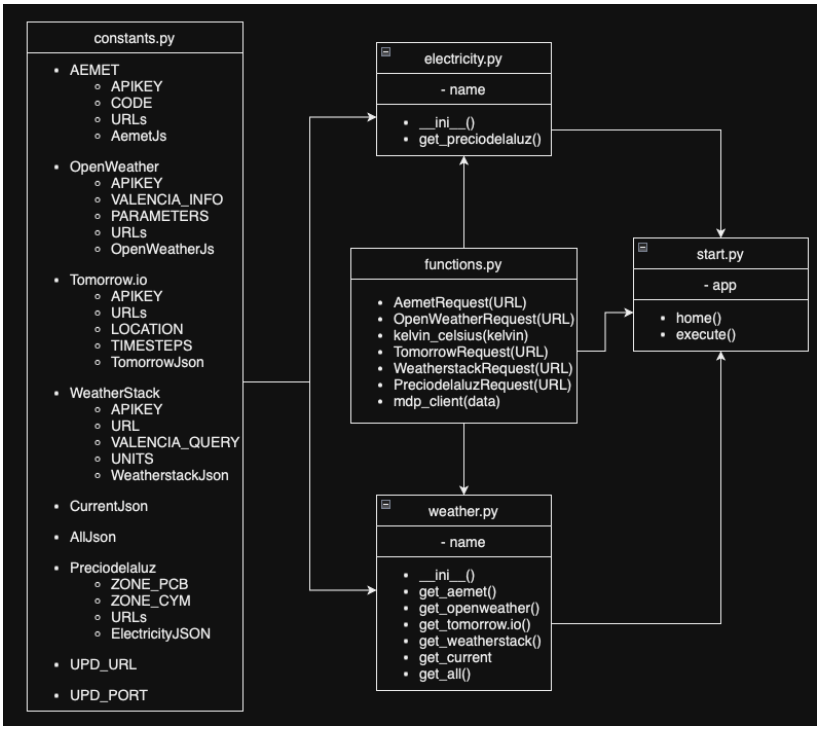

***Figura 27**: diagrama de clases*
***Fuente**: Elaboración propia*

3.2.2.2 Desarrollo de la clase meteorológica

Debido a la estructuración que se decidió seguir para la creación del módulo software, se diseñó un archivo Python llamado **weather.py** para unificar todas las peticiones a las APIs y el procesamiento de las respuestas. En este archivo realiza una importación de funcionalidades de otros módulos, crea una clase denominada **Weather** y se definen en ella un conjunto de funciones.

```
from datetime import datetime
from constants import *
from functions import *

class Weather:
    def __init__(self, name):
        self.name = name

    def get_aemet(self):…

    def get_openweather(self):…

    def get_tomorrowoi(self):…

    def get_weatherstack(self):…

    def get_current(self):

    def get_all(self):
```

A continuación, se definirán que son las importaciones, las clases y las funciones en Python, de forma breve. Posteriormente, se detallará el contenido del archivo weather.py en profundidad.

- **Importaciones**: Sirven para traer funcionalidades definidas en otros módulos o archivos. Las importaciones realizadas en este archivo son:
  - **datetime**: Se importa para obtener la hora y la fecha actual.
  - **constants**: Archivo que contiene variables con valores constantes sobre las APIs (URLs, parámetros, etc.) y estructuras de datos (JSON).
  - **functions**: Archivo que contiene las funciones de las peticiones a las APIs y funciones auxiliares.
- **Clases**: Son estructuras de programación que definen un tipo de objeto, especificando tanto su estructura (datos o atributos) como sus comportamientos (métodos o funciones). Las clases permiten la abstracción y la reutilización del código, facilitando la creación de programas más organizados y modulares. En este archivo se ha desarrollado la clase Weather. La función **__init__(self, name)** actúa como constructor de ella.
- **Funciones**: Bloques de código reutilizables que ejecutan un conjunto definido de instrucciones. En este archivo se han definido las funciones listadas a continuación, sin embargo, es importante destacar que todas las funciones han sido explicadas anteriormente en más detalle en el apartado 3.2.1, salvo **get_current(self)** y **get_all(self)** que se explicarán a continuación.
  - **get_aemet(self)**
  - **get_openweather(self)**
  - **get_tomorrowio(self)**
  - **get_weatherstack(self)**
  - **get_current(self)**: Obtiene los datos actuales de AEMET, Weatherstack, OpenWeather y Tomorrow.io, calcula los promedios de temperatura, velocidad del viento y radiación. Posteriormente actualiza la estructura *CurrentJson* con estos resultados.
  - **get_all(self)**: Obtiene los datos de pronóstico de AEMET, Tomorrow.io y OpenWeather, calcula los promedios de temperatura y velocidad del viento para las próximas 24 horas de las 3 APIs, y actualiza la estructura *AllJson* con estos promedios junto con los datos de la radicación UV promediados de Tomorrow.io y OpenWeather.

```
def get_current(self):
        aemet = self.get_aemet()
        weatherstack = self.get_weatherstack()
        tomorrowio = self.get_tomorrowoi()
        openweather = self.get_openweather()
```

```
        current_temp = (aemet["Current"]["temp"] + weatherstack["Temp
        → erature"] + tomorrowio["Temperature"][0] + openweath
        → er["Current"]["temp"]) / 4
        current_wind = (aemet["Current"]["wind"] + weatherstack["Win
        → d"]["wind_speed"] + tomorrowio["Wind"]["Speed"][0] + openw
        → eather["Current"]["wind"]) / 4

        CurrentJson["Wind"] = round(current_wind, 2)
        CurrentJson["Temperature"] = round(current_temp, 2)
        CurrentJson["UVindex"] = round((weatherstack["UVindex"] + open
        → weather["Current"]["uvi"]) / 2, 2)

        return CurrentJson


def get_all(self):
        aemet = self.get_aemet()
        tomorrowio = self.get_tomorrowoi()
        openweather = self.get_openweather()

        # Temperatura promedio para las próximas 24 horas
        list1 = openweather["Temperature"]
        list2 = aemet["Temperature"][:24]
        list3 = tomorrowio["Temperature"][1:25]

        for i in range(24):
            aux = round(((list1[i] + list2[i] + list3[i]) / 3), 2)
            AllJson["Temperature"].append(aux)

        # Velocidad viento promedio para las próximas 2.4. horas
        list1 = openweather["Wind"]
        list2 = aemet["Wind"][:24]
        list3 = tomorrowio["Wind"]["Speed"][1:25]

        for i in range(24):
            aux = round(((list1[i] + list2[i] + list3[i]) / 3), 2)
            AllJson["Wind"].append(aux)

        # Radición promedio para las próximas 24 horas
        list1 = openweather["UV"]
        list2 = tomorrowio["UV"][1:25]

        for i in range(24):
            aux = round(((list1[i] + list2[i]) / 2), 2)
            AllJson["UV"].append(aux)

        return AllJson
```

3.2.2.3 Desarrollo de la clase eléctrica

Para los datos eléctricos, se ha seguido el mismo proceso explicado en el apartado anterior. En este caso, se ha creado el archivo **electricity.py** que crea una clase denominada **Electricity**, la cual contiene la función **__init__(self, name)** como constructor de la clase y la función

get_preciodelaluz(self), explicada en detalle en el apartado 3.2.1 sección 3.2.1.5, dado que sólo se obtienen datos de una única API.

```
from constants import *
from functions import *

class Electricity:
    def __init__(self, name):
        self.name = name

    def get_preciodelaluz(self):
```

3.2.2.4 Creación de la vista para visualizar los datos

Una parte fundamental del módulo software es la visualización de los datos obtenidos de las APIs. Después de recibir las respuestas de las APIs solicitadas a través de la página web principal denominada **index.html**, los datos son procesados por el mismo servidor Python. El resultado de este procesamiento se presenta al usuario en una nueva página web llamada **data.html**. Ambas páginas webs están enlazadas a una hoja de estilos externa (**styles.css**) que define el aspecto visual.

El archivo index.html define una página web que presenta un módulo de predicción con dos secciones principales dentro de un contenedor. Se pueden observar estas dos secciones en la siguiente figura.

***Figura 28**: página web index.html*
***Fuente**: index.html*

En la primera sección se proporciona una bienvenida y una descripción del propósito del módulo. Además, cómo configurar las predicciones meteorológicas y eléctricas, destacando la posibilidad de elegir entre diferentes fuentes de datos y la opción de obtener información del momento actual.

En la segunda sección se encuentra un formulario, el cual le permite al usuario configurar las preferencias. El formulario está dividido en dos áreas: una para la configuración de la meteorología y otra para la configuración de la electricidad. Para la meteorología, se ofrece un menú desplegable donde seleccionar la fuente de datos (API) y opciones de sí/no para obtener información actual. Para la electricidad, se muestra un campo de texto con una fuente predefinida y opciones similares de sí/no para obtener los datos actuales.

El formulario incluye el botón *Ejecutar código* para enviar la configuración y solicitar los datos, y el botón *Limpiar* para dejar el formulario con los valores predefinidos.

Una vez se ha realizado la configuración y se ha seleccionado el botón de ejecución del código, el módulo realizará las peticiones, obtendrá los resultados, los procesará y los mostrará a través del archivo data.html en una página web. Esta nueva página está estructurada para mostrar una sección sobre la respuesta del servidor UDP y dos secciones independientes de datos.

***Figura 29**: página web data.html*
***Fuente**: data.tml*

La sección superior muestra un mensaje informativo sobre el resultado del envío de los datos al servidor UDP. Las siguientes dos secciones tienen un título principal que describe el tipo de datos que se muestran: *Datos meteorológicos* y *Datos eléctricos*.

A continuación, se muestran dos elementos en cada sección que recogen de manera estructurada los datos. En este caso, para la meteorología encontramos el elemento *Petición*, en el que se recoge la predicción para las futuras 24 horas de la temperatura, el viento y la radiación. Posteriormente, se encuentra el elemento *Valores actuales*, que muestra el valor actual de los mismos tres datos mencionados anteriormente.

En el caso de la sección eléctrica, el primer elemento *Valores de hoy*, recoge el precio medio, máximo, mínimo y los futuros precios de la electricidad para las horas que quedan del día. En el segundo elemento de *Valores actuales*, se encuentra el precio de la electricidad para la hora de la petición.

Los datos específicos se pasan de forma dinámica desde el servidor Flask, en la siguiente sección se verá con detalle este procedimiento.

Finalmente, se encuentra un botón *Volver* que permite acceder a la página anterior para poder realizar una nueva configuración y volver a solicitar datos.

3.2.2.5 Creación de un servidor con Flask

Como se ha comentado anteriormente en este proyecto, el servidor Flask es una de las piezas clave en el desarrollo de la solución, ya que permite la configuración de los datos a solicitar, muestra el resultado de las llamadas a las APIs a través de páginas webs y envía los datos por UDP. A continuación, se detallará el contenido del archivo **start.py**, que es donde se encuentra el desarrollo del servidor.

```
from flask import Flask, render_template, request
import sys
import os

# Obtener la ruta del directorio 'scripts'
scripts_path = os.path.join(os.path.dirname(__file__), 'scripts')

# Añadir 'scripts' al sys.path
if scripts_path not in sys.path:
    sys.path.append(scripts_path)

from scripts import weather, electricity, functions
```

En el código anterior se puede observar los módulos importados para la creación del servidor y la obtención de los datos:

- **Flask**: Es un microframework web que permite la creación de aplicaciones web de manera sencilla y rápida.
- **render_template**: Es una función de Flask que se utiliza para renderizar plantillas *HTML*. Permite enviar datos desde la lógica de la aplicación Python a los archivos *HTML* para generar contenido dinámico.
- **request**: Es un objeto en Flask que contiene todos los datos enviados por el cliente al servidor. Se utiliza para acceder a datos de formularios enviados a través de métodos HTTP como GET y POST

- **sys** y **os**: Permiten definir la ruta del directorio de *scripts* para que se puedan encontrar los módulos situados en este.
- **weather**, **electricity y functions**: Contienen las clases y funciones descritas en las secciones 3.2.2.2 y 3.2.2.3 y la función para crear un cliente UDP.

```
app = Flask(__name__)

# Página de inicio del servidor
@app.route("/")
def home():
    return render_template('index.html')
```

El código muestra parte del archivo **start.yp**, en (*app = Flask(__name__)*) se crea una instancia de Flask llamada *app*, el cual será el punto de entrada principal para la aplicación web.

Del mismo modo, se define la ruta principal del servidor (“/”) y la función **home()** que renderiza el template index.html cuando se accede a esta ruta.

A continuación, se define la ruta */execute* que espera peticiones POST del formulario que se encuentra en el archivo index.html mencionado en al apartado 3.2.2.4. Cuando se recibe una petición, la función **execute()** maneja esta petición.

```
# Llamada a la ejecución de la predicción
@app.route('/execute', methods=['POST'])
def execute():
    param1 = request.form['weather-source']
    param2 = request.form['w-option']
    param3 = request.form['electricity-source']
    param4 = request.form['e-option']

    weather_object = weather.Weather("W")
    electricity_object = electricity.Electricity("E")

    weather_data = None
    current_W_data = None
global_weather = {
    "Current": {
        "wind": None,
        "temp": None,
        "uv": None
    },
    "Wind": [],
    "Temperature": [],
    "UV": []
    }

    # Weather
    if param1 == "Aemet":
        weather_data = weather_object.get_aemet()
        w_temperature_24 = weather_data["Temperature"]
```

```
        w_wind_24 = weather_data["Wind"]
        w_uv_24 = "No hay valores"

        global_weather["Wind"] = w_wind_24
        global_weather["Temperature"] = w_temperature_24
        global_weather["UV"] = []

    elif param1 == "Open Weather":
        weather_data = weather_object.get_openweather()
        w_temperature_24 = weather_data["Temperature"]
        w_wind_24 = weather_data["Wind"]
        w_uv_24 = "No hay valores"

        global_weather["Wind"] = w_wind_24
        global_weather["Temperature"] = w_temperature_24
        global_weather["UV"] = []

    elif param1 == "Tomorrow.io":
        weather_data = weather_object.get_tomorrowoi()
        w_temperature_24 = weather_data["Temperature"]
        w_wind_24 = weather_data["Wind"]["Speed"]
        w_uv_24= weather_data["UV"]

        global_weather["Wind"] = w_wind_24
        global_weather["Temperature"] = w_temperature_24
        global_weather["UV"] = w_uv_24

    elif param1 == "Weather Stack":
        weather_data = weather_object.get_weatherstack()
        w_temperature_24 = str(weather_data["Temperature"]) + " (Solo
        →se ofrece el valor actual)"
        w_wind_24 = str(weather_data["Wind"]["wind_speed"]) + " (Solo
        → se ofrece el valor actual)"
        w_uv_24 = str(weather_data["UVindex"]) + " (Solo se ofrece el
        → valor actual)"

        global_weather["Current"]["wind"] = str(weather_data["Win
        → d"]["wind_speed"])
        global_weather["Current"]["temp"] = str(weather_data["
        → Temperature"])
        global_weather["Current"]["uv"] = str(weather_data["UVindex"])

    elif param1 == "All":
        weather_data = weather_object.get_all()
        w_temperature_24 = weather_data["Temperature"]
        w_wind_24 = weather_data["Wind"]
        w_uv_24= weather_data["UV"]

        global_weather["Wind"] = w_wind_24
        global_weather["Temperature"] = w_temperature_24
        global_weather["UV"] = w_uv_24

    if param2 == "yes":
        current_W_data = weather_object.get_current()
        w_temperature_current = current_W_data["Temperature"]
        w_wind_current = current_W_data["Wind"]
        w_uv_current = current_W_data["UVindex"]
```

```
        global_weather["Current"]["wind"] = w_wind_current
        global_weather["Current"]["temp"] = w_temperature_current
        global_weather["Current"]["uv"] = w_uv_current

    else:
        w_temperature_current = "No selected"
        w_wind_current = "No selected"
        w_uv_current = "No selected"

    # Electricity
    electricity_data = electricity_object.get_preciodelaluz()

    if param4 == "yes":
        electricity_current = electricity_data["Current"]["Time"] + "
    → - " + str(electricity_data["Current"]["Price"])
    else:
        electricity_current = "No selected"

    combined_data = {**electricity_data, **global_weather}

    udp_response = functions.udp_client(combined_data)

    return render_template('data.html', w_temperature_
    → current=w_temperature_current, w_wind_current=w_wind_cur
    → rent, w_uv_current=w_uv_current, electricity_current=electri
    → city_current, w_temperature_24=w_temperature_24, w_wind_24=w_win
    → d_24, w_uv_24=w_uv_24, e_avg=electricity_data["AVG"], e_max=ele
    → ctricity_data["Max"]["Price"], e_min=electricity_da
    → ta["Min"]["Price"], e_nexts=electricity_data["Next"], udp_r
    → esponse=udp_response)
```

Esta función **execute()** recupera los datos enviados mediante el método POST. Posteriormente se crea instancias de los objetos **Weather** y **Electricity** de los módulos **weather** y **electricity**. Además, se crea un objeto JSON llamado *global_weather* que estandarizará la información recibida por parte de las APIs meteorológicas.

Dependiendo de los valores recibidos y almacenados en *param1*, *param2*, *param3* y *param4*, se llaman a los métodos específicos en los objetos instanciados anteriormente para obtener los datos meteorológicos y eléctricos correspondientes. A modo de ejemplo, si se ha seleccionado en el formulario de la página principal el valor AEMET cómo API para obtener la predicción meteorológica, el ***if param1 == "Aemet"*** será verdadero y entonces se llamará, a través del objeto *weather_object*, a la función de la clase **get_aemet()**.

Una vez obtenidos los datos de la llamadas a las APIs, se prepara un objeto JSON llamado *combined_data* con todos los datos para pasarlo a través del cliente UDP (**udp_response = functions.udp_client(combined_data)**) al proceso simulado y unas variables para ser pasadas a

render_template('data.html', *conjunto completo de variables*) que las renderizará en el template data.html.

```
combined_data = {
    "ElectricityJSON": {
        "Current": {
            "Time": None,
            "Price": None
        },
        "AVG": None,
        "Max": {
            "Time": None,
            "Price": None
        },
        "Min": {
            "Time": None,
            "Price": None
        },
        "Next": None
    },
    "global_weather": {
        "Current": {
            "wind": None,
            "temp": None,
            "uv": None
        },
        "Wind": [],
        "Temperature": [],
        "UV": []
    }
}
```

El objeto JSON resultante es la combinación de *ElectricityJSON*, indicado en el apartado 3.2.1.5 y *global_weather*, el cual tiene la siguiente estructura:

- **Current (temp, wind y uv)**: Valor de la temperatura, de la velocidad del viento y de la radiación en el momento actual de la petición.
- **Wind**: Un listado de 24 valores con la predicción de la velocidad del viento para las futuras 24 horas contando desde el momento de la ejecución de la petición.
- **Temperature**: Un listado de 24 valores con el valor de la temperatura en Celsius para las futuras 24 horas contando desde el momento de la ejecución de la petición.
- **UV**: Un listado de 24 valores con el valor de cantidad de radiación solar para las futuras 24 horas contando desde el momento de la ejecución de la petición.

```
if __name__ == "__main__":
    app.run(port=8080)
```

Por último, el código mostrado verifica si el script está siendo ejecutado directamente. Si es así, **app.run(port=8080)** inicia el servidor Flask en el puerto 8080.

3.2.3 Instalación del módulo software en el PLC

Esta parte cubrirá los pasos necesarios para la instalación del módulo software en el PLC industrial ACX F 3152, incluyendo la configuración del entorno, la transferencia del software y las pruebas de funcionamiento en el dispositivo final.

3.2.3.1 Acceso al PLC

La forma de acceso al PLC será desde un ordenador personal haciendo uso del protocolo SSH descrito en el apartado 3.1.4.4, dado que este se encuentra en el laboratorio de la escuela superior de ingeniería industrial, situado en el edificio 5C de la Universidad Politécnica de Valencia.

Para poder realizar el acceso, es necesario que se cumplan los siguientes requisitos:

- Tanto el ordenador personal cómo el PLC deben tener conexión a internet.
- El PLC debe tener una IP pública. En este caso la ID es 158.42.16.178.
- El PLC debe tener un servidor SSH configurado y en ejecución.

Todos los requisitos mencionados anteriormente se cumplían, por ello se ha podido realizar accesos satisfactorios siguiendo las indicaciones que se describirán a continuación. A modo de nota, cabe destacar que no se va a detallar la configuración SSH realizada en el PLC. Esto es debido a que no es parte del desarrollo de este proyecto.

Los pasos para establecer conexión y acceder al PLC son:

1. Conectarse, desde el ordenador del cual se desea establecer la conexión con el PLC, a la VPN de la Universidad Politécnica de Valencia.
2. Abrir la terminal o consola en el ordenador personal y ejecutar el siguiente comando: **ssh admin@158.42.16.178**.
3. Indicar la siguiente contraseña cuando se solicite: 588258f3.
4. Crear un directorio denominado *pythonTest* en el directorio temporal */tmp* para subir el código y ejecutar pruebas. Para ello es necesario ejecutar el siguiente comando: **mkdir /tmp/pythonTest**.

En las siguientes figuras se muestra el proceso de conexión.

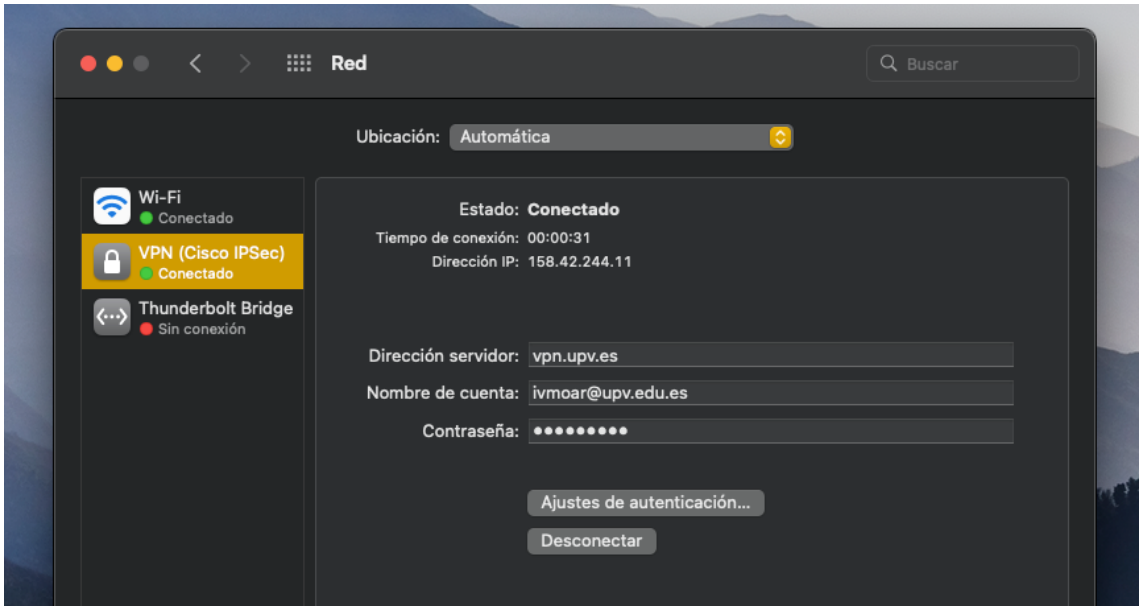

***Figura 30**: conexión a la VPN de la Universidad*
***Fuente**: ordenador*

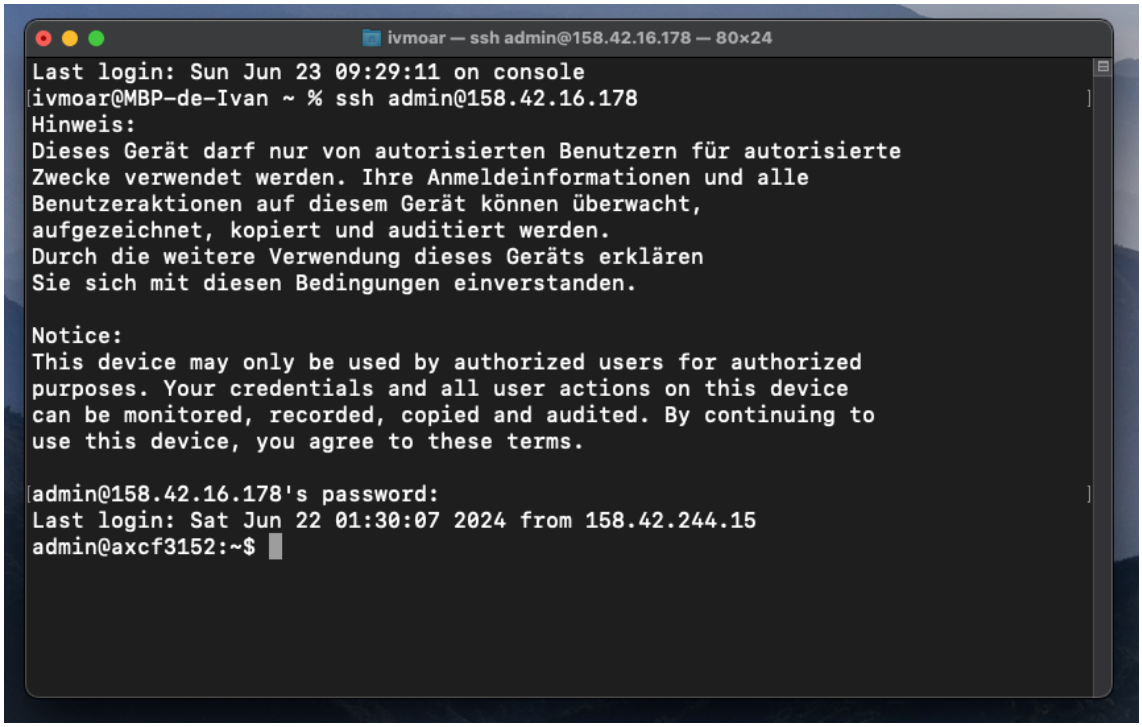

***Figura 31**: acceso mediante el protocolo SSH al PLC*
***Fuente**: ordenador*

3.2.3.2 Configuración del entorno

El siguiente paso después del acceso al PLC consiste en la configuración de este para poder ejecutar el módulo software desarrollado. Para ello se comprobó la versión que había instalada de Python con el siguiente comando: **python3 –-version**. Cómo se puede comprobar en la figura

cincuenta y dos, la versión instalada es la 3.10.8. Esta versión es anterior a la 3.12.3, mencionada en el apartado 3.1.4.2 cómo versión en la que se ha desarrollado el módulo.

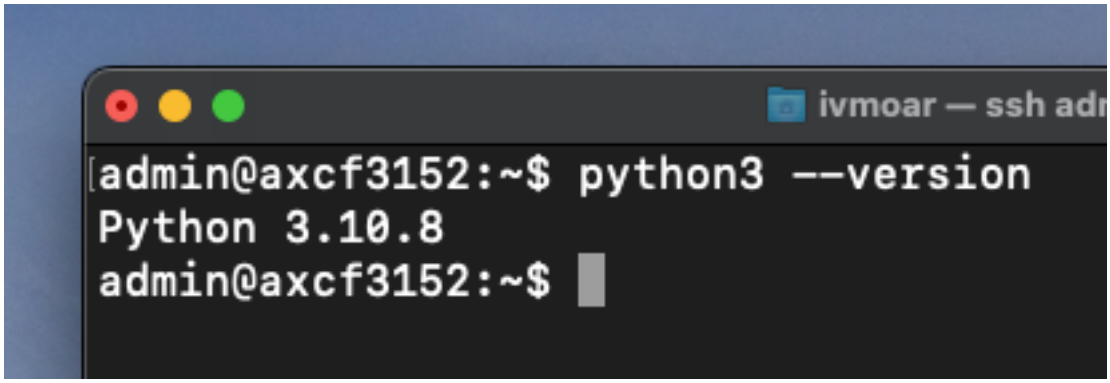

***Figura 32**: versión instalada de Python en el PLC*
***Fuente**: ordenador*

Pese a no disponer de la misma versión, gracias a la compatibilidad del código desarrollado y debido a problemas técnicos en el mismo, no relacionados con este proyecto, es la versión Python con la que se ejecutará el módulo software.

De todas formas, para poder reducir al máximo los futuros posibles problemas derivados de la incompatibilidad de versiones, se creó un entorno virtual Python con los módulos clave del proyecto (requests, numpy y Flask) que se transfirió desde el ordenador de desarrollo al PLC. Los pasos realizados para conseguirlo fueron:

1. Abrir un terminal y crear el entorno virtual en el ordenador de desarrollo con el siguiente comando: **python3 -m venv mytfg**. Esto crea el entorno virtual denominado *mytfg*.
2. Activar el entorno virtual con el siguiente comando: **source mytfg/bin/actívate**.
3. Instalar los módulos necesarios para el desarrollo de este proyecto en el entorno con los siguientes comandos:
    a. **pip install requests**
    b. **pip install Numpy**
    c. **pip install Flask**
4. Exportar el entorno en un archivo *.tar* con el siguiente comando: **tar -czvf mytfg.tar.gz mytfg**.
5. Copiar el archivo *.tar* del entorno virtual en el PLC con el siguiente comando: **scp mytfg.tar.gz** admin@158.42.16.178:/tmp/pythonTest/. Se solicitará la contraseña para poder realizar esta acción.
6. Comprobar la existencia del nuevo archivo *.tar* en el directorio */pythonTest*, Para ello, en el terminal donde se ha establecido la conexión con el PLC por SSH, navegar hasta el directorio con el siguiente comando: **cd /tmp/pythonTest**.

7. Posteriormente, indicar el comando **ls** para listar los archivos existentes y poder comprobar que existe.
8. Descomprimir y extraer el entorno con el siguiente comando: **tar -xzvf mytfg.tar.gz**.
9. Activar el entorno con el siguiente comando: **source mytfg/bin/actívate.**

En la sección de problemas y versiones, se comentará un problema que hubo con el microframework y su correcta instalación en el PLC.

3.2.3.3 Transferencia del software

Una vez ya se había establecido conexión con el PLC y se había configurado el entorno virtual Python para ejecutar el código, se procedió a transferirlo desde el ordenador de desarrollo al PLC mediante el protocolo SSH. Los pasos fueron:

1. Descargar del código en un archivo *.zip* desde el repositorio en *Github*.
2. Transferir el archivo **TFG.zip** al PLC, para ello se debe abrir un terminal en el directorio donde se encuentra el archivo descargado y ejecutar el siguiente comando: **scp TFG.zip admin@158.42.16.178:/tmp/pythonTest**. Se solicitará la contraseña para la ejecución.
3. En la terminal en la que se ha accedido a través de SSH al PLC, acceder a la carpeta *pythonTest* con el siguiente comando: **cd /tmp/pythonTest**. Realizar este paso sólo si no se encontraba ya en la carpeta *pythonTest*.
4. Extraer el contenido del *.zip* con el siguiente comando: **unzip TFG.zip**. Esta acción extraerá una carpeta *TFG_code* en el directorio *pythonTest*.

3.2.3.4 Ejecución y resultado final

Por último, el último paso sería ejecutar el módulo instalado/transferido al PLC después de la configuración y los pasos mencionados previamente. Para ello, es tan sencillo como acceder a la carpeta donde se encuentra el entorno virtual, activarlo y ejecutar el archivo *start.py* del módulo. Los pasos definidos para la ejecución se detallan a continuación.

1. En el terminal conectado al PLC mediante SSH, acceder al directorio /tmp/pythonTest únicamente si no se está ya situado en él. Para ello, ejecutar el comando: **cd /tmp/pythonTest**.
2. Activar el entorno virtual *mytfg* con el siguiente comando: **source mytfg/bin/activate**.
3. Acceder al directorio *TFG_code* con el siguiente comando: **cd TFG_code**.
4. Ejecutar el script *start.py* con el siguiente comando: **python3 start.py**.
5. Acceder desde cualquier dispositivo con acceso a internet a la última URL mostrada por consola. En este caso: http://158.42.16.178.

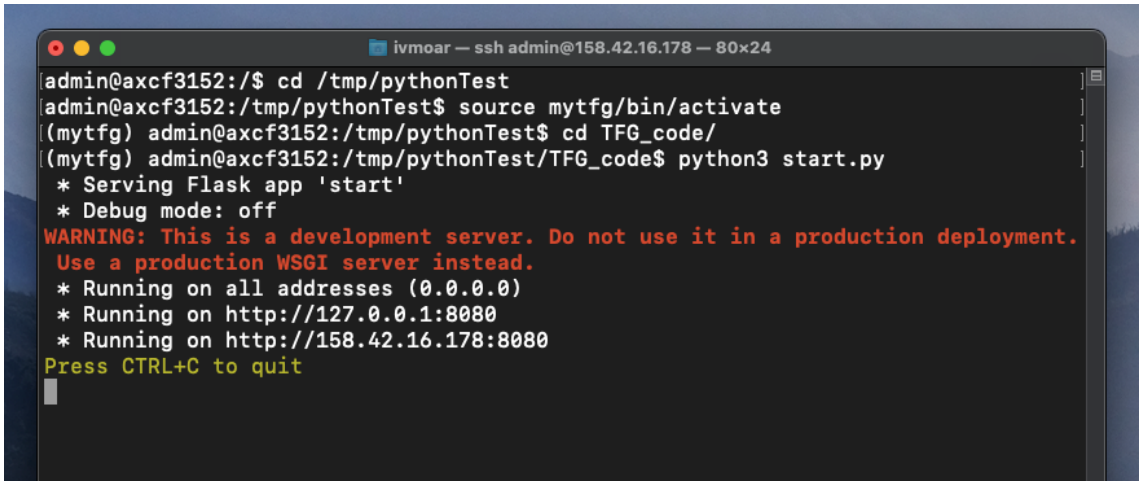

***Figura 33**: ejecución del módulo software en el PLC*
***Fuente**: ordenador*

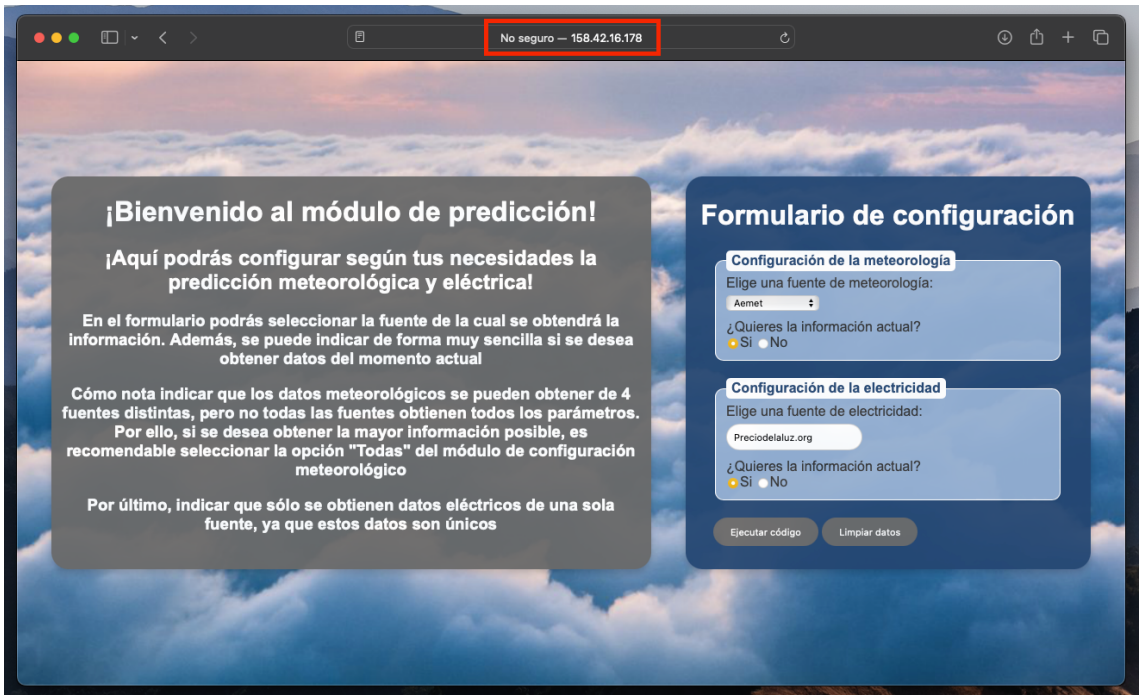

***Figura 34**: acceso a la página web servida desde el PLC*
***Fuente**: http://158.42.16.178:8080*

Con esta ejecución se pone en funcionamiento el resultado final del proyecto, el cual muestra una interfaz amigable a través de un servidor Flask, solicita datos a diferentes APIs meteorológicas y eléctricas, los procesa y presenta haciendo uso de nuevo del servidor Flask. Por último, los envía al Proceso Simulado mediante el protocolo de comunicación UDP.

3.2.4 Desafíos durante el proceso

En cualquier proceso de desarrollo de software, es común encontrarse con una variedad de problemas que pueden afectar al avance del proyecto. Estos problemas pueden surgir en cualquier

fase, desde la planificación inicial hasta la implementación y mantenimiento del software. En esta sección, se expondrán algunos de los problemas que se han encontrado durante el desarrollo del módulo, analizando sus causas y las soluciones propuestas.

3.2.4.1 Problemas de acceso al PLC

Días antes de terminar el desarrollo del código del módulo de predicción, surgieron problemas técnicos en el laboratorio donde se encuentra el PLC que se ha usado en este proyecto. Este problema de conexión retraso de forma considerable la implantación del módulo en el PLC.

Además, las primeras semanas después de la recuperación de los servicios en el laboratorio, el PLC no disponía de acceso a internet. Esto derivó en dos problemas:

1. Al no disponer de internet el PLC, no se podía acceder a través de SSH a no ser que se estuviera en la misma red local. Por suerte, existía la posibilidad de ir de forma presencial al laboratorio. De esta forma se encontró una solución al problema, desplazándose al laboratorio para realizar la transferencia de datos a través de la red local.
2. Este segundo problema tiene la misma causa que el anterior, no disponer de acceso a internet limitaba mucho las acciones a realizar con el PLC. No se podían descargar directamente ni la nueva versión de Python ni los paquetes o frameworks utilizados durante el desarrollo del código. En un primer momento fue frustrante, ya que pese a haber podido transferir el módulo software al PLC sin internet, no existía forma de poder ejecutarlo. Finalmente, después de varias horas de investigación se pudo solucionar el problema realizando el proceso indicado en el apartado 3.2.3.2.

3.2.4.2 Problema con el procesador del ordenador de desarrollo

La primera parte del proceso de desarrollo del código Python se realizó en el programa Visual Studio Code, haciendo uso de un ordenador Macbook Air de 13 pulgadas con procesador M1 de Apple. Todo funcionaba con normalidad hasta que comenzó el desarrollo de las clases.

La figura cincuenta y cinco muestra el error obtenido en la consola de Visual Studio Code después de intentar ejecutar código que hace uso de la librería Numpy. Analizando el error e investigando, se llegó a la conclusión de que la aplicación de Visual Studio Code no estaba preparada para ejecutar ese código en procesadores con arquitectura arm64, cómo la que utiliza el Macbook Air mencionado.

```
Original error was: dlopen(/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/numpy/core/_multiarray_
umath.cpython-312-darwin.so, 0x0002): tried: '/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/nump
y/core/_multiarray_umath.cpython-312-darwin.so' (mach-o file, but is an incompatible architecture (have 'arm64', need 'x86_64')),
'/System/Volumes/Preboot/Cryptexes/OS/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-packages/numpy/core/_m
ultiarray_umath.cpython-312-darwin.so' (no such file), '/Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/site-pac
kages/numpy/core/_multiarray_umath.cpython-312-darwin.so' (mach-o file, but is an incompatible architecture (have 'arm64', need 'x
86_64'))
```

***Figura 35**: error de ejecución de Código*

A partir de este momento, el problema no sólo derivó en la ejecución del código, sino también en el propio desarrollo dentro del IDE. Empezaron a salir incompatibilidades en cada parte del código haciendo imposible su continuidad.

Cómo solución, se cambió de ordenador de desarrollo a un Macbook Pro de 15 pulgadas del 20215 con procesador Intel y arquitectura x86.

4 Conclusiones y trabajos futuros

4.1 Conclusiones

En el presente trabajo se ha logrado desarrollar un módulo software de predicción del coste de la energía eléctrica y de la climatológica para un sistema de gestión de energía basado en PLC industrial. Se han superado varios retos técnicos, incluyendo la integración de diversas APIs, la optimización del código para su ejecución en un entorno de hardware específico y el envío mediante protocolos de comunicación.

A través del proceso de desarrollo, se ha demostrado la viabilidad de utilizar PLCs industriales para tareas complejas de optimización y predicción, lo cual abre nuevas posibilidades para su aplicación en la industria actual gracias a las nuevas capacidades de las que disponen.

Debido al resultado del proyecto presente en este documento, se considera que los objetivos iniciales se han superado adecuadamente. Además, en la relación con los objetivos planteados a nivel personal, se valora que se han cumplido las expectativas con creces. Esto se debe a que se ha podido trabajar en un entorno realista con elementos y herramientas dedicadas a la automatización, y que el módulo software se ha desarrollado haciendo uso de Python, un lenguaje de programación del que no se tenían conocimientos ni se había tenido relación previa.

4.2 Trabajos futuros

Este trabajo representa una versión básica de todo lo que podría llegar a ser un sistema de predicción altamente optimizado y automatizado. A continuación, se indican algunos posibles trabajos futuros y mejoras sobre el presente trabajo:

- Los datos extraídos y presentados cómo solución de las APIs son básicos, se podría considerar el procesamiento de una mayor cantidad de datos, cómo pueden ser la presión atmosférica, la probabilidad de lluvia o incluso la dirección del viento. Además, en caso de necesitar datos más exactos, para periodos más extensos o incluso realizar un mayor número de llamadas, se podría valorar mejorar las subscripciones de las APIs y disponer entonces de mucha más información.
- Un futuro trabajo que podría surgir a raíz de este sería la creación un módulo de predicción de consumo, el cual podría estar constituido por una base de datos y un algoritmo de predicción. La base de datos se utilizaría para recoger la información combinada entre la predicción meteorológica, el precio de la luz y las facturas eléctricas

asociadas a esos datos durante el mes. Por último, el algoritmo podría realizar una estimación económica del consumo dado los datos que se van recogiendo del mes en curso.

- Además, se podría incorporar alguna tecnología de inteligencia artificial. De esta forma, se mejorarían las capacidades predictivas de los algoritmos para obtener resultados más precisos y se podría desarrollar un sistema de recomendaciones en base a los valores actuales que se vayan obteniendo.

5 Bibliografía

[1] Lenguaje de programación Python: https://www.python.org

[2] Información sobre el microframework Flask: https://openwebinars.net/blog/que-es-flask/

[3] Página web Phoenix Contact: https://www.phoenixcontact.com/es-es/

[4] Información específica PLC AXC F 3152: https://www.phoenixcontact.com/es-es/productos/mando-axc-f-3152-1069208

[5] Información sobre las APIs: https://www.geeksforgeeks.org/what-is-an-api/

[6] Información relativa al protocolo SSH: https://medium.com/@aqeelabbas3972/introduction-to-ssh-secure-shell-0d07e18d3149

[7] Página web de información sobre la API de AEMET: https://opendata.aemet.es/centrodedescargas/AEMETApi?

[8] Página web de información sobre la API de OpenWeather: https://openweathermap.org

[9] Página web sobre la API de Tomorrow.io: https://www.tomorrow.io

[10] Página web de Weatherstack: https://weatherstack.com

[11] Página web de la API preciodelaluz.org: https://www.preciodelaluz.org

[12] API meteorológica: https://www.meteomatics.com

[13] API meteorológica: https://www.weatherbit.io

[14] Protocolo UDP: https://www.itmastersmag.com/noticias-analisis/que-es-el-udp-y-para-que-sirve/

[15] Información sobre el Formato JSON: https://www.ibm.com/docs/es/baw/20.x?topic=formats-javascript-object-notation-json-format

[16] Información sobre las herramientas git y GitHub: https://docs.github.com/es/get-started/start-your-journey/about-github-and-git

[17] Información sobre la librería requests de Python: https://j2logo.com/python/python-requests-peticiones-http/

index.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
     → scale=1.0">
    <title>Módulo de predicción - Configuración</title>
    <link rel="stylesheet" href="/static/styles.css">
</head>
<body>
    <div class="container">
        <div class="box-info">
            <h1>¡Bienvenido al módulo de predicción!</h1>
            <h2>
                ¡Aquí podrás configurar según tus necesidades la predi
                 → cción meteorológica y eléctrica!
            </h2>
            <h3>
                En el formulario podrás seleccionar la fuente de la cu
                → al se obtendrá la información. Además, se puede indi
                → car de forma muy sencilla si se desea obtener datos
                → del momento actual
            </h3>
            <h3>
                Cómo nota indicar que los datos meteorológicos se pue
                → den obtener de 4 fuentes distintas, pero no todas la
                → s fuentes obtienen todos los parámetros. Por ello,
                → si se desea obtener la mayor información posible, es
                → recomendable seleccionar la opción "Todas" del mód
                → ulo de configuración meteorológico
            </h3>
            <h3>
                Por último, indicar que sólo se obtienen
                datos eléctricos de una sola fuente, ya que estos da
                → tos son únicos
            </h3>
        </div>
        <div class="box-form">
            <h1>Formulario de configuración</h1>
            <form action="/execute" method="post" id="prediction-fo
            → rm">
                <fieldset>
                    <legend>Configuración de la meteorología</legend>
                    <label for="weather-source">Elige una fuente de me
                    → teorología:</label><br>
                    <select id="weather-source" name="weather-source">
                        <option value="Aemet">Aemet</option>
```

```
                        <option value="Open Weather">Open Weat
                        → her</option>
                        <option value="Tomorrow.io">Tomorrow.io<
                        → /option>
                        <option value="Weather Stack">Weather Sta
                        → ck</option>
                        <option value="All">Todas</option>
                    </select><br>
                    <label>¿Quieres la información actual?</label><br>
                    <label for="w-yes"><input type="radio" id="w-yes"
                    → name="w-option" value="yes" checked>Si</label>
                    <label for="w-no"><input type="radio" id="w-no"
                    → name="w-option" value="no">No</label>
                </fieldset>
                <fieldset>
                    <legend>Configuración de la electricidad</legend>
                    <label for="electricity-source">Elige una fuente
                    → de electricidad:</label><br>
                    <input type="text" id="electricity-source"
                    → name="electricity-source" value="Preciodela
                    → luz.org" readonly><br>
                    <label>¿Quieres la información actual?</label><br>
                    <label for="e-yes"><input type="radio" id="e-yes"
                    → name="e-option" value="yes" checked>Si</label>
                    <label for="e-no"><input type="radio" id="e-no"
                    → name="e-option" value="no">No</label>
                </fieldset>

                <button type="submit">Ejecutar código</button>
                <button type="reset">Limpiar datos</button>
            </form>
        </div>
    </div>
</body>
</html>
```

data.html

```
<!DOCTYPE html>
<html lang="es">
<head>
    <meta charset="UTF-8">
    <meta name="viewport" content="width=device-width, initial-
    → scale=1.0">
    <title>Módulo de predicción - Datos</title>
    <link rel="stylesheet" href="/static/styles.css">
</head>
<body>
    <div class="container">
        <div class="udp-container">
            <h4>Respuesta del servidor UDP: </h4><a class="udp-
            → message">{{udp_response}}</a>
        </div>
        <div class="box-data-w">
            <h1>Datos meteorológicos</h1>
            <fieldset>
                <legend>Predicción</legend>
                <span class="label">Temperatura:</span>
                <span id="w-temperature-24" class="values">{{w_tem
                → perature_24}}</span><br>
                <span class="label">Viento:</span>
                <span id="w-wind-24" class="values">{{w_w
                → ind_24}}</span><br>
                <span class="label">UV:</span>
                <span id="w-uv-24" class="values">{{w_uv_24}}</span>
            </fieldset>
            <fieldset>
                <legend>Valores actuales</legend>
                <span class="label">Temperatura:</span>
                <span id="w-temperature-current" class="values">{{w_
                → temperature_current}}</span>
                <span class="label">Viento:</span>
                <span id="w-wind-current" class="values">{{w_wind_
                → current}}</span>
                <span class="label">UV:</span>
                <span id="w-uv-current" class="values">{{w_uv_c
                → urrent}}</span>
            </fieldset>
        </div>
        <div class="box-data-e">
            <h1>Datos eléctricos</h1>
            <fieldset>
                <legend>Valores de hoy</legend>
                <span class="label">Precio medio:</span>
                <span id="e-avg" class="values">{{e_avg}}</span><br>
                <span class="label">Precio máximo:</span>
                <span id="e-max" class="values">{{e_max}}</span><br>
                <span class="label">Precio mínimo:</span>
                <span id="e-min" class="values">{{e_min}}</span><br>
                <span class="label">Próximos valores:</span>
                <span id="e-nexts" class="values">{{e_nexts}}</span>
```

```
            </fieldset>
            <fieldset>
                <legend>Valores actuales</legend>
                <span class="label">Tiempo - Precio:</span>
                <span id="e-current" class="values">{{elect
                → ricity_current}}</span>
            </fieldset>
        </div>
    </div>
    <div class="button-container">
        <a href="/" class="button-link">Volver</a>
    </div>
</body>
</html>
```

styles.css

```
body, html {
    background-color: lightslategray;
    font-family: Arial, sans-serif;
    margin: 0%;
    height: 100%;
}

body {
    display: flex;
    justify-content: center;
    align-items: center;
}

h1, h2, h3 {
    text-align: center;
    color: rgb(255, 255, 255);
    margin-top: 0;
}

.container {
    display: flex;
    width: 100%;
    height: 100%;
    background-image: url("/static/files/clouds.jpg");
    background-size: cover;
    background-position: center;
    align-items: center;
}

.box-info {
    background-color: dimgray;
    height: 60%;
    margin: 40px 20px 40px 40px;
    flex: 1.5;
    padding: 10px;
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    text-align: center;
    border-radius: 20px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    opacity: 0.95;
}

.box-form {
    background-color: rgb(36, 70, 114);
    height: 60%;
    margin: 40px 40px 40px 20px;
    flex: 1;
    padding: 10px;
    display: flex;
    flex-direction: column;
    justify-content: center;
```

```
    align-items: center;
    border-radius: 20px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    opacity: 0.95;
}

.box-data-w {
    background-color: dimgray;
    height: 60%;
    margin: 40px 20px 40px 40px;
    flex: 1;
    padding: 10px;
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    border-radius: 20px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    opacity: 0.95;
}

.box-data-e {
    background-color: rgb(36, 70, 114);
    height: 60%;
    margin: 40px 40px 40px 20px;
    flex: 1;
    padding: 10px;
    display: flex;
    flex-direction: column;
    justify-content: center;
    align-items: center;
    border-radius: 20px;
    box-shadow: 0 4px 8px rgba(0, 0, 0, 0.2);
    opacity: 0.95;
}

#prediction-form {
    width: 90%;
}

fieldset {
    margin-bottom: 20px;
    border: 1px solid white;
    border-radius: 10px;
    background-color: rgb(147, 174, 209);
}

legend {
    font-weight: bold;
    color: rgb(36, 70, 114);
    background-color: white;
    border-radius: 5px;
    border: 1px solid white;
    padding: 2px 5px 2px 5px;
}

label {
```

```
    color: rgb(49, 49, 49);
}

.values {
    color: white;
}

input[type="text"],
select {
    padding: 8px;
    margin-top: 5px;
    margin-bottom: 10px;
    border: 1px solid #ccc;
    border-radius: 20px;
    box-sizing: border-box;
}

input[type="checkbox"] {
    margin-top: 5px;
    margin-bottom: 10px;
}

button {
    padding: 10px 20px;
    background-color: dimgray;
    color: white;
    border: 0;
    border-radius: 20px;
    cursor: pointer;
}

button:hover {
    background-color: #808080;
}

.button-container {
    position: absolute;
    bottom: 10%;
    width: 100%;
    text-align: center;
}

.button-link {
    padding: 10px 20px;
    background-color: dimgray;
    color: white;
    border: 0;
    border-radius: 20px;
    text-decoration: none;
    display: inline-block;
    cursor: pointer;
}

.button-link:hover {
    background-color: rgb(36, 70, 114);
}
```

```css
/* Estilos para la respuesta UDP */
.udp-container {
    position: absolute;
    top: 5%;
    width: 100%;
    text-align: center;
}

.udp-message {
    padding: 10px 20px;
    background-color: dimgray;
    color: white;
    border: 0;
    border-radius: 20px;
    text-decoration: none;
    display: inline-block;
}
```

constants.py

```
########## APIs DATA ##########
#Aemet
AEMET_KEY = "eyJhbGciOiJIUzI1NiJ9.eyJzdWIiOiJpdmFudGVuaXM2QG
→ hvdG1haWwuY29tIiwianRpIjoiODkwMWVlMWEtNWMyOC00NTEzLW
→ E5YWQtZjNhNzFiODY0MDhlIiwiaXNzIjoiQUVNRVQiLCJpYXQiOj
→ E2OTUyNTA5ODAsInVzZXJJZCI6Ijg5MDFlZTFhLTVjMjgtNDUxM
→ y1hOWFkLWYzYTcxYjg2NDA4ZSIsInJvbGUiOiIifQ.b5rIzkNjXpVvDe
→ swx0jnftViYFiJfIFKdim30oXwukY"
AEMET_VALENCIA_CODE = "46250"
AEMET_URL_OBTAIN_URL = "https://opendata.aemet.es/opendata/api/p
→ rediccion/especifica/municipio/horaria/46250"
AEMET_URL_DATA = "https://opendata.aemet.es/opendata/sh/b331c3c9"
AemetJs = {
    "Current": {
        "wind": None,
        "temp": None
    },
    "Wind": [],
    "Temperature": []
}

#OpenWeather
OPENWEATHER_KEY = "9f74a5005ea5ccfd86c211a8bb40cf70"
OPENWEATHER_VALENCIA_ID = "2509954"
OPENWEATHER_VALENCIA_NAME = "Valencia"
OPENWEATHER_VALENCIA_COD = "200"
OPENWEATHER_VALENCIA_LON = "-0.378"
OPENWEATHER_VALENCIA_LAT = "39.47"
OPENWEATHER_EXCLUDE = "minutely,daily,alerts"
OPENWEATHER_URL_2_5 =
"https://api.openweathermap.org/data/2.5/forecast?lat=39.47&lon=-
0.38&appid=9f74a5005ea5ccfd86c211a8bb40cf70"
OPENWEATHER_URL_3_0 =
"https://api.openweathermap.org/data/3.0/onecall?"
OpenWeatherJs = {
    "Current": {
        "wind": None,
        "temp": None,
        "uvi": None
    },
    "Wind": [],
    "Temperature": [],
    "UV": []
}

#Tomorrow.io
TOMORROWOI_KEY = "1U50XpASQrdgYPG4iEGVel1XkhhR6M7Z"
TOMORROWOI_URL_DAY = "https://api.tomorrow.io/v4/weather/forecast?"
TOMORROWOI_URL_CURRENT =
"https://api.tomorrow.io/v4/weather/realtime?"
TOMORROWOI_LOCATION = "valencia"
TOMORROWOI_TIMESTEPS = "1h"
TomorrowJson = {
```

```
    "Temperature": [],
    "UV": [],
    "Wind": {
        "Speed": [],
        "Dir": [],
    }
}

#WeatherStack
WEATHERSTACK_KEY = "da18d0c4be2e3d33b0cac5fcdd5778d5"
WEATHERSTACK_URL = "http://api.weatherstack.com/current"
WEATHERSTACK_VALENCIA_QUERY = "Valencia, Spain"
WEATHERSTACK_UNITS = "m"
WeatherstackJson = {
    "Wind": {"wind_speed": None, "wind_degree": None, "wind_d
    → ir": None},
    "UVindex": None,
    "Temperature": None
}

#Current
CurrentJson = {
    "Wind": None,
    "UVindex": None,
    "Temperature": None
}

#All
AllJson = {
    "Wind": [],
    "Temperature": [],
    "UV": []
}

###################### - preciodelaluz API - ######################

ZONE_PENINSULA_CANARIAS_BALEARS = "PCB"
ZONE_CEUTA_MELILLA = "CYM"
PRECIODELALUZ_ALL_URL = "https://api.preciodelaluz.org/v1/prices/all?"
PRECIODELALUZ_AVG_URL = "https://api.preciodelaluz.org/v1/prices/avg?"
PRECIODELALUZ_MAX_URL = "https://api.preciodelaluz.org/v1/prices/max?"
PRECIODELALUZ_MIN_URL = "https://api.preciodelaluz.org/v1/prices/min?"
PRECIODELALUZ_NOW_URL = "https://api.preciodelaluz.org/v1/prices/now?"

ElectricityJSON = {
    "Current": {
        "Time": None,
        "Price": None
    },
    "AVG": None,
    "Max": {
        "Time": None,
        "Price": None
    },
    "Min": {
        "Time": None,
        "Price": None
```

```
    },
    "Next": None
}

###################### - UDP - ######################
UDP_URL = "http://158.42.16.178"
UDP_PORT = 15500
```

functions.py

```
import requests
import numpy as np
from constants import *
import socket
import json

# Petición https - AEMET
def AemetRequest(URL):
    querystring = {"api_key": AEMET_KEY}
    headers = {'cache-control': "no-cache"}
    response = requests.request("GET", URL, headers=headers, pa
    → rams=querystring)
    return response.json()

# Petición https - OpenWeather
def OpenWeatherRequest(URL):
    params = {
        "lat": OPENWEATHER_VALENCIA_LAT,
        "lon": OPENWEATHER_VALENCIA_LON,
        "exclude": OPENWEATHER_EXCLUDE,
        "appid": OPENWEATHER_KEY
    }
    response = requests.request("GET", URL, params=params)
    return response.json()

def kelvin_celsius(kelvin):
    return kelvin - 273.15

# Petición https - Tomorrow.io
def TomorrowRequest(URL):
    params = {
        'location': TOMORROWOI_LOCATION,
        'timesteps': TOMORROWOI_TIMESTEPS,
        'apikey': TOMORROWOI_KEY,
    }
    headers = {"accept": "application/json"}
    response = requests.request("GET", URL, headers=headers, par
    → ams=params)
    return response.json()

# Petición https - WeatherStack
def WeatherstackRequest(URL):
    params = {
        'access_key': WEATHERSTACK_KEY,
        'query': WEATHERSTACK_VALENCIA_QUERY,
        'units': WEATHERSTACK_UNITS
    }
    headers = {'cache-control': "no-cache"}
    response = requests.request("GET", URL, headers=headers, par
    → ams=params)
    return response.json()

# Petición https - preciodelaluz
def PreciodelaluzRequest(URL):
```

```
    params = {
        'zone': ZONE_PENINSULA_CANARIAS_BALEARS
    }
    headers = {'cache-control': "no-cache"}
    response = requests.request("GET", URL, headers=headers, par
    → ams=params)
    return response.json()

# UDP
def udp_client(data):
    server_address = (UDP_URL, UDP_PORT)
    client_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
    try:
        message = json.dumps(data)
        client_socket.sendto(message.encode(), server_address)
        response, _ = client_socket.recvfrom(4096)
        return response.decode()
    finally:
        client_socket.close()
```

OBJETIVOS DE DESARROLLO SOSTENIBLE

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenibles	Alto	Medio	Bajo	No Procede
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**				X
ODS 5. **Igualdad de género.**				X
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**	X			
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades**.				X
ODS 11. **Ciudades y comunidades sostenibles**.	X			
ODS 12. **Producción y consumo responsables.**	X			
ODS 13. **Acción por el clima.**			X	
ODS 14. **Vida submarina.**			X	
ODS 15. **Vida de ecosistemas terrestres.**			X	
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Debido a la incorporación de los objetivos a la Agenda 2030, los proyectos de ingeniería han tenido que adaptarse y orientarse hacia la consecución de estos en la medida de lo posible. La Agenda 2030, adoptada por los países miembros de las Naciones Unidas, establece una serie de Objetivos de Desarrollo Sostenible (ODS) que abordan desafíos globales como la pobreza, la desigualdad, el cambio climático, la degradación ambiental y la paz y justicia.

Este compromiso es crucial para todos los ingenieros, ya que tienen el deber ético y profesional de contribuir progresivamente a alcanzar los ODS. Los ingenieros desempeñan un papel fundamental en el diseño, la construcción y la gestión de infraestructuras y tecnologías que pueden tener un impacto significativo en la sostenibilidad y el bienestar global.

Por ello, este trabajo se ha enfocado en cumplir con los Objetivos de Desarrollo Sostenible que están directamente relacionados con el campo de estudio seleccionado.

En este caso, el ODS 9 está vinculado con la industria y la innovación. Uno de los objetivos de este proyecto es aprovechar al máximo la potencia de los PLCs industriales de última generación, estableciendo precedentes para futuras propuestas, y así, puedan utilizar plenamente las capacidades de los PLCs al ejecutar ciertos lenguajes de programación.

Además, el ODS 11 se centra en las ciudades y comunidades sostenibles, este objetivo tiene aplicación directa en este trabajo al mejorar la gestión energética en entornos urbanos o comunitarios. El diseño del módulo software permite optimizar el uso de recursos energéticos, facilitando una gestión más eficiente y sostenible dentro de instalaciones de autoconsumo. Esto podría contribuir a garantizar un suministro energético estable y sostenible para las comunidades locales, al tiempo que ayudar a reducir la huella ambiental asociada con el consumo energético.

Por otro lado, el ODS 12, que promueve la producción y consumo responsables, también se ve fortalecido por este proyecto. El módulo software facilita la predicción precisa de los costos de energía y las condiciones climáticas, permitiendo que el sistema de gestión de energía tome decisiones informadas para optimizar el consumo de recursos y minimizar los desperdicios. Esto promueve prácticas de producción y consumo más eficientes y responsables, contribuyendo así al fomento de patrones sostenibles de desarrollo.

Por último, los ODS 13 y 14 tienen una aplicación directa en este proyecto. El ODS 13, centrado en la acción por el clima, es relevante porque el diseño del módulo software para la predicción del coste de la energía eléctrica y climatológica contribuye a la gestión sostenible de la energía, reduciendo las emisiones de gases de efecto invernadero y promoviendo prácticas energéticas más eficientes. Al mismo tiempo, el ODS 14, que se enfoca en la vida submarina, se beneficia indirectamente al mitigar la contaminación y preservar los recursos naturales mediante el uso responsable de la energía y la reducción de la huella ambiental.

Hay que destacar también el ODS 15, que promueve la vida de los ecosistemas terrestres. La implementación de prácticas de gestión energética sostenible contribuye a la conservación de la biodiversidad y los recursos naturales en entornos urbanos y rurales, garantizando un equilibrio ambiental que sustenta la salud de los ecosistemas terrestres a largo plazo.